wxWidgets 2.6.0: A portable C++ and Python GUI toolkit

Julian Smart, Robert Roebling, Vadim Zeitlin, Robin Dunn, et al

April, 2005

Contents

Chapter 1 Copyright notice

Copyright (c) 1992-2002 Julian Smart, Robert Roebling, Vadim Zeitlin and other members of the wxWidgets team
Portions (c) 1996 Artificial Intelligence Applications Institute

Please also see the wxWindows license files (preamble.txt, lgpl.txt, gpl.txt, licence.txt, licendoc.txt) for conditions of software and documentation use. Note that we use the old name wxWindows in the license, pending recognition of the new name by OSI.

wxWindows Library License, Version 3

Copyright (c) 1992-2004 Julian Smart, Robert Roebling, Vadim Zeitlin et al.

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

WXWINDOWS LIBRARY LICENSE
TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

This library is free software; you can redistribute it and/or modify it under the terms of the GNU Library General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This library is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Library General Public License for more details.

You should have received a copy of the GNU Library General Public License along with this software, usually in a file named COPYING.LIB. If not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA~02111-1307 USA.

EXCEPTION NOTICE

1. As a special exception, the copyright holders of this library give permission for additional uses of the text contained in this release of the library as licensed under the wxWindows Library License, applying either version 3 of the License, or (at your option) any later version of the License as published by the copyright holders of version 3 of the License document.

2. The exception is that you may create binary object code versions of any works using this library or based on this library, and use, copy, modify, link and distribute such binary object code files unrestricted under terms of your choice.

3. If you copy code from files distributed under the terms of the GNU General Public License or the GNU Library General Public License into a copy of this library, as this

license permits, the exception does not apply to the code that you add in this way. To avoid misleading anyone as to the status of such modified files, you must delete this exception notice from such code and/or adjust the licensing conditions notice accordingly.

4. If you write modifications of your own for this library, it is your choice whether to permit this exception to apply to your modifications. If you do not wish that, you must delete the exception notice from such code and/or adjust the licensing conditions notice accordingly.

GNU Library General Public License, Version 2

Copyright (C) 1991 Free Software Foundation, Inc. 675 Mass Ave, Cambridge, MA 02139, USA

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

[This is the first released version of the library GPL. It is numbered 2 because it goes with version 2 of the ordinary GPL.]

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public Licenses are intended to guarantee your freedom to share and change free software -- to make sure the software is free for all its users.

This license, the Library General Public License, applies to some specially designated Free Software Foundation software, and to any other libraries whose authors decide to use it. You can use it for your libraries, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the library, or if you modify it.

For example, if you distribute copies of the library, whether gratis or for a fee, you must give the recipients all the rights that we gave you. You must make sure that they, too, receive or can get the source code. If you link a program with the library, you must provide complete object files to the recipients so that they can relink them with the library, after making changes to the library and recompiling it. And you must show them these terms so they know their rights.

Our method of protecting your rights has two steps: (1) copyright the library, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the library.

Also, for each distributor's protection, we want to make certain that everyone understands that there is no warranty for this free library. If the library is modified by someone else and passed on, we want its recipients to know that what they have is not the original version, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that companies distributing free software will individually obtain patent licenses, thus in effect transforming the program into proprietary software. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

Most GNU software, including some libraries, is covered by the ordinary GNU General Public License, which was designed for utility programs. This license, the GNU Library General Public License, applies to certain designated libraries. This license is quite different from the ordinary one; be sure to read it in full, and don't assume that anything in it is the same as in the ordinary license.

The reason we have a separate public license for some libraries is that they blur the distinction we usually make between modifying or adding to a program and simply using it. Linking a program with a library, without changing the library, is in some sense simply using the library, and is analogous to running a utility program or application program. However, in a textual and legal sense, the linked executable is a combined work, a derivative of the original library, and the ordinary General Public License treats it as such.

Because of this blurred distinction, using the ordinary General Public License for libraries did not effectively promote software sharing, because most developers did not use the libraries. We concluded that weaker conditions might promote sharing better.

However, unrestricted linking of non-free programs would deprive the users of those programs of all benefit from the free status of the libraries themselves. This Library General Public License is intended to permit developers of non-free programs to use free libraries, while preserving your freedom as a user of such programs to change the free libraries that are incorporated in them. (We have not seen how to achieve this as regards changes in header files, but we have achieved it as regards changes in the actual functions of the Library.) The hope is that this will lead to faster development of free libraries.

The precise terms and conditions for copying, distribution and modification follow. Pay close attention to the difference between a "work based on the library" and a "work that uses the library". The former contains code derived from the library, while the latter only works together with the library.

Note that it is possible for a library to be covered by the ordinary General Public License rather than by this special one.

<div align="center">

GNU LIBRARY GENERAL PUBLIC LICENSE
TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

</div>

0. This License Agreement applies to any software library which contains a notice placed by the copyright holder or other authorized party saying it may be distributed under the terms of this Library General Public License (also called "this License"). Each licensee is

addressed as "you".

A "library" means a collection of software functions and/or data prepared so as to be conveniently linked with application programs (which use some of those functions and data) to form executables.

The "Library", below, refers to any such software library or work which has been distributed under these terms. A "work based on the Library" means either the Library or any derivative work under copyright law: that is to say, a work containing the Library or a portion of it, either verbatim or with modifications and/or translated straightforwardly into another language. (Hereinafter, translation is included without limitation in the term "modification".)

"Source code" for a work means the preferred form of the work for making modifications to it. For a library, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the library.

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running a program using the Library is not restricted, and output from such a program is covered only if its contents constitute a work based on the Library (independent of the use of the Library in a tool for writing it). Whether that is true depends on what the Library does and what the program that uses the Library does.

1. You may copy and distribute verbatim copies of the Library's complete source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and distribute a copy of this License along with the Library.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Library or any portion of it, thus forming a work based on the Library, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

    a) The modified work must itself be a software library.

    b) You must cause the files modified to carry prominent notices stating that you changed the files and the date of any change.

    c) You must cause the whole of the work to be licensed at no charge to all third parties under the terms of this License.

    d) If a facility in the modified Library refers to a function or a table of data to be supplied by an application program that uses the facility, other than as an argument passed when the facility is invoked, then you must make a good faith effort to ensure that, in the event an application does not supply such function or table, the facility still operates, and performs whatever part of its purpose remains meaningful.

(For example, a function in a library to compute square roots has a purpose that is entirely well-defined independent of the application. Therefore, Subsection 2d requires that any application-supplied function or table used by this function must be optional: if the application does not supply it, the square root function must still compute square roots.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Library, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Library, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Library.

In addition, mere aggregation of another work not based on the Library with the Library (or with a work based on the Library) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may opt to apply the terms of the ordinary GNU General Public License instead of this License to a given copy of the Library. To do this, you must alter all the notices that refer to this License, so that they refer to the ordinary GNU General Public License, version 2, instead of to this License. (If a newer version than version 2 of the ordinary GNU General Public License has appeared, then you can specify that version instead if you wish.) Do not make any other change in these notices.

Once this change is made in a given copy, it is irreversible for that copy, so the ordinary GNU General Public License applies to all subsequent copies and derivative works made from that copy.

This option is useful when you wish to copy part of the code of the Library into a program that is not a library.

4. You may copy and distribute the Library (or a portion or derivative of it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange.

If distribution of object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place satisfies the requirement to distribute the source code, even though third parties are not compelled to copy the source along with the object code.

5. A program that contains no derivative of any portion of the Library, but is designed to work with the Library by being compiled or linked with it, is called a "work that uses the Library". Such a work, in isolation, is not a derivative work of the Library, and therefore falls outside the scope of this License.

However, linking a "work that uses the Library" with the Library creates an executable

that is a derivative of the Library (because it contains portions of the Library), rather than a "work that uses the library". The executable is therefore covered by this License. Section 6 states terms for distribution of such executables.

When a "work that uses the Library" uses material from a header file that is part of the Library, the object code for the work may be a derivative work of the Library even though the source code is not. Whether this is true is especially significant if the work can be linked without the Library, or if the work is itself a library. The threshold for this to be true is not precisely defined by law.

If such an object file uses only numerical parameters, data structure layouts and accessors, and small macros and small inline functions (ten lines or less in length), then the use of the object file is unrestricted, regardless of whether it is legally a derivative work. (Executables containing this object code plus portions of the Library will still fall under Section 6.)

Otherwise, if the work is a derivative of the Library, you may distribute the object code for the work under the terms of Section 6. Any executables containing that work also fall under Section 6, whether or not they are linked directly with the Library itself.

6. As an exception to the Sections above, you may also compile or link a "work that uses the Library" with the Library to produce a work containing portions of the Library, and distribute that work under terms of your choice, provided that the terms permit modification of the work for the customer's own use and reverse engineering for debugging such modifications.

You must give prominent notice with each copy of the work that the Library is used in it and that the Library and its use are covered by this License. You must supply a copy of this License. If the work during execution displays copyright notices, you must include the copyright notice for the Library among them, as well as a reference directing the user to the copy of this License. Also, you must do one of these things:

  a) Accompany the work with the complete corresponding machine-readable source code for the Library including whatever changes were used in the work (which must be distributed under Sections 1 and 2 above); and, if the work is an executable linked with the Library, with the complete machine-readable "work that uses the Library", as object code and/or source code, so that the user can modify the Library and then relink to produce a modified executable containing the modified Library. (It is understood that the user who changes the contents of definitions files in the Library will not necessarily be able to recompile the application to use the modified definitions.)

  b) Accompany the work with a written offer, valid for at least three years, to give the same user the materials specified in Subsection 6a, above, for a charge no more than the cost of performing this distribution.

  c) If distribution of the work is made by offering access to copy from a designated place, offer equivalent access to copy the above specified materials from the same place.

  d) Verify that the user has already received a copy of these materials or that you have already sent this user a copy.

For an executable, the required form of the "work that uses the Library" must include any data and utility programs needed for reproducing the executable from it. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

It may happen that this requirement contradicts the license restrictions of other proprietary libraries that do not normally accompany the operating system. Such a contradiction means you cannot use both them and the Library together in an executable that you distribute.

7. You may place library facilities that are a work based on the Library side-by-side in a single library together with other library facilities not covered by this License, and distribute such a combined library, provided that the separate distribution of the work based on the Library and of the other library facilities is otherwise permitted, and provided that you do these two things:

> a) Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities. This must be distributed under the terms of the Sections above.

> b) Give prominent notice with the combined library of the fact that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.

8. You may not copy, modify, sublicense, link with, or distribute the Library except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, link with, or distribute the Library is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

9. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Library or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Library (or any work based on the Library), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Library or works based on it.

10. Each time you redistribute the Library (or any work based on the Library), the recipient automatically receives a license from the original licensor to copy, distribute, link with or modify the Library subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

11. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Library at all. For

example, if a patent license would not permit royalty-free redistribution of the Library by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Library.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply, and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

12. If the distribution and/or use of the Library is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Library under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

13. The Free Software Foundation may publish revised and/or new versions of the Library General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Library specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Library does not specify a license version number, you may choose any version ever published by the Free Software Foundation.

14. If you wish to incorporate parts of the Library into other free programs whose distribution conditions are incompatible with these, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

15. BECAUSE THE LIBRARY IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE LIBRARY, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE LIBRARY "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND

FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE LIBRARY IS WITH YOU. SHOULD THE LIBRARY PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE LIBRARY AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE LIBRARY (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE LIBRARY TO OPERATE WITH ANY OTHER SOFTWARE), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

<div align="center">END OF TERMS AND CONDITIONS</div>

Appendix: How to Apply These Terms to Your New Libraries

If you develop a new library, and you want it to be of the greatest possible use to the public, we recommend making it free software that everyone can redistribute and change. You can do so by permitting redistribution under these terms (or, alternatively, under the terms of the ordinary General Public License).

To apply these terms, attach the following notices to the library. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
<one line to give the library's name and a brief idea of what it does.>
Copyright (C) <year> <name of author>

This library is free software; you can redistribute it and/or
modify it under the terms of the GNU Library General Public
License as published by the Free Software Foundation; either
version 2 of the License, or (at your option) any later version.

This library is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
Library General Public License for more details.

You should have received a copy of the GNU Library General Public
License along with this library; if not, write to the Free
Software Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
```

Also add information on how to contact you by electronic and paper mail.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the library, if necessary. Here is a sample; alter the names:

```
Yoyodyne, Inc., hereby disclaims all copyright interest in the
library `Frob' (a library for tweaking knobs) written by James Random
Hacker.
```

```
<signature of Ty Coon>, 1 April 1990
Ty Coon, President of Vice
```

That's all there is to it!

Chapter 2 Introduction

What is wxWidgets?

wxWidgets is a C++ framework providing GUI (Graphical User Interface) and other facilities on more than one platform. Version 2 currently supports all desktop versions of MS Windows, Unix with GTK+, Unix with Motif, and MacOS. An OS/2 port is in progress.

wxWidgets was originally developed at the Artificial Intelligence Applications Institute, University of Edinburgh, for internal use, and was first made publicly available in 1992. Version 2 is a vastly improved version written and maintained by Julian Smart, Robert Roebling, Vadim Zeitlin, Vaclav Slavik and many others.

This manual contains a class reference and topic overviews. For a selection of wxWidgets tutorials, please see the documentation page on the wxWidgets web site (`http://www.wxwidgets.org`).

Please note that in the following, "MS Windows" often refers to all platforms related to Microsoft Windows, including 16-bit and 32-bit variants, unless otherwise stated. All trademarks are acknowledged.

Why another cross-platform development tool?

wxWidgets was developed to provide a cheap and flexible way to maximize investment in GUI application development. While a number of commercial class libraries already existed for cross-platform development, none met all of the following criteria:

1. low price;

2. source availability;

3. simplicity of programming;

4. support for a wide range of compilers.

Since wxWidgets was started, several other free or almost-free GUI frameworks have emerged. However, none has the range of features, flexibility, documentation and the well-established development team that wxWidgets has.

As open source software, wxWidgets has benefited from comments, ideas, bug fixes, enhancements and the sheer enthusiasm of users. This gives wxWidgets a certain advantage over its commercial competitors (and over free libraries without an independent development team), plus a robustness against the transience of one individual or company. This openness and availability of source code is especially important when the future of thousands of lines of application code may depend upon the longevity of the underlying class library.

Version 2 goes much further than previous versions in terms of generality and features, allowing applications to be produced that are often indistinguishable from those produced using single-platform toolkits such as Motif, GTK+ and MFC.

The importance of using a platform-independent class library cannot be overstated, since GUI application development is very time-consuming, and sustained popularity of particular GUIs cannot be guaranteed. Code can very quickly become obsolete if it addresses the wrong platform or audience. wxWidgets helps to insulate the programmer from these winds of change. Although wxWidgets may not be suitable for every application (such as an OLE-intensive program), it provides access to most of the functionality a GUI program normally requires, plus many extras such as network programming, PostScript output, and HTML rendering; and it can of course be extended as needs dictate. As a bonus, it provides a far cleaner and easier programming interface than the native APIs. Programmers may find it worthwhile to use wxWidgets even if they are developing on only one platform.

It is impossible to sum up the functionality of wxWidgets in a few paragraphs, but here are some of the benefits:

- Low cost (free, in fact!)

- You get the source.

- Available on a variety of popular platforms.

- Works with almost all popular C++ compilers and Python.

- Over 50 example programs.

- Over 1000 pages of printable and on-line documentation.

- Includes Tex2RTF, to allow you to produce your own documentation in Windows Help, HTML and Word RTF formats.

- Simple-to-use, object-oriented API.

- Flexible event system.

- Graphics calls include lines, rounded rectangles, splines, polylines, etc.

- Constraint-based and sizer-based layouts.

- Print/preview and document/view architectures.

- Toolbar, notebook, tree control, advanced list control classes.

- PostScript generation under Unix, normal MS Windows printing on the PC.

- MDI (Multiple Document Interface) support.

- Can be used to create DLLs under Windows, dynamic libraries on Unix.

- Common dialogs for file browsing, printing, colour selection, etc.

- Under MS Windows, support for creating metafiles and copying them to the clipboard.

- An API for invoking help from applications.

- Ready-to-use HTML window (supporting a subset of HTML).

- Network support via a family of socket and protocol classes.

- Support for platform independent image processing.

- Built-in support for many file formats (BMP, PNG, JPEG, GIF, XPM, PNM, PCX).

wxWidgets requirements

To make use of wxWidgets, you currently need one of the following setups.

(a) MS-Windows:

1. A 32-bit or 64-bit PC running MS Windows.

2. A Windows compiler: MS Visual C++ (embedded Visual C++ for wxWinCE port), Borland C++, Watcom C++, Cygwin, MinGW, Metrowerks CodeWarrior, Digital Mars C++. See `install.txt` for details about compiler version supported.

3. At least 100 MB of disk space for source tree and additional space for libraries and application building (depends on compiler and build settings).

(b) Unix:

1. Almost any C++ compiler, including GNU C++ (EGCS 1.1.1 or above).

2. Almost any Unix workstation, and one of: GTK+ 1.2, GTK+ 2.0, Motif 1.2 or higher, Lesstif. If using the wxX11 port, no such widget set is required.

3. At least 100 MB of disk space for source tree and additional space for libraries and application building (depends on compiler and build settings).

(c) Mac OS/Mac OS X:

1. A PowerPC Mac running Mac OS 8.6/9.x (eg. Classic) or Mac OS X 10.x.

2. CodeWarrior 5.3, 6 or 7 for Classic Mac OS.

3. The Apple Developer Tools (eg. GNU C++), CodeWarrior 7 or above for Mac OS X.

4. At least 100 MB of disk space for source tree and additional space for libraries and application building (depends on compiler and build settings).

Availability and location of wxWidgets

wxWidgets is available by anonymous FTP and World Wide Web from
ftp://biolpc22.york.ac.uk/pub (`ftp://biolpc22.york.ac.uk/pub`) and/or
http://www.wxwidgets.org (`http://www.wxwidgets.org`).

You can also buy a CD-ROM using the form on the Web site.

Acknowledgements

Thanks are due to AIAI for being willing to release the original version of wxWidgets into the public domain, and to our patient partners.

We would particularly like to thank the following for their contributions to wxWidgets, and the many others who have been involved in the project over the years. Apologies for any unintentional omissions from this list.   Yiorgos Adamopoulos, Jamshid Afshar, Alejandro Aguilar-Sierra, AIAI, Patrick Albert, Karsten Ballueder, Mattia Barbon, Michael Bedward, Kai Bendorf, Yura Bidus, Keith Gary Boyce, Chris Breeze, Pete Britton, Ian Brown, C. Buckley, Marco Cavallini, Dmitri Chubraev, Robin Corbet, Cecil Coupe, Stefan Csomor, Andrew Davison, Gilles Depeyrot, Neil Dudman, Robin Dunn, Hermann Dunkel, Jos van Eijndhoven, Chris Elliott, David Elliott, Tom Felici, Thomas Fettig, Matthew Flatt, Pasquale Foggia, Josep Fortiana, Todd Fries, Dominic Gallagher, Guillermo Rodriguez Garcia, Wolfram Gloger, Norbert Grotz, Stefan Gunter, Bill Hale, Patrick Halke, Stefan Hammes, Guillaume Helle, Harco de Hilster, Cord Hockemeyer, Markus Holzem, Olaf Klein, Leif Jensen, Bart Jourquin, Guilhem Lavaux, Ron Lee, Jan Lessner, Nicholas Liebmann, Torsten Liermann, Per Lindqvist, Thomas Runge, Tatu Männistö, Scott Maxwell, Thomas Myers, Oliver Niedung, Stefan Neis, Hernan Otero, Ian Perrigo, Timothy Peters, Giordano Pezzoli, Harri Pasanen, Thomaso Paoletti, Garrett Potts, Marcel Rasche, Robert Roebling, Dino Scaringella, Jobst Schmalenbach, Arthur Seaton, Paul Shirley, Wlodzimierz 'ABX' Skiba, Vaclav Slavik, Julian Smart, Stein Somers, Petr Smilauer, Neil Smith, Kari Systä, George Tasker, Arthur Tetzlaff-Deas, Jonathan Tonberg, Jyrki Tuomi, Janos Vegh, Andrea Venturoli, David Webster, Otto Wyss, Vadim Zeitlin, Xiaokun Zhu, Edward Zimmermann.

'Graphplace', the basis for the wxGraphLayout library, is copyright Dr. Jos T.J. van Eijndhoven of Eindhoven University of Technology. The code has been used in wxGraphLayout with his permission.

We also acknowledge the author of XFIG, the excellent Unix drawing tool, from the source of which we have borrowed some spline drawing code. His copyright is included below.

XFig2.1 is copyright (c) 1985 by Supoj Sutanthavibul. Permission to use, copy, modify, distribute, and sell this software and its documentation for any purpose is hereby granted without fee, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of M.I.T. not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission. M.I.T. makes no representations about the suitability of this software for any purpose. It is provided "as is" without express or implied warranty.

Chapter 3 Multi-platform development with wxWidgets

This chapter describes the practical details of using wxWidgets. Please see the file install.txt for up-to-date installation instructions, and changes.txt for differences between versions.

Include files

The main include file is "`wx/wx.h`"; this includes the most commonly used modules of wxWidgets.

To save on compilation time, include only those header files relevant to the source file. If you are using precompiled headers, you should include the following section before any other includes:

```
// For compilers that support precompilation, includes "wx.h".
#include <wx/wxprec.h>

#ifdef __BORLANDC__
#pragma hdrstop
#endif

#ifndef WX_PRECOMP
// Include your minimal set of headers here, or wx.h
#include <wx/wx.h>
#endif

... now your other include files ...
```

The file "`wx/wxprec.h`" includes "`wx/wx.h`". Although this incantation may seem quirky, it is in fact the end result of a lot of experimentation, and several Windows compilers to use precompilation which is largely automatic for compilers with necessary support. Currently it is used for Visual C++ (including embedded Visual C++), Borland C++, Open Watcom C++ and newer versions of GCC.

Libraries

Most ports of wxWidgets can create either a static library or a shared library. wxWidgets can also be built in multilib and monolithic variants. See the *libraries list* (p. 15) for more information on these.

Configuration

When using project files and makefiles directly to build wxWidgets, options are

configurable in the file "`wx/XXX/setup.h`" where XXX is the required platform (such as msw, motif, gtk, mac). Some settings are a matter of taste, some help with platform-specific problems, and others can be set to minimize the size of the library. Please see the setup.h file and `install.txt` files for details on configuration.

When using the 'configure' script to configure wxWidgets (on Unix and other platforms where configure is available), the corresponding setup.h files are generated automatically along with suitable makefiles. When using the RPM packages for installing wxWidgets on Linux, a correct setup.h is shipped in the package and this must not be changed.

Makefiles

On Microsoft Windows, wxWidgets has a different set of makefiles for each compiler, because each compiler's 'make' tool is slightly different. Popular Windows compilers that we cater for, and the corresponding makefile extensions, include: Microsoft Visual C++ (.vc), Borland C++ (.bcc), OpenWatcom C++ (.wat) and MinGW/Cygwin (.gcc). Makefiles are provided for the wxWidgets library itself, samples, demos, and utilities.

On Linux, Mac and OS/2, you use the 'configure' command to generate the necessary makefiles. You should also use this method when building with MinGW/Cygwin on Windows.

We also provide project files for some compilers, such as Microsoft VC++. However, we recommend using makefiles to build the wxWidgets library itself, because makefiles can be more powerful and less manual intervention is required.

On Windows using a compiler other than MinGW/Cygwin, you would build the wxWidgets library from the build/msw directory which contains the relevant makefiles.

On Windows using MinGW/Cygwin, and on Unix, MacOS X and OS/2, you invoke 'configure' (found in the top-level of the wxWidgets source hierarchy), from within a suitable empty directory for containing makefiles, object files and libraries.

For details on using makefiles, configure, and project files, please see docs/xxx/install.txt in your distribution, where xxx is the platform of interest, such as msw, gtk, x11, mac.

Windows-specific files

wxWidgets application compilation under MS Windows requires at least two extra files, resource and module definition files.

Resource file

The least that must be defined in the Windows resource file (extension RC) is the following statement:

```
#include "wx/msw/wx.rc"
```

which includes essential internal wxWidgets definitions. The resource script may also contain references to icons, cursors, etc., for example:

```
wxicon icon wx.ico
```

The icon can then be referenced by name when creating a frame icon. See the MS Windows SDK documentation.

Note: include wx.rc *after* any ICON statements so programs that search your executable for icons (such as the Program Manager) find your application icon first.

Allocating and deleting wxWidgets objects

In general, classes derived from wxWindow must dynamically allocated with *new* and deleted with *delete*. If you delete a window, all of its children and descendants will be automatically deleted, so you don't need to delete these descendants explicitly.

When deleting a frame or dialog, use **Destroy** rather than **delete** so that the wxWidgets delayed deletion can take effect. This waits until idle time (when all messages have been processed) to actually delete the window, to avoid problems associated with the GUI sending events to deleted windows.

Don't create a window on the stack, because this will interfere with delayed deletion.

If you decide to allocate a C++ array of objects (such as wxBitmap) that may be cleaned up by wxWidgets, make sure you delete the array explicitly before wxWidgets has a chance to do so on exit, since calling *delete* on array members will cause memory problems.

wxColour can be created statically: it is not automatically cleaned up and is unlikely to be shared between other objects; it is lightweight enough for copies to be made.

Beware of deleting objects such as a wxPen or wxBitmap if they are still in use. Windows is particularly sensitive to this: so make sure you make calls like wxDC::SetPen(wxNullPen) or wxDC::SelectObject(wxNullBitmap) before deleting a drawing object that may be in use. Code that doesn't do this will probably work fine on some platforms, and then fail under Windows.

Architecture dependency

A problem which sometimes arises from writing multi-platform programs is that the basic C types are not defined the same on all platforms. This holds true for both the length in bits of the standard types (such as int and long) as well as their byte order, which might be little endian (typically on Intel computers) or big endian (typically on some Unix workstations). wxWidgets defines types and macros that make it easy to write architecture independent code. The types are:

wxInt32, wxInt16, wxInt8, wxUint32, wxUint16 = wxWord, wxUint8 = wxByte

where wxInt32 stands for a 32-bit signed integer type etc. You can also check which architecture the program is compiled on using the wxBYTE_ORDER define which is either wxBIG_ENDIAN or wxLITTLE_ENDIAN (in the future maybe wxPDP_ENDIAN as well).

The macros handling bit-swapping with respect to the applications endianness are described in the *Byte order macros* (p. 1640) section.

Conditional compilation

One of the purposes of wxWidgets is to reduce the need for conditional compilation in source code, which can be messy and confusing to follow. However, sometimes it is necessary to incorporate platform-specific features (such as metafile use under MS Windows). The symbols listed in the file `symbols.txt` may be used for this purpose, along with any user-supplied ones.

C++ issues

The following documents some miscellaneous C++ issues.

Templates

wxWidgets does not use templates (except for some advanced features that are switched off by default) since it is a notoriously unportable feature.

RTTI

wxWidgets does not use C++ run-time type information since wxWidgets provides its own run-time type information system, implemented using macros.

Type of NULL

Some compilers (e.g. the native IRIX cc) define NULL to be 0L so that no conversion to pointers is allowed. Because of that, all these occurrences of NULL in the GTK+ port use an explicit conversion such as

```
wxWindow *my_window = (wxWindow*) NULL;
```

It is recommended to adhere to this in all code using wxWidgets as this make the code (a bit) more portable.

Precompiled headers

Some compilers, such as Borland C++ and Microsoft C++, support precompiled

headers. This can save a great deal of compiling time. The recommended approach is to precompile `"wx.h"`, using this precompiled header for compiling both wxWidgets itself and any wxWidgets applications. For Windows compilers, two dummy source files are provided (one for normal applications and one for creating DLLs) to allow initial creation of the precompiled header.

However, there are several downsides to using precompiled headers. One is that to take advantage of the facility, you often need to include more header files than would normally be the case. This means that changing a header file will cause more recompilations (in the case of wxWidgets, everything needs to be recompiled since everything includes `"wx.h"`!)

A related problem is that for compilers that don't have precompiled headers, including a lot of header files slows down compilation considerably. For this reason, you will find (in the common X and Windows parts of the library) conditional compilation that under Unix, includes a minimal set of headers; and when using Visual C++, includes `wx.h`. This should help provide the optimal compilation for each compiler, although it is biased towards the precompiled headers facility available in Microsoft C++.

File handling

When building an application which may be used under different environments, one difficulty is coping with documents which may be moved to different directories on other machines. Saving a file which has pointers to full pathnames is going to be inherently unportable. One approach is to store filenames on their own, with no directory information. The application searches through a number of locally defined directories to find the file. To support this, the class **wxPathList** makes adding directories and searching for files easy, and the global function **wxFileNameFromPath** allows the application to strip off the filename from the path if the filename must be stored. This has undesirable ramifications for people who have documents of the same name in different directories.

As regards the limitations of DOS 8+3 single-case filenames versus unrestricted Unix filenames, the best solution is to use DOS filenames for your application, and also for document filenames *if* the user is likely to be switching platforms regularly. Obviously this latter choice is up to the application user to decide. Some programs (such as YACC and LEX) generate filenames incompatible with DOS; the best solution here is to have your Unix makefile rename the generated files to something more compatible before transferring the source to DOS. Transferring DOS files to Unix is no problem, of course, apart from EOL conversion for which there should be a utility available (such as dos2unix).

See also the File Functions section of the reference manual for descriptions of miscellaneous file handling functions.

Chapter 4 Utilities and libraries supplied with wxWidgets

In addition to the core wxWidgets library, a number of further libraries and utilities are supplied with each distribution.

Some are under the 'contrib' hierarchy which mirrors the structure of the main wxWidgets hierarchy. See also the 'utils' hierarchy. The first place to look for documentation about these tools and libraries is under the wxWidgets 'docs' hierarchy, for example `docs/htmlhelp/fl.chm`.

For other user-contributed packages, please see the Contributions page on the wxWidgets Web site (`http://www.wxwidgets.org`).

Helpview Helpview is a program for displaying wxWidgets HTML Help files. In many cases, you may wish to use the wxWidgets HTML Help classes from within your application, but this provides a handy stand-alone viewer. See *wxHTML Notes* (p. 1845) for more details. You can find it in `samples/html/helpview`.

Tex2RTF Supplied with wxWidgets is a utility called Tex2RTF for converting LaTeX manuals HTML, MS HTML Help, wxHTML Help, RTF, and Windows Help RTF formats. Tex2RTF is used for the wxWidgets manuals and can be used independently by authors wishing to create on-line and printed manuals from the same LaTeX source. Please see the separate documentation for Tex2RTF. You can find it under `utils/tex2rtf`.

Helpgen Helpgen takes C++ header files and generates a Tex2RTF-compatible documentation file for each class it finds, using comments as appropriate. This is a good way to start a reference for a set of classes. Helpgen can be found in `utils/HelpGen`.

Emulator Xnest-based display emulator for X11-based PDA applications. On some systems, the Xnest window does not synchronise with the 'skin' window. This program can be found in `utils/emulator`.

Configuration Tool The wxWidgets Configuration Tool is a work in progress intended to make it easier to configure wxWidgets features in detail. It exports setup.h configurations and will eventually generate makefile config files. Invoking compilers is also on the cards. Since configurations are handled one at a time, the tool is of limited used until further development can be done. The program can be found in `utils/configtool`.

XRC resource system This is the sizer-aware resource system, and uses XML-based resource specifications that can be generated by tools such as wxDesigner (`http://www.roebling.de`) and XRC's own wxrcedit. You can find this in `src/xrc`, `include/wx/xrc`, `samples/xrc`, and `utils/wxrcedit`. For more information, see the *XML-based resource system*

overview (p. 1777).

Object Graphics Library OGL defines an API for applications that need to display objects connected by lines. The objects can be moved around and interacted with. You can find this in `contrib/src/ogl`, `contrib/include/wx/ogl`, and `contrib/samples/ogl`.

Frame Layout library FL provides sophisticated pane dragging and docking facilities. You can find this in `contrib/src/fl`, `contrib/include/wx/fl`, and `contrib/samples/fl`.

Gizmos library Gizmos is a collection of useful widgets and other classes. Classes include wxLEDNumberCtrl, wxEditableListBox, wxMultiCellCanvas. You can find this in `contrib/src/gizmos`, `contrib/include/wx/gizmos`, and `contrib/samples/gizmos`.

Net library Net is a collection of very simple mail and web related classes. Currently there is only wxEmail, which makes it easy to send email messages via MAPI on Windows or sendmail on Unix. You can find this in `contrib/src/net` and `contrib/include/wx/net`.

Animate library Animate allows you to load animated GIFs and play them on a window. The library can be extended to use other animation formats. You can find this in `contrib/src/animate`, `contrib/include/wx/animate`, and `contrib/samples/animate`.

MMedia library Mmedia supports a variety of multimedia functionality. The status of this library is currently unclear. You can find this in `contrib/src/mmedia`, `contrib/include/wx/mmedia`, and `contrib/samples/mmedia`.

Styled Text Control library STC is a wrapper around Scintilla, a syntax-highlighting text editor. You can find this in `contrib/src/stc`, `contrib/include/wx/stc`, and `contrib/samples/stc`.

Plot Plot is a simple curve plotting library. You can find this in `contrib/src/plot`, `contrib/include/wx/plot`, and `contrib/samples/plot`.

Chapter 5 Programming strategies

This chapter is intended to list strategies that may be useful when writing and debugging wxWidgets programs. If you have any good tips, please submit them for inclusion here.

Strategies for reducing programming errors

Use ASSERT

Although I haven't done this myself within wxWidgets, it is good practice to use ASSERT statements liberally, that check for conditions that should or should not hold, and print out appropriate error messages. These can be compiled out of a non-debugging version of wxWidgets and your application. Using ASSERT is an example of 'defensive programming': it can alert you to problems later on.

Use wxString in preference to character arrays

Using wxString can be much safer and more convenient than using char *. Again, I haven't practiced what I'm preaching, but I'm now trying to use wxString wherever possible. You can reduce the possibility of memory leaks substantially, and it is much more convenient to use the overloaded operators than functions such as strcmp. wxString won't add a significant overhead to your program; the overhead is compensated for by easier manipulation (which means less code).

The same goes for other data types: use classes wherever possible.

Strategies for portability

Use relative positioning or constraints

Don't use absolute panel item positioning if you can avoid it. Different GUIs have very differently sized panel items. Consider using the constraint system, although this can be complex to program.

Alternatively, you could use alternative .wrc (wxWidgets resource files) on different platforms, with slightly different dimensions in each. Or space your panel items out to avoid problems.

Use wxWidgets resource files

Use .xrc (wxWidgets resource files) where possible, because they can be easily changed independently of source code.

Strategies for debugging

Positive thinking

It is common to blow up the problem in one's imagination, so that it seems to threaten weeks, months or even years of work. The problem you face may seem insurmountable: but almost never is. Once you have been programming for some time, you will be able to remember similar incidents that threw you into the depths of despair. But remember, you always solved the problem, somehow!

Perseverance is often the key, even though a seemingly trivial problem can take an apparently inordinate amount of time to solve. In the end, you will probably wonder why you worried so much. That's not to say it isn't painful at the time. Try not to worry -- there are many more important things in life.

Simplify the problem

Reduce the code exhibiting the problem to the smallest program possible that exhibits the problem. If it is not possible to reduce a large and complex program to a very small program, then try to ensure your code doesn't hide the problem (you may have attempted to minimize the problem in some way: but now you want to expose it).

With luck, you can add a small amount of code that causes the program to go from functioning to non-functioning state. This should give a clue to the problem. In some cases though, such as memory leaks or wrong deallocation, this can still give totally spurious results!

Use a debugger

This sounds like facetious advice, but it is surprising how often people don't use a debugger. Often it is an overhead to install or learn how to use a debugger, but it really is essential for anything but the most trivial programs.

Use logging functions

There is a variety of logging functions that you can use in your program: see *Logging functions* (p. 1648).

Using tracing statements may be more convenient than using the debugger in some circumstances (such as when your debugger doesn't support a lot of debugging code, or you wish to print a bunch of variables).

Use the wxWidgets debugging facilities

You can use wxDebugContext to check for memory leaks and corrupt memory: in fact in debugging mode, wxWidgets will automatically check for memory leaks at the end of the program if wxWidgets is suitably configured. Depending on the operating system and compiler, more or less specific information about the problem will be logged.

You should also use *debug macros* (p. 1657) as part of a 'defensive programming' strategy, scattering wxASSERTs liberally to test for problems in your code as early as possible. Forward thinking will save a surprising amount of time in the long run.

See the *debugging overview* (p. 1746) for further information.

Chapter 6 Libraries list

Starting from version 2.5.0 wxWidgets can be built either as a single large library (this is called the *monolithic build*) or as several smaller libraries (*multilib build*). Multilib build is the default.

wxWidgets library is divided into libraries briefly described below. This diagram show dependencies between them:
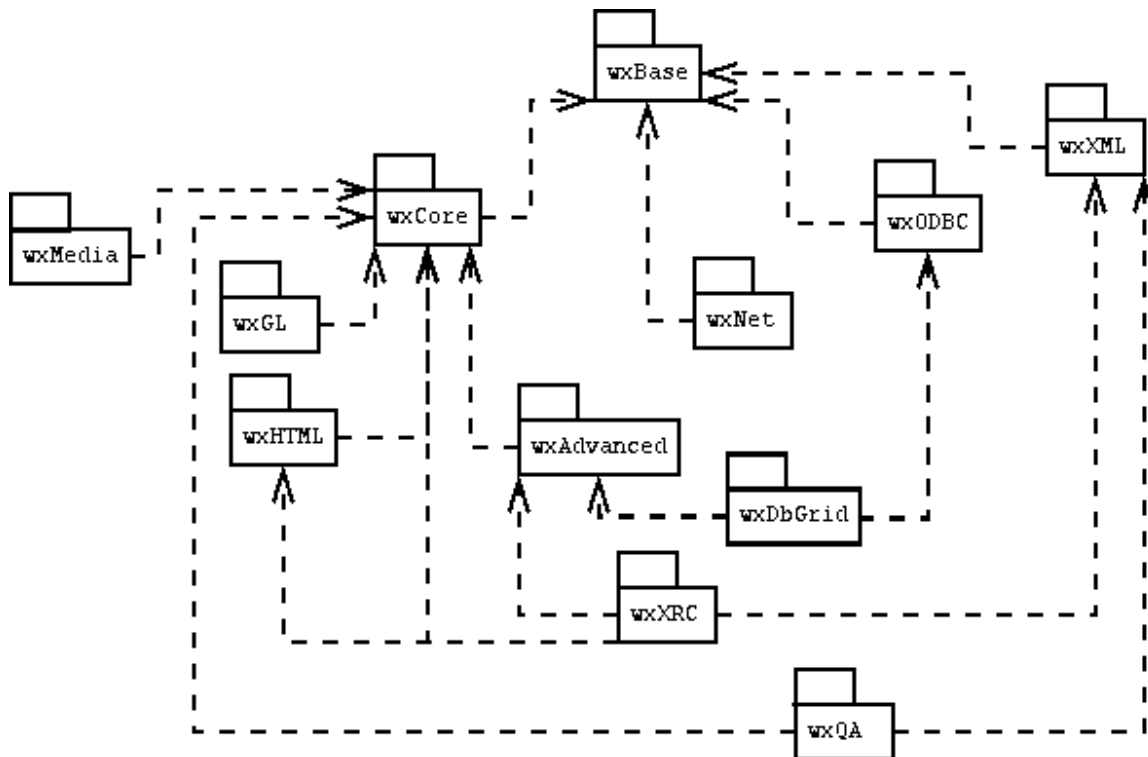
wxBase

Every wxWidgets application must link against this library. It contains mandatory classes that any wxWidgets code depends on (e.g. *wxString* (p. 1284)) and portability classes that abstract differences between platforms. wxBase can be used to develop console mode applications, it does not require any GUI libraries or running X Window System on Unix.

wxNet

Classes for network access:

- wxSocket classes (*wxSocketClient* (p. 1219), *wxSocketServer* (p. 1223) and related classes)

- *wxSocketOutputStream* (p. 1222) and *wxSocketInputStream* (p. 1222)

- sockets-based IPC classes (*wxTCPServer* (p. 391), *wxTCPClient* (p. 386) and *wxTCPConnection* (p. 387))

- *wxURL* (p. 1459)

- wxInternetFSHandler (a *wxFileSystem handler* (p. 1749)) Requires wxBase.

wxXML

This library contains simple classes for parsing XML documents. Note that their API *will* change in the future and backward compatibility will not be preserved. Use of this library in your applications is not recommended, it is only meant for use by XML resources system. Future versions of wxWidgets will contain new XML handling classes with DOM-like API. Requires wxBase.

wxCore

Basic GUI classes such as GDI classes or controls are in this library. All wxWidgets GUI applications must link against this library, only console mode applications don't.

wxAdvanced

Advanced or rarely used GUI classes:

- wxBufferedDC

- *wxCalendarCtrl* (p. 121)

- *wxDragImage* (p. 462)

- *wxGrid classes* (p. 1813)

- *wxJoystick* (p. 821)

- *wxLayoutAlgorithm* (p. 835)

- *wxSplashScreen* (p. 1235)

- *wxTaskBarIcon* (p. 1323)

- *wxSound* (p. 1225)

- *wxWizard* (p. 1549)

- *wxSashLayoutWindow* (p. 1129)

- *wxSashWindow* (p. 1132)

Requires wxCore and wxBase.

wxMedia

Advanced classes that are either low-level or deal with media.

- *wxMediaCtrl* (p. 927)

Requires wxCore and wxBase.

wxGL

This library contains *wxGLCanvas* (p. 614) class for integrating OpenGL library with wxWidgets. Unlike all others, this library is *not* part of the monolithic library, it is always built as separate library. Requires wxCore and wxBase.

wxHTML

Simple HTML renderer and other *HTML rendering classes* (p. 1845) are contained in this library, as well as *wxHtmlHelpController* (p. 722), *wxBestHelpController* (p. 696) and *wxHtmlListBox* (p. 734). Requires wxCore and wxBase.

wxODBC

Database classes (p. 1820). Requires wxBase.

wxQA

This is the library containing extra classes for quality assurance. Currently it only contains *wxDebugReport* (p. 398) and related classes, but more will be added to it in the future.

Requires wxCore, wxBase and wxXML.

wxDbGrid

wxDbGridTableBase (p. 363) class which combines *wxGrid* (p. 619) and *wxDbTable* (p. 324). Requires wxODBC and wxAdvanced.

wxXRC

This library contains *wxXmlResource* (p. 1558) class that provides access to XML resource files in XRC format. Requires wxXML, wxCore, wxAdvanced and wxHTML.

Chapter 7 Alphabetical class reference

wxAcceleratorEntry

An object used by an application wishing to create an *accelerator table* (p. 19).

Derived from

None

Include files

<wx/accel.h>

See also

wxAcceleratorTable (p. 19), *wxWindow::SetAcceleratorTable* (p. 1527)

wxAcceleratorEntry::wxAcceleratorEntry

wxAcceleratorEntry()

Default constructor.

wxAcceleratorEntry(**int** *flags*, **int** *keyCode*, **int** *cmd*)

Constructor.

Parameters

flags

> One of wxACCEL_ALT, wxACCEL_SHIFT, wxACCEL_CTRL and wxACCEL_NORMAL. Indicates which modifier key is held down.

keyCode

> The keycode to be detected. See *Keycodes* (p. 1669) for a full list of keycodes.

cmd

> The menu or control command identifier.

wxAcceleratorEntry::GetCommand

int GetCommand() **const**

Returns the command identifier for the accelerator table entry.

wxAcceleratorEntry::GetFlags

int GetFlags() **const**

Returns the flags for the accelerator table entry.

wxAcceleratorEntry::GetKeyCode

int GetKeyCode() **const**

Returns the keycode for the accelerator table entry.

wxAcceleratorEntry::Set

void Set(**int** *flags*, **int** *keyCode*, **int** *cmd*)

Sets the accelerator entry parameters.

Parameters

flags

> One of wxACCEL_ALT, wxACCEL_SHIFT, wxACCEL_CTRL and
> wxACCEL_NORMAL. Indicates which modifier key is held down.

keyCode

> The keycode to be detected. See *Keycodes* (p. 1669) for a full list of keycodes.

cmd

> The menu or control command identifier.

wxAcceleratorTable

An accelerator table allows the application to specify a table of keyboard shortcuts for menus or other commands. On Windows, menu or button commands are supported; on GTK, only menu commands are supported.

The object **wxNullAcceleratorTable** is defined to be a table with no data, and is the initial accelerator table for a window.

Derived from

wxObject (p. 1012)

Include files

<wx/accel.h>

Example

```
wxAcceleratorEntry entries[4];
entries[0].Set(wxACCEL_CTRL,  (int) 'N',      ID_NEW_WINDOW);
entries[1].Set(wxACCEL_CTRL,  (int) 'X',     wxID_EXIT);
entries[2].Set(wxACCEL_SHIFT, (int) 'A',      ID_ABOUT);
entries[3].Set(wxACCEL_NORMAL,  WXK_DELETE,    wxID_CUT);
wxAcceleratorTable accel(4, entries);
frame->SetAcceleratorTable(accel);
```

Remarks

An accelerator takes precedence over normal processing and can be a convenient way to program some event handling. For example, you can use an accelerator table to enable a dialog with a multi-line text control to accept CTRL-Enter as meaning 'OK' (but not in GTK+ at present).

See also

wxAcceleratorEntry (p. 18), *wxWindow::SetAcceleratorTable* (p. 1527)

wxAcceleratorTable::wxAcceleratorTable

wxAcceleratorTable()

Default constructor.

wxAcceleratorTable(**const wxAcceleratorTable&** *bitmap*)

Copy constructor.

wxAcceleratorTable(**int** *n*, **wxAcceleratorEntry** *entries[]*)

Creates from an array of *wxAcceleratorEntry* (p. 18) objects.

wxAcceleratorTable(**const wxString&** *resource*)

Loads the accelerator table from a Windows resource (Windows only).

Parameters

n

    Number of accelerator entries.

entries

    The array of entries.

resource

Name of a Windows accelerator.

wxPython note: The wxPython constructor accepts a list of wxAcceleratorEntry objects, or 3-tuples consisting of flags, keyCode, and cmd values like you would construct wxAcceleratorEntry objects with.

wxPerl note: The wxPerl constructor accepts a list of either Wx::AcceleratorEntry objects or references to 3-element arrays (flags, keyCode, cmd), like the parameters of Wx::AcceleratorEntry::new.

wxAcceleratorTable::~wxAcceleratorTable

~wxAcceleratorTable()

Destroys the wxAcceleratorTable object.

wxAcceleratorTable::Ok

bool Ok() **const**

Returns true if the accelerator table is valid.

wxAcceleratorTable::operator =

wxAcceleratorTable& operator =(**const wxAcceleratorTable&** *accel*)

Assignment operator. This operator does not copy any data, but instead passes a pointer to the data in *accel* and increments a reference counter. It is a fast operation.

Parameters

accel

Accelerator table to assign.

Return value

Returns reference to this object.

wxAcceleratorTable::operator ==

bool operator ==(**const wxAcceleratorTable&** *accel*)

Equality operator. This operator tests whether the internal data pointers are equal (a fast test).

Parameters

accel

>   Accelerator table to compare with

Return value

Returns true if the accelerator tables were effectively equal, false otherwise.

wxAcceleratorTable::operator !=

bool operator !=(const wxAcceleratorTable& *accel*)

Inequality operator. This operator tests whether the internal data pointers are unequal (a fast test).

Parameters

accel

>   Accelerator table to compare with

Return value

Returns true if the accelerator tables were unequal, false otherwise.

wxAccessible

The wxAccessible class allows wxWidgets applications, and wxWidgets itself, to return extended information about user interface elements to client applications such as screen readers. This is the main way in which wxWidgets implements accessibility features.

At present, only Microsoft Active Accessibility is supported by this class.

To use this class, derive from wxAccessible, implement appropriate functions, and associate an object of the class with a window using *wxWindow::SetAccessible* (p. 1527).

All functions return an indication of success, failure, or not implemented using values of the wxAccStatus enum type.

If you return wxACC_NOT_IMPLEMENTED from any function, the system will try to implement the appropriate functionality. However this will not work with all functions.

Most functions work with an *object id*, which can be zero to refer to 'this' UI element, or greater than zero to refer to the nth child element. This allows you to specify elements that don't have a corresponding wxWindow or wxAccessible; for example, the sash of a splitter window.

For details on the semantics of functions and types, please refer to the Microsoft Active Accessibility 1.2 documentation.

This class is compiled into wxWidgets only if the wxUSE_ACCESSIBILITY setup symbol

is set to 1.

Derived from

wxObject (p. 1012)

Include files

<wx/access.h>

Data structures

Functions return a wxAccStatus error code, which may be one of the following:

```
typedef enum
{
    wxACC_FAIL,             // The function failed
    wxACC_FALSE,            // The function returned false
    wxACC_OK,               // The function completed successfully
    wxACC_NOT_IMPLEMENTED,  // The function is not implemented
    wxACC_NOT_SUPPORTED     // The function is not supported
} wxAccStatus
```

Directions of navigation are represented by the following:

```
typedef enum
{
    wxNAVDIR_DOWN,
    wxNAVDIR_FIRSTCHILD,
    wxNAVDIR_LASTCHILD,
    wxNAVDIR_LEFT,
    wxNAVDIR_NEXT,
    wxNAVDIR_PREVIOUS,
    wxNAVDIR_RIGHT,
    wxNAVDIR_UP
} wxNavDir
```

The role of a user interface element is represented by the following type:

```
typedef enum {
    wxROLE_NONE,
    wxROLE_SYSTEM_ALERT,
    wxROLE_SYSTEM_ANIMATION,
    wxROLE_SYSTEM_APPLICATION,
    wxROLE_SYSTEM_BORDER,
    wxROLE_SYSTEM_BUTTONDROPDOWN,
    wxROLE_SYSTEM_BUTTONDROPDOWNGRID,
    wxROLE_SYSTEM_BUTTONMENU,
    wxROLE_SYSTEM_CARET,
    wxROLE_SYSTEM_CELL,
    wxROLE_SYSTEM_CHARACTER,
    wxROLE_SYSTEM_CHART,
    wxROLE_SYSTEM_CHECKBUTTON,
    wxROLE_SYSTEM_CLIENT,
    wxROLE_SYSTEM_CLOCK,
    wxROLE_SYSTEM_COLUMN,
```

```
        wxROLE_SYSTEM_COLUMNHEADER,
        wxROLE_SYSTEM_COMBOBOX,
        wxROLE_SYSTEM_CURSOR,
        wxROLE_SYSTEM_DIAGRAM,
        wxROLE_SYSTEM_DIAL,
        wxROLE_SYSTEM_DIALOG,
        wxROLE_SYSTEM_DOCUMENT,
        wxROLE_SYSTEM_DROPLIST,
        wxROLE_SYSTEM_EQUATION,
        wxROLE_SYSTEM_GRAPHIC,
        wxROLE_SYSTEM_GRIP,
        wxROLE_SYSTEM_GROUPING,
        wxROLE_SYSTEM_HELPBALLOON,
        wxROLE_SYSTEM_HOTKEYFIELD,
        wxROLE_SYSTEM_INDICATOR,
        wxROLE_SYSTEM_LINK,
        wxROLE_SYSTEM_LIST,
        wxROLE_SYSTEM_LISTITEM,
        wxROLE_SYSTEM_MENUBAR,
        wxROLE_SYSTEM_MENUITEM,
        wxROLE_SYSTEM_MENUPOPUP,
        wxROLE_SYSTEM_OUTLINE,
        wxROLE_SYSTEM_OUTLINEITEM,
        wxROLE_SYSTEM_PAGETAB,
        wxROLE_SYSTEM_PAGETABLIST,
        wxROLE_SYSTEM_PANE,
        wxROLE_SYSTEM_PROGRESSBAR,
        wxROLE_SYSTEM_PROPERTYPAGE,
        wxROLE_SYSTEM_PUSHBUTTON,
        wxROLE_SYSTEM_RADIOBUTTON,
        wxROLE_SYSTEM_ROW,
        wxROLE_SYSTEM_ROWHEADER,
        wxROLE_SYSTEM_SCROLLBAR,
        wxROLE_SYSTEM_SEPARATOR,
        wxROLE_SYSTEM_SLIDER,
        wxROLE_SYSTEM_SOUND,
        wxROLE_SYSTEM_SPINBUTTON,
        wxROLE_SYSTEM_STATICTEXT,
        wxROLE_SYSTEM_STATUSBAR,
        wxROLE_SYSTEM_TABLE,
        wxROLE_SYSTEM_TEXT,
        wxROLE_SYSTEM_TITLEBAR,
        wxROLE_SYSTEM_TOOLBAR,
        wxROLE_SYSTEM_TOOLTIP,
        wxROLE_SYSTEM_WHITESPACE,
        wxROLE_SYSTEM_WINDOW
    } wxAccRole
```

Objects are represented by the following type:

```
    typedef enum {
        wxOBJID_WINDOW =    0x00000000,
        wxOBJID_SYSMENU =   0xFFFFFFFF,
        wxOBJID_TITLEBAR =  0xFFFFFFFE,
        wxOBJID_MENU =      0xFFFFFFFD,
        wxOBJID_CLIENT =    0xFFFFFFFC,
        wxOBJID_VSCROLL =   0xFFFFFFFB,
        wxOBJID_HSCROLL =   0xFFFFFFFA,
        wxOBJID_SIZEGRIP =  0xFFFFFFF9,
        wxOBJID_CARET =     0xFFFFFFF8,
```

```
    wxOBJID_CURSOR =      0xFFFFFFF7,
    wxOBJID_ALERT =       0xFFFFFFF6,
    wxOBJID_SOUND =       0xFFFFFFF5
} wxAccObject
```

Selection actions are identified by this type:

```
typedef enum
{
    wxACC_SEL_NONE            = 0,
    wxACC_SEL_TAKEFOCUS       = 1,
    wxACC_SEL_TAKESELECTION   = 2,
    wxACC_SEL_EXTENDSELECTION = 4,
    wxACC_SEL_ADDSELECTION    = 8,
    wxACC_SEL_REMOVESELECTION = 16
} wxAccSelectionFlags
```

States are represented by the following:

```
#define wxACC_STATE_SYSTEM_ALERT_HIGH      0x00000001
#define wxACC_STATE_SYSTEM_ALERT_MEDIUM    0x00000002
#define wxACC_STATE_SYSTEM_ALERT_LOW       0x00000004
#define wxACC_STATE_SYSTEM_ANIMATED        0x00000008
#define wxACC_STATE_SYSTEM_BUSY            0x00000010
#define wxACC_STATE_SYSTEM_CHECKED         0x00000020
#define wxACC_STATE_SYSTEM_COLLAPSED       0x00000040
#define wxACC_STATE_SYSTEM_DEFAULT         0x00000080
#define wxACC_STATE_SYSTEM_EXPANDED        0x00000100
#define wxACC_STATE_SYSTEM_EXTSELECTABLE   0x00000200
#define wxACC_STATE_SYSTEM_FLOATING        0x00000400
#define wxACC_STATE_SYSTEM_FOCUSABLE       0x00000800
#define wxACC_STATE_SYSTEM_FOCUSED         0x00001000
#define wxACC_STATE_SYSTEM_HOTTRACKED      0x00002000
#define wxACC_STATE_SYSTEM_INVISIBLE       0x00004000
#define wxACC_STATE_SYSTEM_MARQUEED        0x00008000
#define wxACC_STATE_SYSTEM_MIXED           0x00010000
#define wxACC_STATE_SYSTEM_MULTISELECTABLE 0x00020000
#define wxACC_STATE_SYSTEM_OFFSCREEN       0x00040000
#define wxACC_STATE_SYSTEM_PRESSED         0x00080000
#define wxACC_STATE_SYSTEM_PROTECTED       0x00100000
#define wxACC_STATE_SYSTEM_READONLY        0x00200000
#define wxACC_STATE_SYSTEM_SELECTABLE      0x00400000
#define wxACC_STATE_SYSTEM_SELECTED        0x00800000
#define wxACC_STATE_SYSTEM_SELFVOICING     0x01000000
#define wxACC_STATE_SYSTEM_UNAVAILABLE     0x02000000
```

Event identifiers that can be sent via *wxAccessible::NotifyEvent* (p. 29) are as follows:

```
#define wxACC_EVENT_SYSTEM_SOUND              0x0001
#define wxACC_EVENT_SYSTEM_ALERT              0x0002
#define wxACC_EVENT_SYSTEM_FOREGROUND         0x0003
#define wxACC_EVENT_SYSTEM_MENUSTART          0x0004
#define wxACC_EVENT_SYSTEM_MENUEND            0x0005
#define wxACC_EVENT_SYSTEM_MENUPOPUPSTART     0x0006
#define wxACC_EVENT_SYSTEM_MENUPOPUPEND       0x0007
```

```
#define wxACC_EVENT_SYSTEM_CAPTURESTART        0x0008
#define wxACC_EVENT_SYSTEM_CAPTUREEND          0x0009
#define wxACC_EVENT_SYSTEM_MOVESIZESTART       0x000A
#define wxACC_EVENT_SYSTEM_MOVESIZEEND         0x000B
#define wxACC_EVENT_SYSTEM_CONTEXTHELPSTART    0x000C
#define wxACC_EVENT_SYSTEM_CONTEXTHELPEND      0x000D
#define wxACC_EVENT_SYSTEM_DRAGDROPSTART       0x000E
#define wxACC_EVENT_SYSTEM_DRAGDROPEND         0x000F
#define wxACC_EVENT_SYSTEM_DIALOGSTART         0x0010
#define wxACC_EVENT_SYSTEM_DIALOGEND           0x0011
#define wxACC_EVENT_SYSTEM_SCROLLINGSTART      0x0012
#define wxACC_EVENT_SYSTEM_SCROLLINGEND        0x0013
#define wxACC_EVENT_SYSTEM_SWITCHSTART         0x0014
#define wxACC_EVENT_SYSTEM_SWITCHEND           0x0015
#define wxACC_EVENT_SYSTEM_MINIMIZESTART       0x0016
#define wxACC_EVENT_SYSTEM_MINIMIZEEND         0x0017
#define wxACC_EVENT_OBJECT_CREATE                0x8000
#define wxACC_EVENT_OBJECT_DESTROY               0x8001
#define wxACC_EVENT_OBJECT_SHOW                  0x8002
#define wxACC_EVENT_OBJECT_HIDE                  0x8003
#define wxACC_EVENT_OBJECT_REORDER               0x8004
#define wxACC_EVENT_OBJECT_FOCUS                 0x8005
#define wxACC_EVENT_OBJECT_SELECTION             0x8006
#define wxACC_EVENT_OBJECT_SELECTIONADD          0x8007
#define wxACC_EVENT_OBJECT_SELECTIONREMOVE       0x8008
#define wxACC_EVENT_OBJECT_SELECTIONWITHIN       0x8009
#define wxACC_EVENT_OBJECT_STATECHANGE           0x800A
#define wxACC_EVENT_OBJECT_LOCATIONCHANGE        0x800B
#define wxACC_EVENT_OBJECT_NAMECHANGE            0x800C
#define wxACC_EVENT_OBJECT_DESCRIPTIONCHANGE     0x800D
#define wxACC_EVENT_OBJECT_VALUECHANGE           0x800E
#define wxACC_EVENT_OBJECT_PARENTCHANGE          0x800F
#define wxACC_EVENT_OBJECT_HELPCHANGE            0x8010
#define wxACC_EVENT_OBJECT_DEFACTIONCHANGE       0x8011
#define wxACC_EVENT_OBJECT_ACCELERATORCHANGE     0x8012
```

wxAccessible::wxAccessible

wxAccessible(**wxWindow*** *win = NULL*)

Constructor, taking an optional window. The object can be associated with a window later.

wxAccessible::~wxAccessible

~wxAccessible()

Destructor.

wxAccessible::DoDefaultAction

virtual wxAccStatus DoDefaultAction(**int** *childId*)

Performs the default action for the object. *childId* is 0 (the action for this object) or greater than 0 (the action for a child). Return wxACC_NOT_SUPPORTED if there is no default action for this window (e.g. an edit control).

wxAccessible::GetChild

virtual wxAccStatus GetChild(**int** *childId*, **wxAccessible**** *child*)

Gets the specified child (starting from 1). If *child* is NULL and the return value is wxACC_OK, this means that the child is a simple element and not an accessible object.

wxAccessible::GetChildCount

virtual wxAccStatus GetChildCount(**int*** *childCount*)

Returns the number of children in *childCount*.

wxAccessible::GetDefaultAction

virtual wxAccStatus GetDefaultAction(**int** *childId*, **wxString*** *actionName*)

Gets the default action for this object (0) or a child (greater than 0). Return wxACC_OK even if there is no action. *actionName* is the action, or the empty string if there is no action. The retrieved string describes the action that is performed on an object, not what the object does as a result. For example, a toolbar button that prints a document has a default action of "Press" rather than "Prints the current document."

wxAccessible::GetDescription

virtual wxAccStatus GetDescription(**int** *childId*, **wxString*** *description*)

Returns the description for this object or a child.

wxAccessible::GetFocus

virtual wxAccStatus GetFocus(**int*** *childId*, **wxAccessible**** *child*)

Gets the window with the keyboard focus. If childId is 0 and child is NULL, no object in this subhierarchy has the focus. If this object has the focus, child should be 'this'.

wxAccessible::GetHelpText

virtual wxAccStatus GetHelpText(**int** *childId*, **wxString*** *helpText*)

Returns help text for this object or a child, similar to tooltip text.

wxAccessible::GetKeyboardShortcut

virtual wxAccStatus GetKeyboardShortcut(**int** *childId*, **wxString*** *shortcut*)

Returns the keyboard shortcut for this object or child. Return e.g. ALT+K.

wxAccessible::GetLocation

virtual wxAccStatus GetLocation(**wxRect&** *rect*, **int** *elementId*)

Returns the rectangle for this object (id is 0) or a child element (id is greater than 0).*rect* is in screen coordinates.

wxAccessible::GetName

virtual wxAccStatus GetName(**int** *childId*, **wxString*** *name*)

Gets the name of the specified object.

wxAccessible::GetParent

virtual wxAccStatus GetParent(**wxAccessible**** *parent*)

Returns the parent of this object, or NULL.

wxAccessible::GetRole

virtual wxAccStatus GetRole(**int** *childId*, **wxAccRole*** *role*)

Returns a role constant describing this object. See *wxAccessible* (p. 22) for a list of these roles.

wxAccessible::GetSelections

virtual wxAccStatus GetSelections(**wxVariant*** *selections*)

Gets a variant representing the selected children of this object.

Acceptable values are:

- a null variant (IsNull() returns TRUE)

- a list variant (GetType() == wxT("list"))

- an integer representing the selected child element, or 0 if this object is selected (GetType() == wxT("long"))

- a "void*" pointer to a wxAccessible child object

wxAccessible::GetState

virtual wxAccStatus GetState(**int** *childId*, **long*** *state*)

Returns a state constant. See *wxAccessible* (p. 22) for a list of these states.

wxAccessible::GetValue

virtual wxAccStatus GetValue(**int** *childId*, **wxString*** *strValue*)

Returns a localized string representing the value for the object or child.

wxAccessible::GetWindow

wxWindow* GetWindow()

Returns the window associated with this object.

wxAccessible::HitTest

virtual wxAccStatus HitTest(**const wxPoint&** *pt*, **int*** *childId*, **wxAccessible**** *childObject*)

Returns a status value and object id to indicate whether the given point was on this or a child object. Can return either a child object, or an integer representing the child element, starting from 1.

pt is in screen coordinates.

wxAccessible::Navigate

virtual wxAccStatus Navigate(**wxNavDir** *navDir*, **int** *fromId*, **int*** *toId*, **wxAccessible**** *toObject*)

Navigates from *fromId* to *toId*/*toObject*.

wxAccessible::NotifyEvent

virtual static void NotifyEvent(**int** *eventType*, **wxWindow*** *window*, **wxAccObject** *objectType*, **int** *objectType*)

Allows the application to send an event when something changes in an accessible object.

wxAccessible::Select

virtual wxAccStatus Select(**int** *childId*, **wxAccSelectionFlags** *selectFlags*)

Selects the object or child. See *wxAccessible* (p. 22) for a list of the selection actions.

wxAccessible::SetWindow

void SetWindow(**wxWindow*** *window*)

Sets the window associated with this object.

wxActivateEvent

An activate event is sent when a window or application is being activated or deactivated.

Derived from

wxEvent (p. 483)
wxObject (p. 1012)

Include files

<wx/event.h>

Event table macros

To process an activate event, use these event handler macros to direct input to a member function that takes a wxActivateEvent argument.

EVT_ACTIVATE(func)	Process a wxEVT_ACTIVATE event.
EVT_ACTIVATE_APP(func)	Process a wxEVT_ACTIVATE_APP event.
EVT_HIBERNATE(func)	Process a hibernate event, supplying the member function. This event applies to wxApp only, and only on Windows SmartPhone and PocketPC. It is generated when the system is low on memory; the application should free up as much memory as possible, and restore full working when it receives a wxEVT_ACTIVATE or wxEVT_ACTIVATE_APP event.

Remarks
A top-level window (a dialog or frame) receives an activate event when is being activated or deactivated. This is indicated visually by the title bar changing colour, and a subwindow gaining the keyboard focus.

An application is activated or deactivated when one of its frames becomes activated, or a frame becomes inactivate resulting in all application frames being inactive. (Windows only)

Please note that usually you should call *event.Skip()* (p. 486) in your handlers for these events as not doing so can result in strange effects.

See also

Event handling overview (p. 1751), *wxApp::IsActive* (p. 35)

wxActivateEvent::wxActivateEvent

 wxActivateEvent(**WXTYPE** *eventType = 0*, **bool** *active = true*, **int** *id = 0*)

Constructor.

wxActivateEvent::GetActive

bool GetActive() **const**

Returns true if the application or window is being activated, false otherwise.

wxApp

The **wxApp** class represents the application itself. It is used to:

- set and get application-wide properties;

- implement the windowing system message or event loop;

- initiate application processing via *wxApp::OnInit* (p. 38);

- allow default processing of events not handled by other objects in the application.

You should use the macro IMPLEMENT_APP(appClass) in your application implementation file to tell wxWidgets how to create an instance of your application class.

Use DECLARE_APP(appClass) in a header file if you want the wxGetApp function (which returns a reference to your application object) to be visible to other files.

Derived from

wxEvtHandler (p. 486)
wxObject (p. 1012)

Include files

<wx/app.h>

See also

wxApp overview (p. 1717)

wxApp::wxApp

wxApp()

Constructor. Called implicitly with a definition of a wxApp object.

wxApp::~wxApp

~wxApp()

Destructor. Will be called implicitly on program exit if the wxApp object is created on the stack.

wxApp::argc

int argc

Number of command line arguments (after environment-specific processing).

wxApp::argv

wxChar ** argv

Command line arguments (after environment-specific processing).

wxApp::CreateLogTarget

virtual wxLog* CreateLogTarget()

Creates a wxLog class for the application to use for logging errors. The default implementation returns a new wxLogGui class.

See also

wxLog (p. 889)

wxApp::Dispatch

virtual void Dispatch()

Dispatches the next event in the windowing system event queue.

This can be used for programming event loops, e.g.

```
while (app.Pending())
  Dispatch();
```

See also

wxApp::Pending (p. 39)

wxApp::ExitMainLoop

virtual void ExitMainLoop()

Call this to explicitly exit the main message (event) loop. You should normally exit the main loop (and the application) by deleting the top window.

wxApp::FilterEvent

int FilterEvent(**wxEvent&** *event*)

This function is called before processing any event and allows the application to preempt the processing of some events. If this method returns -1 the event is processed normally, otherwise either `true` or `false` should be returned and the event processing stops immediately considering that the event had been already processed (for the former return value) or that it is not going to be processed at all (for the latter one).

wxApp::GetAppName

wxString GetAppName() **const**

Returns the application name.

Remarks

wxWidgets sets this to a reasonable default before calling *wxApp::OnInit* (p. 38), but the application can reset it at will.

wxApp::GetClassName

wxString GetClassName() **const**

Gets the class name of the application. The class name may be used in a platform specific manner to refer to the application.

See also

wxApp::SetClassName (p. 40)

wxApp::GetExitOnFrameDelete

bool GetExitOnFrameDelete() **const**

Returns true if the application will exit when the top-level window is deleted, false otherwise.

See also

wxApp::SetExitOnFrameDelete (p. 40),
wxApp shutdown overview (p. 1718)

wxApp::GetInstance

static wxAppConsole * GetInstance()

Returns the one and only global application object. Usually `wxTheApp` is usead instead.

See also

wxApp::SetInstance (p. 40)

wxApp::GetTopWindow

virtual wxWindow * GetTopWindow() **const**

Returns a pointer to the top window.

Remarks

If the top window hasn't been set using *wxApp::SetTopWindow* (p. 41), this function will find the first top-level window (frame or dialog) and return that.

See also

SetTopWindow (p. 41)

wxApp::GetUseBestVisual

bool GetUseBestVisual() **const**

Returns true if the application will use the best visual on systems that support different visuals, false otherwise.

See also

SetUseBestVisual (p. 41)

wxApp::GetVendorName

wxString GetVendorName() **const**

Returns the application's vendor name.

wxApp::IsActive

bool IsActive() **const**

Returns `true` if the application is active, i.e. if one of its windows is currently in the foreground. If this function returns `false` and you need to attract users attention to the application, you may use *wxTopLevelWindow::RequestUserAttention* (p. 1421) to do it.

wxApp::IsMainLoopRunning

static bool IsMainLoopRunning()

Returns `true` if the main event loop is currently running, i.e. if the application is inside *OnRun* (p. 38).

This can be useful to test whether the events can be dispatched. For example, if this function returns `false`, non-blocking sockets cannot be used because the events from them would never be processed.

wxApp::MainLoop

virtual int MainLoop()

Called by wxWidgets on creation of the application. Override this if you wish to provide your own (environment-dependent) main loop.

Return value

Returns 0 under X, and the wParam of the WM_QUIT message under Windows.

wxApp::OnAssert

void OnAssert(**const wxChar** **file*, **int** *line*, **const wxChar** **cond*, **const wxChar** **msg*)

This function is called when an assert failure occurs, i.e. the condition specified in *wxASSERT* (p. 1657) macro evaluated to `false`. It is only called in debug mode (when `__WXDEBUG__` is defined) as asserts are not left in the release code at all.

The base class version show the default assert failure dialog box proposing to the user to stop the program, continue or ignore all subsequent asserts.

Parameters

file

> the name of the source file where the assert occurred

line

> the line number in this file where the assert occurred

cond

> the condition of the failed assert in string form

msg

> the message specified as argument to *wxASSERT_MSG* (p. 1658) or *wxFAIL_MSG* (p. 1659), will be `NULL` if just *wxASSERT* (p. 1657) or *wxFAIL* (p. 1659) was used

wxApp::OnCmdLineError

bool OnCmdLineError(**wxCmdLineParser&** *parser*)

Called when command line parsing fails (i.e. an incorrect command line option was specified by the user). The default behaviour is to show the program usage text and abort the program.

Return `true` to continue normal execution or `false` to return `false` from *OnInit* (p. 38) thus terminating the program.

See also

OnInitCmdLine (p. 38)

wxApp::OnCmdLineHelp

bool OnCmdLineHelp(**wxCmdLineParser&** *parser*)

Called when the help option (`--help`) was specified on the command line. The default behaviour is to show the program usage text and abort the program.

Return `true` to continue normal execution or `false` to return `false` from *OnInit* (p. 38) thus terminating the program.

See also

OnInitCmdLine (p. 38)

wxApp::OnCmdLineParsed

bool OnCmdLineParsed(**wxCmdLineParser&** *parser*)

Called after the command line had been successfully parsed. You may override this method to test for the values of the various parameters which could be set from the command line.

Don't forget to call the base class version unless you want to suppress processing of the standard command line options.

Return `true` to continue normal execution or `false` to return `false` from *OnInit* (p. 38) thus terminating the program.

See also

OnInitCmdLine (p. 38)

wxApp::OnExceptionInMainLoop

virtual bool OnExceptionInMainLoop()

This function is called if an unhandled exception occurs inside the main application event loop. It can return `true` to ignore the exception and to continue running the loop or `false` to exit the loop and terminate the program. In the latter case it can also use C++ `throw` keyword to rethrow the current exception.

The default behaviour of this function is the latter in all ports except under Windows where a dialog is shown to the user which allows him to choose between the different options. You may override this function in your class to do something more appropriate.

Finally note that if the exception is rethrown from here, it can be caught in *OnUnhandledException* (p. 38).

wxApp::OnExit

virtual int OnExit()

Override this member function for any processing which needs to be done as the application is about to exit. OnExit is called after destroying all application windows and controls, but before wxWidgets cleanup. Note that it is not called at all if *OnInit* (p. 38) failed.

The return value of this function is currently ignored, return the same value as returned by the base class method if you override it.

wxApp::OnFatalException

void OnFatalException()

This function may be called if something fatal happens: an unhandled exception under Win32 or a a fatal signal under Unix, for example. However, this will not happen by default: you have to explicitly call *wxHandleFatalExceptions* (p. 1590) to enable this.

Generally speaking, this function should only show a message to the user and return. You may attempt to save unsaved data but this is not guaranteed to work and, in fact, probably won't.

See also

wxHandleFatalExceptions (p. 1590)

wxApp::OnInit

bool OnInit()

This must be provided by the application, and will usually create the application's main window, optionally calling *wxApp::SetTopWindow* (p. 41). You may use *OnExit* (p. 37) to clean up anything initialized here, provided that the function returns `true`.

Notice that if you want to to use the command line processing provided by wxWidgets you have to call the base class version in the derived class OnInit().

Return `true` to continue processing, `false` to exit the application immediately.

wxApp::OnInitCmdLine

void OnInitCmdLine(**wxCmdLineParser&** *parser*)

Called from *OnInit* (p. 38) and may be used to initialize the parser with the command line options for this application. The base class versions adds support for a few standard options only.

wxApp::OnRun

virtual int OnRun()

This virtual function is where the execution of a program written in wxWidgets starts. The default implementation just enters the main loop and starts handling the events until it terminates, either because *ExitMainLoop* (p. 33) has been explicitly called or because the last frame has been deleted and *GetExitOnFrameDelete* (p. 34) flag is `true` (this is the default).

The return value of this function becomes the exit code of the program, so it should return 0 in case of successful termination.

wxApp::OnUnhandledException

virtual void OnUnhandledException()

This function is called when an unhandled C++ exception occurs inside *OnRun()* (p. 38) (the exceptions which occur during the program startup and shutdown might not be

caught at all). Note that the exception type is lost by now, so if you want to really handle the exception you should override *OnRun()* (p. 38) and put a try/catch clause around the call to the base class version there.

wxApp::ProcessMessage

bool ProcessMessage(**WXMSG ****msg*)

Windows-only function for processing a message. This function is called from the main message loop, checking for windows that may wish to process it. The function returns true if the message was processed, false otherwise. If you use wxWidgets with another class library with its own message loop, you should make sure that this function is called to allow wxWidgets to receive messages. For example, to allow co-existence with the Microsoft Foundation Classes, override the PreTranslateMessage function:

```
// Provide wxWidgets message loop compatibility
BOOL CTheApp::PreTranslateMessage(MSG *msg)
{
  if (wxTheApp && wxTheApp->ProcessMessage((WXMSW *)msg))
    return true;
  else
    return CWinApp::PreTranslateMessage(msg);
}
```

wxApp::Pending

virtual bool Pending()

Returns true if unprocessed events are in the window system event queue.

See also

wxApp::Dispatch (p. 32)

wxApp::SendIdleEvents

bool SendIdleEvents(**wxWindow*** *win*, **wxIdleEvent&** *event*)

Sends idle events to a window and its children.

Please note that this function is internal to wxWidgets and shouldn't be used by user code.

Remarks

These functions poll the top-level windows, and their children, for idle event processing. If true is returned, more OnIdle processing is requested by one or more window.

See also

wxIdleEvent (p. 776)

wxApp::SetAppName

void SetAppName(**const wxString&** *name*)

Sets the name of the application. The name may be used in dialogs (for example by the document/view framework). A default name is set by wxWidgets.

See also

wxApp::GetAppName (p. 33)

wxApp::SetClassName

void SetClassName(**const wxString&** *name*)

Sets the class name of the application. This may be used in a platform specific manner to refer to the application.

See also

wxApp::GetClassName (p. 33)

wxApp::SetExitOnFrameDelete

void SetExitOnFrameDelete(**bool** *flag*)

Allows the programmer to specify whether the application will exit when the top-level frame is deleted.

Parameters

flag

> If true (the default), the application will exit when the top-level frame is deleted. If false, the application will continue to run.

See also

wxApp::GetExitOnFrameDelete (p. 34),
wxApp shutdown overview (p. 1718)

wxApp::SetInstance

static void SetInstance(**wxAppConsole*** *app*)

Allows external code to modify global `wxTheApp`, but you should really know what you're doing if you call it.

Parameters

app

Replacement for the global application object.

See also

wxApp::GetInstance (p. 34)

wxApp::SetTopWindow

void SetTopWindow(wxWindow* *window*)

Sets the 'top' window. You can call this from within *wxApp::OnInit* (p. 38) to let wxWidgets know which is the main window. You don't have to set the top window; it is only a convenience so that (for example) certain dialogs without parents can use a specific window as the top window. If no top window is specified by the application, wxWidgets just uses the first frame or dialog in its top-level window list, when it needs to use the top window.

Parameters

window

The new top window.

See also

wxApp::GetTopWindow (p. 34), *wxApp::OnInit* (p. 38)

wxApp::SetVendorName

void SetVendorName(const wxString& *name*)

Sets the name of application's vendor. The name will be used in registry access. A default name is set by wxWidgets.

See also

wxApp::GetVendorName (p. 35)

wxApp::SetUseBestVisual

void SetUseBestVisual(bool *flag*)

Allows the programmer to specify whether the application will use the best visual on systems that support several visual on the same display. This is typically the case under Solaris and IRIX, where the default visual is only 8-bit whereas certain applications are supposed to run in TrueColour mode.

Note that this function has to be called in the constructor of the `wxApp` instance and won't have any effect when called later on.

This function currently only has effect under GTK.

Parameters

flag

> If true, the app will use the best visual.

wxApp::HandleEvent

virtual void HandleEvent(**wxEvtHandler** **handler*, **wxEventFunction** *func*, **wxEvent&** *event*) **const**

This function simply invokes the given method *func* of the specified event handler *handler* with the *event* as parameter. It exists solely to allow to catch the C++ exceptions which could be thrown by all event handlers in the application in one place: if you want to do this, override this function in your wxApp-derived class and add try/catch clause(s) to it.

wxApp::Yield

bool Yield(**bool** *onlyIfNeeded = false*)

Yields control to pending messages in the windowing system. This can be useful, for example, when a time-consuming process writes to a text window. Without an occasional yield, the text window will not be updated properly, and on systems with cooperative multitasking, such as Windows 3.1 other processes will not respond.

Caution should be exercised, however, since yielding may allow the user to perform actions which are not compatible with the current task. Disabling menu items or whole menus during processing can avoid unwanted reentrance of code: see *::wxSafeYield* (p. 1591) for a better function.

Note that Yield() will not flush the message logs. This is intentional as calling Yield() is usually done to quickly update the screen and popping up a message box dialog may be undesirable. If you do wish to flush the log messages immediately (otherwise it will be done during the next idle loop iteration), call *wxLog::FlushActive* (p. 894).

Calling Yield() recursively is normally an error and an assert failure is raised in debug build if such situation is detected. However if the *onlyIfNeeded* parameter is `true`, the method will just silently return `false` instead.

wxArchiveClassFactory

An abstract base class which serves as a common interface to archive class factories such as *wxZipClassFactory* (p. 1567).

For each supported archive type (such as zip) there is a class factory derived from wxArchiveClassFactory, which allows archive objects to be created in a generic way,

without knowing the particular type of archive being used.

Derived from

wxObject (p. 1012)

Include files

<wx/archive.h>

See also

Archive formats such as zip (p. 1879)
Generic archive programming (p. 1883)
wxArchiveEntry (p. 44)
wxArchiveInputStream (p. 47)
wxArchiveOutputStream (p. 51)

wxArchiveClassFactory::Get/SetConv

wxMBConv& GetConv() **const**

void SetConv(**wxMBConv&** *conv*)

The *wxMBConv* (p. 909) object that the created streams will use when translating meta-data. The initial default, set by the constructor, is wxConvLocal.

wxArchiveClassFactory::GetInternalName

wxString GetInternalName(**const wxString&** *name*, **wxPathFormat** *format* = *wxPATH_NATIVE*) **const**

Calls the static GetInternalName() function for the archive entry type, for example *wxZipEntry::GetInternalName()* (p. 1572).

wxArchiveClassFactory::NewEntry

wxArchiveEntry* NewEntry() **const**

Create a new *wxArchiveEntry* (p. 44) object of the appropriate type.

wxArchiveClassFactory::NewStream

wxArchiveInputStream* NewStream(**wxInputStream&** *stream*) **const**

wxArchiveOutputStream* NewStream(**wxOutputStream&** *stream*) **const**

Create a new *wxArchiveInputStream* (p. 47)or *wxArchiveOutputStream* (p. 51) of the appropriate type.

wxArchiveEntry

An abstract base class which serves as a common interface to archive entry classes such as *wxZipEntry* (p. 1568). These hold the meta-data (filename, timestamp, etc.), for entries in archive files such as zips and tars.

Derived from

wxObject (p. 1012)

Include files

<wx/archive.h>

See also

Archive formats such as zip (p. 1879)
Generic archive programming (p. 1883)
wxArchiveInputStream (p. 47)
wxArchiveOutputStream (p. 51)
wxArchiveNotifier (p. 50)

Non-seekable streams

This information applies only when reading archives from non-seekable streams. When the stream is seekable *GetNextEntry()* (p. 47)returns a fully populated *wxArchiveEntry* (p. 44). See '*Archives on non-seekable streams* (p. 1884)' for more information.

For generic programming, when the worst case must be assumed, you can rely on all the fields of wxArchiveEntry being fully populated when GetNextEntry() returns, with the the following exceptions:

GetSize() (p. 46) Guaranteed to be available after the entry has been read to *Eof()* (p. 815), or *CloseEntry()* (p. 47) has been called

IsReadOnly() (p. 46)   Guaranteed to be available after the end of the archive has been reached, i.e. after GetNextEntry() returns NULL and Eof() is true

wxArchiveEntry::Clone

wxArchiveEntry* Clone() **const**

Returns a copy of this entry object.

wxArchiveEntry::Get/SetDateTime

wxDateTime GetDateTime() **const**

void SetDateTime(**const wxDateTime&** *dt*)

The entry's timestamp.

wxArchiveEntry::GetInternalFormat

wxPathFormat GetInternalFormat() **const**

Returns the path format used internally within the archive to store filenames.

wxArchiveEntry::GetInternalName

wxString GetInternalName() **const**

Returns the entry's filename in the internal format used within the archive. The name can include directory components, i.e. it can be a full path.

The names of directory entries are returned without any trailing path separator. This gives a canonical name that can be used in comparisons.

See also

Looking up an archive entry by name (p. 1881)

wxArchiveEntry::Get/SetName

wxString GetName(**wxPathFormat** *format* = *wxPATH_NATIVE*) **const**

void SetName(**const wxString&** *name*, **wxPathFormat** *format* = *wxPATH_NATIVE*)

The entry's name, by default in the native format. The name can include directory components, i.e. it can be a full path.

If this is a directory entry, (i.e. if *IsDir()* (p. 46)is true) then GetName() returns the name with a trailing path separator.

Similarly, setting a name with a trailing path separator sets IsDir().

wxArchiveEntry::GetOffset

off_t GetOffset() **const**

Returns a numeric value unique to the entry within the archive.

wxArchiveEntry::Get/SetSize

off_t GetSize() **const**

void SetSize(**off_t** *size*)

The size of the entry's data in bytes.

wxArchiveEntry::IsDir/SetIsDir

bool IsDir() **const**

void SetIsDir(**bool** *isDir = true*)

True if this is a directory entry.

Directory entries are entries with no data, which are used to store the meta-data of directories. They also make it possible for completely empty directories to be stored.

The names of entries within an archive can be complete paths, and unarchivers typically create whatever directories are necessary as they restore files, even if the archive contains no explicit directory entries.

wxArchiveEntry::IsReadOnly/SetIsReadOnly

bool IsReadOnly() **const**

void SetIsReadOnly(**bool** *isReadOnly = true*)

True if the entry is a read-only file.

wxArchiveEntry::Set/UnsetNotifier

void SetNotifier(**wxArchiveNotifier&** *notifier*)

void UnsetNotifier()

Sets the *notifier* (p. 50) for this entry. Whenever the *wxArchiveInputStream* (p. 47) updates this entry, it will then invoke the associated notifier's *OnEntryUpdated* (p. 51)method.

Setting a notifier is not usually necessary. It is used to handle certain cases when modifying an archive in a pipeline (i.e. between non-seekable streams).

See also

Archives on non-seekable streams (p. 1884)
wxArchiveNotifier (p. 50)

wxArchiveInputStream

An abstract base class which serves as a common interface to archive input streams such as *wxZipInputStream* (p. 1574).

GetNextEntry() (p. 47) returns an *wxArchiveEntry* (p. 44) object containing the meta-data for the next entry in the archive (and gives away ownership). Reading from the wxArchiveInputStream then returns the entry's data. Eof() becomes true after an attempt has been made to read past the end of the entry's data. When there are no more entries, GetNextEntry() returns NULL and sets Eof().

Derived from

wxFilterInputStream (p. 549)

Include files

<wx/archive.h>

Data structurestypedef wxArchiveEntry entry_type

See also

Archive formats such as zip (p. 1879)
wxArchiveEntry (p. 44)
wxArchiveOutputStream (p. 51)

wxArchiveInputStream::CloseEntry

bool CloseEntry()

Closes the current entry. On a non-seekable stream reads to the end of the current entry first.

wxArchiveInputStream::GetNextEntry

wxArchiveEntry* GetNextEntry()

Closes the current entry if one is open, then reads the meta-data for the next entry and returns it in a *wxArchiveEntry* (p. 44)object, giving away ownership. Reading this wxArchiveInputStream then returns the entry's data.

wxArchiveInputStream::OpenEntry

bool OpenEntry(**wxArchiveEntry&** *entry*)

Closes the current entry if one is open, then opens the entry specified by the *wxArchiveEntry* (p. 44) object.

entry must be from the same archive file that this wxArchiveInputStream is reading, and it must be reading it from a seekable stream.

See also

Looking up an archive entry by name (p. 1881)

wxArchiveIterator

An input iterator template class that can be used to transfer an archive's catalogue to a container. It is only available if wxUSE_STL is set to 1 in setup.h, and the uses for it outlined below require a compiler which supports member templates.

```
template <class Arc, class T = typename Arc::entry_type*>
class wxArchiveIterator
{
    // this constructor creates an 'end of sequence' object
    wxArchiveIterator();

    // template parameter 'Arc' should be the type of an archive input
stream
    wxArchiveIterator(Arc& arc) {

    /* ... */
};
```

The first template parameter should be the type of archive input stream (e.g. *wxArchiveInputStream* (p. 47)) and the second can either be a pointer to an entry (e.g. *wxArchiveEntry* (p. 44)*), or a string/pointer pair (e.g. std::pair<wxString, wxArchiveEntry*>).

The `<wx/archive.h>` header defines the following typedefs:

```
typedef wxArchiveIterator<wxArchiveInputStream> wxArchiveIter;

typedef wxArchiveIterator<wxArchiveInputStream,
        std::pair<wxString, wxArchiveEntry*> > wxArchivePairIter;
```

The header for any implementation of this interface should define similar typedefs for its types, for example in `<wx/zipstrm.h>` there is:

```
typedef wxArchiveIterator<wxZipInputStream> wxZipIter;

typedef wxArchiveIterator<wxZipInputStream,
        std::pair<wxString, wxZipEntry*> > wxZipPairIter;
```

Transferring the catalogue of an archive *arc* to a vector *cat*, can then be done something like this:

```
std::vector<wxArchiveEntry*> cat((wxArchiveIter)arc, wxArchiveIter());
```

When the iterator is dereferenced, it gives away ownership of an entry object. So in the above example, when you have finished with *cat* you must delete the pointers it contains.

If you have smart pointers with normal copy semantics (i.e. not auto_ptr or *wxScopedPtr* (p. 1139)), then you can create an iterator which uses them instead. For example, with a smart pointer class for zip entries *ZipEntryPtr*:

```
typedef std::vector<ZipEntryPtr> ZipCatalog;
typedef wxArchiveIterator<wxZipInputStream, ZipEntryPtr> ZipIter;
ZipCatalog cat((ZipIter)zip, ZipIter());
```

Iterators that return std::pair objects can be used to populate a std::multimap, to allow entries to be looked up by name. The string is initialised using the wxArchiveEntry object's *GetInternalName()* (p. 45) function.

```
typedef std::multimap<wxString, wxZipEntry*> ZipCatalog;
ZipCatalog cat((wxZipPairIter)zip, wxZipPairIter());
```

Note that this iterator also gives away ownership of an entry object each time it is dereferenced. So in the above example, when you have finished with *cat* you must delete the pointers it contains.

Or if you have them, a pair containing a smart pointer can be used (again *ZipEntryPtr*), no worries about ownership:

```
typedef std::multimap<wxString, ZipEntryPtr> ZipCatalog;
typedef wxArchiveIterator<wxZipInputStream,
              std::pair<wxString, ZipEntryPtr> > ZipPairIter;
ZipCatalog cat((ZipPairIter)zip, ZipPairIter());
```

Derived from

No base class

Include files

<wx/archive.h>

See also

wxArchiveEntry (p. 44)
wxArchiveInputStream (p. 47)
wxArchiveOutputStream (p. 51)

```
Data structurestypedef std::input_iterator_tag iterator_category
typedef T value_type
typedef ptrdiff_t difference_type
typedef T* pointer
typedef T& reference
```

wxArchiveIterator::wxArchiveIterator

wxArchiveIterator()

Construct an 'end of sequence' instance.

wxArchiveIterator(**Arc&** *arc*)

Construct iterator that returns all the entries in the archive input stream *arc*.

wxArchiveIterator::operator*

const T& operator*() **const**

Returns an entry object from the archive input stream, giving away ownership.

wxArchiveIterator::operator++

wxArchiveIterator& operator++()

wxArchiveIterator& operator++(**int**)

Position the input iterator at the next entry in the archive input stream.

wxArchiveNotifier

If you need to know when a *wxArchiveInputStream* (p. 47) updates a *wxArchiveEntry* (p. 44) object, you can create a notifier by deriving from this abstract base class, overriding *OnEntryUpdated()* (p. 51). An instance of your notifier class can then be assigned to the wxArchiveEntry object using *wxArchiveEntry::SetNotifier()* (p. 46). Your OnEntryUpdated() method will then be invoked whenever the input stream updates the entry.

Setting a notifier is not usually necessary. It is used to handle certain cases when modifying an archive in a pipeline (i.e. between non-seekable streams). See *Archives on non-seekable streams* (p. 1884).

Derived from

No base class

Include files

<wx/archive.h>

See also

Archives on non-seekable streams (p. 1884)
wxArchiveEntry (p. 44)
wxArchiveInputStream (p. 47)

wxArchiveOutputStream (p. 51)

wxArchiveNotifier::OnEntryUpdated

void OnEntryUpdated(**class wxArchiveEntry&** *entry*)

This method must be overridden in your derived class.

wxArchiveOutputStream

An abstract base class which serves as a common interface to archive output streams such as *wxZipOutputStream* (p. 1577).

PutNextEntry() (p. 53) is used to create a new entry in the output archive, then the entry's data is written to the wxArchiveOutputStream. Another call to PutNextEntry() closes the current entry and begins the next.

Derived from

wxFilterOutputStream (p. 549)

Include files

<wx/archive.h>

See also

Archive formats such as zip (p. 1879)
wxArchiveEntry (p. 44)
wxArchiveInputStream (p. 47)

wxArchiveOutputStream::~wxArchiveOutputStream

~wxArchiveOutputStream()

Calls *Close()* (p. 51) if it has not already been called.

wxArchiveOutputStream::Close

bool Close()

Closes the archive, returning true if it was successfully written. Called by the destructor if not called explicitly.

wxArchiveOutputStream::CloseEntry

bool CloseEntry()

Close the current entry. It is called implicitly whenever another new entry is created with *CopyEntry()* (p. 52)or *PutNextEntry()* (p. 53), or when the archive is closed.

wxArchiveOutputStream::CopyArchiveMetaData

bool CopyArchiveMetaData(**wxArchiveInputStream&** *stream*)

Some archive formats have additional meta-data that applies to the archive as a whole. For example in the case of zip there is a comment, which is stored at the end of the zip file. CopyArchiveMetaData() can be used to transfer such information when writing a modified copy of an archive.

Since the position of the meta-data can vary between the various archive formats, it is best to call CopyArchiveMetaData() before transferring the entries. The *wxArchiveOutputStream* (p. 51)will then hold on to the meta-data and write it at the correct point in the output file.

When the input archive is being read from a non-seekable stream, the meta-data may not be available when CopyArchiveMetaData() is called, in which case the two streams set up a link and transfer the data when it becomes available.

wxArchiveOutputStream::CopyEntry

bool CopyEntry(**wxArchiveEntry*** *entry*, **wxArchiveInputStream&** *stream*)

Takes ownership of *entry* and uses it to create a new entry in the archive. *entry* is then opened in the input stream *stream*and its contents copied to this stream.

For archive types which compress entry data, CopyEntry() is likely to be much more efficient than transferring the data using Read() and Write() since it will copy them without decompressing and recompressing them.

entry must be from the same archive file that *stream* is accessing. For non-seekable streams, *entry* must also be the last thing read from *stream*.

wxArchiveOutputStream::PutNextDirEntry

bool PutNextDirEntry(**const wxString&** *name*, **const wxDateTime&** *dt* = *wxDateTime::Now()*)

Create a new directory entry (see *wxArchiveEntry::IsDir()* (p. 46)) with the given name and timestamp.

PutNextEntry() (p. 53) can also be used to create directory entries, by supplying a name with a trailing path separator.

wxArchiveOutputStream::PutNextEntry

bool PutNextEntry(**wxArchiveEntry*** *entry*)

Takes ownership of *entry* and uses it to create a new entry in the archive. The entry's data can then be written by writing to this wxArchiveOutputStream.

bool PutNextEntry(**const wxString&** *name*, **const wxDateTime&** *dt =
wxDateTime::Now()*, **off_t** *size = wxInvalidOffset*)

Create a new entry with the given name, timestamp and size. The entry's data can then be written by writing to this wxArchiveOutputStream.

wxArray

This section describes the so called *dynamic arrays*. This is a C array-like data structure i.e. the member access time is constant (and not linear according to the number of container elements as for linked lists). However, these arrays are dynamic in the sense that they will automatically allocate more memory if there is not enough of it for adding a new element. They also perform range checking on the index values but in debug mode only, so please be sure to compile your application in debug mode to use it (see *debugging overview* (p. 1746) for details). So, unlike the arrays in some other languages, attempt to access an element beyond the arrays bound doesn't automatically expand the array but provokes an assertion failure instead in debug build and does nothing (except possibly crashing your program) in the release build.

The array classes were designed to be reasonably efficient, both in terms of run-time speed and memory consumption and the executable size. The speed of array item access is, of course, constant (independent of the number of elements) making them much more efficient than linked lists (*wxList* (p. 840)). Adding items to the arrays is also implemented in more or less constant time - but the price is preallocating the memory in advance. In the *memory management* (p. 56) section you may find some useful hints about optimizing wxArray memory usage. As for executable size, all wxArray functions are inline, so they do not take *any space at all*.

wxWidgets has three different kinds of array. All of them derive from wxBaseArray class which works with untyped data and can not be used directly. The standard macros WX_DEFINE_ARRAY(), WX_DEFINE_SORTED_ARRAY() and WX_DEFINE_OBJARRAY() are used to define a new class deriving from it. The classes declared will be called in this documentation wxArray, wxSortedArray and wxObjArray but you should keep in mind that no classes with such names actually exist, each time you use one of WX_DEFINE_XXXARRAY macro you define a class with a new name. In fact, these names are "template" names and each usage of one of the macros mentioned above creates a template specialization for the given element type.

wxArray is suitable for storing integer types and pointers which it does not treat as objects in any way, i.e. the element pointed to by the pointer is not deleted when the element is removed from the array. It should be noted that all of wxArray's functions are inline, so it costs strictly nothing to define as many array types as you want (either in terms of the executable size or the speed) as long as at least one of them is defined and this is always the case because wxArrays are used by wxWidgets internally. This class

has one serious limitation: it can only be used for storing integral types (bool, char, short, int, long and their unsigned variants) or pointers (of any kind). An attempt to use with objects of sizeof() greater than sizeof(long) will provoke a runtime assertion failure, however declaring a wxArray of floats will not (on the machines where sizeof(float) <= sizeof(long)), yet it will **not** work, please use wxObjArray for storing floats and doubles (NB: a more efficient wxArrayDouble class is scheduled for the next release of wxWidgets).

wxSortedArray is a wxArray variant which should be used when searching in the array is a frequently used operation. It requires you to define an additional function for comparing two elements of the array element type and always stores its items in the sorted order (according to this function). Thus, it is *Index()* (p. 62) function execution time is O(log(N)) instead ofO(N) for the usual arrays but the *Add()* (p. 61) method is slower: it is O(log(N)) instead of constant time (neglecting time spent in memory allocation routine). However, in a usual situation elements are added to an array much less often than searched inside it, so wxSortedArray may lead to huge performance improvements compared to wxArray. Finally, it should be noticed that, as wxArray, wxSortedArray can be only used for storing integral types or pointers.

wxObjArray class treats its elements like "objects". It may delete them when they are removed from the array (invoking the correct destructor) and copies them using the objects copy constructor. In order to implement this behaviour the definition of the wxObjArray arrays is split in two parts: first, you should declare the new wxObjArray class using WX_DECLARE_OBJARRAY() macro and then you must include the file defining the implementation of template type: <wx/arrimpl.cpp> and define the array class with WX_DEFINE_OBJARRAY() macro from a point where the full (as opposed to 'forward') declaration of the array elements class is in scope. As it probably sounds very complicated here is an example:

```
#include <wx/dynarray.h>

// we must forward declare the array because it is used inside the class
// declaration
class MyDirectory;
class MyFile;

// this defines two new types: ArrayOfDirectories and ArrayOfFiles which
can be
// now used as shown below
WX_DECLARE_OBJARRAY(MyDirectory, ArrayOfDirectories);
WX_DECLARE_OBJARRAY(MyFile,      ArrayOfFiles);

class MyDirectory
{
...
    ArrayOfDirectories m_subdirectories; // all subdirectories
    ArrayOfFiles       m_files;          // all files in this directory
};

...

// now that we have MyDirectory declaration in scope we may finish the
// definition of ArrayOfDirectories -- note that this expands into some
C++
// code and so should only be compiled once (i.e., don't put this in the
// header, but into a source file or you will get linking errors)
#include <wx/arrimpl.cpp> // this is a magic incantation which must be
```

```
done!
WX_DEFINE_OBJARRAY(ArrayOfDirectories);

// that's all!
```

It is not as elegant as writing

```
typedef std::vector<MyDirectory> ArrayOfDirectories;
```

but is not that complicated and allows the code to be compiled with any, however dumb, C++ compiler in the world.

Things are much simpler for wxArray and wxSortedArray however: it is enough just to write

```
WX_DEFINE_ARRAY(int, ArrayOfDirectories);
WX_DEFINE_SORTED_ARRAY(int, ArrayOfFiles);
```

i.e. there is only one `DEFINE` macro and no need for separate `DECLARE` one.

See also:

Container classes overview (p. 1740), *wxList* (p. 840)

Include files

<wx/dynarray.h> for wxArray and wxSortedArray and additionally <wx/arrimpl.cpp> for wxObjArray.

Macros for template array definition

To use an array you must first define the array class. This is done with the help of the macros in this section. The class of array elements must be (at least) forward declared for WX_DEFINE_ARRAY, WX_DEFINE_SORTED_ARRAY and WX_DECLARE_OBJARRAY macros and must be fully declared before you use WX_DEFINE_OBJARRAY macro.

WX_DEFINE_ARRAY (p. 57)
WX_DEFINE_EXPORTED_ARRAY (p. 57)
WX_DEFINE_USER_EXPORTED_ARRAY (p. 57)
WX_DEFINE_SORTED_ARRAY (p. 58)
WX_DEFINE_SORTED_EXPORTED_ARRAY (p. 58)
WX_DEFINE_SORTED_USER_EXPORTED_ARRAY (p. 58)
WX_DECLARE_EXPORTED_OBJARRAY (p. 59)
WX_DECLARE_USER_EXPORTED_OBJARRAY (p. 59)
WX_DEFINE_OBJARRAY (p. 59)
WX_DEFINE_EXPORTED_OBJARRAY (p. 59)
WX_DEFINE_USER_EXPORTED_OBJARRAY (p. 59)

To slightly complicate the matters even further, the operator -> defined by default for the array iterators by these macros only makes sense if the array element type is not a

pointer itself and, although it still works, this provokes warnings from some compilers and to avoid them you should use the `_PTR` versions of the macros above. For example, to define an array of pointers to `double` you should use.

Note that the above macros are generally only useful for wxObject types. There are seperate macros for declaring an array of a simple type, such as an int.

The following simple types are supported:
int
long
size_t
double

To create an array of a simple type, simply append the type you want in CAPS to the array definition.

For example, for an integer array, you'd use one of the following variants:

WX_DEFINE_ARRAY_INT (p. 57)
WX_DEFINE_EXPORTED_ARRAY_INT (p. 57)
WX_DEFINE_USER_EXPORTED_ARRAY_INT (p. 57)
WX_DEFINE_SORTED_ARRAY_INT (p. 58)
WX_DEFINE_SORTED_EXPORTED_ARRAY_INT (p. 58)
WX_DEFINE_SORTED_USER_EXPORTED_ARRAY_INT (p. 58)

```
WX_DEFINE_ARRAY_PTR(double *, MyArrayOfDoublePointers);
```

Constructors and destructors

Array classes are 100% C++ objects and as such they have the appropriate copy constructors and assignment operators. Copying wxArray just copies the elements but copying wxObjArray copies the arrays items. However, for memory-efficiency sake, neither of these classes has virtual destructor. It is not very important for wxArray which has trivial destructor anyhow, but it does mean that you should avoid deleting wxObjArray through a wxBaseArray pointer (as you would never use wxBaseArray anyhow it shouldn't be a problem) and that you should not derive your own classes from the array classes.

wxArray default constructor (p. 60)
wxArray copy constructors and assignment operators (p. 60)
~wxArray (p. 61)

Memory management

Automatic array memory management is quite trivial: the array starts by preallocating some minimal amount of memory (defined by WX_ARRAY_DEFAULT_INITIAL_SIZE) and when further new items exhaust already allocated memory it reallocates it adding 50% of the currently allocated amount, but no more than some maximal number which is defined by ARRAY_MAXSIZE_INCREMENT constant. Of course, this may lead to some

memory being wasted (ARRAY_MAXSIZE_INCREMENT in the worst case, i.e. 4Kb in the current implementation), so the *Shrink()* (p. 64) function is provided to deallocate the extra memory. The *Alloc()* (p. 61) function can also be quite useful if you know in advance how many items you are going to put in the array and will prevent the array code from reallocating the memory more times than needed.

Alloc (p. 61)
Shrink (p. 64)

Number of elements and simple item access

Functions in this section return the total number of array elements and allow to retrieve them - possibly using just the C array indexing [] operator which does exactly the same as *Item()* (p. 63) method.

Count (p. 62)
GetCount (p. 62)
IsEmpty (p. 63)
Item (p. 63)
Last (p. 63)

Adding items

Add (p. 61)
Insert (p. 63)
SetCount (p. 64)
WX_APPEND_ARRAY (p. 60)

Removing items

WX_CLEAR_ARRAY (p. 60)
Empty (p. 62)
Clear (p. 61)
RemoveAt (p. 64)
Remove (p. 63)

Searching and sorting

Index (p. 62)
Sort (p. 65)

WX_DEFINE_ARRAY

WX_DEFINE_ARRAY(*T*, *name*)

WX_DEFINE_EXPORTED_ARRAY(*T*, *name*)

WX_DEFINE_USER_EXPORTED_ARRAY(*T*, *name*, *exportspec*)

This macro defines a new array class named *name* and containing the elements of type *T*. The second form is used when compiling wxWidgets as a DLL under Windows and array needs to be visible outside the DLL. The third is needed for exporting an array from a user DLL.

Example:

```
WX_DEFINE_ARRAY_INT(wxArrayInt);

class MyClass;
WX_DEFINE_ARRAY(MyClass *, wxArrayOfMyClass);
```

Note that wxWidgets predefines the following standard array classes: wxArrayInt, wxArrayLong and wxArrayPtrVoid.

WX_DEFINE_SORTED_ARRAY

WX_DEFINE_SORTED_ARRAY(*T*, *name*)

WX_DEFINE_SORTED_EXPORTED_ARRAY(*T*, *name*)

WX_DEFINE_SORTED_USER_EXPORTED_ARRAY(*T*, *name*)

This macro defines a new sorted array class named *name* and containing the elements of type *T*. The second form is used when compiling wxWidgets as a DLL under Windows and array needs to be visible outside the DLL. The third is needed for exporting an array from a user DLL.

Example:

```
WX_DEFINE_SORTED_ARRAY_INT(wxSortedArrayInt);

class MyClass;
WX_DEFINE_SORTED_ARRAY(MyClass *, wxArrayOfMyClass);
```

You will have to initialize the objects of this class by passing a comparison function to the array object constructor like this:

```
int CompareInts(int n1, int n2)
{
    return n1 - n2;
}

wxSortedArrayInt sorted(CompareInts);

int CompareMyClassObjects(MyClass *item1, MyClass *item2)
{
    // sort the items by their address...
    return Stricmp(item1->GetAddress(), item2->GetAddress());
}

wxArrayOfMyClass another(CompareMyClassObjects);
```

WX_DECLARE_OBJARRAY

WX_DECLARE_OBJARRAY(*T*, *name*)

WX_DECLARE_EXPORTED_OBJARRAY(*T*, *name*)

WX_DECLARE_USER_EXPORTED_OBJARRAY(*T*, *name*)

This macro declares a new object array class named *name* and containing the elements of type *T*. The second form is used when compiling wxWidgets as a DLL under Windows and array needs to be visible outside the DLL. The third is needed for exporting an array from a user DLL.

Example:

```
class MyClass;
WX_DECLARE_OBJARRAY(MyClass, wxArrayOfMyClass); // note: not "MyClass *"!
```

You must use *WX_DEFINE_OBJARRAY()* (p. 59) macro to define the array class - otherwise you would get link errors.

WX_DEFINE_OBJARRAY

WX_DEFINE_OBJARRAY(*name*)

WX_DEFINE_EXPORTED_OBJARRAY(*name*)

WX_DEFINE_USER_EXPORTED_OBJARRAY(*name*)

This macro defines the methods of the array class *name* not defined by the *WX_DECLARE_OBJARRAY()* (p. 59) macro. You must include the file <wx/arrimpl.cpp> before using this macro and you must have the full declaration of the class of array elements in scope! If you forget to do the first, the error will be caught by the compiler, but, unfortunately, many compilers will not give any warnings if you forget to do the second - but the objects of the class will not be copied correctly and their real destructor will not be called. The latter two forms are merely aliases of the first to satisfy some people's sense of symmetry when using the exported declarations.

Example of usage:

```
// first declare the class!
class MyClass
{
public:
   MyClass(const MyClass&);

   ...

   virtual ~MyClass();
};

#include <wx/arrimpl.cpp>
WX_DEFINE_OBJARRAY(wxArrayOfMyClass);
```

WX_APPEND_ARRAY

void WX_APPEND_ARRAY(**wxArray&** *array*, **wxArray&** *other*)

This macro may be used to append all elements of the *other* array to the *array*. The two arrays must be of the same type.

WX_CLEAR_ARRAY

void WX_CLEAR_ARRAY(**wxArray&** *array*)

This macro may be used to delete all elements of the array before emptying it. It can not be used with wxObjArrays - but they will delete their elements anyhow when you call Empty().

Default constructors

wxArray()

wxObjArray()

Default constructor initializes an empty array object.

wxSortedArray(**int (*)(T first, T second)***compareFunction*)

There is no default constructor for wxSortedArray classes - you must initialize it with a function to use for item comparison. It is a function which is passed two arguments of type *T* where *T* is the array element type and which should return a negative, zero or positive value according to whether the first element passed to it is less than, equal to or greater than the second one.

wxArray copy constructor and assignment operator

wxArray(**const wxArray&** *array*)

wxSortedArray(**const wxSortedArray&** *array*)

wxObjArray(**const wxObjArray&** *array*)

wxArray& operator=(**const wxArray&** *array*)

wxSortedArray& operator=(**const wxSortedArray&** *array*)

wxObjArray& operator=(**const wxObjArray&** *array*)

The copy constructors and assignment operators perform a shallow array copy (i.e. they don't copy the objects pointed to even if the source array contains the items of pointer type) for wxArray and wxSortedArray and a deep copy (i.e. the array element are copied too) for wxObjArray.

wxArray::~wxArray

~wxArray()

~wxSortedArray()

~wxObjArray()

The wxObjArray destructor deletes all the items owned by the array. This is not done by wxArray and wxSortedArray versions - you may use *WX_CLEAR_ARRAY* (p. 60) macro for this.

wxArray::Add

void Add(**T** *item*, **size_t** *copies = 1*)

void Add(**T** **item*)

void Add(**T &***item*, **size_t** *copies = 1*)

Appends the given number of *copies* of the *item* to the array consisting of the elements of type *T*.

The first version is used with wxArray and wxSortedArray. The second and the third are used with wxObjArray. There is an important difference between them: if you give a pointer to the array, it will take ownership of it, i.e. will delete it when the item is deleted from the array. If you give a reference to the array, however, the array will make a copy of the item and will not take ownership of the original item. Once again, it only makes sense for wxObjArrays because the other array types never take ownership of their elements. Also note that you cannot append more than one pointer as reusing it would lead to deleting it twice (or more) and hence to a crash.

You may also use *WX_APPEND_ARRAY* (p. 60) macro to append all elements of one array to another one but it is more efficient to use *copies* parameter and modify the elements in place later if you plan to append a lot of items.

wxArray::Alloc

void Alloc(**size_t** *count*)

Preallocates memory for a given number of array elements. It is worth calling when the number of items which are going to be added to the array is known in advance because it will save unneeded memory reallocation. If the array already has enough memory for the given number of items, nothing happens.

wxArray::Clear

void Clear()

This function does the same as *Empty()* (p. 62) and additionally frees the memory

allocated to the array.

wxArray::Count

size_t Count() **const**

Same as *GetCount()* (p. 62). This function is deprecated - it exists only for compatibility.

wxObjArray::Detach

T * Detach(**size_t** *index*)

Removes the element from the array, but, unlike, *Remove()* (p. 63) doesn't delete it. The function returns the pointer to the removed element.

wxArray::Empty

void Empty()

Empties the array. For wxObjArray classes, this destroys all of the array elements. For wxArray and wxSortedArray this does nothing except marking the array of being empty - this function does not free the allocated memory, use *Clear()* (p. 61) for this.

wxArray::GetCount

size_t GetCount() **const**

Return the number of items in the array.

wxArray::Index

int Index(**T&** *item*, **bool** *searchFromEnd = false*)

int Index(**T&** *item*)

The first version of the function is for wxArray and wxObjArray, the second is for wxSortedArray only.

Searches the element in the array, starting from either beginning or the end depending on the value of *searchFromEnd* parameter. `wxNOT_FOUND` is returned if the element is not found, otherwise the index of the element is returned.

Linear search is used for the wxArray and wxObjArray classes but binary search in the sorted array is used for wxSortedArray (this is why searchFromEnd parameter doesn't make sense for it).

NB: even for wxObjArray classes, the operator==() of the elements in the array is **not**

used by this function. It searches exactly the given element in the array and so will only succeed if this element had been previously added to the array, but fail even if another, identical, element is in the array.

wxArray::Insert

void Insert(**T** *item*, **size_t** *n*, **size_t** *copies = 1*)

void Insert(**T** **item*, **size_t** *n*)

void Insert(**T** *&item*, **size_t** *n*, **size_t** *copies = 1*)

Insert the given number of *copies* of the *item* into the array before the existing item *n* - thus, *Insert(something, 0u)* will insert an item in such way that it will become the first array element.

Please see *Add()* (p. 61) for explanation of the differences between the overloaded versions of this function.

wxArray::IsEmpty

bool IsEmpty() **const**

Returns true if the array is empty, false otherwise.

wxArray::Item

T& Item(**size_t** *index*) **const**

Returns the item at the given position in the array. If *index* is out of bounds, an assert failure is raised in the debug builds but nothing special is done in the release build.

The returned value is of type "reference to the array element type" for all of the array classes.

wxArray::Last

T& Last() **const**

Returns the last element in the array, i.e. is the same as Item(GetCount() - 1). An assert failure is raised in the debug mode if the array is empty.

The returned value is of type "reference to the array element type" for all of the array classes.

wxArray::Remove

Remove(**T** *item*)

Removes an element from the array by value: the first item of the array equal to *item* is removed, an assert failure will result from an attempt to remove an item which doesn't exist in the array.

When an element is removed from wxObjArray it is deleted by the array - use *Detach()* (p. 62) if you don't want this to happen. On the other hand, when an object is removed from a wxArray nothing happens - you should delete it manually if required:

```
T *item = array[n];
delete item;
array.Remove(n)
```

See also *WX_CLEAR_ARRAY* (p. 60) macro which deletes all elements of a wxArray (supposed to contain pointers).

wxArray::RemoveAt

RemoveAt(**size_t** *index*, **size_t** *count = 1*)

Removes *count* elements starting at *index* from the array. When an element is removed from wxObjArray it is deleted by the array - use*Detach()* (p. 62) if you don't want this to happen. On the other hand, when an object is removed from a wxArray nothing happens - you should delete it manually if required:

```
T *item = array[n];
delete item;
array.RemoveAt(n)
```

See also *WX_CLEAR_ARRAY* (p. 60) macro which deletes all elements of a wxArray (supposed to contain pointers).

wxArray::SetCount

void SetCount(**size_t** *count*, **T** *defval = T(0)*)

This function ensures that the number of array elements is at least *count*. If the array has already *count* or more items, nothing is done. Otherwise, `count - GetCount()` elements are added and initialized to the value *defval*.

See also

GetCount (p. 62)

wxArray::Shrink

void Shrink()

Frees all memory unused by the array. If the program knows that no new items will be added to the array it may call Shrink() to reduce its memory usage. However, if a new

item is added to the array, some extra memory will be allocated again.

wxArray::Sort

void Sort(**CMPFUNC<T>** *compareFunction*)

The notation CMPFUNC<T> should be read as if we had the following declaration:

```
template int CMPFUNC(T *first, T *second);
```

where *T* is the type of the array elements. I.e. it is a function returning *int* which is passed two arguments of type *T **.

Sorts the array using the specified compare function: this function should return a negative, zero or positive value according to whether the first element passed to it is less than, equal to or greater than the second one.

wxSortedArray doesn't have this function because it is always sorted.

wxArrayString

wxArrayString is an efficient container for storing *wxString* (p. 1284) objects. It has the same features as all *wxArray* (p. 53) classes, i.e. it dynamically expands when new items are added to it (so it is as easy to use as a linked list), but the access time to the elements is constant, instead of being linear in number of elements as in the case of linked lists. It is also very size efficient and doesn't take more space than a C array *wxString[]* type (wxArrayString uses its knowledge of internals of wxString class to achieve this).

This class is used in the same way as other dynamic *arrays* (p. 53), except that no *WX_DEFINE_ARRAY* declaration is needed for it. When a string is added or inserted in the array, a copy of the string is created, so the original string may be safely deleted (e.g. if it was a *char ** pointer the memory it was using can be freed immediately after this). In general, there is no need to worry about string memory deallocation when using this class - it will always free the memory it uses itself.

The references returned by *Item* (p. 69), *Last* (p. 69) or *operator[]* (p. 67) are not constant, so the array elements may be modified in place like this

```
array.Last().MakeUpper();
```

There is also a variant of wxArrayString called wxSortedArrayString which has exactly the same methods as wxArrayString, but which always keeps the string in it in (alphabetical) order. wxSortedArrayString uses binary search in its *Index* (p. 68) function (instead of linear search for wxArrayString::Index) which makes it much more efficient if you add strings to the array rarely (because, of course, you have to pay for Index() efficiency by having Add() be slower) but search for them often. Several methods should not be used with sorted array (basically, all which break the order of items) which is mentioned in their description.

Final word: none of the methods of wxArrayString is virtual including its destructor, so this class should not be used as a base class.

Derived from

Although this is not true strictly speaking, this class may be considered as a specialization of *wxArray* (p. 53) class for the wxString member data: it is not implemented like this, but it does have all of the wxArray functions.

Include files

<wx/arrstr.h>

See also

wxArray (p. 53), *wxString* (p. 1284), *wxString overview* (p. 1721)

wxArrayString::wxArrayString

wxArrayString()

Default constructor.

wxArrayString(**const wxArrayString&** *array*)

Copy constructor. Note that when an array is assigned to a sorted array, its contents is automatically sorted during construction.

wxArrayString(**size_t** *sz*, **const wxChar**** *arr*)

Constructor from a C string array. Pass a size *sz* and array *arr*.

wxArrayString(**size_t** *sz*, **const wxString*** *arr*)

Constructor from a wxString array. Pass a size *sz* and array *arr*.

wxArrayString::~wxArrayString

~wxArrayString()

Destructor frees memory occupied by the array strings. For the performance reasons it is not virtual, so this class should not be derived from.

wxArrayString::operator=

wxArrayString & operator =(**const wxArrayString&** *array*)

Assignment operator.

wxArrayString::operator==

bool operator ==(**const wxArrayString&** *array*) **const**

Compares 2 arrays respecting the case. Returns true only if the arrays have the same number of elements and the same strings in the same order.

wxArrayString::operator!=

bool operator !=(**const wxArrayString&** *array*) **const**

Compares 2 arrays respecting the case. Returns true if the arrays have different number of elements or if the elements don't match pairwise.

wxArrayString::operator[]

wxString& operator[](**size_t** *nIndex*)

Return the array element at position *nIndex*. An assert failure will result from an attempt to access an element beyond the end of array in debug mode, but no check is done in release mode.

This is the operator version of *Item* (p. 69) method.

wxArrayString::Add

size_t Add(**const wxString&** *str*, **size_t** *copies = 1*)

Appends the given number of *copies* of the new item *str* to the array and returns the index of the first new item in the array.

Warning: For sorted arrays, the index of the inserted item will not be, in general, equal to *GetCount()* (p. 68) - 1 because the item is inserted at the correct position to keep the array sorted and not appended.

See also: *Insert* (p. 68)

wxArrayString::Alloc

void Alloc(**size_t** *nCount*)

Preallocates enough memory to store *nCount* items. This function may be used to improve array class performance before adding a known number of items consecutively.

See also: *Dynamic array memory management* (p. 56)

wxArrayString::Clear

void Clear()

Clears the array contents and frees memory.

See also: *Empty* (p. 68)

wxArrayString::Count

size_t Count() **const**

Returns the number of items in the array. This function is deprecated and is for backwards compatibility only, please use *GetCount* (p. 68) instead.

wxArrayString::Empty

void Empty()

Empties the array: after a call to this function *GetCount* (p. 68) will return 0. However, this function does not free the memory used by the array and so should be used when the array is going to be reused for storing other strings. Otherwise, you should use *Clear* (p. 67) to empty the array and free memory.

wxArrayString::GetCount

size_t GetCount() **const**

Returns the number of items in the array.

wxArrayString::Index

int Index(**const char *** *sz*, **bool** *bCase = true*, **bool** *bFromEnd = false*)

Search the element in the array, starting from the beginning if*bFromEnd* is false or from end otherwise. If *bCase*, comparison is case sensitive (default), otherwise the case is ignored.

This function uses linear search for wxArrayString and binary search for wxSortedArrayString, but it ignores the *bCase* and *bFromEnd* parameters in the latter case.

Returns index of the first item matched or wxNOT_FOUND if there is no match.

wxArrayString::Insert

void Insert(**const wxString&** *str*, **size_t** *nIndex*, **size_t** *copies = 1*)

Insert the given number of *copies* of the new element in the array before the position

nIndex. Thus, for example, to insert the string in the beginning of the array you would write

```
Insert("foo", 0);
```

If *nIndex* is equal to *GetCount()* this function behaves as *Add* (p. 67).

Warning: this function should not be used with sorted arrays because it could break the order of items and, for example, subsequent calls to *Index()* (p. 68) would then not work!

wxArrayString::IsEmpty

bool IsEmpty()

Returns true if the array is empty, false otherwise. This function returns the same result as *GetCount() == 0* but is probably easier to read.

wxArrayString::Item

wxString& Item(**size_t** *nIndex*) **const**

Return the array element at position *nIndex*. An assert failure will result from an attempt to access an element beyond the end of array in debug mode, but no check is done in release mode.

See also *operator[]* (p. 67) for the operator version.

wxArrayString::Last

wxString& Last()

Returns the last element of the array. Attempt to access the last element of an empty array will result in assert failure in debug build, however no checks are done in release mode.

wxArrayString::Remove

void Remove(**const char *** *sz*)

Removes the first item matching this value. An assert failure is provoked by an attempt to remove an element which does not exist in debug build.

See also: *Index* (p. 68)

wxArrayString::RemoveAt

void RemoveAt(**size_t** *nIndex*, **size_t** *count = 1*)

Removes *count* items starting at position *nIndex* from the array.

wxArrayString::Shrink

void Shrink()

Releases the extra memory allocated by the array. This function is useful to minimize the array memory consumption.

See also: *Alloc* (p. 67), *Dynamic array memory management* (p. 56)

wxArrayString::Sort

void Sort(**bool** *reverseOrder = false*)

Sorts the array in alphabetical order or in reverse alphabetical order if *reverseOrder* is true. The sort is case-sensitive.

Warning: this function should not be used with sorted array because it could break the order of items and, for example, subsequent calls to *Index()* (p. 68) would then not work!

void Sort(**CompareFunction** *compareFunction*)

Sorts the array using the specified *compareFunction* for item comparison.*CompareFunction* is defined as a function taking two *const wxString&* parameters and returning an *int* value less than, equal to or greater than 0 if the first string is less than, equal to or greater than the second one.

Example

The following example sorts strings by their length.

```
static int CompareStringLen(const wxString& first, const wxString& second)
{
    return first.length() - second.length();
}

...

wxArrayString array;

array.Add("one");
array.Add("two");
array.Add("three");
array.Add("four");

array.Sort(CompareStringLen);
```

Warning: this function should not be used with sorted array because it could break the order of items and, for example, subsequent calls to *Index()* (p. 68) would then not work!

wxArtProvider

wxArtProvider class is used to customize the look of wxWidgets application. When wxWidgets needs to display an icon or a bitmap (e.g. in the standard file dialog), it does not use a hard-coded resource but asks wxArtProvider for it instead. This way users can plug in their own wxArtProvider class and easily replace standard art with their own version. All that is needed is to derive a class from wxArtProvider, override its*CreateBitmap* (p. 73) method and register the provider with*wxArtProvider::PushProvider* (p. 75):

```
class MyProvider : public wxArtProvider
{
protected:
  wxBitmap CreateBitmap(const wxArtID& id,
                        const wxArtClient& client,
                        const wxSize size)
  { ... }
};
...
wxArtProvider::PushProvider(new MyProvider);
```

There's another way of taking advantage of this class: you can use it in your code and use platform native icons as provided by *wxArtProvider::GetBitmap* (p. 74) or *wxArtProvider::GetIcon* (p. 74) (NB: this is not yet really possible as of wxWidgets 2.3.3, the set of wxArtProvider bitmaps is too small).

Identifying art resources

Every bitmap is known to wxArtProvider under an unique ID that is used by when requesting a resource from it. The ID is represented by wxArtID type and can have one of these predefined values (you can see bitmaps represented by these constants in the *artprov* (p. 1707) sample):

- wxART_ADD_BOOKMARK

- wxART_DEL_BOOKMARK

- wxART_HELP_SIDE_PANEL

- wxART_HELP_SETTINGS

- wxART_HELP_BOOK

- wxART_HELP_FOLDER

- wxART_HELP_PAGE

- wxART_GO_BACK

- wxART_GO_FORWARD

- wxART_GO_UP

- wxART_GO_DOWN

- wxART_GO_TO_PARENT

- • wxART_GO_HOME

- • wxART_FILE_OPEN

- • wxART_PRINT

- • wxART_HELP

- • wxART_TIP

- • wxART_REPORT_VIEW

- • wxART_LIST_VIEW

- • wxART_NEW_DIR

- • wxART_FOLDER

- • wxART_GO_DIR_UP

- • wxART_EXECUTABLE_FILE

- • wxART_NORMAL_FILE

- • wxART_TICK_MARK

- • wxART_CROSS_MARK

- • wxART_ERROR

- • wxART_QUESTION

- • wxART_WARNING

- • wxART_INFORMATION

- • wxART_MISSING_IMAGE

Additionally, any string recognized by custom art providers registered using*PushProvider* (p. 75) may be used.

GTK+ Note

When running under GTK+ 2, GTK+ stock item IDs (e.g. `"gtk-cdrom"`) may be used as well. Additionally, if wxGTK was compiled against GTK+ >= 2.4, then it is also possible to load icons from current icon theme by specifying their name (without extension and directory components). Icon themes recognized by GTK+ follow thefreedesktop.org Icon Themes specification (`http://freedesktop.org/Standards/icon-theme-spec`). Note that themes are not guaranteed to contain all icons, so wxArtProvider may return `wxNullBitmap` or `wxNullIcon`. Default theme is typically installed in `/usr/share/icons/hicolor`.

Clients

Client is the entity that calls wxArtProvider's GetBitmap or GetIcon function. It is represented by wxClientID type and can have one of these values:

- wxART_TOOLBAR

- wxART_MENU

- wxART_BUTTON

- wxART_FRAME_ICON

- wxART_CMN_DIALOG

- wxART_HELP_BROWSER

- wxART_MESSAGE_BOX

- wxART_OTHER (used for all requests that don't fit into any of the categories above)Client ID servers as a hint to wxArtProvider that is supposed to help it to choose the best looking bitmap. For example it is often desirable to use slightly different icons in menus and toolbars even though they represent the same action (e.g. `wx_ART_FILE_OPEN`). Remember that this is really only a hint for wxArtProvider -- it is common that*wxArtProvider::GetBitmap* (p. 74) returns identical bitmap for different *client* values!

See also

See the *artprov* (p. 1707) sample for an example of wxArtProvider usage.

Derived from

wxObject (p. 1012)

Include files

<wx/artprov.h>

wxArtProvider::CreateBitmap

wxBitmap CreateBitmap(**const wxArtID&** *id*, **const wxArtClient&** *client*, **const wxSize&** *size*)

Derived art provider classes must override this method to create requested art resource. Note that returned bitmaps are cached by wxArtProvider and it is therefore not necessary to optimize CreateBitmap for speed (e.g. you may create wxBitmap objects from XPMs here).

Parameters

id

>   wxArtID unique identifier of the bitmap.

client

>   wxArtClient identifier of the client (i.e. who is asking for the bitmap). This only
>   servers as a hint.

size

>   Preferred size of the bitmap. The function may return a bitmap of different
>   dimensions, it will be automatically rescaled to meet client's request.

Note

This is **not** part of wxArtProvider's public API, use*wxArtProvider::GetBitmap* (p. 74) or
wxArtProvider::GetIcon (p. 74)to query wxArtProvider for a resource.

wxArtProvider::GetBitmap

static wxBitmap GetBitmap(**const wxArtID&** *id*, **const wxArtClient&** *client* =
wxART_OTHER, **const wxSize&** *size* = *wxDefaultSize*)

Query registered providers for bitmap with given ID.

Parameters

id

>   wxArtID unique identifier of the bitmap.

client

>   wxArtClient identifier of the client (i.e. who is asking for the bitmap).

size

>   Size of the returned bitmap or `wxDefaultSize` if size doesn't matter.

Return value

The bitmap if one of registered providers recognizes the ID or wxNullBitmap otherwise.

wxArtProvider::GetIcon

static wxIcon GetIcon(**const wxArtID&** *id*, **const wxArtClient&** *client* =
wxART_OTHER, **const wxSize&** *size* = *wxDefaultSize*)

Same as *wxArtProvider::GetBitmap* (p. 74), but return a wxIcon object (or wxNullIcon on
failure).

static wxSize GetSizeHint(**const wxArtClient&** *client*, **bool** *platform_default = false*)

Returns a suitable size hint for the given *wxArtClient*. If *platform_default* is `true`, return a size based on the current platform, otherwise return the size from the topmost wxArtProvider. *wxDefaultSize* may be returned if the client doesn't have a specified size, like wxART_OTHER for example.

wxArtProvider::PopProvider

static bool PopProvider()

Remove latest added provider and delete it.

wxArtProvider::PushProvider

static void PushProvider(**wxArtProvider*** *provider*)

Register new art provider (add it to the top of providers stack).

wxArtProvider::RemoveProvider

static bool RemoveProvider(**wxArtProvider*** *provider*)

Remove a provider from the stack. The provider must have been added previously and is *not* deleted.

wxAutomationObject

The **wxAutomationObject** class represents an OLE automation object containing a single data member, an IDispatch pointer. It contains a number of functions that make it easy to perform automation operations, and set and get properties. The class makes heavy use of the *wxVariant* (p. 1463) class.

The usage of these classes is quite close to OLE automation usage in Visual Basic. The API is high-level, and the application can specify multiple properties in a single string. The following example gets the current Excel instance, and if it exists, makes the active cell bold.

```
wxAutomationObject excelObject;
if (excelObject.GetInstance("Excel.Application"))
    excelObject.PutProperty("ActiveCell.Font.Bold", true);
```

Note that this class obviously works under Windows only.

Derived from

wxObject (p. 1012)

Include files

<wx/msw/ole/automtn.h>

wxAutomationObject::wxAutomationObject

 wxAutomationObject(**WXIDISPATCH*** *dispatchPtr = NULL*)

Constructor, taking an optional IDispatch pointer which will be released when the object is deleted.

wxAutomationObject::~wxAutomationObject

 ~wxAutomationObject()

Destructor. If the internal IDispatch pointer is non-null, it will be released.

wxAutomationObject::CallMethod

wxVariant CallMethod(**const wxString&** *method*, **int** *noArgs*, **wxVariant** *args[]*) **const**

wxVariant CallMethod(**const wxString&** *method*, **...**) **const**

Calls an automation method for this object. The first form takes a method name, number of arguments, and an array of variants. The second form takes a method name and zero to six constant references to variants. Since the variant class has constructors for the basic data types, and C++ provides temporary objects automatically, both of the following lines are syntactically valid:

```
wxVariant res = obj.CallMethod("Sum", wxVariant(1.2), wxVariant(3.4));
wxVariant res = obj.CallMethod("Sum", 1.2, 3.4);
```

Note that *method* can contain dot-separated property names, to save the application needing to call GetProperty several times using several temporary objects. For example:

```
object.CallMethod("ActiveCell.Font.ShowDialog", "My caption");
```

wxAutomationObject::CreateInstance

bool CreateInstance(**const wxString&** *classId*) **const**

Creates a new object based on the class id, returning true if the object was successfully created, or false if not.

wxAutomationObject::GetDispatchPtr

IDispatch* GetDispatchPtr() **const**

Gets the IDispatch pointer.

wxAutomationObject::GetInstance

bool GetInstance(**const wxString&** *classId*) **const**

Retrieves the current object associated with a class id, and attaches the IDispatch pointer to this object. Returns true if a pointer was successfully retrieved, false otherwise.

Note that this cannot cope with two instances of a given OLE object being active simultaneously, such as two copies of Excel running. Which object is referenced cannot currently be specified.

wxAutomationObject::GetObject

bool GetObject(**wxAutomationObject&***obj* **const wxString&** *property*, **int** *noArgs = 0*, **wxVariant** *args[] = NULL*) **const**

Retrieves a property from this object, assumed to be a dispatch pointer, and initialises *obj* with it. To avoid having to deal with IDispatch pointers directly, use this function in preference to *wxAutomationObject::GetProperty* (p. 77) when retrieving objects from other objects.

Note that an IDispatch pointer is stored as a void* pointer in wxVariant objects.

See also

wxAutomationObject::GetProperty (p. 77)

wxAutomationObject::GetProperty

wxVariant GetProperty(**const wxString&** *property*, **int** *noArgs*, **wxVariant** *args[]*) **const**

wxVariant GetProperty(**const wxString&** *property*, **...**) **const**

Gets a property value from this object. The first form takes a property name, number of arguments, and an array of variants. The second form takes a property name and zero to six constant references to variants. Since the variant class has constructors for the basic data types, and C++ provides temporary objects automatically, both of the

following lines are syntactically valid:

```
wxVariant res = obj.GetProperty("Range", wxVariant("A1"));
wxVariant res = obj.GetProperty("Range", "A1");
```

Note that *property* can contain dot-separated property names, to save the application needing to call GetProperty several times using several temporary objects.

wxAutomationObject::Invoke

bool Invoke(**const wxString&** *member*, **int** *action*, **wxVariant&** *retValue*, **int** *noArgs*, **wxVariant** *args[]*, **const wxVariant*** *ptrArgs[] = 0*) **const**

This function is a low-level implementation that allows access to the IDispatch Invoke function. It is not meant to be called directly by the application, but is used by other convenience functions.

Parameters

member

The member function or property name.

action

Bitlist: may contain DISPATCH_PROPERTYPUT, DISPATCH_PROPERTYPUTREF, DISPATCH_METHOD.

retValue

Return value (ignored if there is no return value)

.

noArgs

Number of arguments in *args* or *ptrArgs*.

args

If non-null, contains an array of variants.

ptrArgs

If non-null, contains an array of constant pointers to variants.

Return value

true if the operation was successful, false otherwise.

Remarks

Two types of argument array are provided, so that when possible pointers are used for efficiency.

wxAutomationObject::PutProperty

bool PutProperty(**const wxString&** *property*, **int** *noArgs*, **wxVariant** *args[]*) **const**

bool PutProperty(**const wxString&** *property*, **...**)

Puts a property value into this object. The first form takes a property name, number of arguments, and an array of variants. The second form takes a property name and zero to six constant references to variants. Since the variant class has constructors for the basic data types, and C++ provides temporary objects automatically, both of the following lines are syntactically valid:

```
obj.PutProperty("Value", wxVariant(23));
obj.PutProperty("Value", 23);
```

Note that *property* can contain dot-separated property names, to save the application needing to call GetProperty several times using several temporary objects.

wxAutomationObject::SetDispatchPtr

void SetDispatchPtr(**WXIDISPATCH*** *dispatchPtr*)

Sets the IDispatch pointer. This function does not check if there is already an IDispatch pointer.

You may need to cast from IDispatch* to WXIDISPATCH* when calling this function.

wxBitmap

This class encapsulates the concept of a platform-dependent bitmap, either monochrome or colour or colour with alpha channel support.

Derived from

wxGDIObject (p. 607)
wxObject (p. 1012)

Include file

<wx/bitmap.h>

Predefined objects

Objects:

wxNullBitmap

See also

wxBitmap overview (p. 1788),*supported bitmap file formats* (p. 1789),*wxDC::Blit* (p. 367),*wxIcon* (p. 766), *wxCursor* (p. 226), *wxBitmap* (p. 79),*wxMemoryDC* (p. 935)

wxBitmap::wxBitmap

wxBitmap()

Default constructor.

wxBitmap(**const wxBitmap&** *bitmap*)

Copy constructor. Note that this does not take a fresh copy of the data, but instead makes the internal data point to *bitmap*'s data. So changing one bitmap will change the other. To make a real copy, you can use:

```
wxBitmap newBitmap = oldBitmap.GetSubBitmap(
                           wxRect(0, 0, oldBitmap.GetWidth(),
oldBitmap.GetHeight()));
```

wxBitmap(**void*** *data*, **int** *type*, **int** *width*, **int** *height*, **int** *depth = -1*)

Creates a bitmap from the given data which is interpreted in platform-dependent manner.

wxBitmap(**const char** *bits[]*, **int** *width*, **int** *height*
 int *depth = 1*)

Creates a bitmap from an array of bits.

You should only use this function for monochrome bitmaps (*depth* 1) in portable programs: in this case the *bits* parameter should contain an XBM image.

For other bit depths, the behaviour is platform dependent: under Windows, the data is passed without any changes to the underlying `CreateBitmap()` API. Under other platforms, only monochrome bitmaps may be created using this constructor and *wxImage* (p. 779) should be used for creating colour bitmaps from static data.

wxBitmap(**int** *width*, **int** *height*, **int** *depth = -1*)

Creates a new bitmap. A depth of -1 indicates the depth of the current screen or visual. Some platforms only support 1 for monochrome and -1 for the current colour setting. Beginning with version 2.5.4 of wxWidgets a depth of 32 including an alpha channel is supported under MSW, Mac and GTK+.

wxBitmap(**const char**** *bits*)

Creates a bitmap from XPM data.

wxBitmap(**const wxString&** *name*, **long** *type*)

Loads a bitmap from a file or resource.

wxBitmap(**const wxImage&** *img*, **int** *depth = -1*)

Creates bitmap object from the image. This has to be done to actually display an image as you cannot draw an image directly on a window. The resulting bitmap will use the provided colour depth (or that of the current system if depth is -1) which entails that a colour reduction has to take place.

When in 8-bit mode (PseudoColour mode), the GTK port will use a color cube created on program start-up to look up colors. This ensures a very fast conversion, but the image quality won't be perfect (and could be better for photo images using more sophisticated dithering algorithms).

On Windows, if there is a palette present (set with SetPalette), it will be used when creating the wxBitmap (most useful in 8-bit display mode). On other platforms, the palette is currently ignored.

Parameters

bits

> Specifies an array of pixel values.

width

> Specifies the width of the bitmap.

height

> Specifies the height of the bitmap.

depth

> Specifies the depth of the bitmap. If this is omitted, the display depth of the screen is used.

name

> This can refer to a resource name under MS Windows, or a filename under MS Windows and X. Its meaning is determined by the *type* parameter.

type

> May be one of the following:
>
> wxBITMAP_TYPE_BMP   Load a Windows bitmap file.
>
> wxBITMAP_TYPE_BMP_RESOURCELoad a Windows bitmap resource from the executable. Windows only.
>
> wxBITMAP_TYPE_PICT_RESOURCE   Load a PICT image resource from the executable. Mac OS only.

wxBITMAP_TYPE_GIF   Load a GIF bitmap file.

wxBITMAP_TYPE_XBM   Load an X bitmap file.

wxBITMAP_TYPE_XPM   Load an XPM bitmap file.

The validity of these flags depends on the platform and wxWidgets configuration. If all possible wxWidgets settings are used, the Windows platform supports BMP file, BMP resource, XPM data, and XPM. Under wxGTK, the available formats are BMP file, XPM data, XPM file, and PNG file. Under wxMotif, the available formats are XBM data, XBM file, XPM data, XPM file.

In addition, wxBitmap can read all formats that *wxImage* (p. 779) can, which currently include wxBITMAP_TYPE_JPEG, wxBITMAP_TYPE_TIF, wxBITMAP_TYPE_PNG, wxBITMAP_TYPE_GIF, wxBITMAP_TYPE_PCX, and wxBITMAP_TYPE_PNM. Of course, you must have wxImage handlers loaded.

img

Platform-independent wxImage object.

Remarks

The first form constructs a bitmap object with no data; an assignment or another member function such as Create or LoadFile must be called subsequently.

The second and third forms provide copy constructors. Note that these do not copy the bitmap data, but instead a pointer to the data, keeping a reference count. They are therefore very efficient operations.

The fourth form constructs a bitmap from data whose type and value depends on the value of the *type* argument.

The fifth form constructs a (usually monochrome) bitmap from an array of pixel values, under both X and Windows.

The sixth form constructs a new bitmap.

The seventh form constructs a bitmap from pixmap (XPM) data, if wxWidgets has been configured to incorporate this feature.

To use this constructor, you must first include an XPM file. For example, assuming that the file `mybitmap.xpm` contains an XPM array of character pointers called mybitmap:

```
#include "mybitmap.xpm"

...

wxBitmap *bitmap = new wxBitmap(mybitmap);
```

The eighth form constructs a bitmap from a file or resource. *name* can refer to a resource name under MS Windows, or a filename under MS Windows and X.

Under Windows, *type* defaults to wxBITMAP_TYPE_BMP_RESOURCE. Under X, *type* defaults to wxBITMAP_TYPE_XPM.

See also

wxBitmap::LoadFile (p. 88)

wxPython note: Constructors supported by wxPython are:

wxBitmap(name, flag) Loads a bitmap from a file

wxEmptyBitmap(width, height, depth = -1) Creates an empty bitmap with the given specifications

wxBitmapFromXPMData(listOfStrings) Create a bitmap from a Python list of strings whose contents are XPM data.

wxBitmapFromBits(bits, width, height, depth=-1) Create a bitmap from an array of bits contained in a string.

wxBitmapFromImage(image, depth=-1) Convert a wxImage to a wxBitmap.

wxPerl note: Constructors supported by wxPerl are:

• ::Bitmap->new(width, height, depth = -1)

• ::Bitmap->new(name, type)

• ::Bitmap->new(icon)

• ::Bitmap->newFromBits(bits, width, height, depth = 1)

• ::Bitmap->newFromXPM(data)

wxBitmap::~wxBitmap

~wxBitmap()

Destroys the wxBitmap object and possibly the underlying bitmap data. Because reference counting is used, the bitmap may not actually be destroyed at this point - only when the reference count is zero will the data be deleted.

If the application omits to delete the bitmap explicitly, the bitmap will be destroyed automatically by wxWidgets when the application exits.

Do not delete a bitmap that is selected into a memory device context.

wxBitmap::AddHandler

static void AddHandler(**wxBitmapHandler*** *handler*)

Adds a handler to the end of the static list of format handlers.

handler

> A new bitmap format handler object. There is usually only one instance of a given handler class in an application session.

See also

wxBitmapHandler (p. 98)

wxBitmap::CleanUpHandlers

static void CleanUpHandlers()

Deletes all bitmap handlers.

This function is called by wxWidgets on exit.

wxBitmap::ConvertToImage

wxImage ConvertToImage()

Creates an image from a platform-dependent bitmap. This preserves mask information so that bitmaps and images can be converted back and forth without loss in that respect.

wxBitmap::CopyFromIcon

bool CopyFromIcon(**const wxIcon&** *icon*)

Creates the bitmap from an icon.

wxBitmap::Create

virtual bool Create(**int** *width*, **int** *height*, **int** *depth = -1*)

Creates a fresh bitmap. If the final argument is omitted, the display depth of the screen is used.

virtual bool Create(**void*** *data*, **int** *type*, **int** *width*, **int** *height*, **int** *depth = -1*)

Creates a bitmap from the given data, which can be of arbitrary type.

Parameters

width

> The width of the bitmap in pixels.

height

> The height of the bitmap in pixels.

depth

> The depth of the bitmap in pixels. If this is -1, the screen depth is used.

data

> Data whose type depends on the value of *type*.

type

> A bitmap type identifier - see *wxBitmap::wxBitmap* (p. 80) for a list of possible values.

Return value

true if the call succeeded, false otherwise.

Remarks

The first form works on all platforms. The portability of the second form depends on the type of data.

See also

wxBitmap::wxBitmap (p. 80)

wxBitmap::FindHandler

static wxBitmapHandler* FindHandler(**const wxString&** *name*)

Finds the handler with the given name.

static wxBitmapHandler* FindHandler(**const wxString&** *extension*, **wxBitmapType** *bitmapType*)

Finds the handler associated with the given extension and type.

static wxBitmapHandler* FindHandler(**wxBitmapType** *bitmapType*)

Finds the handler associated with the given bitmap type.

name

> The handler name.

extension

> The file extension, such as "bmp".

bitmapType

> The bitmap type, such as wxBITMAP_TYPE_BMP.

Return value

A pointer to the handler if found, NULL otherwise.

See also

wxBitmapHandler (p. 98)

wxBitmap::GetDepth

int GetDepth() **const**

Gets the colour depth of the bitmap. A value of 1 indicates a monochrome bitmap.

wxBitmap::GetHandlers

static wxList& GetHandlers()

Returns the static list of bitmap format handlers.

See also

wxBitmapHandler (p. 98)

wxBitmap::GetHeight

int GetHeight() **const**

Gets the height of the bitmap in pixels.

wxBitmap::GetPalette

wxPalette* GetPalette() **const**

Gets the associated palette (if any) which may have been loaded from a file or set for the bitmap.

See also

wxPalette (p. 1027)

wxBitmap::GetMask

wxMask* GetMask() **const**

Gets the associated mask (if any) which may have been loaded from a file or set for the bitmap.

See also

wxBitmap::SetMask (p. 90), *wxMask* (p. 906)

wxBitmap::GetWidth

int GetWidth() **const**

Gets the width of the bitmap in pixels.

See also

wxBitmap::GetHeight (p. 86)

wxBitmap::GetSubBitmap

wxBitmap GetSubBitmap(**const wxRect&***rect*) **const**

Returns a sub bitmap of the current one as long as the rect belongs entirely to the bitmap. This function preserves bit depth and mask information.

wxBitmap::InitStandardHandlers

static void InitStandardHandlers()

Adds the standard bitmap format handlers, which, depending on wxWidgets configuration, can be handlers for Windows bitmap, Windows bitmap resource, and XPM.

This function is called by wxWidgets on startup.

See also

wxBitmapHandler (p. 98)

wxBitmap::InsertHandler

static void InsertHandler(**wxBitmapHandler*** *handler*)

Adds a handler at the start of the static list of format handlers.

handler

    A new bitmap format handler object. There is usually only one instance of a given handler class in an application session.

See also

wxBitmapHandler (p. 98)

wxBitmap::LoadFile

bool LoadFile(**const wxString&** *name*, **wxBitmapType** *type*)

Loads a bitmap from a file or resource.

Parameters

name

> Either a filename or a Windows resource name. The meaning of *name* is determined by the *type* parameter.

type

> One of the following values:

> **wxBITMAP_TYPE_BMP**   Load a Windows bitmap file.

> **wxBITMAP_TYPE_BMP_RESOURCE**   Load a Windows bitmap resource from the executable.

> **wxBITMAP_TYPE_PICT_RESOURCE**   Load a PICT image resource from the executable. Mac OS only.

> **wxBITMAP_TYPE_GIF**   Load a GIF bitmap file.

> **wxBITMAP_TYPE_XBM**   Load an X bitmap file.

> **wxBITMAP_TYPE_XPM**   Load an XPM bitmap file.

> The validity of these flags depends on the platform and wxWidgets configuration.

> In addition, wxBitmap can read all formats that *wxImage* (p. 779) can (wxBITMAP_TYPE_JPEG, wxBITMAP_TYPE_PNG, wxBITMAP_TYPE_GIF, wxBITMAP_TYPE_PCX, wxBITMAP_TYPE_PNM). (Of course you must have wxImage handlers loaded.)

Return value

true if the operation succeeded, false otherwise.

Remarks

A palette may be associated with the bitmap if one exists (especially for colour Windows bitmaps), and if the code supports it. You can check if one has been created by using the *GetPalette* (p. 86) member.

See also

wxBitmap::SaveFile (p. 89)

wxBitmap::Ok

bool Ok() **const**

Returns true if bitmap data is present.

wxBitmap::RemoveHandler

static bool RemoveHandler(**const wxString&** *name*)

Finds the handler with the given name, and removes it. The handler is not deleted.

name

> The handler name.

Return value

true if the handler was found and removed, false otherwise.

See also

wxBitmapHandler (p. 98)

wxBitmap::SaveFile

bool SaveFile(**const wxString&** *name*, **wxBitmapType** *type*, **wxPalette*** *palette* = *NULL*)

Saves a bitmap in the named file.

Parameters

name

> A filename. The meaning of *name* is determined by the *type* parameter.

type

> One of the following values:

> | **wxBITMAP_TYPE_BMP** | Save a Windows bitmap file. |
> |---|---|
> | **wxBITMAP_TYPE_GIF** | Save a GIF bitmap file. |
> | **wxBITMAP_TYPE_XBM** | Save an X bitmap file. |
> | **wxBITMAP_TYPE_XPM** | Save an XPM bitmap file. |

> The validity of these flags depends on the platform and wxWidgets configuration.

> In addition, wxBitmap can save all formats that *wxImage* (p. 779) can (wxBITMAP_TYPE_JPEG, wxBITMAP_TYPE_PNG). (Of course you must have wxImage handlers loaded.)

palette

An optional palette used for saving the bitmap.

Return value

true if the operation succeeded, false otherwise.

Remarks

Depending on how wxWidgets has been configured, not all formats may be available.

See also

wxBitmap::LoadFile (p. 88)

wxBitmap::SetDepth

void SetDepth(**int** *depth*)

Sets the depth member (does not affect the bitmap data).

Parameters

depth

Bitmap depth.

wxBitmap::SetHeight

void SetHeight(**int** *height*)

Sets the height member (does not affect the bitmap data).

Parameters

height

Bitmap height in pixels.

wxBitmap::SetMask

void SetMask(**wxMask*** *mask*)

Sets the mask for this bitmap.

Remarks

The bitmap object owns the mask once this has been called.

See also

wxBitmap::GetMask (p. 86), *wxMask* (p. 906)

wxBitmap::SetPalette

void SetPalette(**const wxPalette&** *palette*)

Sets the associated palette. (Not implemented under GTK+).

Parameters

palette

The palette to set.

See also

wxPalette (p. 1027)

wxBitmap::SetWidth

void SetWidth(**int** *width*)

Sets the width member (does not affect the bitmap data).

Parameters

width

Bitmap width in pixels.

wxBitmap::operator =

wxBitmap& operator =(**const wxBitmap&** *bitmap*)

Assignment operator. This operator does not copy any data, but instead passes a pointer to the data in *bitmap* and increments a reference counter. It is a fast operation.

Parameters

bitmap

Bitmap to assign.

Return value

Returns 'this' object.

wxBitmap::operator ==

bool operator ==(**const wxBitmap&** *bitmap*)

Equality operator. This operator tests whether the internal data pointers are equal (a fast test).

Parameters

bitmap

Bitmap to compare with 'this'

Return value

Returns true if the bitmaps were effectively equal, false otherwise.

---

wxBitmap::operator !=

bool operator !=(**const wxBitmap&** *bitmap*)

Inequality operator. This operator tests whether the internal data pointers are unequal (a fast test).

Parameters

bitmap

Bitmap to compare with 'this'

Return value

Returns true if the bitmaps were unequal, false otherwise.

---

wxBitmapButton

A bitmap button is a control that contains a bitmap. It may be placed on a *dialog box* (p. 406) or *panel* (p. 1030), or indeed almost any other window.

Derived from

wxButton (p. 117)
wxControl (p. 215)
wxWindow (p. 1490)
wxEvtHandler (p. 486)
wxObject (p. 1012)

Include files

<wx/bmpbuttn.h>

Remarks

A bitmap button can be supplied with a single bitmap, and wxWidgets will draw all button states using this bitmap. If the application needs more control, additional bitmaps for the selected state, unpressed focused state, and greyed-out state may be supplied.

Window styles

wxBU_AUTODRAW	If this is specified, the button will be drawn automatically using the label bitmap only, providing a 3D-look border. If this style is not specified, the button will be drawn without borders and using all provided bitmaps. WIN32 only.
wxBU_LEFT	Left-justifies the bitmap label. WIN32 only.
wxBU_TOP	Aligns the bitmap label to the top of the button. WIN32 only.
wxBU_RIGHT	Right-justifies the bitmap label. WIN32 only.
wxBU_BOTTOM	Aligns the bitmap label to the bottom of the button. WIN32 only.

Note that **wxBU_EXACTFIT** supported by *wxButton* (p. 117) is *not* used by this class as bitmap buttons don't have any minimal standard size by default.

See also *window styles overview* (p. 1762).

Event handling

EVT_BUTTON(id, func)	Process a wxEVT_COMMAND_BUTTON_CLICKED event, when the button is clicked.

See also

wxButton (p. 117)

wxBitmapButton::wxBitmapButton

 wxBitmapButton()

Default constructor.

 wxBitmapButton(**wxWindow*** *parent*, **wxWindowID** *id*, **const wxBitmap&** *bitmap*, **const wxPoint&** *pos* = *wxDefaultPosition*, **const wxSize&** *size* = *wxDefaultSize*, **long** *style* = *wxBU_AUTODRAW*, **const wxValidator&** *validator* = *wxDefaultValidator*, **const wxString&** *name* = *"button"*)

Constructor, creating and showing a button.

Parameters

parent

　　Parent window. Must not be NULL.

id

Button identifier. A value of -1 indicates a default value.

bitmap

Bitmap to be displayed.

pos

Button position.

size

Button size. If the default size (-1, -1) is specified then the button is sized appropriately for the bitmap.

style

Window style. See *wxBitmapButton* (p. 92).

validator

Window validator.

name

Window name.

Remarks

The *bitmap* parameter is normally the only bitmap you need to provide, and wxWidgets will draw the button correctly in its different states. If you want more control, call any of the functions *wxBitmapButton::SetBitmapSelected* (p. 97), *wxBitmapButton::SetBitmapFocus* (p. 96), *wxBitmapButton::SetBitmapDisabled* (p. 96).

Note that the bitmap passed is smaller than the actual button created.

See also

wxBitmapButton::Create (p. 94), *wxValidator* (p. 1461)

wxBitmapButton::~wxBitmapButton

 ~wxBitmapButton()

Destructor, destroying the button.

wxBitmapButton::Create

bool Create(**wxWindow*** *parent*, **wxWindowID** *id*, **const wxBitmap&** *bitmap*, **const wxPoint&** *pos*, **const wxSize&** *size = wxDefaultSize*, **long** *style = 0*, **const wxValidator&** *validator*, **const wxString&** *name = "button"*)

Button creation function for two-step creation. For more details, see

wxBitmapButton::wxBitmapButton (p. 93).

wxBitmapButton::GetBitmapDisabled

wxBitmap& GetBitmapDisabled() **const**

Returns the bitmap for the disabled state.

Return value

A reference to the disabled state bitmap.

See also

wxBitmapButton::SetBitmapDisabled (p. 96)

wxBitmapButton::GetBitmapFocus

wxBitmap& GetBitmapFocus() **const**

Returns the bitmap for the focused state.

Return value

A reference to the focused state bitmap.

See also

wxBitmapButton::SetBitmapFocus (p. 96)

wxBitmapButton::GetBitmapLabel

wxBitmap& GetBitmapLabel() **const**

Returns the label bitmap (the one passed to the constructor).

Return value

A reference to the button's label bitmap.

See also

wxBitmapButton::SetBitmapLabel (p. 96)

wxBitmapButton::GetBitmapSelected

wxBitmap& GetBitmapSelected() **const**

Returns the bitmap for the selected state.

Return value

A reference to the selected state bitmap.

See also

wxBitmapButton::SetBitmapSelected (p. 97)

wxBitmapButton::SetBitmapDisabled

void SetBitmapDisabled(**const wxBitmap&** *bitmap*)

Sets the bitmap for the disabled button appearance.

Parameters

bitmap

The bitmap to set.

See also

wxBitmapButton::GetBitmapDisabled (p. 95), *wxBitmapButton::SetBitmapLabel* (p. 96), *wxBitmapButton::SetBitmapSelected* (p. 97), *wxBitmapButton::SetBitmapFocus* (p. 96)

wxBitmapButton::SetBitmapFocus

void SetBitmapFocus(**const wxBitmap&** *bitmap*)

Sets the bitmap for the button appearance when it has the keyboard focus.

Parameters

bitmap

The bitmap to set.

See also

wxBitmapButton::GetBitmapFocus (p. 95), *wxBitmapButton::SetBitmapLabel* (p. 96), *wxBitmapButton::SetBitmapSelected* (p. 97), *wxBitmapButton::SetBitmapDisabled* (p. 96)

wxBitmapButton::SetBitmapLabel

void SetBitmapLabel(**const wxBitmap&** *bitmap*)

Sets the bitmap label for the button.

Parameters

bitmap

> The bitmap label to set.

Remarks

This is the bitmap used for the unselected state, and for all other states if no other bitmaps are provided.

See also

wxBitmapButton::GetBitmapLabel (p. 95)

wxBitmapButton::SetBitmapSelected

void SetBitmapSelected(**const wxBitmap&** *bitmap*)

Sets the bitmap for the selected (depressed) button appearance.

Parameters

bitmap

> The bitmap to set.

See also

wxBitmapButton::GetBitmapSelected (p. 95), *wxBitmapButton::SetBitmapLabel* (p. 96), *wxBitmapButton::SetBitmapFocus* (p. 96), *wxBitmapButton::SetBitmapDisabled* (p. 96)

wxBitmapDataObject

wxBitmapDataObject is a specialization of wxDataObject for bitmap data. It can be used without change to paste data into the *wxClipboard* (p. 150) or a *wxDropSource* (p. 468). A user may wish to derive a new class from this class for providing a bitmap on-demand in order to minimize memory consumption when offering data in several formats, such as a bitmap and GIF.

wxPython note: If you wish to create a derived wxBitmapDataObject class in wxPython you should derive the class from wxPyBitmapDataObject in order to get Python-aware capabilities for the various virtual methods.

Virtual functions to override

This class may be used as is, but *GetBitmap* (p. 98) may be overridden to increase efficiency.

Derived from

wxDataObjectSimple (p. 243)
wxDataObject (p. 239)

Include files

<wx/dataobj.h>

See also

Clipboard and drag and drop overview (p. 1817), *wxDataObject* (p. 239),
wxDataObjectSimple (p. 243), *wxFileDataObject* (p. 510), *wxTextDataObject* (p. 1359),
wxDataObject (p. 239)

 wxBitmapDataObject(**const wxBitmap&** *bitmap = wxNullBitmap*)

Constructor, optionally passing a bitmap (otherwise use *SetBitmap* (p. 98) later).

wxBitmapDataObject::GetBitmap

virtual wxBitmap GetBitmap() **const**

Returns the bitmap associated with the data object. You may wish to override this
method when offering data on-demand, but this is not required by wxWidgets' internals.
Use this method to get data in bitmap form from the *wxClipboard* (p. 150).

wxBitmapDataObject::SetBitmap

virtual void SetBitmap(**const wxBitmap&** *bitmap*)

Sets the bitmap associated with the data object. This method is called when the data
object receives data. Usually there will be no reason to override this function.

wxBitmapHandler

Overview (p. 1788)

This is the base class for implementing bitmap file loading/saving, and bitmap creation
from data. It is used within wxBitmap and is not normally seen by the application.

If you wish to extend the capabilities of wxBitmap, derive a class from wxBitmapHandler
and add the handler using *wxBitmap::AddHandler* (p. 83) in your application initialisation.

Derived from

wxObject (p. 1012)

Include files

<wx/bitmap.h>

See also

wxBitmap (p. 79), *wxIcon* (p. 766), *wxCursor* (p. 226)

wxBitmapHandler::wxBitmapHandler

wxBitmapHandler()

Default constructor. In your own default constructor, initialise the members m_name, m_extension and m_type.

wxBitmapHandler::~wxBitmapHandler

~wxBitmapHandler()

Destroys the wxBitmapHandler object.

wxBitmapHandler::Create

virtual bool Create(**wxBitmap*** *bitmap*, **void*** *data*, **int** *type*, **int** *width*, **int** *height*, **int** *depth = -1*)

Creates a bitmap from the given data, which can be of arbitrary type. The wxBitmap object *bitmap* is manipulated by this function.

Parameters

bitmap

> The wxBitmap object.

width

> The width of the bitmap in pixels.

height

> The height of the bitmap in pixels.

depth

> The depth of the bitmap in pixels. If this is -1, the screen depth is used.

data

> Data whose type depends on the value of *type*.

type

> A bitmap type identifier - see *wxBitmapHandler::wxBitmapHandler* (p. 80) for a list of possible values.

Return value

true if the call succeeded, false otherwise (the default).

wxBitmapHandler::GetName

wxString GetName() **const**

Gets the name of this handler.

wxBitmapHandler::GetExtension

wxString GetExtension() **const**

Gets the file extension associated with this handler.

wxBitmapHandler::GetType

long GetType() **const**

Gets the bitmap type associated with this handler.

wxBitmapHandler::LoadFile

bool LoadFile(**wxBitmap*** *bitmap*, **const wxString&** *name*, **long** *type*)

Loads a bitmap from a file or resource, putting the resulting data into *bitmap*.

Parameters

bitmap

> The bitmap object which is to be affected by this operation.

name

> Either a filename or a Windows resource name. The meaning of *name* is determined by the *type* parameter.

type

> See *wxBitmap::wxBitmap* (p. 80) for values this can take.

Return value

true if the operation succeeded, false otherwise.

See also

wxBitmap::LoadFile (p. 88)
wxBitmap::SaveFile (p. 89)
wxBitmapHandler::SaveFile (p. 101)

wxBitmapHandler::SaveFile

bool SaveFile(**wxBitmap*** *bitmap*, **const wxString&** *name*, **int** *type*, **wxPalette*** *palette* = *NULL*)

Saves a bitmap in the named file.

Parameters

bitmap

> The bitmap object which is to be affected by this operation.

name

> A filename. The meaning of *name* is determined by the *type* parameter.

type

> See *wxBitmap::wxBitmap* (p. 80) for values this can take.

palette

> An optional palette used for saving the bitmap.

Return value

true if the operation succeeded, false otherwise.

See also

wxBitmap::LoadFile (p. 88)
wxBitmap::SaveFile (p. 89)
wxBitmapHandler::LoadFile (p. 100)

wxBitmapHandler::SetName

void SetName(**const wxString&** *name*)

Sets the handler name.

Parameters

name

> Handler name.

wxBitmapHandler::SetExtension

void SetExtension(**const wxString&** *extension*)

Sets the handler extension.

Parameters

extension

> Handler extension.

wxBitmapHandler::SetType

void SetType(**long** *type*)

Sets the handler type.

Parameters

name

> Handler type.

wxBoxSizer

The basic idea behind a box sizer is that windows will most often be laid out in rather simple basic geometry, typically in a row or a column or several hierarchies of either.

For more information, please see *Programming with wxBoxSizer* (p. 1774).

Derived from

wxSizer (p. 1176)
wxObject (p. 1012)

Include files

<wx/sizer.h>

See also

wxSizer (p. 1176), *Sizer overview* (p. 1771)

wxBoxSizer::wxBoxSizer

 wxBoxSizer(**int** *orient*)

Constructor for a wxBoxSizer. *orient* may be either of wxVERTICAL or wxHORIZONTAL for creating either a column sizer or a row sizer.

wxBoxSizer::RecalcSizes

void RecalcSizes()

Implements the calculation of a box sizer's dimensions and then sets the size of its children (calling *wxWindow::SetSize* (p. 1538) if the child is a window). It is used internally only and must not be called by the user. Documented for information.

wxBoxSizer::CalcMin

wxSize CalcMin()

Implements the calculation of a box sizer's minimal. It is used internally only and must not be called by the user. Documented for information.

wxBoxSizer::GetOrientation

int GetOrientation()

Returns the orientation of the box sizer, either wxVERTICAL or wxHORIZONTAL.

wxBrush

A brush is a drawing tool for filling in areas. It is used for painting the background of rectangles, ellipses, etc. It has a colour and a style.

Derived from

wxGDIObject (p. 607)
wxObject (p. 1012)

Include files

<wx/brush.h>

Predefined objects

Objects:

wxNullBrush

Pointers:

wxBLUE_BRUSH
wxGREEN_BRUSH
wxWHITE_BRUSH
wxBLACK_BRUSH
wxGREY_BRUSH
wxMEDIUM_GREY_BRUSH
wxLIGHT_GREY_BRUSH
wxTRANSPARENT_BRUSH
wxCYAN_BRUSH
wxRED_BRUSH

Remarks

On a monochrome display, wxWidgets shows all brushes as white unless the colour is really black.

Do not initialize objects on the stack before the program commences, since other required structures may not have been set up yet. Instead, define global pointers to objects and create them in *wxApp::OnInit* (p. 38) or when required.

An application may wish to create brushes with different characteristics dynamically, and there is the consequent danger that a large number of duplicate brushes will be created. Therefore an application may wish to get a pointer to a brush by using the global list of brushes **wxTheBrushList**, and calling the member function **FindOrCreateBrush**.

wxBrush uses a reference counting system, so assignments between brushes are very cheap. You can therefore use actual wxBrush objects instead of pointers without efficiency problems. Once one wxBrush object changes its data it will create its own brush data internally so that other brushes, which previously shared the data using the reference counting, are not affected.

See also

wxBrushList (p. 109), *wxDC* (p. 365), *wxDC::SetBrush* (p. 382)

wxBrush::wxBrush

wxBrush()

Default constructor. The brush will be uninitialised, and *wxBrush::Ok* (p. 107) will return false.

wxBrush(**const wxColour&** *colour*, **int** *style* = `wxSOLID`)

Constructs a brush from a colour object and style.

wxBrush(**const wxString&** *colourName*, **int** *style*)

Constructs a brush from a colour name and style.

wxBrush(**const wxBitmap&** *stippleBitmap*)

Constructs a stippled brush using a bitmap.

wxBrush(**const wxBrush&** *brush*)

Copy constructor. This uses reference counting so is a cheap operation.

Parameters

colour

>   Colour object.

colourName

> Colour name. The name will be looked up in the colour database.

style

> One of:

wxTRANSPARENT	Transparent (no fill).
wxSOLID	Solid.
wxSTIPPLE	Uses a bitmap as a stipple.
wxBDIAGONAL_HATCH	Backward diagonal hatch.
wxCROSSDIAG_HATCH	Cross-diagonal hatch.
wxFDIAGONAL_HATCH	Forward diagonal hatch.
wxCROSS_HATCH	Cross hatch.
wxHORIZONTAL_HATCH	Horizontal hatch.
wxVERTICAL_HATCH	Vertical hatch.

brush

> Pointer or reference to a brush to copy.

stippleBitmap

> A bitmap to use for stippling.

Remarks

If a stipple brush is created, the brush style will be set to wxSTIPPLE.

See also

wxBrushList (p. 109), *wxColour* (p. 164), *wxColourDatabase* (p. 169)

wxBrush::~wxBrush

~wxBrush()

Destructor.

Remarks

The destructor may not delete the underlying brush object of the native windowing system, since wxBrush uses a reference counting system for efficiency.

Although all remaining brushes are deleted when the application exits, the application

should try to clean up all brushes itself. This is because wxWidgets cannot know if a pointer to the brush object is stored in an application data structure, and there is a risk of double deletion.

wxBrush::GetColour

wxColour& GetColour() **const**

Returns a reference to the brush colour.

See also

wxBrush::SetColour (p. 107)

wxBrush::GetStipple

wxBitmap * GetStipple() **const**

Gets a pointer to the stipple bitmap. If the brush does not have a wxSTIPPLE style, this bitmap may be non-NULL but uninitialised (*wxBitmap::Ok* (p. 88) returns false).

See also

wxBrush::SetStipple (p. 107)

wxBrush::GetStyle

int GetStyle() **const**

Returns the brush style, one of:

wxTRANSPARENT	Transparent (no fill).
wxSOLID	Solid.
wxBDIAGONAL_HATCH	Backward diagonal hatch.
wxCROSSDIAG_HATCH	Cross-diagonal hatch.
wxFDIAGONAL_HATCH	Forward diagonal hatch.
wxCROSS_HATCH	Cross hatch.
wxHORIZONTAL_HATCH	Horizontal hatch.
wxVERTICAL_HATCH	Vertical hatch.
wxSTIPPLE	Stippled using a bitmap.
wxSTIPPLE_MASK_OPAQUE	Stippled using a bitmap's mask.

See also

wxBrush::SetStyle (p. 108), *wxBrush::SetColour* (p. 107), *wxBrush::SetStipple* (p. 107)

wxBrush::IsHatch

bool IsHatch() **const**

Returns true if the style of the brush is any of hatched fills.

See also

wxBrush::GetStyle (p. 106)

wxBrush::Ok

bool Ok() **const**

Returns true if the brush is initialised. It will return false if the default constructor has been used (for example, the brush is a member of a class, or NULL has been assigned to it).

wxBrush::SetColour

void SetColour(**wxColour&** *colour*)

Sets the brush colour using a reference to a colour object.

void SetColour(**const wxString&** *colourName*)

Sets the brush colour using a colour name from the colour database.

void SetColour(**const unsigned char** *red*, **const unsigned char** *green*, **const unsigned char** *blue*)

Sets the brush colour using red, green and blue values.

See also

wxBrush::GetColour (p. 106)

wxBrush::SetStipple

void SetStipple(**const wxBitmap&** *bitmap*)

Sets the stipple bitmap.

Parameters

bitmap

The bitmap to use for stippling.

Remarks

The style will be set to wxSTIPPLE, unless the bitmap has a mask associated to it, in which case the style will be set to wxSTIPPLE_MASK_OPAQUE.

If the wxSTIPPLE variant is used, the bitmap will be used to fill out the area to be drawn. If the wxSTIPPLE_MASK_OPAQUE is used, the current text foreground and text background determine what colours are used for displaying and the bits in the mask (which is a mono-bitmap actually) determine where to draw what.

Note that under Windows 95, only 8x8 pixel large stipple bitmaps are supported, Windows 98 and NT as well as GTK support arbitrary bitmaps.

See also

wxBitmap (p. 79)

wxBrush::SetStyle

void SetStyle(**int** *style*)

Sets the brush style.

style

One of:

wxTRANSPARENT	Transparent (no fill).
wxSOLID	Solid.
wxBDIAGONAL_HATCH	Backward diagonal hatch.
wxCROSSDIAG_HATCH	Cross-diagonal hatch.
wxFDIAGONAL_HATCH	Forward diagonal hatch.
wxCROSS_HATCH	Cross hatch.
wxHORIZONTAL_HATCH	Horizontal hatch.
wxVERTICAL_HATCH	Vertical hatch.
wxSTIPPLE	Stippled using a bitmap.
wxSTIPPLE_MASK_OPAQUE	Stippled using a bitmap's mask.

See also

wxBrush::GetStyle (p. 106)

wxBrush::operator =

wxBrush& operator =(const wxBrush& *brush*)

Assignment operator, using reference counting. Returns a reference to 'this'.

wxBrush::operator ==

bool operator ==(const wxBrush& *brush*)

Equality operator. Two brushes are equal if they contain pointers to the same underlying brush data. It does not compare each attribute, so two independently-created brushes using the same parameters will fail the test.

wxBrush::operator !=

bool operator !=(const wxBrush& *brush*)

Inequality operator. Two brushes are not equal if they contain pointers to different underlying brush data. It does not compare each attribute.

wxBrushList

A brush list is a list containing all brushes which have been created.

Derived from

wxList (p. 840)
wxObject (p. 1012)

Include files

<wx/gdicmn.h>

Remarks

There is only one instance of this class: **wxTheBrushList**. Use this object to search for a previously created brush of the desired type and create it if not already found. In some windowing systems, the brush may be a scarce resource, so it can pay to reuse old resources if possible. When an application finishes, all brushes will be deleted and their resources freed, eliminating the possibility of 'memory leaks'. However, it is best not to rely on this automatic cleanup because it can lead to double deletion in some circumstances.

There are two mechanisms in recent versions of wxWidgets which make the brush list less useful than it once was. Under Windows, scarce resources are cleaned up internally if they are not being used. Also, a reference counting mechanism applied to all GDI objects means that some sharing of underlying resources is possible. You don't have to keep track of pointers, working out when it is safe delete a brush, because the reference

counting does it for you. For example, you can set a brush in a device context, and then immediately delete the brush you passed, because the brush is 'copied'.

So you may find it easier to ignore the brush list, and instead create and copy brushes as you see fit. If your Windows resource meter suggests your application is using too many resources, you can resort to using GDI lists to share objects explicitly.

The only compelling use for the brush list is for wxWidgets to keep track of brushes in order to clean them up on exit. It is also kept for backward compatibility with earlier versions of wxWidgets.

See also

wxBrush (p. 103)

wxBrushList::wxBrushList

void wxBrushList()

Constructor. The application should not construct its own brush list: use the object pointer **wxTheBrushList**.

wxBrushList::AddBrush

void AddBrush(**wxBrush ****brush*)

Used internally by wxWidgets to add a brush to the list.

wxBrushList::FindOrCreateBrush

wxBrush * FindOrCreateBrush(**const wxColour&** *colour*, **int** *style = wxSOLID*)

Finds a brush with the specified attributes and returns it, else creates a new brush, adds it to the brush list, and returns it.

Parameters

colour

Colour object.

style

Brush style. See *wxBrush::SetStyle* (p. 108) for a list of styles.

wxBrushList::RemoveBrush

void RemoveBrush(**wxBrush** *brush)

Used by wxWidgets to remove a brush from the list.

wxBufferedDC

This simple class provides a simple way to avoid flicker: when drawing on it, everything is in fact first drawn on an in-memory buffer (a *wxBitmap* (p. 79)) and then copied to the screen only once, when this object is destroyed.

It can be used in the same way as any other device context. wxBufferedDC itself typically replaces *wxClientDC* (p. 147), if you want to use it in your `OnPaint()` handler, you should look at*wxBufferedPaintDC* (p. 112).

Derived from

wxMemoryDC (p. 935)
wxDC (p. 365)
wxObject (p. 1012)

Include files

<wx/dcbuffer.h>

See also

wxDC (p. 365)

wxBufferedDC::wxBufferedDC

wxBufferedDC()

wxBufferedDC(**wxDC** *dc, **const wxSize&** *area*, **int** *style* = *wxBUFFER_CLIENT_AREA*)

wxBufferedDC(**wxDC** *dc, **const wxBitmap&** *buffer*, **int** *style* = *wxBUFFER_CLIENT_AREA*)

If you use the first, default, constructor, you must call one of the *Init* (p. 112) methods later in order to use the object.

The other constructors initialize the object immediately and `Init()` must not be called after using them.

Parameters

dc

The underlying DC: everything drawn to this object will be flushed to this DC when

this object is destroyed. You may pass NULL in order to just initialize the buffer, and not flush it.

area

The size of the bitmap to be used for buffering (this bitmap is created internally when it is not given explicitly).

buffer

Explicitly provided bitmap to be used for buffering: this is the most efficient solution as the bitmap doesn't have to be recreated each time but it also requires more memory as the bitmap is never freed. The bitmap should have appropriate size, anything drawn outside of its bounds is clipped.

style

wxBUFFER_CLIENT_AREA to indicate that just the client area of the window is buffered, or wxBUFFER_VIRTUAL_AREA to indicate that the buffer bitmap covers the virtual area (in which case PrepareDC is automatically called for the actual window device context).

wxBufferedDC::Init

void Init(**wxDC ***dc*, **const wxSize&** *area*, **int** *style = wxBUFFER_CLIENT_AREA*)

void Init(**wxDC ***dc*, **const wxBitmap&** *buffer*, **int** *style = wxBUFFER_CLIENT_AREA*)

These functions initialize the object created using the default constructor. Please see *constructors documentation* (p. 111) for details.

wxBufferedDC::~wxBufferedDC

Copies everything drawn on the DC so far to the underlying DC associated with this object, if any.

wxBufferedPaintDC

This is a subclass of *wxBufferedDC* (p. 111) which can be used inside of an `OnPaint()` event handler. Just create an object of this class instead of *wxPaintDC* (p. 1024) and that's all you have to do to (mostly) avoid flicker. The only thing to watch out for is that if you are using this class together with *wxScrolledWindow* (p. 1150), you probably do **not** want to call *PrepareDC* (p. 1157) on it as it already does this internally for the real underlying wxPaintDC.

Derived from

wxMemoryDC (p. 935)
wxDC (p. 365)

wxObject (p. 1012)

Include files

<wx/dcbuffer.h>

wxBufferedPaintDC::wxBufferedPaintDC

 wxBufferedPaintDC(**wxWindow** *window*, **const wxBitmap&** *buffer*, **int** *style* = *wxBUFFER_CLIENT_AREA*)

 wxBufferedPaintDC(**wxWindow** *window*, **int** *style* = *wxBUFFER_CLIENT_AREA*)

As with *wxBufferedDC* (p. 111), you may either provide the bitmap to be used for buffering or let this object create one internally (in the latter case, the size of the client part of the window is used).

Pass wxBUFFER_CLIENT_AREA for the *style* parameter to indicate that just the client area of the window is buffered, or wxBUFFER_VIRTUAL_AREA to indicate that the buffer bitmap covers the virtual area (in which case PrepareDC is automatically called for the actual window device context).

wxBufferedPaintDC::~wxBufferedPaintDC

Copies everything drawn on the DC so far to the window associated with this object, using a *wxPaintDC* (p. 1024).

wxBufferedInputStream

This stream acts as a cache. It caches the bytes read from the specified input stream (See *wxFilterInputStream* (p. 549)). It uses wxStreamBuffer and sets the default in-buffer size to 1024 bytes. This class may not be used without some other stream to read the data from (such as a file stream or a memory stream).

Derived from

wxFilterInputStream (p. 549)

Include files

<wx/stream.h>

See also

wxStreamBuffer (p. 1277), *wxInputStream* (p. 814),*wxBufferedOutputStream* (p. 114)

wxBufferedOutputStream

This stream acts as a cache. It caches the bytes to be written to the specified output stream (See *wxFilterOutputStream* (p. 549)). The data is only written when the cache is full, when the buffered stream is destroyed or when calling SeekO().

This class may not be used without some other stream to write the data to (such as a file stream or a memory stream).

Derived from

wxFilterOutputStream (p. 549)

Include files

<wx/stream.h>

See also

wxStreamBuffer (p. 1277), *wxOutputStream* (p. 1016)

wxBufferedOutputStream::wxBufferedOutputStream

wxBufferedOutputStream(**const wxOutputStream&** *parent*)

Creates a buffered stream using a buffer of a default size of 1024 bytes for cashing the stream *parent*.

wxBufferedOutputStream::~wxBufferedOutputStream

~wxBufferedOutputStream()

Destructor. Calls Sync() and destroys the internal buffer.

wxBufferedOutputStream::SeekO

off_t SeekO(**off_t** *pos*, **wxSeekMode** *mode*)

Calls Sync() and changes the stream position.

wxBufferedOutputStream::Sync

void Sync()

Flushes the buffer and calls Sync() on the parent stream.

wxBusyCursor

This class makes it easy to tell your user that the program is temporarily busy. Just create a wxBusyCursor object on the stack, and within the current scope, the hourglass will be shown.

For example:

```
wxBusyCursor wait;

for (int i = 0; i < 100000; i++)
  DoACalculation();
```

It works by calling *wxBeginBusyCursor* (p. 1614) in the constructor, and *wxEndBusyCursor* (p. 1616) in the destructor.

Derived from

None

Include files

<wx/utils.h>

See also

wxBeginBusyCursor (p. 1614), *wxEndBusyCursor* (p. 1616), *wxWindowDisabler* (p. 1548)

wxBusyCursor::wxBusyCursor

wxBusyCursor(**wxCursor*** *cursor = wxHOURGLASS_CURSOR*)

Constructs a busy cursor object, calling *wxBeginBusyCursor* (p. 1614).

wxBusyCursor::~wxBusyCursor

~wxBusyCursor()

Destroys the busy cursor object, calling *wxEndBusyCursor* (p. 1616).

wxBusyInfo

This class makes it easy to tell your user that the program is temporarily busy. Just create a wxBusyInfo object on the stack, and within the current scope, a message window will be shown.

For example:

```
wxBusyInfo wait("Please wait, working...");

for (int i = 0; i < 100000; i++)
{
    DoACalculation();
}
```

It works by creating a window in the constructor, and deleting it in the destructor.

You may also want to call wxTheApp->Yield() to refresh the window periodically (in case it had been obscured by other windows, for example) like this:

```
wxWindowDisabler disableAll;

wxBusyInfo wait("Please wait, working...");

for (int i = 0; i < 100000; i++)
{
    DoACalculation();

    if ( !(i % 1000) )
        wxTheApp->Yield();
}
```

but take care to not cause undesirable reentrancies when doing it (see *wxApp::Yield()* (p. 42) for more details). The simplest way to do it is to use *wxWindowDisabler* (p. 1548) class as illustrated in the above example.

Derived from

None

Include files

<wx/busyinfo.h>

wxBusyInfo::wxBusyInfo

wxBusyInfo(**const wxString&** *msg*, **wxWindow*** *parent* = *NULL*)

Constructs a busy info window as child of *parent* and displays *msg*in it.

NB: If *parent* is not NULL you must ensure that it is not closed while the busy info is shown.

wxBusyInfo::~wxBusyInfo

~wxBusyInfo()

Hides and closes the window containing the information text.

wxButton

A button is a control that contains a text string, and is one of the most common elements of a GUI. It may be placed on a *dialog box* (p. 406) or *panel* (p. 1030), or indeed almost any other window.

Derived from

wxControl (p. 215)
wxWindow (p. 1490)
wxEvtHandler (p. 486)
wxObject (p. 1012)

Include files

<wx/button.h>

Window styles

wxBU_LEFT	Left-justifies the label. Windows and GTK+ only.
wxBU_TOP	Aligns the label to the top of the button. Windows and GTK+ only.
wxBU_RIGHT	Right-justifies the bitmap label. Windows and GTK+ only.
wxBU_BOTTOM	Aligns the label to the bottom of the button. Windows and GTK+ only.
wxBU_EXACTFIT	Creates the button as small as possible instead of making it of the standard size (which is the default behaviour).
wxNO_BORDER	Creates a flat button. Windows and GTK+ only.

See also *window styles overview* (p. 1762).

Event handling

EVT_BUTTON(id, func)	Process a wxEVT_COMMAND_BUTTON_CLICKED event, when the button is clicked.

See also

wxBitmapButton (p. 92)

wxButton::wxButton

wxButton()

Default constructor.

 wxButton(**wxWindow*** *parent*, **wxWindowID** *id*, **const wxString&** *label* = *wxEmptyString*, **const wxPoint&** *pos* = *wxDefaultPosition*, **const wxSize&** *size* = *wxDefaultSize*, **long** *style = 0*, **const wxValidator&** *validator = wxDefaultValidator*, **const wxString&** *name = "button"*)

Constructor, creating and showing a button.

The preferred way to create standard buttons is to use default value of*label*. If no label is supplied and *id* is one of standard IDs from*this list* (p. 1679), standard label will be used. In addition to that, the button will be decorated with stock icons under GTK+ 2.

Parameters

parent

> Parent window. Must not be NULL.

id

> Button identifier. A value of `wxID_ANY` indicates a default value.

label

> Text to be displayed on the button.

pos

> Button position.

size

> Button size. If the default size is specified then the button is sized appropriately for the text.

style

> Window style. See *wxButton* (p. 117).

validator

> Window validator.

name

> Window name.

See also

wxButton::Create (p. 119), *wxValidator* (p. 1461)

wxButton::~wxButton

 ~wxButton()

Destructor, destroying the button.

wxButton::Create

bool Create(**wxWindow*** *parent*, **wxWindowID** *id*, **const wxString&** *label* = *wxEmptyString*, **const wxPoint&** *pos* = *wxDefaultPosition*, **const wxSize&** *size* = *wxDefaultSize*, **long** *style = 0*, **const wxValidator&** *validator*, **const wxString&** *name* = *"button"*)

Button creation function for two-step creation. For more details, see*wxButton::wxButton* (p. 117).

wxButton::GetLabel

wxString GetLabel() **const**

Returns the string label for the button.

Return value

The button's label.

See also

wxButton::SetLabel (p. 120)

wxButton::GetDefaultSize

wxSize GetDefaultSize()

Returns the default size for the buttons. It is advised to make all the dialog buttons of the same size and this function allows to retrieve the (platform and current font dependent size) which should be the best suited for this.

wxButton::SetDefault

void SetDefault()

This sets the button to be the default item for the panel or dialog box.

Remarks

Under Windows, only dialog box buttons respond to this function. As normal under Windows and Motif, pressing return causes the default button to be depressed when the return key is pressed. See also *wxWindow::SetFocus* (p. 1533) which sets the keyboard focus for windows and text panel items, and *wxPanel::SetDefaultItem* (p. 1033).

Note that under Motif, calling this function immediately after creation of a button and before the creation of other buttons will cause misalignment of the row of buttons, since

default buttons are larger. To get around this, call *SetDefault* after you have created a row of buttons: wxWidgets will then set the size of all buttons currently on the panel to the same size.

wxButton::SetLabel

void SetLabel(**const wxString&** *label*)

Sets the string label for the button.

Parameters

label

      The label to set.

See also

wxButton::GetLabel (p. 119)

wxCalculateLayoutEvent

This event is sent by *wxLayoutAlgorithm* (p. 835) to calculate the amount of the remaining client area that the window should occupy.

Derived from

wxEvent (p. 483)
wxObject (p. 1012)

Include files

<wx/laywin.h>

Event table macros

EVT_CALCULATE_LAYOUT(func)	Process a wxEVT_CALCULATE_LAYOUT event, which asks the window to take a 'bite' out of a rectangle provided by the algorithm.

See also

wxQueryLayoutInfoEvent (p. 1087), *wxSashLayoutWindow* (p. 1129), *wxLayoutAlgorithm* (p. 835).

wxCalculateLayoutEvent::wxCalculateLayoutEvent

wxCalculateLayoutEvent(**wxWindowID** *id = 0*)

Constructor.

wxCalculateLayoutEvent::GetFlags

int GetFlags() **const**

Returns the flags associated with this event. Not currently used.

wxCalculateLayoutEvent::GetRect

wxRect GetRect() **const**

Before the event handler is entered, returns the remaining parent client area that the window could occupy. When the event handler returns, this should contain the remaining parent client rectangle, after the event handler has subtracted the area that its window occupies.

wxCalculateLayoutEvent::SetFlags

void SetFlags(**int** *flags*)

Sets the flags associated with this event. Not currently used.

wxCalculateLayoutEvent::SetRect

void SetRect(**const wxRect&** *rect*)

Call this to specify the new remaining parent client area, after the space occupied by the window has been subtracted.

wxCalendarCtrl

The calendar control allows the user to pick a date. For this, it displays a window containing several parts: a control at the top to pick the month and the year (either or both of them may be disabled), and a month area below them which shows all the days in the month. The user can move the current selection using the keyboard and select the date (generating EVT_CALENDAR event) by pressing <Return> or double clicking it.

It has advanced possibilities for the customization of its display. All global settings (such as colours and fonts used) can, of course, be changed. But also, the display style for each day in the month can be set independently using *wxCalendarDateAttr* (p. 127) class.

An item without custom attributes is drawn with the default colours and font and without border, but setting custom attributes with *SetAttr* (p. 126) allows to modify its appearance. Just create a custom attribute object and set it for the day you want to be

displayed specially (note that the control will take ownership of the pointer, i.e. it will delete it itself). A day may be marked as being a holiday, even if it is not recognized as one by *wxDateTime* (p. 1730) using *SetHoliday* (p. 128) method.

As the attributes are specified for each day, they may change when the month is changed, so you will often want to update them in EVT_CALENDAR_MONTH event handler.

Derived from

wxControl (p. 215)
wxWindow (p. 1490)
wxEvtHandler (p. 486)
wxObject (p. 1012)

Include files

<wx/calctrl.h>

Window styles

wxCAL_SUNDAY_FIRST   Show Sunday as the first day in the week

wxCAL_MONDAY_FIRST   Show Monday as the first day in the week

wxCAL_SHOW_HOLIDAYS Highlight holidays in the calendar

wxCAL_NO_YEAR_CHANGE   Disable the year changing

wxCAL_NO_MONTH_CHANGE   Disable the month (and, implicitly, the year) changing

wxCAL_SHOW_SURROUNDING_WEEKS Show the neighbouring weeks in the previous and next months

wxCAL_SEQUENTIAL_MONTH_SELECTION   Use alternative, more compact, style for the month and year selection controls.

The default calendar style is wxCAL_SHOW_HOLIDAYS.

Event table macros

To process input from a calendar control, use these event handler macros to direct input to member functions that take a *wxCalendarEvent* (p. 130) argument.

EVT_CALENDAR(id, func)   A day was double clicked in the calendar.

EVT_CALENDAR_SEL_CHANGED(id, func)   The selected date changed.

EVT_CALENDAR_DAY(id, func)   The selected day changed.

EVT_CALENDAR_MONTH(id, func)   The selected month changed.

EVT_CALENDAR_YEAR(id, func)   The selected year changed.

EVT_CALENDAR_WEEKDAY_CLICKED(id, func)　　　User clicked on the week day
header

Note that changing the selected date will result in either of EVT_CALENDAR_DAY, MONTH
or YEAR events and EVT_CALENDAR_SEL_CHANGED one.

Constants

The following are the possible return values for *HitTest* (p. 127) method:

```
enum wxCalendarHitTestResult
{
    wxCAL_HITTEST_NOWHERE,      // outside of anything
    wxCAL_HITTEST_HEADER,       // on the header (weekdays)
    wxCAL_HITTEST_DAY           // on a day in the calendar
}
```

See also

Calendar sample (p. 1708)
wxCalendarDateAttr (p. 127)
wxCalendarEvent (p. 130)

wxCalendarCtrl::wxCalendarCtrl

wxCalendarCtrl()

Default constructor, use *Create* (p. 123) after it.

wxCalendarCtrl(**wxWindow*** *parent*, **wxWindowID** *id*, **const wxDateTime&** *date* =
wxDefaultDateTime, **const wxPoint&** *pos* = *wxDefaultPosition*, **const wxSize&** *size* =
wxDefaultSize, **long** *style* = *wxCAL_SHOW_HOLIDAYS*, **const wxString&** *name* =
wxCalendarNameStr)

Does the same as *Create* (p. 123) method.

wxCalendarCtrl::Create

bool Create(**wxWindow*** *parent*, **wxWindowID** *id*, **const wxDateTime&** *date* =
wxDefaultDateTime, **const wxPoint&** *pos* = *wxDefaultPosition*, **const wxSize&** *size* =
wxDefaultSize, **long** *style* = *wxCAL_SHOW_HOLIDAYS*, **const wxString&** *name* =
wxCalendarNameStr)

Creates the control. See *wxWindow* (p. 1492) for the meaning of the parameters and the
control overview for the possible styles.

wxCalendarCtrl::~wxCalendarCtrl

 ~wxCalendarCtrl()

Destroys the control.

wxCalendarCtrl::SetDate

void SetDate(**const wxDateTime&** *date*)

Sets the current date.

wxCalendarCtrl::GetDate

const wxDateTime& GetDate() **const**

Gets the currently selected date.

wxCalendarCtrl::EnableYearChange

void EnableYearChange(**bool** *enable = true*)

This function should be used instead of changing `wxCAL_NO_YEAR_CHANGE` style bit directly. It allows or disallows the user to change the year interactively.

wxCalendarCtrl::EnableMonthChange

void EnableMonthChange(**bool** *enable = true*)

This function should be used instead of changing `wxCAL_NO_MONTH_CHANGE` style bit. It allows or disallows the user to change the month interactively. Note that if the month can not be changed, the year can not be changed neither.

wxCalendarCtrl::EnableHolidayDisplay

void EnableHolidayDisplay(**bool** *display = true*)

This function should be used instead of changing `wxCAL_SHOW_HOLIDAYS` style bit directly. It enables or disables the special highlighting of the holidays.

wxCalendarCtrl::SetHeaderColours

void SetHeaderColours(**const wxColour&** *colFg*, **const wxColour&** *colBg*)

Set the colours used for painting the weekdays at the top of the control.

wxCalendarCtrl::GetHeaderColourFg

const wxColour& GetHeaderColourFg() **const**

Gets the foreground colour of the header part of the calendar window.

See also

SetHeaderColours (p. 124)

wxCalendarCtrl::GetHeaderColourBg

const wxColour& GetHeaderColourBg() **const**

Gets the background colour of the header part of the calendar window.

See also

SetHeaderColours (p. 124)

wxCalendarCtrl::SetHighlightColours

void SetHighlightColours(**const wxColour&** *colFg*, **const wxColour&** *colBg*)

Set the colours to be used for highlighting the currently selected date.

wxCalendarCtrl::GetHighlightColourFg

const wxColour& GetHighlightColourFg() **const**

Gets the foreground highlight colour.

See also

SetHighlightColours (p. 125)

wxCalendarCtrl::GetHighlightColourBg

const wxColour& GetHighlightColourBg() **const**

Gets the background highlight colour.

See also

SetHighlightColours (p. 125)

wxCalendarCtrl::SetHolidayColours

void SetHolidayColours(**const wxColour&** *colFg*, **const wxColour&** *colBg*)

Sets the colours to be used for the holidays highlighting (only used if the window style includes `wxCAL_SHOW_HOLIDAYS` flag).

wxCalendarCtrl::GetHolidayColourFg

const wxColour& GetHolidayColourFg() **const**

Return the foreground colour currently used for holiday highlighting.

See also

SetHolidayColours (p. 125)

wxCalendarCtrl::GetHolidayColourBg

const wxColour& GetHolidayColourBg() **const**

Return the background colour currently used for holiday highlighting.

See also

SetHolidayColours (p. 125)

wxCalendarCtrl::GetAttr

wxCalendarDateAttr * GetAttr(**size_t** *day*) **const**

Returns the attribute for the given date (should be in the range 1...31).

The returned pointer may be `NULL`.

wxCalendarCtrl::SetAttr

void SetAttr(**size_t** *day*, **wxCalendarDateAttr*** *attr*)

Associates the attribute with the specified date (in the range 1...31).

If the pointer is `NULL`, the items attribute is cleared.

wxCalendarCtrl::SetHoliday

void SetHoliday(**size_t** *day*)

Marks the specified day as being a holiday in the current month.

wxCalendarCtrl::ResetAttr

void ResetAttr(**size_t** *day*)

Clears any attributes associated with the given day (in the range1...31).

wxCalendarCtrl::HitTest

wxCalendarHitTestResult HitTest(**const wxPoint&** *pos*, **wxDateTime*** *date* = *NULL*, **wxDateTime::WeekDay*** *wd* = *NULL*)

Returns one of `wxCAL_HITTEST_XXX` *constants* (p. 121) and fills either *date* or *wd* pointer with the corresponding value depending on the hit test code.

wxCalendarDateAttr

wxCalendarDateAttr is a custom attributes for a calendar date. The objects of this class are used with *wxCalendarCtrl* (p. 121).

Derived from

No base class

Constants

Here are the possible kinds of borders which may be used to decorate a date:

```
enum wxCalendarDateBorder
{
    wxCAL_BORDER_NONE,          // no border (default)
    wxCAL_BORDER_SQUARE,        // a rectangular border
    wxCAL_BORDER_ROUND          // a round border
}
```

See also

wxCalendarCtrl (p. 121)

Include files

<wx/calctrl.h>

wxCalendarDateAttr::wxCalendarDateAttr

wxCalendarDateAttr()

wxCalendarDateAttr(**const wxColour&** *colText*, **const wxColour&** *colBack* = *wxNullColour*, **const wxColour&** *colBorder* = *wxNullColour*, **const wxFont&** *font* = *wxNullFont*, **wxCalendarDateBorder** *border* = *wxCAL_BORDER_NONE*)

 wxCalendarDateAttr(**wxCalendarDateBorder** *border*, **const wxColour&** *colBorder =* *wxNullColour*)

The constructors.

wxCalendarDateAttr::SetTextColour

void SetTextColour(**const wxColour&** *colText*)

Sets the text (foreground) colour to use.

wxCalendarDateAttr::SetBackgroundColour

void SetBackgroundColour(**const wxColour&** *colBack*)

Sets the text background colour to use.

wxCalendarDateAttr::SetBorderColour

void SetBorderColour(**const wxColour&** *col*)

Sets the border colour to use.

wxCalendarDateAttr::SetFont

void SetFont(**const wxFont&** *font*)

Sets the font to use.

wxCalendarDateAttr::SetBorder

void SetBorder(**wxCalendarDateBorder** *border*)

Sets the *border kind* (p. 127)

wxCalendarDateAttr::SetHoliday

void SetHoliday(**bool** *holiday*)

Display the date with this attribute as a holiday.

wxCalendarDateAttr::HasTextColour

bool HasTextColour() **const**

Returns `true` if this item has a non default text foreground colour.

wxCalendarDateAttr::HasBackgroundColour

bool HasBackgroundColour() **const**

Returns `true` if this attribute specifies a non default text background colour.

wxCalendarDateAttr::HasBorderColour

bool HasBorderColour() **const**

Returns `true` if this attribute specifies a non default border colour.

wxCalendarDateAttr::HasFont

bool HasFont() **const**

Returns `true` if this attribute specifies a non default font.

wxCalendarDateAttr::HasBorder

bool HasBorder() **const**

Returns `true` if this attribute specifies a non default (i.e. any) border.

wxCalendarDateAttr::IsHoliday

bool IsHoliday() **const**

Returns `true` if this attribute specifies that this item should be displayed as a holiday.

wxCalendarDateAttr::GetTextColour

const wxColour& GetTextColour() **const**

Returns the text colour to use for the item with this attribute.

wxCalendarDateAttr::GetBackgroundColour

const wxColour& GetBackgroundColour() **const**

Returns the background colour to use for the item with this attribute.

wxCalendarDateAttr::GetBorderColour

const wxColour& GetBorderColour() **const**

Returns the border colour to use for the item with this attribute.

wxCalendarDateAttr::GetFont

const wxFont& GetFont() **const**

Returns the font to use for the item with this attribute.

wxCalendarDateAttr::GetBorder

wxCalendarDateBorder GetBorder() **const**

Returns the *border* (p. 127) to use for the item with this attribute.

wxCalendarEvent

The wxCalendarEvent class is used together with *wxCalendarCtrl* (p. 121).

Derived from

wxDateEvent (p. 248)
wxCommandEvent (p. 180)
wxEvent (p. 483)
wxObject (p. 1012)

Include files

<wx/calctrl.h>

See also

wxCalendarCtrl (p. 121)

wxCalendarEvent::GetWeekDay

wxDateTime::WeekDay GetWeekDay() **const**

Returns the week day on which the user clicked in `EVT_CALENDAR_WEEKDAY_CLICKED` handler. It doesn't make sense to call this function in other handlers.

wxCalendarEvent::SetWeekDay

void SetWeekDay(**wxDateTime::WeekDay** *day*)

Sets the week day carried by the event, normally only used by the library internally.

wxCaret

A caret is a blinking cursor showing the position where the typed text will appear. The text controls usually have a caret but wxCaret class also allows to use a caret in other windows.

Currently, the caret appears as a rectangle of the given size. In the future, it will be possible to specify a bitmap to be used for the caret shape.

A caret is always associated with a window and the current caret can be retrieved using *wxWindow::GetCaret* (p. 1504). The same caret can't be reused in two different windows.

Derived from

No base class

Include files

<wx/caret.h>

Data structures

wxCaret::wxCaret

wxCaret()

Default constructor: you must use one of Create() functions later.

wxCaret(**wxWindow*** *window*, **int** *width*, **int** *height*)

wxCaret(**wxWindowBase*** *window*, **const wxSize&** *size*)

Create the caret of given (in pixels) width and height and associates it with the given window.

wxCaret::Create

bool Create(**wxWindowBase*** *window*, **int** *width*, **int** *height*)

bool Create(**wxWindowBase*** *window*, **const wxSize&** *size*)

Create the caret of given (in pixels) width and height and associates it with the given window (same as constructor).

wxCaret::GetBlinkTime

static int GetBlinkTime()

Returns the blink time which is measured in milliseconds and is the time elapsed between 2 inversions of the caret (blink time of the caret is the same for all carets, so this functions is static).

wxCaret::GetPosition

void GetPosition(**int*** *x*, **int*** *y*) **const**

wxPoint GetPosition() **const**

Get the caret position (in pixels).

wxPerl note: In wxPerl there are two methods instead of a single overloaded method:

GetPosition()	Returns a Wx::Point
GetPositionXY()	Returns a 2-element list (`x, y`)

wxCaret::GetSize

void GetSize(**int*** *width*, **int*** *height*) **const**

wxSize GetSize() **const**

Get the caret size.

wxPerl note: In wxPerl there are two methods instead of a single overloaded method:

GetSize()	Returns a Wx::Size
GetSizeWH()	Returns a 2-element list (`width, height`)

wxCaret::GetWindow

wxWindow* GetWindow() **const**

Get the window the caret is associated with.

wxCaret::Hide

void Hide()

Same as *wxCaret::Show(false)* (p. 134).

wxCaret::IsOk

bool IsOk() **const**

Returns true if the caret was created successfully.

wxCaret::IsVisible

bool IsVisible() **const**

Returns true if the caret is visible and false if it is permanently hidden (if it is is blinking and not shown currently but will be after the next blink, this method still returns true).

wxCaret::Move

void Move(**int** *x*, **int** *y*)

void Move(**const wxPoint&** *pt*)

Move the caret to given position (in logical coordinates).

wxCaret::SetBlinkTime

static void SetBlinkTime(**int** *milliseconds*)

Sets the blink time for all the carets.

Remarks

Under Windows, this function will change the blink time for **all** carets permanently (until the next time it is called), even for the carets in other applications.

See also

GetBlinkTime (p. 132)

wxCaret::SetSize

void SetSize(**int** *width*, **int** *height*)

void SetSize(**const wxSize&** *size*)

Changes the size of the caret.

wxCaret::Show

void Show(**bool** *show = true*)

Shows or hides the caret. Notice that if the caret was hidden N times, it must be shown N times as well to reappear on the screen.

wxCheckBox

A checkbox is a labelled box which by default is either on (checkmark is visible) or off (no checkmark). Optionally (when the wxCHK_3STATE style flag is set) it can have a third state, called the mixed or undetermined state. Often this is used as a "Does Not Apply" state.

Derived from

wxControl (p. 215)
wxWindow (p. 1490)
wxEvtHandler (p. 486)
wxObject (p. 1012)

Include files

<wx/checkbox.h>

Window styles

wxCHK_2STATE	Create a 2-state checkbox. This is the default.
wxCHK_3STATE	Create a 3-state checkbox. Not implemented in wxMGL, wxOS2 and wxGTK built against GTK+ 1.2.
wxCHK_ALLOW_3RD_STATE_FOR_USER	By default a user can't set a 3-state checkbox to the third state. It can only be done from code. Using this flags allows the user to set the checkbox to the third state by clicking.
wxALIGN_RIGHT	Makes the text appear on the left of the checkbox.

See also *window styles overview* (p. 1762).

Event handling

EVT_CHECKBOX(id, func)	Process a wxEVT_COMMAND_CHECKBOX_CLICKED event, when the checkbox is clicked.

See also

wxRadioButton (p. 1096), *wxCommandEvent* (p. 180)

wxCheckBox::wxCheckBox

wxCheckBox()

Default constructor.

wxCheckBox(**wxWindow*** *parent*, **wxWindowID** *id*, **const wxString&** *label*, **const wxPoint&** *pos* = *wxDefaultPosition*, **const wxSize&** *size* = *wxDefaultSize*, **long** *style* = *0*, **const wxValidator&** *val*, **const wxString&** *name* = *"checkBox"*)

Constructor, creating and showing a checkbox.

Parameters

parent

Parent window. Must not be NULL.

id

Checkbox identifier. A value of -1 indicates a default value.

label

Text to be displayed next to the checkbox.

pos

Checkbox position. If the position (-1, -1) is specified then a default position is chosen.

size

Checkbox size. If the default size (-1, -1) is specified then a default size is chosen.

style

Window style. See *wxCheckBox* (p. 134).

validator

Window validator.

name

Window name.

See also

wxCheckBox::Create (p. 136), *wxValidator* (p. 1461)

wxCheckBox::~wxCheckBox

~wxCheckBox()

Destructor, destroying the checkbox.

wxCheckBox::Create

bool Create(**wxWindow*** *parent*, **wxWindowID** *id*, **const wxString&** *label*, **const wxPoint&** *pos* = *wxDefaultPosition*, **const wxSize&** *size* = *wxDefaultSize*, **long** *style* = *0*, **const wxValidator&** *val*, **const wxString&** *name* = *"checkBox"*)

Creates the checkbox for two-step construction. See *wxCheckBox::wxCheckBox* (p. 135) for details.

wxCheckBox::GetValue

bool GetValue() **const**

Gets the state of a 2-state checkbox.

Return value

Returns `true` if it is checked, `false` otherwise.

wxCheckBox::Get3StateValue

wxCheckBoxState Get3StateValue() **const**

Gets the state of a 3-state checkbox.

Return value

Returns wxCHK_UNCHECKED when the checkbox is unchecked, wxCHK_CHECKED when it is checked and wxCHK_UNDETERMINED when it's in the undetermined state. Asserts when the function is used with a 2-state checkbox.

wxCheckBox::Is3rdStateAllowedForUser

bool Is3rdStateAllowedForUser() **const**

Returns whether or not the user can set the checkbox to the third state.

Return value

Returns `true` if the user can set the third state of this checkbox, `false` if it can only be set programmatically or if it's a 2-state checkbox.

wxCheckBox::Is3State

bool Is3State() **const**

Returns whether or not the checkbox is a 3-state checkbox.

Return value

Returns `true` if this checkbox is a 3-state checkbox, `false` if it's a 2-state checkbox.

wxCheckBox::IsChecked

bool IsChecked() **const**

This is just a maybe more readable synonym for *GetValue* (p. 136): just as the latter, it returns `true` if the checkbox is checked and `false` otherwise.

wxCheckBox::SetValue

void SetValue(**bool** *state*)

Sets the checkbox to the given state. This does not cause a wxEVT_COMMAND_CHECKBOX_CLICKED event to get emitted.

Parameters

state

> If `true`, the check is on, otherwise it is off.

wxCheckBox::Set3StateValue

void Set3StateValue(**const wxCheckBoxState** *state*)

Sets the checkbox to the given state. This does not cause a wxEVT_COMMAND_CHECKBOX_CLICKED event to get emitted.

Parameters

state

> Can be one of: wxCHK_UNCHECKED (Check is off), wxCHK_CHECKED (Check is on) or wxCHK_UNDETERMINED (Check is mixed). Asserts when the checkbox is a 2-state checkbox and setting the state to wxCHK_UNDETERMINED.

wxCheckListBox

A checklistbox is like a listbox, but allows items to be checked or unchecked.

When using this class under Windows wxWidgets must be compiled with USE_OWNER_DRAWN set to 1.

Only the new functions for this class are documented; see also *wxListBox* (p. 847).

Please note that wxCheckListBox uses client data in its implementation, and therefore this is not available to the application.

Derived from

wxListBox (p. 847)
wxControl (p. 215)
wxWindow (p. 1490)
wxEvtHandler (p. 486)
wxObject (p. 1012)

Include files

<wx/checklst.h>

Window styles

See *wxListBox* (p. 847).

Event handling

EVT_CHECKLISTBOX(id, func)	Process a wxEVT_COMMAND_CHECKLISTBOX_TOGG LED event, when an item in the check list box is checked or unchecked.

See also

wxListBox (p. 847), *wxChoice* (p. 140), *wxComboBox* (p. 172), *wxListCtrl* (p. 852), *wxCommandEvent* (p. 180)

---

wxCheckListBox::wxCheckListBox

wxCheckListBox()

Default constructor.

wxCheckListBox(**wxWindow*** *parent*, **wxWindowID** *id*, **const wxPoint&** *pos* = *wxDefaultPosition*, **const wxSize&** *size* = *wxDefaultSize*, **int** *n*, **const wxString** *choices[] = NULL*, **long** *style = 0*, **const wxValidator&** *validator = wxDefaultValidator*, **const wxString&** *name = "listBox"*)

wxCheckListBox(**wxWindow*** *parent*, **wxWindowID** *id*, **const wxPoint&** *pos*, **const wxSize&** *size*, **const wxArrayString&** *choices*, **long** *style = 0*, **const wxValidator&** *validator = wxDefaultValidator*, **const wxString&** *name = "listBox"*)

Constructor, creating and showing a list box.

Parameters

parent

> Parent window. Must not be NULL.

id

> Window identifier. A value of -1 indicates a default value.

pos

> Window position.

size

> Window size. If the default size (-1, -1) is specified then the window is sized appropriately.

n

> Number of strings with which to initialise the control.

choices

> An array of strings with which to initialise the control.

style

> Window style. See *wxCheckListBox* (p. 137).

validator

> Window validator.

name

> Window name.

wxPython note: The wxCheckListBox constructor in wxPython reduces the `n`and `choices` arguments are to a single argument, which is a list of strings.

wxPerl note: In wxPerl there is just an array reference in place of `n`and `choices`.

wxCheckListBox::~wxCheckListBox

void ~wxCheckListBox()

Destructor, destroying the list box.

wxCheckListBox::Check

void Check(**int** *item*, **bool** *check = true*)

Checks the given item. Note that calling this method doesn't result in wxEVT_COMMAND_CHECKLISTBOX_TOGGLE being emitted.

Parameters

item

> Index of item to check.

check

> true if the item is to be checked, false otherwise.

wxCheckListBox::IsChecked

bool IsChecked(**int** *item*) **const**

Returns true if the given item is checked, false otherwise.

Parameters

item

> Index of item whose check status is to be returned.

wxChoice

A choice item is used to select one of a list of strings. Unlike a listbox, only the selection is visible until the user pulls down the menu of choices.

Derived from

wxControlWithItems (p. 215)
wxControl (p. 215)
wxWindow (p. 1490)
wxEvtHandler (p. 486)
wxObject (p. 1012)

Include files

<wx/choice.h>

Window styles

There are no special styles for wxChoice.

See also *window styles overview* (p. 1762).

Event handling

EVT_CHOICE(id, func)   Process a wxEVT_COMMAND_CHOICE_SELECTED event, when an item on the list is selected.

See also

wxListBox (p. 847), *wxComboBox* (p. 172), *wxCommandEvent* (p. 180)

wxChoice::wxChoice

wxChoice()

Default constructor.

wxChoice(**wxWindow** **parent*, **wxWindowID** *id*, **const wxPoint&** *pos*, **const wxSize&** *size*, **int** *n*, **const wxString** *choices[]*, **long** *style = 0*, **const wxValidator&** *validator = wxDefaultValidator*, **const wxString&** *name = "choice"*)

wxChoice(**wxWindow** **parent*, **wxWindowID** *id*, **const wxPoint&** *pos*, **const wxSize&** *size*, **const wxArrayString&** *choices*, **long** *style = 0*, **const wxValidator&** *validator = wxDefaultValidator*, **const wxString&** *name = "choice"*)

Constructor, creating and showing a choice.

Parameters

parent

Parent window. Must not be NULL.

id

Window identifier. A value of -1 indicates a default value.

pos

Window position.

size

Window size. If the default size (-1, -1) is specified then the choice is sized appropriately.

n

Number of strings with which to initialise the choice control.

choices

An array of strings with which to initialise the choice control.

style

Window style. See *wxChoice* (p. 140).

validator

Window validator.

name

Window name.

See also

wxChoice::Create (p. 142), *wxValidator* (p. 1461)

wxPython note: The wxChoice constructor in wxPython reduces the `n` and `choices` arguments are to a single argument, which is a list of strings.

wxPerl note: In wxPerl there is just an array reference in place of `n` and `choices`.

wxChoice::~wxChoice

 ~wxChoice()

Destructor, destroying the choice item.

wxChoice::Create

bool Create(**wxWindow** **parent*, **wxWindowID** *id*, **const wxPoint&** *pos*, **const wxSize&** *size*, **int** *n*, **const wxString** *choices[]*, **long** *style = 0*, **const wxValidator&** *validator = wxDefaultValidator*, **const wxString&** *name = "choice"*)

bool Create(**wxWindow** **parent*, **wxWindowID** *id*, **const wxPoint&** *pos*, **const wxSize&** *size*, **const wxArrayString&** *choices*, **long** *style = 0*, **const wxValidator&** *validator = wxDefaultValidator*, **const wxString&** *name = "choice"*)

Creates the choice for two-step construction. See *wxChoice::wxChoice* (p. 141).

wxChoice::Delete

void Delete(**int** *n*)

Deletes the item with the given index from the control.

Parameters

n

The item to delete.

wxChoice::GetColumns

int GetColumns() **const**

Gets the number of columns in this choice item.

Remarks

This is implemented for Motif only and always returns 1 for the other platforms.

wxChoice::SetColumns

void SetColumns(**int** *n = 1*)

Sets the number of columns in this choice item.

Parameters

n

Number of columns.

Remarks

This is implemented for Motif only and doesn't do anything under other platforms.

wxChoicebook

wxChoicebook is a class similar to *wxNotebook* (p. 999) but which uses a *wxChoice* (p. 140) to show the labels instead of the tabs.

There is no documentation for this class yet but its usage is identical to wxNotebook (except for the features clearly related to tabs only), so please refer to that class documentation for now. You can also use the *notebook sample* (p. 1713) to see wxChoicebook in action.

Derived from

wxControl (p. 215)
wxWindow (p. 1490)
wxEvtHandler (p. 486)
wxObject (p. 1012)

Include files

<wx/choicebk.h>

Window styles

wxCHB_DEFAULT	Choose the default location for the labels depending on the current platform (left everywhere except Mac where it is top).

wxCHB_TOP	Place labels above the page area.
wxCHB_LEFT	Place labels on the left side.
wxCHB_RIGHT	Place labels on the right side.
wxCHB_BOTTOM	Place labels below the page area.

See also

wxBookCtrl (p. 1797), *wxNotebook* (p. 999), *notebook sample* (p. 1713)

wxClassInfo

This class stores meta-information about classes. Instances of this class are not generally defined directly by an application, but indirectly through use of macros such as **DECLARE_DYNAMIC_CLASS** and **IMPLEMENT_DYNAMIC_CLASS**.

Derived from

No parent class.

Include files

<wx/object.h>

See also

Overview (p. 1720), *wxObject* (p. 1012)

wxClassInfo::wxClassInfo

 wxClassInfo(**const wxChar *** *className*, **const wxClassInfo *** *baseClass1*, **const wxClassInfo *** *baseClass2*, **int** *size*, **wxObjectConstructorFn** *fn*)

Constructs a wxClassInfo object. The supplied macros implicitly construct objects of this class, so there is no need to create such objects explicitly in an application.

wxClassInfo::CreateObject

wxObject* CreateObject()

Creates an object of the appropriate kind. Returns NULL if the class has not been declared dynamically creatable (typically, it is an abstract class).

wxClassInfo::FindClass

static wxClassInfo * FindClass(**wxChar *** *name*)

Finds the wxClassInfo object for a class of the given string name.

wxClassInfo::GetBaseClassName1

wxChar * GetBaseClassName1() **const**

Returns the name of the first base class (NULL if none).

wxClassInfo::GetBaseClassName2

wxChar * GetBaseClassName2() **const**

Returns the name of the second base class (NULL if none).

wxClassInfo::GetClassName

wxChar * GetClassName() **const**

Returns the string form of the class name.

wxClassInfo::GetSize

int GetSize() **const**

Returns the size of the class.

wxClassInfo::InitializeClasses

static void InitializeClasses()

Initializes pointers in the wxClassInfo objects for fast execution of IsKindOf. Called in base wxWidgets library initialization.

wxClassInfo::IsKindOf

bool IsKindOf(**wxClassInfo*** *info*)

Returns true if this class is a kind of (inherits from) the given class.

wxClient

A wxClient object represents the client part of a client-server DDE-like (Dynamic Data

Exchange) conversation. The actual DDE-based implementation using wxDDEClient is available on Windows only, but a platform-independent, socket-based version of this API is available using wxTCPClient, which has the same API.

To create a client which can communicate with a suitable server, you need to derive a class from wxConnection and another from wxClient. The custom wxConnection class will intercept communications in a 'conversation' with a server, and the custom wxClient is required so that a user-overridden*wxClient::OnMakeConnection* (p. 147) member can return a wxConnection of the required class, when a connection is made. Look at the IPC sample and the *Interprocess communications overview* (p. 1841) for an example of how to do this.

Derived from

wxClientBase
wxObject (p. 1012)

Include files

<wx/ipc.h>

See also

wxServer (p. 1167), *wxConnection* (p. 207), *Interprocess communications overview* (p. 1841)

wxClient::wxClient

wxClient()

Constructs a client object.

wxClient::MakeConnection

wxConnectionBase * MakeConnection(**const wxString&** *host*, **const wxString&** *service*, **const wxString&** *topic*)

Tries to make a connection with a server by host (machine name under UNIX - use 'localhost' for same machine; ignored when using native DDE in Windows), service name and topic string. If the server allows a connection, a wxConnection object will be returned. The type of wxConnection returned can be altered by overriding the *wxClient::OnMakeConnection* (p. 147) member to return your own derived connection object.

Under Unix, the service name may be either an integer port identifier in which case an Internet domain socket will be used for the communications, or a valid file name (which shouldn't exist and will be deleted afterwards) in which case a Unix domain socket is created.

SECURITY NOTE: Using Internet domain sockets if extremely insecure for IPC as there is absolutely no access control for them, use Unix domain sockets whenever possible!

wxClient::OnMakeConnection

wxConnectionBase * OnMakeConnection()

Called by *wxClient::MakeConnection* (p. 146), by default this simply returns a new wxConnection object. Override this method to return a wxConnection descendant customised for the application.

The advantage of deriving your own connection class is that it will enable you to intercept messages initiated by the server, such as *wxConnection::OnAdvise* (p. 209). You may also want to store application-specific data in instances of the new class.

wxClient::ValidHost

bool ValidHost(**const wxString&** *host*)

Returns true if this is a valid host name, false otherwise. This always returns true under MS Windows.

wxClientDC

A wxClientDC must be constructed if an application wishes to paint on the client area of a window from outside an **OnPaint** event. This should normally be constructed as a temporary stack object; don't store a wxClientDC object.

To draw on a window from within **OnPaint**, construct a *wxPaintDC* (p. 1024) object.

To draw on the whole window including decorations, construct a *wxWindowDC* (p. 1547) object (Windows only).

Derived from

wxWindowDC (p. 1547)
wxDC (p. 365)

Include files

<wx/dcclient.h>

See also

wxDC (p. 365), *wxMemoryDC* (p. 935), *wxPaintDC* (p. 1024), *wxWindowDC* (p. 1547), *wxScreenDC* (p. 1142)

wxClientDC::wxClientDC

wxClientDC(**wxWindow*** *window*)

Constructor. Pass a pointer to the window on which you wish to paint.

wxClientData

All classes deriving from *wxEvtHandler* (p. 486)(such as all controls and *wxApp* (p. 31)) can hold arbitrary data which is here referred to as "client data". This is useful e.g. for scripting languages which need to handle shadow objects for most of wxWidgets' classes and which store a handle to such a shadow class as client data in that class. This data can either be of type void - in which case the data*container* does not take care of freeing the data again or it is of type wxClientData or its derivatives. In that case the container (e.g. a control) will free the memory itself later. Note that you *must not* assign both void data and data derived from the wxClientData class to a container.

Some controls can hold various items and these controls can additionally hold client data for each item. This is the case for*wxChoice* (p. 140), *wxComboBox* (p. 172)and *wxListBox* (p. 847). *wxTreeCtrl* (p. 1424)has a specialized class *wxTreeItemData* (p. 1446)for each item in the tree.

If you want to add client data to your own classes, you may use the mix-in class *wxClientDataContainer* (p. 149).

Include files

<wx/clntdata.h>

See also

wxEvtHandler (p. 486), *wxTreeItemData* (p. 1446),*wxStringClientData* (p. 1310), *wxClientDataContainer* (p. 149)

wxClientData::wxClientData

wxClientData()

Constructor.

wxClientData::~wxClientData

~wxClientData()

Virtual destructor.

wxClientDataContainer

This class is a mixin that provides storage and management of "client data." This data can either be of type void - in which case the data*container* does not take care of freeing the data again or it is of type wxClientData or its derivatives. In that case the container will free the memory itself later. Note that you *must not* assign both void data and data derived from the wxClientData class to a container.

NOTE: This functionality is currently duplicated in wxEvtHandler in order to avoid having more than one vtable in that class hierarchy.

See also

wxEvtHandler (p. 486), *wxClientData* (p. 148)

Derived from

No base class

Include files

<wx/clntdata.h>

Data structures

wxClientDataContainer::wxClientDataContainer

 wxClientDataContainer()

wxClientDataContainer::~wxClientDataContainer

 ~wxClientDataContainer()

wxClientDataContainer::GetClientData

void* GetClientData() **const**

Get the untyped client data.

wxClientDataContainer::GetClientObject

wxClientData* GetClientObject() **const**

Get a pointer to the client data object.

wxClientDataContainer::SetClientData

void SetClientData(**void*** *data*)

Set the untyped client data.

wxClientDataContainer::SetClientObject

void SetClientObject(**wxClientData*** *data*)

Set the client data object. Any previous object will be deleted.

wxClipboard

A class for manipulating the clipboard. Note that this is not compatible with the clipboard class from wxWidgets 1.xx, which has the same name but a different implementation.

To use the clipboard, you call member functions of the global **wxTheClipboard** object.

See also the *wxDataObject overview* (p. 1819) for further information.

Call *wxClipboard::Open* (p. 152) to get ownership of the clipboard. If this operation returns true, you now own the clipboard. Call *wxClipboard::SetData* (p. 152) to put data on the clipboard, or *wxClipboard::GetData* (p. 152) to retrieve data from the clipboard. Call *wxClipboard::Close* (p. 151) to close the clipboard and relinquish ownership. You should keep the clipboard open only momentarily.

For example:

```
// Write some text to the clipboard
if (wxTheClipboard->Open())
{
  // This data objects are held by the clipboard,
  // so do not delete them in the app.
  wxTheClipboard->SetData( new wxTextDataObject("Some text") );
  wxTheClipboard->Close();
}

// Read some text
if (wxTheClipboard->Open())
{
  if (wxTheClipboard->IsSupported( wxDF_TEXT ))
  {
    wxTextDataObject data;
    wxTheClipboard->GetData( data );
    wxMessageBox( data.GetText() );
  }
  wxTheClipboard->Close();
}
```

Derived from

wxObject (p. 1012)

Include files

<wx/clipbrd.h>

See also

Drag and drop overview (p. 1817), *wxDataObject* (p. 239)

wxClipboard::wxClipboard

wxClipboard()

Constructor.

wxClipboard::~wxClipboard

~wxClipboard()

Destructor.

wxClipboard::AddData

bool AddData(**wxDataObject*** *data*)

Call this function to add the data object to the clipboard. You may call this function repeatedly after having cleared the clipboard using *wxClipboard::Clear* (p. 151).

After this function has been called, the clipboard owns the data, so do not delete the data explicitly.

See also

wxClipboard::SetData (p. 152)

wxClipboard::Clear

void Clear()

Clears the global clipboard object and the system's clipboard if possible.

wxClipboard::Close

void Close()

Call this function to close the clipboard, having opened it with *wxClipboard::Open* (p. 152).

wxClipboard::Flush

bool Flush()

Flushes the clipboard: this means that the data which is currently on clipboard will stay available even after the application exits (possibly eating memory), otherwise the clipboard will be emptied on exit. Returns false if the operation is unsuccessful for any reason.

wxClipboard::GetData

bool GetData(**wxDataObject&** *data*)

Call this function to fill *data* with data on the clipboard, if available in the required format. Returns true on success.

wxClipboard::IsOpened

bool IsOpened() **const**

Returns true if the clipboard has been opened.

wxClipboard::IsSupported

bool IsSupported(**const wxDataFormat&** *format*)

Returns true if there is data which matches the data format of the given data object currently **available** (IsSupported sounds like a misnomer, FIXME: better deprecate this name?) on the clipboard.

wxClipboard::Open

bool Open()

Call this function to open the clipboard before calling *wxClipboard::SetData* (p. 152) and *wxClipboard::GetData* (p. 152).

Call *wxClipboard::Close* (p. 151) when you have finished with the clipboard. You should keep the clipboard open for only a very short time.

Returns true on success. This should be tested (as in the sample shown above).

wxClipboard::SetData

bool SetData(**wxDataObject*** *data*)

Call this function to set the data object to the clipboard. This function will clear all

previous contents in the clipboard, so calling it several times does not make any sense.

After this function has been called, the clipboard owns the data, so do not delete the data explicitly.

See also

wxClipboard::AddData (p. 151)

wxClipboard::UsePrimarySelection

void UsePrimarySelection(**bool** *primary* = *true*)

On platforms supporting it (currently only GTK), selects the so called PRIMARY SELECTION as the clipboard as opposed to the normal clipboard, if *primary* is true.

wxCloseEvent

This event class contains information about window and session close events.

The handler function for EVT_CLOSE is called when the user has tried to close a a frame or dialog box using the window manager (X) or system menu (Windows). It can also be invoked by the application itself programmatically, for example by calling the *wxWindow::Close* (p. 1496) function.

You should check whether the application is forcing the deletion of the window using *wxCloseEvent::CanVeto* (p. 154). If this is `false`, you *must* destroy the window using *wxWindow::Destroy* (p. 1499). If the return value is true, it is up to you whether you respond by destroying the window.

If you don't destroy the window, you should call *wxCloseEvent::Veto* (p. 155) to let the calling code know that you did not destroy the window. This allows the *wxWindow::Close* (p. 1496) function to return `true` or `false` depending on whether the close instruction was honoured or not.

Derived from

wxEvent (p. 483)

Include files

<wx/event.h>

Event table macros

To process a close event, use these event handler macros to direct input to member functions that take a wxCloseEvent argument.

EVT_CLOSE(func)	Process a close event, supplying the member function. This event applies to wxFrame and wxDialog classes.

| **EVT_QUERY_END_SESSION(func)** | Process a query end session event, supplying the member function. This event applies to wxApp only. |
| **EVT_END_SESSION(func)** | Process an end session event, supplying the member function. This event applies to wxApp only. |

See also

wxWindow::Close (p. 1496), *Window deletion overview* (p. 1763)

wxCloseEvent::wxCloseEvent

 wxCloseEvent(**WXTYPE** *commandEventType = 0*, **int** *id = 0*)

Constructor.

wxCloseEvent::CanVeto

bool CanVeto()

Returns true if you can veto a system shutdown or a window close event. Vetoing a window close event is not possible if the calling code wishes to force the application to exit, and so this function must be called to check this.

wxCloseEvent::GetLoggingOff

bool GetLoggingOff() **const**

Returns true if the user is logging off.

wxCloseEvent::GetForce

bool GetForce() **const**

Returns true if the application wishes to force the window to close. This will shortly be obsolete, replaced by CanVeto.

wxCloseEvent::SetCanVeto

void SetCanVeto(**bool** *canVeto*)

Sets the 'can veto' flag.

wxCloseEvent::SetForce

void SetForce(**bool** *force*) **const**

Sets the 'force' flag.

wxCloseEvent::SetLoggingOff

void SetLoggingOff(**bool** *loggingOff*) **const**

Sets the 'logging off' flag.

wxCloseEvent::Veto

void Veto(**bool** *veto* = *true*)

Call this from your event handler to veto a system shutdown or to signal to the calling application that a window close did not happen.

You can only veto a shutdown if *wxCloseEvent::CanVeto* (p. 154) returns true.

wxCmdLineParser

wxCmdLineParser is a class for parsing the command line.

It has the following features:

1. distinguishes options, switches and parameters; allows option grouping

2. allows both short and long options

3. automatically generates the usage message from the command line description

4. does type checks on the options values (number, date, ...).

To use it you should follow these steps:

1. *construct* (p. 157) an object of this class giving it the command line to parse and optionally its description or use `AddXXX()` functions later

2. call `Parse()`

3. use `Found()` to retrieve the results

In the documentation below the following terminology is used:

switch
This is a boolean option which can be given or not, but which doesn't have any value. We use the word switch to distinguish such boolean options from more generic options like those

described below. For example, $-v$ might be a switch meaning "enable verbose mode".

option   Option for us here is something which comes with a value 0 unlike a switch. For example, $-o:filename$ might be an option which allows to specify the name of the output file.

parameter   This is a required program argument.

Derived from

No base class

Include files

<wx/cmdline.h>

Constants

The structure wxCmdLineEntryDesc is used to describe the one command line switch, option or parameter. An array of such structures should be passed to *SetDesc()* (p. 161). Also, the meanings of parameters of the `AddXXX()` functions are the same as of the corresponding fields in this structure:

```
struct wxCmdLineEntryDesc
{
    wxCmdLineEntryType kind;
    const wxChar *shortName;
    const wxChar *longName;
    const wxChar *description;
    wxCmdLineParamType type;
    int flags;
};
```

The type of a command line entity is in the `kind` field and may be one of the following constants:

```
enum wxCmdLineEntryType
{
    wxCMD_LINE_SWITCH,
    wxCMD_LINE_OPTION,
    wxCMD_LINE_PARAM,
    wxCMD_LINE_NONE          // use this to terminate the list
}
```

The field `shortName` is the usual, short, name of the switch or the option.`longName` is the corresponding long name or NULL if the option has no long name. Both of these fields are unused for the parameters. Both the short and long option names can contain only letters, digits and the underscores.

`description` is used by the *Usage()* (p. 163) method to construct a help message explaining the syntax of the program.

The possible values of `type` which specifies the type of the value accepted by an option or parameter are:

```
enum wxCmdLineParamType
{
    wxCMD_LINE_VAL_STRING,  // default
    wxCMD_LINE_VAL_NUMBER,
    wxCMD_LINE_VAL_DATE,
    wxCMD_LINE_VAL_NONE
}
```

Finally, the `flags` field is a combination of the following bit masks:

```
enum
{
    wxCMD_LINE_OPTION_MANDATORY = 0x01, // this option must be given
    wxCMD_LINE_PARAM_OPTIONAL   = 0x02, // the parameter may be omitted
    wxCMD_LINE_PARAM_MULTIPLE   = 0x04, // the parameter may be repeated
    wxCMD_LINE_OPTION_HELP      = 0x08, // this option is a help request
    wxCMD_LINE_NEEDS_SEPARATOR  = 0x10, // must have sep before the value
}
```

Notice that by default (i.e. if flags are just 0), options are optional (sic) and each call to *AddParam()* (p. 162) allows one more parameter - this may be changed by giving non-default flags to it, i.e. use `wxCMD_LINE_OPTION_MANDATORY` to require that the option is given and `wxCMD_LINE_PARAM_OPTIONAL` to make a parameter optional. Also, `wxCMD_LINE_PARAM_MULTIPLE` may be specified if the programs accepts a variable number of parameters - but it only can be given for the last parameter in the command line description. If you use this flag, you will probably need to use *GetParamCount* (p. 164) to retrieve the number of parameters effectively specified after calling *Parse* (p. 162).

The last flag `wxCMD_LINE_NEEDS_SEPARATOR` can be specified to require a separator (either a colon, an equal sign or white space) between the option name and its value. By default, no separator is required.

See also

wxApp::argc (p. 32) and *wxApp::argv* (p. 32)
console sample

Construction

Before *Parse* (p. 162) can be called, the command line parser object must have the command line to parse and also the rules saying which switches, options and parameters are valid - this is called command line description in what follows.

You have complete freedom of choice as to when specify the required information, the

only restriction is that it must be done before calling *Parse* (p. 162).

To specify the command line to parse you may use either one of constructors accepting it (*wxCmdLineParser(argc, argv)* (p. 159) or *wxCmdLineParser* (p. 159) usually) or, if you use *the default constructor* (p. 159), you can do it later by calling *SetCmdLine* (p. 160).

The same holds for command line description: it can be specified either in the constructor (*without command line* (p. 159) or *together with it* (p. 159)) or constructed later using either *SetDesc* (p. 161) or combination of *AddSwitch* (p. 162), *AddOption* (p. 162) and *AddParam* (p. 162) methods.

Using constructors or *SetDesc* (p. 161) uses a (usually `const static`) table containing the command line description. If you want to decide which options to accept during the run-time, using one of the `AddXXX()` functions above might be preferable.

Customization

wxCmdLineParser has several global options which may be changed by the application. All of the functions described in this section should be called before *Parse* (p. 162).

First global option is the support for long (also known as GNU-style) options. The long options are the ones which start with two dashes (`"--"`) and look like this: `--verbose`, i.e. they generally are complete words and not some abbreviations of them. As long options are used by more and more applications, they are enabled by default, but may be disabled with *DisableLongOptions* (p. 161).

Another global option is the set of characters which may be used to start an option (otherwise, the word on the command line is assumed to be a parameter). Under Unix, `'-'` is always used, but Windows has at least two common choices for this: `'-'` and `'/'`. Some programs also use `'+'`. The default is to use what suits most the current platform, but may be changed with *SetSwitchChars* (p. 161) method.

Finally, *SetLogo* (p. 161) can be used to show some application-specific text before the explanation given by *Usage* (p. 163) function.

Parsing command line

After the command line description was constructed and the desired options were set, you can finally call *Parse* (p. 162) method. It returns 0 if the command line was correct and was parsed, -1 if the help option was specified (this is a separate case as, normally, the program will terminate after this) or a positive number if there was an error during the command line parsing.

In the latter case, the appropriate error message and usage information are logged by wxCmdLineParser itself using the standard wxWidgets logging functions.

Getting results

After calling *Parse* (p. 162) (and if it returned 0), you may access the results of parsing using one of overloaded `Found()` methods.

For a simple switch, you will simply call *Found* (p. 163) to determine if the switch was given or not, for an option or a parameter, you will call a version of `Found()` which also returns the associated value in the provided variable. All `Found()` functions return true if the switch or option were found in the command line or false if they were not specified.

wxCmdLineParser::wxCmdLineParser

wxCmdLineParser()

Default constructor. You must use *SetCmdLine* (p. 160) later.

wxCmdLineParser::wxCmdLineParser

wxCmdLineParser(**int** *argc*, **char**** *argv*)

wxCmdLineParser(**int** *argc*, **wchar_t**** *argv*)

Constructor specifies the command line to parse. This is the traditional (Unix) command line format. The parameters *argc* and *argv* have the same meaning as for `main()` function.

The second overloaded constructor is only available in Unicode build. The first one is available in both ANSI and Unicode modes because under some platforms the command line arguments are passed as ASCII strings even to Unicode programs.

wxCmdLineParser::wxCmdLineParser

wxCmdLineParser(**const wxString&** *cmdline*)

Constructor specifies the command line to parse in Windows format. The parameter *cmdline* has the same meaning as the corresponding parameter of `WinMain()`.

wxCmdLineParser::wxCmdLineParser

wxCmdLineParser(**const wxCmdLineEntryDesc*** *desc*)

Same as *wxCmdLineParser* (p. 159), but also specifies the *command line description* (p. 161).

wxCmdLineParser::wxCmdLineParser

wxCmdLineParser(**const wxCmdLineEntryDesc*** *desc*, **int** *argc*, **char**** *argv*)

Same as *wxCmdLineParser* (p. 159), but also specifies the *command line description* (p. 161).

wxCmdLineParser::wxCmdLineParser

wxCmdLineParser(**const wxCmdLineEntryDesc*** *desc*, **const wxString&** *cmdline*)

Same as *wxCmdLineParser* (p. 159), but also specifies the *command line description* (p. 161).

wxCmdLineParser::ConvertStringToArgs

static wxArrayString ConvertStringToArgs(**const wxChar** **cmdline*)

Breaks down the string containing the full command line in words. The words are separated by whitespace. The quotes can be used in the input string to quote the white space and the back slashes can be used to quote the quotes.

wxCmdLineParser::SetCmdLine

void SetCmdLine(**int** *argc*, **char**** *argv*)

void SetCmdLine(**int** *argc*, **wchar_t**** *argv*)

Set command line to parse after using one of the constructors which don't do it. The second overload of this function is only available in Unicode build.

See also

wxCmdLineParser (p. 159)

wxCmdLineParser::SetCmdLine

void SetCmdLine(**const wxString&** *cmdline*)

Set command line to parse after using one of the constructors which don't do it.

See also

wxCmdLineParser (p. 159)

wxCmdLineParser::~wxCmdLineParser

~wxCmdLineParser()

Frees resources allocated by the object.

NB: destructor is not virtual, don't use this class polymorphically.

wxCmdLineParser::SetSwitchChars

void SetSwitchChars(**const wxString&** *switchChars*)

switchChars contains all characters with which an option or switch may start. Default is "-" for Unix, "-/" for Windows.

wxCmdLineParser::EnableLongOptions

void EnableLongOptions(**bool** *enable = true*)

Enable or disable support for the long options.

As long options are not (yet) POSIX-compliant, this option allows to disable them.

See also

Customization (p. 158) and *AreLongOptionsEnabled* (p. 161)

wxCmdLineParser::DisableLongOptions

void DisableLongOptions()

Identical to *EnableLongOptions(false)* (p. 161).

wxCmdLineParser::AreLongOptionsEnabled

bool AreLongOptionsEnabled()

Returns true if long options are enabled, otherwise false.

See also

EnableLongOptions (p. 161)

wxCmdLineParser::SetLogo

void SetLogo(**const wxString&** *logo*)

logo is some extra text which will be shown by *Usage* (p. 163) method.

wxCmdLineParser::SetDesc

void SetDesc(**const wxCmdLineEntryDesc*** *desc*)

Construct the command line description

Take the command line description from the wxCMD_LINE_NONE terminated table.

Example of usage:

```
static const wxCmdLineEntryDesc cmdLineDesc[] =
{
    { wxCMD_LINE_SWITCH, "v", "verbose", "be verbose" },
    { wxCMD_LINE_SWITCH, "q", "quiet",   "be quiet" },

    { wxCMD_LINE_OPTION, "o", "output",  "output file" },
    { wxCMD_LINE_OPTION, "i", "input",   "input dir" },
    { wxCMD_LINE_OPTION, "s", "size",    "output block size",
wxCMD_LINE_VAL_NUMBER },
    { wxCMD_LINE_OPTION, "d", "date",    "output file date",
wxCMD_LINE_VAL_DATE },

    { wxCMD_LINE_PARAM,  NULL, NULL, "input file", wxCMD_LINE_VAL_STRING,
wxCMD_LINE_PARAM_MULTIPLE },

    { wxCMD_LINE_NONE }
};

wxCmdLineParser parser;

parser.SetDesc(cmdLineDesc);
```

wxCmdLineParser::AddSwitch

void AddSwitch(**const wxString&** *name*, **const wxString&** *lng = wxEmptyString*, **const wxString&** *desc = wxEmptyString*, **int** *flags = 0*)

Add a switch *name* with an optional long name *lng* (no long name if it is empty, which is default), description *desc* and flags *flags* to the command line description.

wxCmdLineParser::AddOption

void AddOption(**const wxString&** *name*, **const wxString&** *lng = wxEmptyString*, **const wxString&** *desc = wxEmptyString*, **wxCmdLineParamType** *type = wxCMD_LINE_VAL_STRING*, **int** *flags = 0*)

Add an option *name* with an optional long name *lng* (no long name if it is empty, which is default) taking a value of the given type (string by default) to the command line description.

wxCmdLineParser::AddParam

void AddParam(**const wxString&** *desc = wxEmptyString*, **wxCmdLineParamType** *type = wxCMD_LINE_VAL_STRING*, **int** *flags = 0*)

Add a parameter of the given *type* to the command line description.

wxCmdLineParser::Parse

int Parse(**bool** *giveUsage* = *true*)

Parse the command line, return 0 if ok, -1 if "-h" or "--help" option was encountered and the help message was given or a positive value if a syntax error occurred.

Parameters

giveUsage

> If true (default), the usage message is given if a syntax error was encountered while parsing the command line or if help was requested. If false, only error messages about possible syntax errors are given, use *Usage* (p. 163) to show the usage message from the caller if needed.

wxCmdLineParser::Usage

void Usage()

Give the standard usage message describing all program options. It will use the options and parameters descriptions specified earlier, so the resulting message will not be helpful to the user unless the descriptions were indeed specified.

See also

SetLogo (p. 161)

wxCmdLineParser::Found

bool Found(**const wxString&** *name*) **const**

Returns true if the given switch was found, false otherwise.

wxCmdLineParser::Found

bool Found(**const wxString&** *name*, **wxString*** *value*) **const**

Returns true if an option taking a string value was found and stores the value in the provided pointer (which should not be NULL).

wxCmdLineParser::Found

bool Found(**const wxString&** *name*, **long*** *value*) **const**

Returns true if an option taking an integer value was found and stores the value in the provided pointer (which should not be NULL).

wxCmdLineParser::Found

bool Found(**const wxString&** *name*, **wxDateTime*** *value*) **const**

Returns true if an option taking a date value was found and stores the value in the provided pointer (which should not be NULL).

wxCmdLineParser::GetParamCount

size_t GetParamCount() **const**

Returns the number of parameters found. This function makes sense mostly if you had used `wxCMD_LINE_PARAM_MULTIPLE` flag.

wxCmdLineParser::GetParam

wxString GetParam(**size_t** *n = 0u*) **const**

Returns the value of Nth parameter (as string only for now).

See also

GetParamCount (p. 164)

wxColour

A colour is an object representing a combination of Red, Green, and Blue (RGB) intensity values, and is used to determine drawing colours. See the entry for *wxColourDatabase* (p. 169) for how a pointer to a predefined, named colour may be returned instead of creating a new colour.

Valid RGB values are in the range 0 to 255.

You can retrieve the current system colour settings with *wxSystemSettings* (p. 1318).

Derived from

wxObject (p. 1012)

Include files

<wx/colour.h>

Predefined objects

Objects:

wxNullColour

Pointers:

wxBLACK

wxWHITE
wxRED
wxBLUE
wxGREEN
wxCYAN
wxLIGHT_GREY

See also

wxColourDatabase (p. 169)*, wxPen* (p. 1036)*, wxBrush* (p. 103)*, wxColourDialog* (p. 171)*, wxSystemSettings* (p. 1318)

wxColour::wxColour

wxColour()

Default constructor.

wxColour(**const unsigned char** *red*, **const unsigned char** *green*, **const unsigned char** *blue*)

Constructs a colour from red, green and blue values.

wxColour(**const wxString&** *colourNname*)

Constructs a colour object using a colour name listed in **wxTheColourDatabase**.

wxColour(**const wxColour&** *colour*)

Copy constructor.

Parameters

red

> The red value.

green

> The green value.

blue

> The blue value.

colourName

> The colour name.

colour

> The colour to copy.

See also

wxColourDatabase (p. 169)

wxPython note: Constructors supported by wxPython are:

       wxColour(red=0, green=0, blue=0)

       wxNamedColour(name)

wxColour::Blue

unsigned char Blue() **const**

Returns the blue intensity.

wxColour::GetPixel

long GetPixel() **const**

Returns a pixel value which is platform-dependent. On Windows, a COLORREF is returned. On X, an allocated pixel value is returned.

-1 is returned if the pixel is invalid (on X, unallocated).

wxColour::Green

unsigned char Green() **const**

Returns the green intensity.

wxColour::Ok

bool Ok() **const**

Returns `true` if the colour object is valid (the colour has been initialised with RGB values).

wxColour::Red

unsigned char Red() **const**

Returns the red intensity.

wxColour::Set

void Set(**const unsigned char** *red,* **const unsigned char** *green,* **const unsigned char** *blue*)

Sets the RGB intensity values.

wxColour::operator =

wxColour& operator =(**const wxColour&** *colour*)

Assignment operator, taking another colour object.

wxColour& operator =(**const wxString&** *colourName*)

Assignment operator, using a colour name to be found in the colour database.

See also

wxColourDatabase (p. 169)

wxColour::operator ==

bool operator ==(**const wxColour&** *colour*)

Tests the equality of two colours by comparing individual red, green blue colours.

wxColour::operator !=

bool operator !=(**const wxColour&** *colour*)

Tests the inequality of two colours by comparing individual red, green blue colours.

wxColourData

This class holds a variety of information related to colour dialogs.

Derived from

wxObject (p. 1012)

Include files

<wx/cmndata.h>

See also

wxColour (p. 164), *wxColourDialog* (p. 171), *wxColourDialog overview* (p. 1798)

wxColourData::wxColourData

wxColourData()

Constructor. Initializes the custom colours to `wxNullColour`, the *data colour* setting to black, and the *choose full* setting to true.

wxColourData::~wxColourData

~wxColourData()

Destructor.

wxColourData::GetChooseFull

bool GetChooseFull() **const**

Under Windows, determines whether the Windows colour dialog will display the full dialog with custom colour selection controls. Under PalmOS, determines whether colour dialog will display full rgb colour picker or only available palette indexer. Has no meaning under other platforms.

The default value is true.

wxColourData::GetColour

wxColour& GetColour() **const**

Gets the current colour associated with the colour dialog.

The default colour is black.

wxColourData::GetCustomColour

wxColour& GetCustomColour(**int** *i*) **const**

Gets the *i*th custom colour associated with the colour dialog. *i* should be an integer between 0 and 15.

The default custom colours are invalid colours.

wxColourData::SetChooseFull

void SetChooseFull(**const bool** *flag*)

Under Windows, tells the Windows colour dialog to display the full dialog with custom colour selection controls. Under other platforms, has no effect.

The default value is true.

wxColourData::SetColour

void SetColour(**const wxColour&** *colour*)

Sets the default colour for the colour dialog.

The default colour is black.

wxColourData::SetCustomColour

void SetCustomColour(**int** *i*, **const wxColour&** *colour*)

Sets the *i*th custom colour for the colour dialog. *i* should be an integer between 0 and 15.

The default custom colours are invalid colours.

wxColourData::operator =

void operator =(**const wxColourData&** *data*)

Assignment operator for the colour data.

wxColourDatabase

wxWidgets maintains a database of standard RGB colours for a predefined set of named colours (such as "BLACK", "LIGHT GREY"). The application may add to this set if desired by using *AddColour* (p. 170) and may use it to look up colours by names using *Find* (p. 170) or find the names for the standard colour suing *FindName* (p. 170).

There is one predefined instance of this class called **wxTheColourDatabase**.

Derived from

None

Include files

<wx/gdicmn.h>

Remarks

The standard database contains at least the following colours:

AQUAMARINE, BLACK, BLUE, BLUE VIOLET, BROWN, CADET BLUE, CORAL, CORNFLOWER BLUE, CYAN, DARK GREY, DARK GREEN, DARK OLIVE GREEN, DARK ORCHID, DARK SLATE BLUE, DARK SLATE GREY DARK TURQUOISE, DIM GREY, FIREBRICK, FOREST GREEN, GOLD, GOLDENROD, GREY, GREEN, GREEN

YELLOW, INDIAN RED, KHAKI, LIGHT BLUE, LIGHT GREY, LIGHT STEEL BLUE, LIME GREEN, MAGENTA, MAROON, MEDIUM AQUAMARINE, MEDIUM BLUE, MEDIUM FOREST GREEN, MEDIUM GOLDENROD, MEDIUM ORCHID, MEDIUM SEA GREEN, MEDIUM SLATE BLUE, MEDIUM SPRING GREEN, MEDIUM TURQUOISE, MEDIUM VIOLET RED, MIDNIGHT BLUE, NAVY, ORANGE, ORANGE RED, ORCHID, PALE GREEN, PINK, PLUM, PURPLE, RED, SALMON, SEA GREEN, SIENNA, SKY BLUE, SLATE BLUE, SPRING GREEN, STEEL BLUE, TAN, THISTLE, TURQUOISE, VIOLET, VIOLET RED, WHEAT, WHITE, YELLOW, YELLOW GREEN.

See also

wxColour (p. 164)

wxColourDatabase::wxColourDatabase

 wxColourDatabase()

Constructs the colour database. It will be initialized at the first use.

wxColourDatabase::AddColour

void AddColour(**const wxString&** *colourName*, **const wxColour&***colour*)

void AddColour(**const wxString&** *colourName*, **wxColour*** *colour*)

Adds a colour to the database. If a colour with the same name already exists, it is replaced.

Please note that the overload taking a pointer is deprecated and will be removed in the next wxWidgets version, please don't use it.

wxColourDatabase::Find

wxColour Find(**const wxString&** *colourName*)

Finds a colour given the name. Returns an invalid colour object (that is, such that its *Ok()* (p. 166) method returns `false`) if the colour wasn't found in the database.

wxColourDatabase::FindName

wxString FindName(**const wxColour&** *colour*) **const**

Finds a colour name given the colour. Returns an empty string if the colour is not found in the database.

wxColourDialog

This class represents the colour chooser dialog.

Derived from

wxDialog (p. 406)
wxWindow (p. 1490)
wxEvtHandler (p. 486)
wxObject (p. 1012)

Include files

<wx/colordlg.h>

See also

wxColourDialog Overview (p. 1798),
wxColour (p. 164),
wxColourData (p. 167),
wxGetColourFromUser (p. 1616)

wxColourDialog::wxColourDialog

wxColourDialog(**wxWindow*** *parent*, **wxColourData*** *data* = *NULL*)

Constructor. Pass a parent window, and optionally a pointer to a block of colour data, which will be copied to the colour dialog's colour data. Custom colours from colour data object will be be used in dialog's colour palette. Invalid entries in custom colours list will be ignored on some platforms (GTK) or replaced with white colour on platforms where custom colours palette has fixed size (MSW).

See also

wxColourData (p. 167)

wxColourDialog::~wxColourDialog

~wxColourDialog()

Destructor.

wxColourDialog::Create

bool Create(**wxWindow*** *parent*, **wxColourData*** *data* = *NULL*)

Same as *constructor* (p. 171).

wxColourDialog::GetColourData

wxColourData& GetColourData()

Returns the *colour data* (p. 167) associated with the colour dialog.

wxColourDialog::ShowModal

int ShowModal()

Shows the dialog, returning wxID_OK if the user pressed OK, and wxID_CANCEL otherwise.

wxComboBox

A combobox is like a combination of an edit control and a listbox. It can be displayed as static list with editable or read-only text field; or a drop-down list with text field; or a drop-down list without a text field.

A combobox permits a single selection only. Combobox items are numbered from zero.

Derived from

wxControlWithItems (p. 215)
wxControl (p. 215)
wxWindow (p. 1490)
wxEvtHandler (p. 486)
wxObject (p. 1012)

Include files

<wx/combobox.h>

Window styles

wxCB_SIMPLE	Creates a combobox with a permanently displayed list. Windows only.
wxCB_DROPDOWN	Creates a combobox with a drop-down list.
wxCB_READONLY	Same as wxCB_DROPDOWN but only the strings specified as the combobox choices can be selected, it is impossible to select (even from a program) a string which is not in the choices list.
wxCB_SORT	Sorts the entries in the list alphabetically.

See also *window styles overview* (p. 1762).

Event handling

EVT_COMBOBOX(id, func)	Process a wxEVT_COMMAND_COMBOBOX_SELECTED event, when an item on the list is selected. Note that calling*GetValue* (p. 176) returns the new value of selection.
EVT_TEXT(id, func)	Process a wxEVT_COMMAND_TEXT_UPDATED event, when the combobox text changes.
EVT_TEXT_ENTER(id, func)	Process a wxEVT_COMMAND_TEXT_ENTER event, when <RETURN> is pressed in the combobox.

See also

wxListBox (p. 847), *wxTextCtrl* (p. 1340), *wxChoice* (p. 140), *wxCommandEvent* (p. 180)

wxComboBox::wxComboBox

wxComboBox()

Default constructor.

wxComboBox(**wxWindow*** *parent*, **wxWindowID** *id*, **const wxString&** *value = ""*, **const wxPoint&** *pos = wxDefaultPosition*, **const wxSize&** *size = wxDefaultSize*, **int** *n*, **const wxString** *choices[]*, **long** *style = 0*, **const wxValidator&** *validator = wxDefaultValidator*, **const wxString&** *name = "comboBox"*)

wxComboBox(**wxWindow*** *parent*, **wxWindowID** *id*, **const wxString&** *value*, **const wxPoint&** *pos*, **const wxSize&** *size*, **const wxArrayString&** *choices*, **long** *style = 0*, **const wxValidator&** *validator = wxDefaultValidator*, **const wxString&** *name = "comboBox"*)

Constructor, creating and showing a combobox.

Parameters

parent

Parent window. Must not be NULL.

id

Window identifier. A value of -1 indicates a default value.

value

Initial selection string. An empty string indicates no selection.

pos

Window position.

size

Window size. If the default size (-1, -1) is specified then the window is sized appropriately.

n

Number of strings with which to initialise the control.

choices

An array of strings with which to initialise the control.

style

Window style. See *wxComboBox* (p. 172).

validator

Window validator.

name

Window name.

See also

wxComboBox::Create (p. 174), *wxValidator* (p. 1461)

wxPython note: The wxComboBox constructor in wxPython reduces the `n`and `choices` arguments are to a single argument, which is a list of strings.

wxPerl note: In wxPerl there is just an array reference in place of `n`and `choices`.

wxComboBox::~wxComboBox

 ~wxComboBox()

Destructor, destroying the combobox.

wxComboBox::Create

bool Create(**wxWindow*** *parent*, **wxWindowID** *id*, **const wxString&** *value* = *""*, **const wxPoint&** *pos* = *wxDefaultPosition*, **const wxSize&** *size* = *wxDefaultSize*, **int** *n*, **const wxString** *choices[]*, **long** *style = 0*, **const wxValidator&** *validator* = *wxDefaultValidator*, **const wxString&** *name* = *"comboBox"*)

bool Create(**wxWindow*** *parent*, **wxWindowID** *id*, **const wxString&** *value*, **const wxPoint&** *pos*, **const wxSize&** *size*, **const wxArrayString&** *choices*, **long** *style = 0*, **const wxValidator&** *validator* = *wxDefaultValidator*, **const wxString&** *name* =

"comboBox")

Creates the combobox for two-step construction. Derived classes should call or replace this function. See *wxComboBox::wxComboBox* (p. 173) for further details.

wxComboBox::CanCopy

bool CanCopy() **const**

Returns true if the combobox is editable and there is a text selection to copy to the clipboard. Only available on Windows.

wxComboBox::CanCut

bool CanCut() **const**

Returns true if the combobox is editable and there is a text selection to copy to the clipboard. Only available on Windows.

wxComboBox::CanPaste

bool CanPaste() **const**

Returns true if the combobox is editable and there is text on the clipboard that can be pasted into the text field. Only available on Windows.

wxComboBox::CanRedo

bool CanRedo() **const**

Returns true if the combobox is editable and the last undo can be redone. Only available on Windows.

wxComboBox::CanUndo

bool CanUndo() **const**

Returns true if the combobox is editable and the last edit can be undone. Only available on Windows.

wxComboBox::Copy

void Copy()

Copies the selected text to the clipboard.

wxComboBox::Cut

void Cut()

Copies the selected text to the clipboard and removes the selection.

wxComboBox::GetInsertionPoint

long GetInsertionPoint() **const**

Returns the insertion point for the combobox's text field.

wxComboBox::GetLastPosition

virtual wxTextPos GetLastPosition() **const**

Returns the last position in the combobox text field.

wxComboBox::GetValue

wxString GetValue() **const**

Returns the current value in the combobox text field.

wxComboBox::Paste

void Paste()

Pastes text from the clipboard to the text field.

wxComboBox::Redo

void Redo()

Redoes the last undo in the text field. Windows only.

wxComboBox::Replace

void Replace(**long** *from*, **long** *to*, **const wxString&** *text*)

Replaces the text between two positions with the given text, in the combobox text field.

Parameters

from

The first position.

to

The second position.

text

The text to insert.

wxComboBox::Remove

void Remove(**long** *from*, **long** *to*)

Removes the text between the two positions in the combobox text field.

Parameters

from

The first position.

to

The last position.

wxComboBox::SetInsertionPoint

void SetInsertionPoint(**long** *pos*)

Sets the insertion point in the combobox text field.

Parameters

pos

The new insertion point.

wxComboBox::SetInsertionPointEnd

void SetInsertionPointEnd()

Sets the insertion point at the end of the combobox text field.

wxComboBox::SetSelection

void SetSelection(**long** *from*, **long** *to*)

Selects the text between the two positions, in the combobox text field.

Parameters

from

> The first position.

to

> The second position.

wxPython note: This method is called SetMark in wxPython, SetSelection name is kept for *wxControlWithItems::SetSelection* (p. 221).

wxComboBox::SetValue

void SetValue(**const wxString&** *text*)

Sets the text for the combobox text field.

NB: For a combobox with wxCB_READONLY style the string must be in the combobox choices list, otherwise the call to SetValue() is ignored.

Parameters

text

> The text to set.

wxComboBox::Undo

void Undo()

Undoes the last edit in the text field. Windows only.

wxCommand

wxCommand is a base class for modelling an application command, which is an action usually performed by selecting a menu item, pressing a toolbar button or any other means provided by the application to change the data or view.

Derived from

wxObject (p. 1012)

Include files

<wx/cmdproc.h>

See also

Overview (p. 1806)

wxCommand::wxCommand

wxCommand(**bool** *canUndo = false*, **const wxString&** *name = NULL*)

Constructor. wxCommand is an abstract class, so you will need to derive a new class and call this constructor from your own constructor.

canUndo tells the command processor whether this command is undo-able. You can achieve the same functionality by overriding the CanUndo member function (if for example the criteria for undoability is context-dependent).

name must be supplied for the command processor to display the command name in the application's edit menu.

wxCommand::~wxCommand

~wxCommand()

Destructor.

wxCommand::CanUndo

bool CanUndo()

Returns true if the command can be undone, false otherwise.

wxCommand::Do

bool Do()

Override this member function to execute the appropriate action when called. Return true to indicate that the action has taken place, false otherwise. Returning false will indicate to the command processor that the action is not undoable and should not be added to the command history.

wxCommand::GetName

wxString GetName()

Returns the command name.

wxCommand::Undo

bool Undo()

Override this member function to un-execute a previous Do. Return true to indicate that the action has taken place, false otherwise. Returning false will indicate to the command processor that the action is not redoable and no change should be made to the command history.

How you implement this command is totally application dependent, but typical strategies include:

- Perform an inverse operation on the last modified piece of data in the document. When redone, a copy of data stored in command is pasted back or some operation reapplied. This relies on the fact that you know the ordering of Undos; the user can never Undo at an arbitrary position in the command history.

- Restore the entire document state (perhaps using document transactioning). Potentially very inefficient, but possibly easier to code if the user interface and data are complex, and an 'inverse execute' operation is hard to write.

The docview sample uses the first method, to remove or restore segments in the drawing.

wxCommandEvent

This event class contains information about command events, which originate from a variety of simple controls. More complex controls, such as *wxTreeCtrl* (p. 1424), have separate command event classes.

Derived from

wxEvent (p. 483)

Include files

<wx/event.h>

Event table macros

To process a menu command event, use these event handler macros to direct input to member functions that take a wxCommandEvent argument.

EVT_COMMAND(id, event, func)	Process a command, supplying the window identifier, command event identifier, and member function.
EVT_COMMAND_RANGE(id1, id2, event, func)	Process a command for a range of window identifiers, supplying the minimum and maximum window identifiers, command event identifier, and member function.
EVT_BUTTON(id, func)	Process a wxEVT_COMMAND_BUTTON_CLICKED command, which is generated by a wxButton control.

EVT_CHECKBOX(id, func)	Process a wxEVT_COMMAND_CHECKBOX_CLICKED command, which is generated by a wxCheckBox control.
EVT_CHOICE(id, func)	Process a wxEVT_COMMAND_CHOICE_SELECTED command, which is generated by a wxChoice control.
EVT_COMBOBOX(id, func)	Process a wxEVT_COMMAND_COMBOBOX_SELECTED command, which is generated by a wxComboBox control.
EVT_LISTBOX(id, func)	Process a wxEVT_COMMAND_LISTBOX_SELECTED command, which is generated by a wxListBox control.
EVT_LISTBOX_DCLICK(id, func)	Process a wxEVT_COMMAND_LISTBOX_DOUBLECLICKED command, which is generated by a wxListBox control.
EVT_MENU(id, func)	Process a wxEVT_COMMAND_MENU_SELECTED command, which is generated by a menu item.
EVT_MENU_RANGE(id1, id2, func)	Process a wxEVT_COMMAND_MENU_RANGE command, which is generated by a range of menu items.
EVT_CONTEXT_MENU(func)	Process the event generated when the user has requested a popup menu to appear by pressing a special keyboard key (under Windows) or by right clicking the mouse.
EVT_RADIOBOX(id, func)	Process a wxEVT_COMMAND_RADIOBOX_SELECTED command, which is generated by a wxRadioBox control.
EVT_RADIOBUTTON(id, func)	Process a wxEVT_COMMAND_RADIOBUTTON_SELECTED command, which is generated by a wxRadioButton control.
EVT_SCROLLBAR(id, func)	Process a wxEVT_COMMAND_SCROLLBAR_UPDATED command, which is generated by a wxScrollBar control. This is provided for compatibility only;

	more specific scrollbar event macros should be used instead (see *wxScrollEvent* (p. 1159)).
EVT_SLIDER(id, func)	Process a wxEVT_COMMAND_SLIDER_UPDATED command, which is generated by a wxSlider control.
EVT_TEXT(id, func)	Process a wxEVT_COMMAND_TEXT_UPDATED command, which is generated by a wxTextCtrl control.
EVT_TEXT_ENTER(id, func)	Process a wxEVT_COMMAND_TEXT_ENTER command, which is generated by a wxTextCtrl control. Note that you must use wxTE_PROCESS_ENTER flag when creating the control if you want it to generate such events.
EVT_TEXT_MAXLEN(id, func)	Process a wxEVT_COMMAND_TEXT_MAXLEN command, which is generated by a wxTextCtrl control when the user tries to enter more characters into it than the limit previously set with *SetMaxLength* (p. 1355).
EVT_TOOL(id, func)	Process a wxEVT_COMMAND_TOOL_CLICKED event (a synonym for wxEVT_COMMAND_MENU_SELECTED). Pass the id of the tool.
EVT_TOOL_RANGE(id1, id2, func)	Process a wxEVT_COMMAND_TOOL_CLICKED event for a range of identifiers. Pass the ids of the tools.
EVT_TOOL_RCLICKED(id, func)	Process a wxEVT_COMMAND_TOOL_RCLICKED event. Pass the id of the tool.
EVT_TOOL_RCLICKED_RANGE(id1, id2, func)	Process a wxEVT_COMMAND_TOOL_RCLICKED event for a range of ids. Pass the ids of the tools.
EVT_TOOL_ENTER(id, func)	Process a wxEVT_COMMAND_TOOL_ENTER event. Pass the id of the toolbar itself. The value of wxCommandEvent::GetSelection is the tool id, or -1 if the mouse cursor has moved off a tool.
EVT_COMMAND_LEFT_CLICK(id, func)	Process a

wxEVT_COMMAND_LEFT_CLICK command, which is generated by a control (Windows 95 and NT only).

EVT_COMMAND_LEFT_DCLICK(id, func) Process a wxEVT_COMMAND_LEFT_DCLICK command, which is generated by a control (Windows 95 and NT only).

EVT_COMMAND_RIGHT_CLICK(id, func) Process a wxEVT_COMMAND_RIGHT_CLICK command, which is generated by a control (Windows 95 and NT only).

EVT_COMMAND_SET_FOCUS(id, func) Process a wxEVT_COMMAND_SET_FOCUS command, which is generated by a control (Windows 95 and NT only).

EVT_COMMAND_KILL_FOCUS(id, func) Process a wxEVT_COMMAND_KILL_FOCUS command, which is generated by a control (Windows 95 and NT only).

EVT_COMMAND_ENTER(id, func) Process a wxEVT_COMMAND_ENTER command, which is generated by a control.

wxCommandEvent::wxCommandEvent

wxCommandEvent(**WXTYPE** *commandEventType = 0*, **int** *id = 0*)

Constructor.

wxCommandEvent::Checked

bool Checked() **const**

Deprecated, use *IsChecked* (p. 184) instead.

wxCommandEvent::GetClientData

void* GetClientData()

Returns client data pointer for a listbox or choice selection event (not valid for a deselection).

wxCommandEvent::GetClientObject

wxClientData * GetClientObject()

Returns client object pointer for a listbox or choice selection event (not valid for a deselection).

wxCommandEvent::GetExtraLong

long GetExtraLong()

Returns extra information dependant on the event objects type. If the event comes from a listbox selection, it is a boolean determining whether the event was a selection (true) or a deselection (false). A listbox deselection only occurs for multiple-selection boxes, and in this case the index and string values are indeterminate and the listbox must be examined by the application.

wxCommandEvent::GetInt

int GetInt()

Returns the integer identifier corresponding to a listbox, choice or radiobox selection (only if the event was a selection, not a deselection), or a boolean value representing the value of a checkbox.

wxCommandEvent::GetSelection

int GetSelection()

Returns item index for a listbox or choice selection event (not valid for a deselection).

wxCommandEvent::GetString

wxString GetString()

Returns item string for a listbox or choice selection event (not valid for a deselection).

wxCommandEvent::IsChecked

bool IsChecked() **const**

This method can be used with checkbox and menu events: for the checkboxes, the method returns `true` for a selection event and `false` for a deselection one. For the menu events, this method indicates if the menu item just has become checked or unchecked (and thus only makes sense for checkable menu items).

wxCommandEvent::IsSelection

bool IsSelection()

For a listbox or similar event, returns true if it is a selection, false if it is a deselection.

wxCommandEvent::SetClientData

void SetClientData(**void*** *clientData*)

Sets the client data for this event.

wxCommandEvent::SetClientObject

void SetClientObject(**wxClientData*** *clientObject*)

Sets the client object for this event. The client object is *not* owned by the event object and the event object will not delete the client object in its destructor. The client object must be owned and deleted by another object (e.g. a control) that has longer life time than the event object.

wxCommandEvent::SetExtraLong

void SetExtraLong(**int** *extraLong*)

Sets the **m_extraLong** member.

wxCommandEvent::SetInt

void SetInt(**int** *intCommand*)

Sets the **m_commandInt** member.

wxCommandEvent::SetString

void SetString(**const wxString&** *string*)

Sets the **m_commandString** member.

wxCommandProcessor

wxCommandProcessor is a class that maintains a history of wxCommands, with undo/redo functionality built-in. Derive a new class from this if you want different behaviour.

Derived from

wxObject (p. 1012)

Include files

<wx/cmdproc.h>

See also

wxCommandProcessor overview (p. 1806), *wxCommand* (p. 178)

wxCommandProcessor::wxCommandProcessor

wxCommandProcessor(**int** *maxCommands = -1*)

Constructor.

maxCommands may be set to a positive integer to limit the number of commands stored to it, otherwise (and by default) the list of commands can grow arbitrarily.

wxCommandProcessor::~wxCommandProcessor

~wxCommandProcessor()

Destructor.

wxCommandProcessor::CanUndo

virtual bool CanUndo()

Returns true if the currently-active command can be undone, false otherwise.

wxCommandProcessor::ClearCommands

virtual void ClearCommands()

Deletes all the commands in the list and sets the current command pointer to NULL.

wxCommandProcessor::Redo

virtual bool Redo()

Executes (redoes) the current command (the command that has just been undone if any).

wxCommandProcessor::GetCommands

wxList& GetCommands() **const**

Returns the list of commands.

wxCommandProcessor::GetMaxCommands

int GetMaxCommands() **const**

Returns the maximum number of commands that the command processor stores.

wxCommandProcessor::GetEditMenu

wxMenu* GetEditMenu() **const**

Returns the edit menu associated with the command processor.

wxCommandProcessor::GetRedoAccelerator

const wxString& GetRedoAccelerator() **const**

Returns the string that will be appended to the Redo menu item.

wxCommandProcessor::GetRedoMenuLabel

wxString GetRedoMenuLabel() **const**

Returns the string that will be shown for the redo menu item.

wxCommandProcessor::GetUndoAccelerator

const wxString& GetUndoAccelerator() **const**

Returns the string that will be appended to the Undo menu item.

wxCommandProcessor::GetUndoMenuLabel

wxString GetUndoMenuLabel() **const**

Returns the string that will be shown for the undo menu item.

wxCommandProcessor::Initialize

virtual void Initialize()

Initializes the command processor, setting the current command to the last in the list (if

any), and updating the edit menu (if one has been specified).

wxCommandProcessor::IsDirty

virtual bool IsDirty()

Returns a boolean value that indicates if changes have been made since the last save operation. This only works if *wxCommandProcessor::MarkAsSaved* (p. 188)is called whenever the project is saved.

wxCommandProcessor::MarkAsSaved

virtual void MarkAsSaved()

You must call this method whenever the project is saved if you plan to use *wxCommandProcessor::IsDirty* (p. 188).

wxCommandProcessor::SetEditMenu

void SetEditMenu(**wxMenu*** *menu*)

Tells the command processor to update the Undo and Redo items on this menu as appropriate. Set this to NULL if the menu is about to be destroyed and command operations may still be performed, or the command processor may try to access an invalid pointer.

wxCommandProcessor::SetMenuStrings

void SetMenuStrings()

Sets the menu labels according to the currently set menu and the current command state.

wxCommandProcessor::SetRedoAccelerator

void SetRedoAccelerator(**const wxString&***accel*)

Sets the string that will be appended to the Redo menu item.

wxCommandProcessor::SetUndoAccelerator

void SetUndoAccelerator(**const wxString&***accel*)

Sets the string that will be appended to the Undo menu item.

wxCommandProcessor::Submit

virtual bool Submit(**wxCommand** *command, **bool** *storeIt = true*)

Submits a new command to the command processor. The command processor calls wxCommand::Do to execute the command; if it succeeds, the command is stored in the history list, and the associated edit menu (if any) updated appropriately. If it fails, the command is deleted immediately. Once Submit has been called, the passed command should not be deleted directly by the application.

storeIt indicates whether the successful command should be stored in the history list.

wxCommandProcessor::Undo

virtual bool Undo()

Undoes the command just executed.

wxCondition

wxCondition variables correspond to pthread conditions or to Win32 event objects. They may be used in a multithreaded application to wait until the given condition becomes true which happens when the condition becomes signaled.

For example, if a worker thread is doing some long task and another thread has to wait until it is finished, the latter thread will wait on the condition object and the worker thread will signal it on exit (this example is not perfect because in this particular case it would be much better to just *Wait()* (p. 1384) for the worker thread, but if there are several worker threads it already makes much more sense).

Note that a call to *Signal()* (p. 191) may happen before the other thread calls *Wait()* (p. 192) and, just as with the pthread conditions, the signal is then lost and so if you want to be sure that you don't miss it you must keep the mutex associated with the condition initially locked and lock it again before calling *Signal()* (p. 191). Of course, this means that this call is going to block until *Wait()* (p. 192) is called by another thread.

Example

This example shows how a main thread may launch a worker thread which starts running and then waits until the main thread signals it to continue:

```
class MySignallingThread : public wxThread
{
public:
    MySignallingThread(wxMutex *mutex, wxCondition *condition)
    {
        m_mutex = mutex;
        m_condition = condition;

        Create();
    }
```

```
    virtual ExitCode Entry()
    {
        ... do our job ...

        // tell the other(s) thread(s) that we're about to terminate: we
must
        // lock the mutex first or we might signal the condition before
the
        // waiting threads start waiting on it!
        wxMutexLocker lock(m_mutex);
        m_condition.Broadcast(); // same as Signal() here -- one waiter
only

        return 0;
    }

private:
    wxCondition *m_condition;
    wxMutex *m_mutex;
};

int main()
{
    wxMutex mutex;
    wxCondition condition(mutex);

    // the mutex should be initially locked
    mutex.Lock();

    // create and run the thread but notice that it won't be able to
    // exit (and signal its exit) before we unlock the mutex below
    MySignallingThread *thread = new MySignallingThread(&mutex,
&condition);

    thread->Run();

    // wait for the thread termination: Wait() atomically unlocks the
mutex
    // which allows the thread to continue and starts waiting
    condition.Wait();

    // now we can exit
    return 0;
}
```

Of course, here it would be much better to simply use a joinable thread and call *wxThread::Wait* (p. 1384) on it, but this example does illustrate the importance of properly locking the mutex when using wxCondition.

Constants

The following return codes are returned by wxCondition member functions:

```
enum wxCondError
{
    wxCOND_NO_ERROR = 0,    // successful completion
    wxCOND_INVALID,         // object hasn't been initialized successfully
    wxCOND_TIMEOUT,         // WaitTimeout() has timed out
    wxCOND_MISC_ERROR       // some other error
};
```

Derived from

None.

Include files

<wx/thread.h>

See also

wxThread (p. 1377), *wxMutex* (p. 994)

wxCondition::wxCondition

 wxCondition(**wxMutex&** *mutex*)

Default and only constructor. The *mutex* must be locked by the caller before calling *Wait* (p. 192) function.

Use *IsOk* (p. 191) to check if the object was successfully initialized.

wxCondition::~wxCondition

 ~wxCondition()

Destroys the wxCondition object. The destructor is not virtual so this class should not be used polymorphically.

wxCondition::Broadcast

void Broadcast()

Broadcasts to all waiting threads, waking all of them up. Note that this method may be called whether the mutex associated with this condition is locked or not.

See also

wxCondition::Signal (p. 191)

wxCondition::IsOk

bool IsOk() **const**

Returns `true` if the object had been initialized successfully, `false` if an error occurred.

wxCondition::Signal

void Signal()

Signals the object waking up at most one thread. If several threads are waiting on the same condition, the exact thread which is woken up is undefined. If no threads are waiting, the signal is lost and the condition would have to be signalled again to wake up any thread which may start waiting on it later.

Note that this method may be called whether the mutex associated with this condition is locked or not.

See also

wxCondition::Broadcast (p. 191)

wxCondition::Wait

wxCondError Wait()

Waits until the condition is signalled.

This method atomically releases the lock on the mutex associated with this condition (this is why it must be locked prior to calling Wait) and puts the thread to sleep until *Signal* (p. 191) or *Broadcast* (p. 191) is called.

Note that even if *Signal* (p. 191) had been called before Wait without waking up any thread, the thread would still wait for another one and so it is important to ensure that the condition will be signalled after Wait or the thread may sleep forever.

Return value

Returns wxCOND_NO_ERROR on success, another value if an error occurred.

See also

WaitTimeout (p. 192)

wxCondition::WaitTimeout

wxCondError WaitTimeout(**unsigned long** *milliseconds*)

Waits until the condition is signalled or the timeout has elapsed.

This method is identical to *Wait* (p. 192) except that it returns, with the return code of wxCOND_TIMEOUT as soon as the given timeout expires.

Parameters

milliseconds

    Timeout in milliseconds

Return value

Returns `wxCOND_NO_ERROR` if the condition was signalled, `wxCOND_TIMEOUT` if the timeout elapsed before this happened or another error code from wxCondError enum.

wxConfigBase

wxConfigBase class defines the basic interface of all config classes. It can not be used by itself (it is an abstract base class) and you will always use one of its derivations: *wxFileConfig* (p. 509), wxRegConfig or any other.

However, usually you don't even need to know the precise nature of the class you're working with but you would just use the wxConfigBase methods. This allows you to write the same code regardless of whether you're working with the registry under Win32 or text-based config files under Unix (or even Windows 3.1 .INI files if you're really unlucky). To make writing the portable code even easier, wxWidgets provides a typedef wxConfig which is mapped onto the native wxConfigBase implementation on the given platform: i.e. wxRegConfig under Win32 and wxFileConfig otherwise.

See *config overview* (p. 1748) for the descriptions of all features of this class.

It is highly recommended to use static functions *Get()* and/or *Set()*, so please have a *look at them.* (p. 194)

Derived from

No base class

Include files

<wx/config.h> (to let wxWidgets choose a wxConfig class for your platform)
<wx/confbase.h> (base config class)
<wx/fileconf.h> (wxFileConfig class)
<wx/msw/regconf.h> (wxRegConfig class)

Example

Here is how you would typically use this class:

```
  // using wxConfig instead of writing wxFileConfig or wxRegConfig
enhances
  // portability of the code
  wxConfig *config = new wxConfig("MyAppName");

  wxString str;
  if ( config->Read("LastPrompt", &str) ) {
    // last prompt was found in the config file/registry and its value is
now
    // in str
    ...
  }
  else {
    // no last prompt...
  }

  // another example: using default values and the full path instead of
just
```

```
        // key name: if the key is not found , the value 17 is returned
        long value = config->Read("/LastRun/CalculatedValues/MaxValue", 17);
        ...
        ...
        ...
        // at the end of the program we would save everything back
        config->Write("LastPrompt", str);
        config->Write("/LastRun/CalculatedValues/MaxValue", value);

        // the changes will be written back automatically
        delete config;
```

This basic example, of course, doesn't show all wxConfig features, such as enumerating, testing for existence and deleting the entries and groups of entries in the config file, its abilities to automatically store the default values or expand the environment variables on the fly. However, the main idea is that using this class is easy and that it should normally do what you expect it to.

NB: in the documentation of this class, the words "config file" also mean "registry hive" for wxRegConfig and, generally speaking, might mean any physical storage where a wxConfigBase-derived class stores its data.

Static functions

These functions deal with the "default" config object. Although its usage is not at all mandatory it may be convenient to use a global config object instead of creating and deleting the local config objects each time you need one (especially because creating a wxFileConfig object might be a time consuming operation). In this case, you may create this global config object in the very start of the program and *Set()* it as the default. Then, from anywhere in your program, you may access it using the *Get()* function. Note that you must delete this object (usually in *wxApp::OnExit* (p. 37)) in order to avoid memory leaks, wxWidgets won't do it automatically.

As it happens, you may even further simplify the procedure described above: you may forget about calling *Set()*. When *Get()* is called and there is no current object, it will create one using *Create()* function. To disable this behaviour *DontCreateOnDemand()* is provided.

Note: You should use either *Set()* or *Get()* because wxWidgets library itself would take advantage of it and could save various information in it. For example *wxFontMapper* (p. 575) or Unix version of *wxFileDialog* (p. 511) have the ability to use wxConfig class.

Set (p. 205)
Get (p. 201)
Create (p. 200)
DontCreateOnDemand (p. 200)

Constructor and destructor

wxConfigBase (p. 198)

~wxConfigBase (p. 200)

Path management

As explained in *config overview* (p. 1748), the config classes support a file system-like hierarchy of keys (files) and groups (directories). As in the file system case, to specify a key in the config class you must use a path to it. Config classes also support the notion of the current group, which makes it possible to use the relative paths. To clarify all this, here is an example (it is only for the sake of demonstration, it doesn't do anything sensible!):

```
wxConfig *config = new wxConfig("FooBarApp");

// right now the current path is '/'
conf->Write("RootEntry", 1);

// go to some other place: if the group(s) don't exist, they will be
created
conf->SetPath("/Group/Subgroup");

// create an entry in subgroup
conf->Write("SubgroupEntry", 3);

// '..' is understood
conf->Write("../GroupEntry", 2);
conf->SetPath("..");

wxASSERT( conf->Read("Subgroup/SubgroupEntry", 0l) == 3 );

// use absolute path: it is allowed, too
wxASSERT( conf->Read("/RootEntry", 0l) == 1 );
```

Warning: it is probably a good idea to always restore the path to its old value on function exit:

```
void foo(wxConfigBase *config)
{
  wxString strOldPath = config->GetPath();

  config->SetPath("/Foo/Data");
  ...

  config->SetPath(strOldPath);
}
```

because otherwise the assert in the following example will surely fail (we suppose here that *foo()* function is the same as above except that it doesn't save and restore the path):

```
void bar(wxConfigBase *config)
{
  config->Write("Test", 17);

  foo(config);

  // we're reading "/Foo/Data/Test" here! -1 will probably be
returned...
  wxASSERT( config->Read("Test", -1) == 17 );
}
```

Finally, the path separator in wxConfigBase and derived classes is always '/', regardless of the platform (i.e. it is **not** '\\' under Windows).

SetPath (p. 206)
GetPath (p. 203)

Enumeration

The functions in this section allow to enumerate all entries and groups in the config file. All functions here return `false` when there are no more items.

You must pass the same index to GetNext and GetFirst (don't modify it). Please note that it is **not** the index of the current item (you will have some great surprises with wxRegConfig if you assume this) and you shouldn't even look at it: it is just a "cookie" which stores the state of the enumeration. It can't be stored inside the class because it would prevent you from running several enumerations simultaneously, that's why you must pass it explicitly.

Having said all this, enumerating the config entries/groups is very simple:

```
wxConfigBase *config = ...;
wxArrayString aNames;

// enumeration variables
wxString str;
long dummy;

// first enum all entries
bool bCont = config->GetFirstEntry(str, dummy);
while ( bCont ) {
  aNames.Add(str);

  bCont = GetConfig()->GetNextEntry(str, dummy);
}

... we have all entry names in aNames...

// now all groups...
bCont = GetConfig()->GetFirstGroup(str, dummy);
while ( bCont ) {
  aNames.Add(str);

  bCont = GetConfig()->GetNextGroup(str, dummy);
}

... we have all group (and entry) names in aNames...
```

There are also functions to get the number of entries/subgroups without actually enumerating them, but you will probably never need them.

GetFirstGroup (p. 201)
GetNextGroup (p. 202)
GetFirstEntry (p. 202)
GetNextEntry (p. 202)

Tests of existence

Miscellaneous functions

Key access

These function are the core of wxConfigBase class: they allow you to read and write config file data. All *Read* function take a default value which will be returned if the specified key is not found in the config file.

Currently, only two types of data are supported: string and long (but it might change in the near future). To work with other types: for *int* or *bool* you can work with function taking/returning *long* and just use the casts. Better yet, just use *long* for all variables which you're going to save in the config file: chances are that `sizeof(bool) ==` `sizeof(int) == sizeof(long)` anyhow on your system. For *float*, *double* and, in general, any other type you'd have to translate them to/from string representation and use string functions.

Try not to read long values into string variables and vice versa: although it just might work with wxFileConfig, you will get a system error with wxRegConfig because in the Windows registry the different types of entries are indeed used.

Final remark: the *szKey* parameter for all these functions can contain an arbitrary path (either relative or absolute), not just the key name.

Rename entries/groups

The functions in this section allow to rename entries or subgroups of the current group. They will return `false` on error. typically because either the entry/group with the original name doesn't exist, because the entry/group with the new name already exists or because the function is not supported in this wxConfig implementation.

Delete entries/groups

The functions in this section delete entries and/or groups of entries from the config file. *DeleteAll()* is especially useful if you want to erase all traces of your program presence: for example, when you uninstall it.

Options

Some aspects of wxConfigBase behaviour can be changed during run-time. The first of them is the expansion of environment variables in the string values read from the config file: for example, if you have the following in your config file:

```
# config file for my program
UserData = $HOME/data

# the following syntax is valud only under Windows
UserData = %windir%\\data.dat
```

the call to `config->Read("UserData")` will return something like`"/home/zeitlin/data"` if you're lucky enough to run a Linux system ;-)

Although this feature is very useful, it may be annoying if you read a value which containts '$' or '%' symbols (% is used for environment variables expansion under Windows) which are not used for environment variable expansion. In this situation you may call SetExpandEnvVars(false) just before reading this value and SetExpandEnvVars(true) just after. Another solution would be to prefix the offending symbols with a backslash.

The following functions control this option:

wxConfigBase::wxConfigBase

 wxConfigBase(**const wxString&** *appName* = *wxEmptyString*, **const wxString&** *vendorName* = *wxEmptyString*, **const wxString&** *localFilename* = *wxEmptyString*, **const wxString&** *globalFilename* = *wxEmptyString*, **long** *style* = 0, **wxMBConv&** *conv* = *wxConvUTF8*)

This is the default and only constructor of the wxConfigBase class, and derived classes.

Parameters

appName

The application name. If this is empty, the class will normally use *wxApp::GetAppName* (p. 33) to set it. The application name is used in the registry key on Windows, and can be used to deduce the local filename parameter if that is missing.

vendorName

The vendor name. If this is empty, it is assumed that no vendor name is wanted, if this is optional for the current config class. The vendor name is appended to the application name for wxRegConfig.

localFilename

Some config classes require a local filename. If this is not present, but required, the application name will be used instead.

globalFilename

Some config classes require a global filename. If this is not present, but required, the application name will be used instead.

style

Can be one of wxCONFIG_USE_LOCAL_FILE and wxCONFIG_USE_GLOBAL_FILE. The style interpretation depends on the config class and is ignored by some. For wxFileConfig, these styles determine whether a local or global config file is created or used. If the flag is present but the parameter is empty, the parameter will be set to a default. If the parameter is present but the style flag not, the relevant flag will be added to the style. For wxFileConfig you can also add wxCONFIG_USE_RELATIVE_PATH by logically or'ing it to either of the _FILE options to tell wxFileConfig to use relative instead of absolute paths. For wxFileConfig, you can also add wxCONFIG_USE_NO_ESCAPE_CHARACTERS which will turn off character escaping for the values of entries stored in the config file: for example a *foo* key with some backslash characters will be stored as `foo=C:\mydir` instead of the usual storage of `foo=C:\\mydir`. For wxRegConfig, this flag refers to HKLM, and provides read-only access.

The wxCONFIG_USE_NO_ESCAPE_CHARACTERS style can be helpful if your config file must be read or written to by a non-wxWidgets program (which might not understand the escape characters). Note, however, that if wxCONFIG_USE_NO_ESCAPE_CHARACTERS style is used, it is is now your application's responsibility to ensure that there is no newline or other illegal characters in a value, before writing that value to the file.

conv

This parameter is only used by wxFileConfig when compiled in Unicode mode. It specifies the encoding in which the configuration file is written.

Remarks

By default, environment variable expansion is on and recording defaults is off.

wxConfigBase::~wxConfigBase

~wxConfigBase()

Empty but ensures that dtor of all derived classes is virtual.

wxConfigBase::Create

static wxConfigBase * Create()

Create a new config object: this function will create the "best" implementation of wxConfig available for the current platform, see comments near the definition of wxCONFIG_WIN32_NATIVE for details. It returns the created object and also sets it as the current one.

wxConfigBase::DontCreateOnDemand

void DontCreateOnDemand()

Calling this function will prevent *Get()* from automatically creating a new config object if the current one is NULL. It might be useful to call it near the program end to prevent "accidental" creation of a new config object.

wxConfigBase::DeleteAll

bool DeleteAll()

Delete the whole underlying object (disk file, registry key, ...). Primarly for use by uninstallation routine.

wxConfigBase::DeleteEntry

bool DeleteEntry(**const wxString&** *key*, **bool** *bDeleteGroupIfEmpty = true*)

Deletes the specified entry and the group it belongs to if it was the last key in it and the second parameter is true.

wxConfigBase::DeleteGroup

bool DeleteGroup(**const wxString&** *key*)

Delete the group (with all subgroups)

wxConfigBase::Exists

bool Exists(**wxString&** *strName*) **const**

returns `true` if either a group or an entry with a given name exists

wxConfigBase::Flush

bool Flush(**bool** *bCurrentOnly = false*)

permanently writes all changes (otherwise, they're only written from object's destructor)

wxConfigBase::Get

static wxConfigBase * Get(**bool** *CreateOnDemand = true*)

Get the current config object. If there is no current object and*CreateOnDemand* is true, creates one (using *Create*) unless DontCreateOnDemand was called previously.

wxConfigBase::GetAppName

wxString GetAppName() **const**

Returns the application name.

wxConfigBase::GetEntryType

enum wxConfigBase::EntryType GetEntryType(**const wxString&** *name*) **const**

Returns the type of the given entry or *Unknown* if the entry doesn't exist. This function should be used to decide which version of Read() should be used because some of wxConfig implementations will complain about type mismatch otherwise: e.g., an attempt to read a string value from an integer key with wxRegConfig will fail.

The result is an element of enum EntryType:

```
enum EntryType
{
  Type_Unknown,
  Type_String,
  Type_Boolean,
  Type_Integer,
  Type_Float
};
```


wxConfigBase::GetFirstGroup

bool GetFirstGroup(**wxString&** *str*, **long&** *index*) **const**

Gets the first group.

wxPython note: The wxPython version of this method returns a 3-tuple consisting of the continue flag, the value string, and the index for the next call.

wxPerl note: In wxPerl this method takes no arguments and returns a 3-element list (`continue, str, index`).

wxConfigBase::GetFirstEntry

bool GetFirstEntry(**wxString&** *str*, **long&** *index*) **const**

Gets the first entry.

wxPython note: The wxPython version of this method returns a 3-tuple consisting of the continue flag, the value string, and the index for the next call.

wxPerl note: In wxPerl this method takes no arguments and returns a 3-element list (`continue, str, index`).

wxConfigBase::GetNextGroup

bool GetNextGroup(**wxString&** *str*, **long&** *index*) **const**

Gets the next group.

wxPython note: The wxPython version of this method returns a 3-tuple consisting of the continue flag, the value string, and the index for the next call.

wxPerl note: In wxPerl this method only takes the `index` parameter and returns a 3-element list (`continue, str, index`).

wxConfigBase::GetNextEntry

bool GetNextEntry(**wxString&** *str*, **long&** *index*) **const**

Gets the next entry.

wxPython note: The wxPython version of this method returns a 3-tuple consisting of the continue flag, the value string, and the index for the next call.

wxPerl note: In wxPerl this method only takes the `index` parameter and returns a 3-element list (`continue, str, index`).

wxConfigBase::GetNumberOfEntries

uint GetNumberOfEntries(**bool** *bRecursive* = *false*) **const**

wxConfigBase::GetNumberOfGroups

uint GetNumberOfGroups(**bool** *bRecursive = false*) **const**

Get number of entries/subgroups in the current group, with or without its subgroups.

wxConfigBase::GetPath

const wxString& GetPath() **const**

Retrieve the current path (always as absolute path).

wxConfigBase::GetVendorName

wxString GetVendorName() **const**

Returns the vendor name.

wxConfigBase::HasEntry

bool HasEntry(**wxString&** *strName*) **const**

returns `true` if the entry by this name exists

wxConfigBase::HasGroup

bool HasGroup(**const wxString&** *strName*) **const**

returns `true` if the group by this name exists

wxConfigBase::IsExpandingEnvVars

bool IsExpandingEnvVars() **const**

Returns `true` if we are expanding environment variables in key values.

wxConfigBase::IsRecordingDefaults

bool IsRecordingDefaults() **const**

Returns `true` if we are writing defaults back to the config file.

wxConfigBase::Read

bool Read(**const wxString&** *key*, **wxString*** *str*) **const**

Read a string from the key, returning `true` if the value was read. If the key was not found, *str* is not changed.

bool Read(**const wxString&** *key*, **wxString*** *str*, **const wxString&** *defaultVal*) **const**

Read a string from the key. The default value is returned if the key was not found.

Returns `true` if value was really read, `false` if the default was used.

wxString Read(**const wxString&** *key*, **const wxString&** *defaultVal*) **const**

Another version of *Read()*, returning the string value directly.

bool Read(**const wxString&** *key*, **long*** *l*) **const**

Reads a long value, returning `true` if the value was found. If the value was not found, *l* is not changed.

bool Read(**const wxString&** *key*, **long*** *l*,**long** *defaultVal*) **const**

Reads a long value, returning `true` if the value was found. If the value was not found, *defaultVal* is used instead.

long Read(**const wxString&** *key*, **long** *defaultVal*) **const**

Reads a long value from the key and returns it. *defaultVal* is returned if the key is not found.

NB: writing

```
conf->Read("key", 0);
```

won't work because the call is ambiguous: compiler can not choose between two*Read* functions. Instead, write:

```
conf->Read("key", 0l);
```

bool Read(**const wxString&** *key*, **double*** *d*) **const**

Reads a double value, returning `true` if the value was found. If the value was not found, *d* is not changed.

bool Read(**const wxString&** *key*, **double*** *d*, **double** *defaultVal*) **const**

Reads a double value, returning `true` if the value was found. If the value was not found, *defaultVal* is used instead.

bool Read(**const wxString&** *key*, **bool*** *b*) **const**

Reads a bool value, returning `true` if the value was found. If the value was not found, *b* is not changed.

bool Read(**const wxString&** *key*, **bool*** *d*,**bool** *defaultVal*) **const**

Reads a bool value, returning `true` if the value was found. If the value was not found, *defaultVal* is used instead.

wxPython note: In place of a single overloaded method name, wxPython implements the following methods:

Read(key, default="")	Returns a string.
ReadInt(key, default=0)	Returns an int.
ReadFloat(key, default=0.0)	Returns a floating point number.

wxPerl note: In place of a single overloaded method, wxPerl uses:

Read(key, default="")	Returns a string
ReadInt(key, default=0)	Returns an integer
ReadFloat(key, default=0.0)	Returns a floating point number
ReadBool(key, default=0)	Returns a boolean

wxConfigBase::RenameEntry

bool RenameEntry(**const wxString&** *oldName*, **const wxString&** *newName*)

Renames an entry in the current group. The entries names (both the old and the new one) shouldn't contain backslashes, i.e. only simple names and not arbitrary paths are accepted by this function.

Returns `false` if *oldName* doesn't exist or if *newName* already exists.

wxConfigBase::RenameGroup

bool RenameGroup(**const wxString&** *oldName*, **const wxString&** *newName*)

Renames a subgroup of the current group. The subgroup names (both the old and the new one) shouldn't contain backslashes, i.e. only simple names and not arbitrary paths are accepted by this function.

Returns `false` if *oldName* doesn't exist or if *newName* already exists.

wxConfigBase::Set

static wxConfigBase * Set(**wxConfigBase** **pConfig*)

Sets the config object as the current one, returns the pointer to the previous current object (both the parameter and returned value may be NULL)

wxConfigBase::SetExpandEnvVars

void SetExpandEnvVars (**bool** *bDoIt = true*)

Determine whether we wish to expand environment variables in key values.

wxConfigBase::SetPath

void SetPath(**const wxString&** *strPath*)

Set current path: if the first character is '/', it is the absolute path, otherwise it is a relative path. '..' is supported. If strPath doesn't exist it is created.

wxConfigBase::SetRecordDefaults

void SetRecordDefaults(**bool** *bDoIt = true*)

Sets whether defaults are recorded to the config file whenever an attempt to read the value which is not present in it is done.

If on (default is off) all default values for the settings used by the program are written back to the config file. This allows the user to see what config options may be changed and is probably useful only for wxFileConfig.

wxConfigBase::Write

bool Write(**const wxString&** *key*, **const wxString&** *value*)

bool Write(**const wxString&** *key*, **long** *value*)

bool Write(**const wxString&** *key*, **double** *value*)

bool Write(**const wxString&** *key*, **bool** *value*)

These functions write the specified value to the config file and return `true` on success.

wxPython note: In place of a single overloaded method name, wxPython implements the following methods:

Write(key, value)	Writes a string.
WriteInt(key, value)	Writes an int.
WriteFloat(key, value)	Writes a floating point number.

wxPerl note: In place of a single overloaded method, wxPerl uses:

Write(key, value)	Writes a string
WriteInt(key, value)	Writes an integer
WriteFloat(key, value)	Writes a floating point number
WriteBool(key, value)	Writes a boolean

wxConnection

A wxConnection object represents the connection between a client and a server. It is created by making a connection using a *wxClient* (p. 145) object, or by the acceptance of a connection by a *wxServer* (p. 1167) object. The bulk of a DDE-like (Dynamic Data Exchange) conversation is controlled by calling members in a **wxConnection** object or by overriding its members. The actual DDE-based implementation using wxDDEConnection is available on Windows only, but a platform-independent, socket-based version of this API is available using wxTCPConnection, which has the same API.

An application should normally derive a new connection class from wxConnection, in order to override the communication event handlers to do something interesting.

Derived from

wxConnectionBase
wxObject (p. 1012)

Include files

<wx/ipc.h>

Types

wxIPCFormat is defined as follows:

```
enum wxIPCFormat
{
  wxIPC_INVALID =          0,
  wxIPC_TEXT =             1,  /* CF_TEXT */
  wxIPC_BITMAP =           2,  /* CF_BITMAP */
  wxIPC_METAFILE =         3,  /* CF_METAFILEPICT */
  wxIPC_SYLK =             4,
  wxIPC_DIF =              5,
  wxIPC_TIFF =             6,
  wxIPC_OEMTEXT =          7,  /* CF_OEMTEXT */
  wxIPC_DIB =              8,  /* CF_DIB */
  wxIPC_PALETTE =          9,
  wxIPC_PENDATA =          10,
  wxIPC_RIFF =             11,
  wxIPC_WAVE =             12,
  wxIPC_UNICODETEXT =      13,
  wxIPC_ENHMETAFILE =      14,
```

```
      wxIPC_FILENAME =          15, /* CF_HDROP */
      wxIPC_LOCALE =            16,
      wxIPC_PRIVATE =           20
    };
```

See also

wxClient (p. 145), *wxServer* (p. 1167),*Interprocess communications overview* (p. 1841)

wxConnection::wxConnection

wxConnection()

wxConnection(**char*** *buffer*, **int** *size*)

Constructs a connection object. If no user-defined connection object is to be derived from wxConnection, then the constructor should not be called directly, since the default connection object will be provided on requesting (or accepting) a connection. However, if the user defines his or her own derived connection object, the *wxServer::OnAcceptConnection* (p. 1168) and/or *wxClient::OnMakeConnection* (p. 147) members should be replaced by functions which construct the new connection object.

If the arguments of the wxConnection constructor are void then the wxConnection object manages its own connection buffer, allocating memory as needed. A programmer-supplied buffer cannot be increased if necessary, and the program will assert if it is not large enough. The programmer-supplied buffer is included mainly for backwards compatibility.

wxConnection::Advise

bool Advise(**const wxString&** *item*, **char*** *data*, **int** *size = -1*, **wxIPCFormat** *format = wxCF_TEXT*)

Called by the server application to advise the client of a change in the data associated with the given item. Causes the client connection's *wxConnection::OnAdvise* (p. 209) member to be called. Returns true if successful.

wxConnection::Execute

bool Execute(**char*** *data*, **int** *size = -1*, **wxIPCFormat** *format = wxCF_TEXT*)

Called by the client application to execute a command on the server. Can also be used to transfer arbitrary data to the server (similar to *wxConnection::Poke* (p. 210) in that respect). Causes the server connection's *wxConnection::OnExecute* (p. 209) member to be called. Returns true if successful.

wxConnection::Disconnect

bool Disconnect()

Called by the client or server application to disconnect from the other program; it causes the *wxConnection::OnDisconnect* (p. 209) message to be sent to the corresponding connection object in the other program. Returns true if successful or already disconnected. The application that calls **Disconnect** must explicitly delete its side of the connection.

wxConnection::OnAdvise

virtual bool OnAdvise(**const wxString&** *topic*, **const wxString&** *item*, **char*** *data*, **int** *size*, **wxIPCFormat** *format*)

Message sent to the client application when the server notifies it of a change in the data associated with the given item, using*Advise* (p. 208).

wxConnection::OnDisconnect

virtual bool OnDisconnect()

Message sent to the client or server application when the other application notifies it to end the connection. The default behaviour is to delete the connection object and return true, so applications should generally override **OnDisconnect**(finally calling the inherited method as well) so that they know the connection object is no longer available.

wxConnection::OnExecute

virtual bool OnExecute(**const wxString&** *topic*, **char*** *data*, **int** *size*, **wxIPCFormat** *format*)

Message sent to the server application when the client notifies it to execute the given data, using *Execute* (p. 208). Note that there is no item associated with this message.

wxConnection::OnPoke

virtual bool OnPoke(**const wxString&** *topic*, **const wxString&** *item*, **char*** *data*, **int** *size*, **wxIPCFormat** *format*)

Message sent to the server application when the client notifies it to accept the given data.

wxConnection::OnRequest

virtual char* OnRequest(**const wxString&** *topic*, **const wxString&** *item*, **int ****size*, **wxIPCFormat** *format*)

Message sent to the server application when the client calls *wxConnection::Request* (p.

210). The server's *OnRequest* (p. 209) method should respond by returning a character string, or NULL to indicate no data, and setting *size. The character string must of course persist after the call returns.

wxConnection::OnStartAdvise

virtual bool OnStartAdvise(**const wxString&** *topic*, **const wxString&** *item*)

Message sent to the server application by the client, when the client wishes to start an 'advise loop' for the given topic and item. The server can refuse to participate by returning false.

wxConnection::OnStopAdvise

virtual bool OnStopAdvise(**const wxString&** *topic*, **const wxString&** *item*)

Message sent to the server application by the client, when the client wishes to stop an 'advise loop' for the given topic and item. The server can refuse to stop the advise loop by returning false, although this doesn't have much meaning in practice.

wxConnection::Poke

bool Poke(**const wxString&** *item*, **char*** *data*, **int** *size = -1*, **wxIPCFormat** *format = wxCF_TEXT*)

Called by the client application to poke data into the server. Can be used to transfer arbitrary data to the server. Causes the server connection's *wxConnection::OnPoke* (p. 209) member to be called. If size is -1 the size is computed from the string length of data.

Returns true if successful.

wxConnection::Request

char* **Request**(**const wxString&** *item*, **int ***size*, **wxIPCFormat** *format = wxIPC_TEXT*)

Called by the client application to request data from the server. Causes the server connection's *wxConnection::OnRequest* (p. 209) member to be called. Size may be NULL or a pointer to a variable to receive the size of the requested item.

Returns a character string (actually a pointer to the connection's buffer) if successful, NULL otherwise. This buffer does not need to be deleted.

wxConnection::StartAdvise

bool StartAdvise(**const wxString&** *item*)

Called by the client application to ask if an advise loop can be started with the server. Causes the server connection's *wxConnection::OnStartAdvise* (p. 210) member to be called. Returns true if the server okays it, false otherwise.

wxConnection::StopAdvise

bool StopAdvise(**const wxString&** *item*)

Called by the client application to ask if an advise loop can be stopped. Causes the server connection's *wxConnection::OnStopAdvise* (p. 210) member to be called. Returns true if the server okays it, false otherwise.

wxContextMenuEvent

This class is used for context menu events, sent to give the application a chance to show a context (popup) menu.

Derived from

wxCommandEvent (p. 180)
wxEvent (p. 483)
wxObject (p. 1012)

Include files

<wx/event.h>

Event table macros

To process a menu event, use these event handler macros to direct input to member functions that take a wxContextMenuEvent argument.

EVT_CONTEXT_MENU(func)   A right click (or other context menu command depending on platform) has been detected.

See also

Command events (p. 180),
Event handling overview (p. 1751)

wxContextMenuEvent::wxContextMenuEvent

 wxContextMenuEvent(**WXTYPE** *id = 0*, **int** *id = 0*, **const wxPoint&** *pos=wxDefaultPosition*)

Constructor.

wxContextMenuEvent::GetPosition

wxPoint GetPosition() **const**

Returns the position at which the menu should be shown.

wxContextMenuEvent::SetPosition

void SetPosition(**const wxPoint&** *point*)

Sets the position at which the menu should be shown.

wxContextHelp

This class changes the cursor to a query and puts the application into a 'context-sensitive help mode'. When the user left-clicks on a window within the specified window, a wxEVT_HELP event is sent to that control, and the application may respond to it by popping up some help.

For example:

```
wxContextHelp contextHelp(myWindow);
```

There are a couple of ways to invoke this behaviour implicitly:

- Use the wxDIALOG_EX_CONTEXTHELP style for a dialog (Windows only). This will put a question mark in the titlebar, and Windows will put the application into context-sensitive help mode automatically, with further programming.

- Create a *wxContextHelpButton* (p. 213), whose predefined behaviour is to create a context help object. Normally you will write your application so that this button is only added to a dialog for non-Windows platforms (use wxDIALOG_EX_CONTEXTHELP on Windows).

Note that on Mac OS X, the cursor does not change when in context-sensitive help mode.

Derived from

wxObject (p. 1012)

Include files

<wx/cshelp.h>

See also

wxHelpEvent (p. 702), *wxHelpController* (p. 696), *wxContextHelpButton* (p. 213)

wxContextHelp::wxContextHelp

 wxContextHelp(**wxWindow*** *window = NULL*, **bool** *doNow = true*)

Constructs a context help object, calling *BeginContextHelp* (p. 213) if *doNow* is true (the default).

If *window* is NULL, the top window is used.

wxContextHelp::~wxContextHelp

 ~wxContextHelp()

Destroys the context help object.

wxContextHelp::BeginContextHelp

 bool BeginContextHelp(**wxWindow*** *window = NULL*)

Puts the application into context-sensitive help mode. *window* is the window which will be used to catch events; if NULL, the top window will be used.

Returns true if the application was successfully put into context-sensitive help mode. This function only returns when the event loop has finished.

wxContextHelp::EndContextHelp

 bool EndContextHelp()

Ends context-sensitive help mode. Not normally called by the application.

wxContextHelpButton

Instances of this class may be used to add a question mark button that when pressed, puts the application into context-help mode. It does this by creating a *wxContextHelp* (p. 212) object which itself generates a wxEVT_HELP event when the user clicks on a window.

On Windows, you may add a question-mark icon to a dialog by use of the wxDIALOG_EX_CONTEXTHELP extra style, but on other platforms you will have to add a button explicitly, usually next to OK, Cancel or similar buttons.

Derived from

wxBitmapButton (p. 92)
wxButton (p. 117)
wxControl (p. 215)
wxWindow (p. 1490)

wxEvtHandler (p. 486)
wxObject (p. 1012)

Include files

<wx/cshelp.h>

See also

wxBitmapButton (p. 92), *wxContextHelp* (p. 212)

wxContextHelpButton::wxContextHelpButton

wxContextHelpButton()

Default constructor.

wxContextHelpButton(**wxWindow*** *parent*, **wxWindowID** *id* = *wxID_CONTEXT_HELP*, **const wxPoint&** *pos* = *wxDefaultPosition*, **const wxSize&** *size* = *wxDefaultSize*, **long** *style* = *wxBU_AUTODRAW*)

Constructor, creating and showing a context help button.

Parameters

parent

  Parent window. Must not be NULL.

id

  Button identifier. Defaults to wxID_CONTEXT_HELP.

pos

  Button position.

size

  Button size. If the default size (-1, -1) is specified then the button is sized appropriately for the question mark bitmap.

style

  Window style.

Remarks

Normally you need pass only the parent window to the constructor, and use the defaults for the remaining parameters.

wxControl

This is the base class for a control or "widget".

A control is generally a small window which processes user input and/or displays one or more item of data.

Derived from

wxWindow (p. 1490)
wxEvtHandler (p. 486)
wxObject (p. 1012)

Include files

<wx/control.h>

See also

wxValidator (p. 1461)

wxControl::Command

void Command(**wxCommandEvent&** *event*)

Simulates the effect of the user issuing a command to the item. See *wxCommandEvent* (p. 180).

wxControl::GetLabel

wxString& GetLabel()

Returns the control's text.

wxControl::SetLabel

void SetLabel(**const wxString&** *label*)

Sets the item's text.

wxControlWithItems

This class is an abstract base class for some wxWidgets controls which contain several items, such as *wxListBox* (p. 847) and *wxCheckListBox* (p. 137) derived from it, *wxChoice* (p. 140) and *wxComboBox* (p. 172).

It defines the methods for accessing the controls items and although each of the derived classes implements them differently, they still all conform to the same interface.

The items in a wxControlWithItems have (non empty) string labels and, optionally, client data associated with them. Client data may be of two different kinds: either simple untyped (`void *`) pointers which are simply stored by the control but not used in any way by it, or typed pointers (`wxClientData *`) which are owned by the control meaning that the typed client data (and only it) will be deleted when an item is *deleted* (p. 217) or the entire control is *cleared* (p. 217) (which also happens when it is destroyed). Finally note that in the same control all items must have client data of the same type (typed or untyped), if any. This type is determined by the first call to *Append* (p. 216) (the version with client data pointer) or *SetClientData* (p. 221).

Derived from

wxControl (p. 215)
wxWindow (p. 1490)
wxEvtHandler (p. 486)
wxObject (p. 1012)

Include files

<wx/ctrlsub.h> but usually never included directly

wxControlWithItems::Append

int Append(**const wxString&** *item*)

Adds the item to the end of the list box.

int Append(**const wxString&** *item*, **void ****clientData*)

int Append(**const wxString&** *item*, **wxClientData ****clientData*)

Adds the item to the end of the list box, associating the given, typed or untyped, client data pointer with the item.

void Append(**const wxArrayString&** *strings*)

Appends several items at once to the control. Notice that calling this method may be much faster than appending the items one by one if you need to add a lot of items.

Parameters

item

    String to add.

clientData

    Client data to associate with the item.

Return value

When appending a single item, the return value is the index of the newly added item which may be different from the last one if the control is sorted (e.g. has `wxLB_SORT` or `wxCB_SORT` style).

wxControlWithItems::Clear

void Clear()

Removes all items from the control.

Clear() also deletes the client data of the existing items if it is owned by the control.

wxControlWithItems::Delete

void Delete(**int** *n*)

Deletes an item from the control. The client data associated with the item will be also deleted if it is owned by the control.

Note that it is an error (signalled by an assert failure in debug builds) to remove an item with the index negative or greater or equal than the number of items in the control.

Parameters

n

The zero-based item index.

See also

Clear (p. 217)

wxControlWithItems::FindString

int FindString(**const wxString&** *string*)

Finds an item whose label matches the given string.

Parameters

string

String to find.

Return value

The zero-based position of the item, or `wxNOT_FOUND` if the string was not found.

wxControlWithItems::GetClientData

void * GetClientData(**int** *n*) **const**

Returns a pointer to the client data associated with the given item (if any). It is an error to call this function for a control which doesn't have untyped client data at all although it is ok to call it even if the given item doesn't have any client data associated with it (but other items do).

Parameters

n

> The zero-based position of the item.

Return value

A pointer to the client data, or NULL if not present.

wxControlWithItems::GetClientObject

wxClientData * GetClientObject(**int** *n*) **const**

Returns a pointer to the client data associated with the given item (if any). It is an error to call this function for a control which doesn't have typed client data at all although it is ok to call it even if the given item doesn't have any client data associated with it (but other items do).

Parameters

n

> The zero-based position of the item.

Return value

A pointer to the client data, or NULL if not present.

wxControlWithItems::GetCount

int GetCount() **const**

Returns the number of items in the control.

See also

IsEmpty (p. 220)

wxControlWithItems::GetSelection

int GetSelection() **const**

Returns the index of the selected item or `wxNOT_FOUND` if no item is selected.

Return value

The position of the current selection.

Remarks

This method can be used with single selection list boxes only, you should use *wxListBox::GetSelections* (p. 850) for the list boxes with `wxLB_MULTIPLE` style.

See also

SetSelection (p. 221), *GetStringSelection* (p. 219)

wxControlWithItems::GetString

wxString GetString(**int** *n*) **const**

Returns the label of the item with the given index.

Parameters

n

> The zero-based index.

Return value

The label of the item or an empty string if the position was invalid.

wxControlWithItems::GetStringSelection

wxString GetStringSelection() **const**

Returns the label of the selected item or an empty string if no item is selected.

See also

GetSelection (p. 218)

wxControlWithItems::Insert

int Insert(**const wxString&** *item*, **int** *pos*)

Inserts the item into the list before pos. Not valid for `wxLB_SORT` or `wxCB_SORT` styles, use Append instead.

int Insert(**const wxString&** *item*, **int** *pos*, **void ****clientData*)

int Insert(**const wxString&** *item*, **int** *pos*, **wxClientData ****clientData*)

Inserts the item into the list before pos, associating the given, typed or untyped, client data pointer with the item. Not valid for `wxLB_SORT` or `wxCB_SORT` styles, use Append instead.

Parameters

item

> String to add.

pos

> Position to insert item before, zero based.

clientData

> Client data to associate with the item.

Return value

The return value is the index of the newly inserted item. If the insertion failed for some reason, -1 is returned.

wxControlWithItems::IsEmpty

bool IsEmpty() **const**

Returns `true` if the control is empty or `false` if it has some items.

See also

GetCount (p. 218)

wxControlWithItems::Number

int Number() **const**

Obsolescence note: This method is obsolete and was replaced with*GetCount* (p. 218), please use the new method in the new code. This method is only available if wxWidgets was compiled with `WXWIN_COMPATIBILITY_2_2` defined and will disappear completely in future versions.

wxControlWithItems::Select

void Select(**int** *n*)

This is the same as *SetSelection* (p. 221) and exists only because it is slightly more natural for controls which support multiple selection.

wxControlWithItems::SetClientData

void SetClientData(**int** *n*, **void ****data*)

Associates the given untyped client data pointer with the given item. Note that it is an error to call this function if any typed client data pointers had been associated with the control items before.

Parameters

n

> The zero-based item index.

data

> The client data to associate with the item.

wxControlWithItems::SetClientObject

void SetClientObject(**int** *n*, **wxClientData ****data*)

Associates the given typed client data pointer with the given item: the *data* object will be deleted when the item is deleted (either explicitly by using *Deletes* (p. 217) or implicitly when the control itself is destroyed).

Note that it is an error to call this function if any untyped client data pointers had been associated with the control items before.

Parameters

n

> The zero-based item index.

data

> The client data to associate with the item.

wxControlWithItems::SetSelection

void SetSelection(**int** *n*)

Sets the selection to the given item *n* or removes the selection entirely if *n* == `wxNOT_FOUND`.

Note that this does not cause any command events to be emitted nor does it deselect any other items in the controls which support multiple selections.

Parameters

n

The string position to select, starting from zero.

wxControlWithItems::SetString

void SetString(**int** *n*, **const wxString&** *string*)

Sets the label for the given item.

Parameters

n

The zero-based item index.

string

The label to set.

wxControlWithItems::SetStringSelection

bool SetStringSelection(**const wxString&** *string*)

Selects the item with the specified string in the control. This doesn't cause any command events being emitted.

Parameters

string

The string to select.

Return value

`true` if the specified string has been selected, `false` if it wasn't found in the control.

wxCountingOutputStream

wxCountingOutputStream is a specialized output stream which does not write any data anyway, instead it counts how many bytes would get written if this were a normal stream. This can sometimes be useful or required if some data gets serialized to a stream or compressed by using stream compression and thus the final size of the stream cannot be known other than pretending to write the stream. One case where the

resulting size would have to be known is if the data has to be written to a piece of memory and the memory has to be allocated before writing to it (which is probably always the case when writing to a memory stream).

Derived from

wxOutputStream (p. 1016)*wxStreamBase* (p. 1275)

Include files

<wx/stream.h>

wxCountingOutputStream::wxCountingOutputStream

wxCountingOutputStream()

Creates a wxCountingOutputStream object.

wxCountingOutputStream::~wxCountingOutputStream

~wxCountingOutputStream()

Destructor.

wxCountingOutputStream::GetSize

size_t GetSize() **const**

Returns the current size of the stream.

wxCriticalSection

A critical section object is used for exactly the same purpose as *mutexes* (p. 994). The only difference is that under Windows platform critical sections are only visible inside one process, while mutexes may be shared between processes, so using critical sections is slightly more efficient. The terminology is also slightly different: mutex may be locked (or acquired) and unlocked (or released) while critical section is entered and left by the program.

Finally, you should try to use *wxCriticalSectionLocker* (p. 224) class whenever possible instead of directly using wxCriticalSection for the same reasons *wxMutexLocker* (p. 997) is preferrable to *wxMutex* (p. 994) - please see wxMutex for an example.

Derived from

None.

Include files

<wx/thread.h>

See also

wxThread (p. 1377), *wxCondition* (p. 189), *wxCriticalSectionLocker* (p. 224)

wxCriticalSection::wxCriticalSection

 wxCriticalSection()

Default constructor initializes critical section object.

wxCriticalSection::~wxCriticalSection

 ~wxCriticalSection()

Destructor frees the resources.

wxCriticalSection::Enter

void Enter()

Enter the critical section (same as locking a mutex). There is no error return for this function. After entering the critical section protecting some global data the thread running in critical section may safely use/modify it.

wxCriticalSection::Leave

void Leave()

Leave the critical section allowing other threads use the global data protected by it. There is no error return for this function.

wxCriticalSectionLocker

This is a small helper class to be used with *wxCriticalSection* (p. 223) objects. A wxCriticalSectionLocker enters the critical section in the constructor and leaves it in the destructor making it much more difficult to forget to leave a critical section (which, in general, will lead to serious and difficult to debug problems).

Example of using it:

```
void Set Foo()
{
```

```
      // gs_critSect is some (global) critical section guarding access to
the
      // object "foo"
      wxCriticalSectionLocker locker(gs_critSect);

      if ( ... )
      {
          // do something
          ...

          return;
      }

      // do something else
      ...

      return;
}
```

Without wxCriticalSectionLocker, you would need to remember to manually leave the critical section before each return.

Derived from

None.

Include files

<wx/thread.h>

See also

wxCriticalSection (p. 223), *wxMutexLocker* (p. 997)

wxCriticalSectionLocker::wxCriticalSectionLocker

wxCriticalSectionLocker(**wxCriticalSection&** *criticalsection*)

Constructs a wxCriticalSectionLocker object associated with given*criticalsection* and enters it.

wxCriticalSectionLocker::~wxCriticalSectionLocker

~wxCriticalSectionLocker()

Destructor leaves the critical section.

wxCSConv

This class converts between any character sets and Unicode. It has one predefined

instance, **wxConvLocal**, for the default user character set.

Derived from

wxMBConv (p. 909)

Include files

<wx/strconv.h>

See also

wxMBConv (p. 909), *wxEncodingConverter* (p. 478), *wxMBConv classes overview* (p. 1733)

wxCSConv::wxCSConv

wxCSConv(**const wxChar*** *charset*)

wxCSConv(**wxFontEncoding** *encoding*)

Constructor. You may specify either the name of the character set you want to convert from/to or an encoding constant. If the character set name is not recognized, ISO 8859-1 is used as fall back.

wxCSConv::~wxCSConv

~wxCSConv()

Destructor frees any resources needed to perform the conversion.

wxCSConv::MB2WC

size_t MB2WC(**wchar_t*** *buf*, **const char*** *psz*, **size_t** *n*) **const**

Converts from the selected character set to Unicode. Returns length of string written to destination buffer.

wxCSConv::WC2MB

size_t WC2MB(**char*** *buf*, **const wchar_t*** *psz*, **size_t** *n*) **const**

Converts from Unicode to the selected character set. Returns length of string written to destination buffer.

wxCursor

A cursor is a small bitmap usually used for denoting where the mouse pointer is, with a picture that might indicate the interpretation of a mouse click. As with icons, cursors in X and MS Windows are created in a different manner. Therefore, separate cursors will be created for the different environments. Platform-specific methods for creating a **wxCursor** object are catered for, and this is an occasion where conditional compilation will probably be required (see *wxIcon* (p. 766) for an example).

A single cursor object may be used in many windows (any subwindow type). The wxWidgets convention is to set the cursor for a window, as in X, rather than to set it globally as in MS Windows, although a global *::wxSetCursor* (p. 1625) is also available for MS Windows use.

Derived from

wxBitmap (p. 79)
wxGDIObject (p. 607)
wxObject (p. 1012)

Include files

<wx/cursor.h>

Predefined objects

Objects:

wxNullCursor

Pointers:

wxSTANDARD_CURSOR
wxHOURGLASS_CURSOR
wxCROSS_CURSOR

See also

wxBitmap (p. 79), *wxIcon* (p. 766), *wxWindow::SetCursor* (p. 1530), *::wxSetCursor* (p. 1625)

wxCursor::wxCursor

wxCursor()

Default constructor.

wxCursor(**const char** *bits[]*, **int** *width*, **int** *height*, **int** *hotSpotX=-1*, **int** *hotSpotY=-1*, **const char** *maskBits[]=NULL*, **wxColour*** *fg=NULL*, **wxColour*** *bg=NULL*)

Constructs a cursor by passing an array of bits (Motif and GTK+ only). *maskBits* is used only under Motif and GTK+. The parameters *fg* and *bg* are only present on GTK+, and force the cursor to use particular background and foreground colours.

If either *hotSpotX* or *hotSpotY* is -1, the hotspot will be the centre of the cursor image (Motif only).

wxCursor(**const wxString&** *cursorName*, **long** *type*, **int** *hotSpotX=0*, **int** *hotSpotY=0*)

Constructs a cursor by passing a string resource name or filename.

On MacOS when specifying a string resource name, first the color cursors 'crsr' and then the black/white cursors 'CURS' in the resource chain are scanned through.

hotSpotX and *hotSpotY* are currently only used under Windows when loading from an icon file, to specify the cursor hotspot relative to the top left of the image.

wxCursor(**int** *cursorId*)

Constructs a cursor using a cursor identifier.

wxCursor(**const wxImage&** *image*)

Constructs a cursor from a wxImage. The cursor is monochrome, colors with the RGB elements all greater than 127 will be foreground, colors less than this background. The mask (if any) will be used as transparent.

In MSW the foreground will be white and the background black. If the cursor is larger than 32x32 it is resized. In GTK, the two most frequent colors will be used for foreground and background. The cursor will be displayed at the size of the image. On MacOS if the cursor is larger than 16x16 it is resized and currently only shown as black/white (mask respected).

wxCursor(**const wxCursor&** *cursor*)

Copy constructor. This uses reference counting so is a cheap operation.

Parameters

bits

> An array of bits.

maskBits

> Bits for a mask bitmap.

width

> Cursor width.

height

> Cursor height.

hotSpotX

> Hotspot x coordinate.

hotSpotY

Hotspot y coordinate.

type

Icon type to load. Under Motif, *type* defaults to **wxBITMAP_TYPE_XBM**. Under Windows, it defaults to **wxBITMAP_TYPE_CUR_RESOURCE**. Under MacOS, it defaults to **wxBITMAP_TYPE_MACCURSOR_RESOURCE**.

Under X, the permitted cursor types are:

wxBITMAP_TYPE_XBM   Load an X bitmap file.

Under Windows, the permitted types are:

wxBITMAP_TYPE_CUR   Load a cursor from a .cur cursor file (only if USE_RESOURCE_LOADING_IN_MSW is enabled in setup.h).

wxBITMAP_TYPE_CUR_RESOURCE   Load a Windows resource (as specified in the .rc file).

wxBITMAP_TYPE_ICO   Load a cursor from a .ico icon file (only if USE_RESOURCE_LOADING_IN_MSW is enabled in setup.h). Specify *hotSpotX* and *hotSpotY*.

cursorId

A stock cursor identifier. May be one of:

wxCURSOR_ARROW   A standard arrow cursor.

wxCURSOR_RIGHT_ARROW   A standard arrow cursor pointing to the right.

wxCURSOR_BLANK   Transparent cursor.

wxCURSOR_BULLSEYE   Bullseye cursor.

wxCURSOR_CHAR   Rectangular character cursor.

wxCURSOR_CROSS   A cross cursor.

wxCURSOR_HAND   A hand cursor.

wxCURSOR_IBEAM   An I-beam cursor (vertical line).

wxCURSOR_LEFT_BUTTON   Represents a mouse with the left button depressed.

wxCURSOR_MAGNIFIER   A magnifier icon.

wxCURSOR_MIDDLE_BUTTON   Represents a mouse with the middle button depressed.

wxCURSOR_NO_ENTRY	A no-entry sign cursor.
wxCURSOR_PAINT_BRUSH	A paintbrush cursor.
wxCURSOR_PENCIL	A pencil cursor.
wxCURSOR_POINT_LEFT	A cursor that points left.
wxCURSOR_POINT_RIGHT	A cursor that points right.
wxCURSOR_QUESTION_ARROW	An arrow and question mark.
wxCURSOR_RIGHT_BUTTON	Represents a mouse with the right button depressed.
wxCURSOR_SIZENESW	A sizing cursor pointing NE-SW.
wxCURSOR_SIZENS	A sizing cursor pointing N-S.
wxCURSOR_SIZENWSE	A sizing cursor pointing NW-SE.
wxCURSOR_SIZEWE	A sizing cursor pointing W-E.
wxCURSOR_SIZING	A general sizing cursor.
wxCURSOR_SPRAYCAN	A spraycan cursor.
wxCURSOR_WAIT	A wait cursor.
wxCURSOR_WATCH	A watch cursor.
wxCURSOR_ARROWWAIT	A cursor with both an arrow and an hourglass, (windows.)

Note that not all cursors are available on all platforms.

cursor

Pointer or reference to a cursor to copy.

wxPython note: Constructors supported by wxPython are:

wxCursor(name, flags, hotSpotX=0, hotSpotY=0)	Constructs a cursor from a filename
wxStockCursor(id)	Constructs a stock cursor

wxPerl note: Constructors supported by wxPerl are:

- ::Cursor->new(name, type, hotSpotX = 0, hotSpotY = 0)

- ::Cursor->new(id)

- ::Cursor->new(image)

•::Cursor->newData(bits, width, height, hotSpotX = -1, hotSpotY = -1, maskBits = 0)

Example

The following is an example of creating a cursor from 32x32 bitmap data (`down_bits`) and a mask (`down_mask`) where 1 is black and 0 is white for the bits, and 1 is opaque and 0 is transparent for the mask. It works on Windows and GTK+.

```
static char down_bits[] = { 255, 255, 255, 255, 31,
  255, 255, 255, 31, 255, 255, 255, 31, 255, 255, 255,
  31, 255, 255, 255, 31, 255, 255, 255, 31, 255, 255,
  255, 31, 255, 255, 255, 31, 255, 255, 255, 25, 243,
  255, 255, 19, 249, 255, 255, 7, 252, 255, 255, 15, 254,
  255, 255, 31, 255, 255, 255, 191, 255, 255, 255, 255,
  255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
  255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
  255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
  255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
  255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
  255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
  255 };

static char down_mask[] = { 240, 1, 0, 0, 240, 1,
  0, 0, 240, 1, 0, 0, 240, 1, 0, 0, 240, 1, 0, 0, 240, 1,
  0, 0, 240, 1, 0, 0, 240, 1, 0, 0, 255, 31, 0, 0, 255,
  31, 0, 0, 254, 15, 0, 0, 252, 7, 0, 0, 248, 3, 0, 0,
  240, 1, 0, 0, 224, 0, 0, 0, 64, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0 };

#ifdef __WXMSW__
wxBitmap down_bitmap(down_bits, 32, 32);
wxBitmap down_mask_bitmap(down_mask, 32, 32);

down_bitmap.SetMask(new wxMask(down_mask_bitmap));
wxImage down_image = down_bitmap.ConvertToImage();
down_image.SetOption(wxIMAGE_OPTION_CUR_HOTSPOT_X, 6);
down_image.SetOption(wxIMAGE_OPTION_CUR_HOTSPOT_Y, 14);
wxCursor down_cursor = wxCursor(down_image);
#else
wxCursor down_cursor = wxCursor(down_bits, 32, 32,
  6, 14, down_mask, wxWHITE, wxBLACK);
#endif
```

wxCursor::~wxCursor

~wxCursor()

Destroys the cursor. A cursor can be reused for more than one window, and does not get destroyed when the window is destroyed. wxWidgets destroys all cursors on application exit, although it is best to clean them up explicitly.

wxCursor::Ok

bool Ok() const

Returns true if cursor data is present.

wxCursor::operator =

wxCursor& operator =(const wxCursor& *cursor*)

Assignment operator, using reference counting. Returns a reference to 'this'.

wxCursor::operator ==

bool operator ==(const wxCursor& *cursor*)

Equality operator. Two cursors are equal if they contain pointers to the same underlying cursor data. It does not compare each attribute, so two independently-created cursors using the same parameters will fail the test.

wxCursor::operator !=

bool operator !=(const wxCursor& *cursor*)

Inequality operator. Two cursors are not equal if they contain pointers to different underlying cursor data. It does not compare each attribute.

wxCustomDataObject

wxCustomDataObject is a specialization of *wxDataObjectSimple* (p. 243) for some application-specific data in arbitrary (either custom or one of the standard ones). The only restriction is that it is supposed that this data can be copied bitwise (i.e. with `memcpy()`), so it would be a bad idea to make it contain a C++ object (though C struct is fine).

By default, wxCustomDataObject stores the data inside in a buffer. To put the data into the buffer you may use either *SetData* (p. 234) or *TakeData* (p. 234) depending on whether you want the object to make a copy of data or not.

If you already store the data in another place, it may be more convenient and efficient to provide the data on-demand which is possible too if you override the virtual functions mentioned below.

Virtual functions to override

This class may be used as is, but if you don't want store the data inside the object but provide it on demand instead, you should override *GetSize* (p. 233), *GetData* (p. 234) and *SetData* (p. 234) (or may be only the first two or only the last one if you only allow reading/writing the data)

Derived from

wxDataObjectSimple (p. 243)
wxDataObject (p. 239)

Include files

<wx/dataobj.h>

See also

wxDataObject (p. 239)

wxCustomDataObject::wxCustomDataObject

wxCustomDataObject(**const wxDataFormat&** *format* = *wxFormatInvalid*)

The constructor accepts a *format* argument which specifies the (single) format supported by this object. If it isn't set here, *SetFormat* (p. 244) should be used.

wxCustomDataObject::~wxCustomDataObject

~wxCustomDataObject()

The destructor will free the data hold by the object. Notice that although it calls a virtual *Free()* (p. 233) function, the base class version will always be called (C++ doesn't allow calling virtual functions from constructors or destructors), so if you override `Free()`, you should override the destructor in your class as well (which would probably just call the derived class' version of `Free()`).

wxCustomDataObject::Alloc

virtual void * Alloc(**size_t** *size*)

This function is called to allocate *size* bytes of memory from SetData(). The default version just uses the operator new.

wxCustomDataObject::Free

virtual void Free()

This function is called when the data is freed, you may override it to anything you want (or may be nothing at all). The default version calls operator delete[] on the data.

wxCustomDataObject::GetSize

virtual size_t GetSize() **const**

Returns the data size in bytes.

wxCustomDataObject::GetData

virtual void * GetData() **const**

Returns a pointer to the data.

wxCustomDataObject::SetData

virtual void SetData(**size_t** *size*, **const void** **data*)

Set the data. The data object will make an internal copy.

wxPython note: This method expects a string in wxPython. You can pass nearly any object by pickling it first.

wxCustomDataObject::TakeData

virtual void TakeData(**size_t** *size*, **const void** **data*)

Like *SetData* (p. 234), but doesn't copy the data - instead the object takes ownership of the pointer.

wxPython note: This method expects a string in wxPython. You can pass nearly any object by pickling it first.

wxDataFormat

A wxDataFormat is an encapsulation of a platform-specific format handle which is used by the system for the clipboard and drag and drop operations. The applications are usually only interested in, for example, pasting data from the clipboard only if the data is in a format the program understands and a data format is something which uniquely identifies this format.

On the system level, a data format is usually just a number (CLIPFORMATunder Windows or Atom under X11, for example) and the standard formats are, indeed, just numbers which can be implicitly converted to wxDataFormat. The standard formats are:

wxDF_INVALID	An invalid format - used as default argument for functions taking a wxDataFormat argument sometimes
wxDF_TEXT	Text format (wxString)
wxDF_BITMAP	A bitmap (wxBitmap)

wxDF_METAFILE	A metafile (wxMetafile, Windows only)
wxDF_FILENAME	A list of filenames
wxDF_HTML	An HTML string. This is only valid when passed to wxSetClipboardData when compiled with Visual C++ in non-Unicode mode

As mentioned above, these standard formats may be passed to any function taking wxDataFormat argument because wxDataFormat has an implicit conversion from them (or, to be precise from the type `wxDataFormat::NativeFormat` which is the type used by the underlying platform for data formats).

Aside the standard formats, the application may also use custom formats which are identified by their names (strings) and not numeric identifiers. Although internally custom format must be created (or *registered*) first, you shouldn't care about it because it is done automatically the first time the wxDataFormat object corresponding to a given format name is created. The only implication of this is that you should avoid having global wxDataFormat objects with non-default constructor because their constructors are executed before the program has time to perform all necessary initialisations and so an attempt to do clipboard format registration at this time will usually lead to a crash!

Virtual functions to override

None

Derived from

None

See also

Clipboard and drag and drop overview (p. 1817), *DnD sample* (p. 1710), *wxDataObject* (p. 239)

Include files

<wx/dataobj.h>

wxDataFormat::wxDataFormat

 wxDataFormat(**NativeFormat** *format* = *wxDF_INVALID*)

Constructs a data format object for one of the standard data formats or an empty data object (use *SetType* (p. 236) or *SetId* (p. 236) later in this case)

wxPerl note: In wxPerl this function is named `newNative`.

wxDataFormat::wxDataFormat

wxDataFormat(**const wxChar** **format*)

Constructs a data format object for a custom format identified by its name *format*.

wxPerl note: In wxPerl this function is named `newUser`.

wxDataFormat::operator ==

bool operator ==(**const wxDataFormat&** *format*) **const**

Returns true if the formats are equal.

wxDataFormat::operator !=

bool operator !=(**const wxDataFormat&** *format*) **const**

Returns true if the formats are different.

wxDataFormat::GetId

wxString GetId() **const**

Returns the name of a custom format (this function will fail for a standard format).

wxDataFormat::GetType

NativeFormat GetType() **const**

Returns the platform-specific number identifying the format.

wxDataFormat::SetId

void SetId(**const wxChar** **format*)

Sets the format to be the custom format identified by the given name.

wxDataFormat::SetType

void SetType(**NativeFormat** *format*)

Sets the format to the given value, which should be one of wxDF_XXX constants.

wxDataInputStream

This class provides functions that read binary data types in a portable way. Data can be

read in either big-endian or little-endian format, little-endian being the default on all architectures.

If you want to read data from text files (or streams) use *wxTextInputStream* (p. 1369) instead.

The >> operator is overloaded and you can use this class like a standard C++ iostream. Note, however, that the arguments are the fixed size types wxUint32, wxInt32 etc and on a typical 32-bit computer, none of these match to the "long" type (wxInt32 is defined as signed int on 32-bit architectures) so that you cannot use long. To avoid problems (here and elsewhere), make use of the wxInt32, wxUint32, etc types.

For example:

```
wxFileInputStream input( "mytext.dat" );
wxDataInputStream store( input );
wxUint8 i1;
float f2;
wxString line;

store >> i1;        // read a 8 bit integer.
store >> i1 >> f2; // read a 8 bit integer followed by float.
store >> line;      // read a text line
```

See also *wxDataOutputStream* (p. 245).

Derived from

None

Include files

<wx/datstrm.h>

wxDataInputStream::wxDataInputStream

 wxDataInputStream(**wxInputStream&** *stream*)

 wxDataInputStream(**wxInputStream&** *stream*, **wxMBConv&** *conv* = *wxMBConvUTF8*)

Constructs a datastream object from an input stream. Only read methods will be available. The second form is only available in Unicode build of wxWidgets.

Parameters

stream

> The input stream.

conv

> Charset conversion object object used to decode strings in Unicode mode (see *wxDataInputStream::ReadString* (p. 239)documentation for detailed description).

Note that you must not destroy*conv* before you destroy this wxDataInputStream instance!

wxDataInputStream::~wxDataInputStream

~wxDataInputStream()

Destroys the wxDataInputStream object.

wxDataInputStream::BigEndianOrdered

void BigEndianOrdered(**bool** *be_order*)

If *be_order* is true, all data will be read in big-endian order, such as written by programs on a big endian architecture (e.g. Sparc) or written by Java-Streams (which always use big-endian order). wxDataInputStream::Read8

wxUint8 Read8()

Reads a single byte from the stream.

void Read8(**wxUint8 ****buffer*, **size_t** *size*)

Reads bytes from the stream in a specified buffer. The amount of bytes to read is specified by the *size* variable.

wxDataInputStream::Read16

wxUint16 Read16()

Reads a 16 bit unsigned integer from the stream.

void Read16(**wxUint16 ****buffer*, **size_t** *size*)

Reads 16 bit unsigned integers from the stream in a specified buffer. the amount of 16 bit unsigned integer to read is specified by the *size* variable.

wxDataInputStream::Read32

wxUint32 Read32()

Reads a 32 bit unsigned integer from the stream.

void Read32(**wxUint32 ****buffer*, **size_t** *size*)

Reads 32 bit unsigned integers from the stream in a specified buffer. the amount of 32 bit unsigned integer to read is specified by the *size* variable.

wxDataInputStream::Read64

wxUint64 Read64()

Reads a 64 bit unsigned integer from the stream.

void Read64(**wxUint64 ****buffer*, **size_t** *size*)

Reads 64 bit unsigned integers from the stream in a specified buffer. the amount of 64 bit unsigned integer to read is specified by the *size* variable.

wxDataInputStream::ReadDouble

double ReadDouble()

Reads a double (IEEE encoded) from the stream.

void ReadDouble(**double ****buffer*, **size_t** *size*)

Reads double data (IEEE encoded) from the stream in a specified buffer. the amount of double to read is specified by the *size* variable.

wxDataInputStream::ReadString

wxString ReadString()

Reads a string from a stream. Actually, this function first reads a long integer specifying the length of the string (without the last null character) and then reads the string.

In Unicode build of wxWidgets, the fuction first reads multibyte (char*) string from the stream and then converts it to Unicode using the *conv*object passed to constructor and returns the result as wxString. You are responsible for using the same convertor as when writing the stream.

See also *wxDataOutputStream::WriteString* (p. 247).

wxDataObject

A wxDataObject represents data that can be copied to or from the clipboard, or dragged and dropped. The important thing about wxDataObject is that this is a 'smart' piece of data unlike 'dumb' data containers such as memory buffers or files. Being 'smart' here means that the data object itself should know what data formats it supports and how to render itself in each of its supported formats.

A supported format, incidentally, is exactly the format in which the data can be requested from a data object or from which the data object may be set. In the general case, an object may support different formats on 'input' and 'output', i.e. it may be able to render itself in a given format but not be created from data on this format or vice versa. wxDataObject defines an enumeration type

```
enum Direction
{
    Get  = 0x01,    // format is supported by GetDataHere()
    Set  = 0x02     // format is supported by SetData()
};
```

which distinguishes between them. See *wxDataFormat* (p. 234) documentation for more about formats.

Not surprisingly, being 'smart' comes at a price of added complexity. This is reasonable for the situations when you really need to support multiple formats, but may be annoying if you only want to do something simple like cut and paste text.

To provide a solution for both cases, wxWidgets has two predefined classes which derive from wxDataObject: *wxDataObjectSimple* (p. 243) and *wxDataObjectComposite* (p. 243). *wxDataObjectSimple* (p. 243) is the simplest wxDataObject possible and only holds data in a single format (such as HTML or text) and *wxDataObjectComposite* (p. 243) is the simplest way to implement a wxDataObject that does support multiple formats because it achieves this by simply holding several wxDataObjectSimple objects.

So, you have several solutions when you need a wxDataObject class (and you need one as soon as you want to transfer data via the clipboard or drag and drop):

1. Use one of the built-in classes   You may use wxTextDataObject, wxBitmapDataObject or wxFileDataObject in the simplest cases when you only need to support one format and your data is either text, bitmap or list of files.

2. Use wxDataObjectSimple   Deriving from wxDataObjectSimple is the simplest solution for custom data - you will only support one format and so probably won't be able to communicate with other programs, but data transfer will work in your program (or between different copies of it).

3. Use wxDataObjectComposite   This is a simple but powerful solution which allows you to support any number of formats (either standard or custom if you combine it with the previous solution).

4. Use wxDataObject directly   This is the solution for maximal flexibility and efficiency, but it is also the most difficult to implement.

Please note that the easiest way to use drag and drop and the clipboard with multiple formats is by using wxDataObjectComposite, but it is not the most efficient one as each wxDataObjectSimple would contain the whole data in its respective formats. Now imagine that you want to paste 200 pages of text in your proprietary format, as well as Word, RTF, HTML, Unicode and plain text to the clipboard and even today's computers are in trouble. For this case, you will have to derive from wxDataObject directly and make it enumerate its formats and provide the data in the requested format on demand.

Note that neither the GTK+ data transfer mechanisms for clipboard and drag and drop, nor OLE data transfer, copy any data until another application actually requests the data. This is in contrast to the 'feel' offered to the user of a program who would normally think that the data resides in the clipboard after having pressed 'Copy' - in reality it is only declared to be available.

There are several predefined data object classes derived from wxDataObjectSimple: *wxFileDataObject* (p. 510), *wxTextDataObject* (p. 1359) and *wxBitmapDataObject* (p. 97) which can be used without change.

You may also derive your own data object classes from *wxCustomDataObject* (p. 232) for user-defined types. The format of user-defined data is given as a mime-type string literal, such as "application/word" or "image/png". These strings are used as they are under Unix (so far only GTK+) to identify a format and are translated into their Windows equivalent under Win32 (using the OLE IDataObject for data exchange to and from the clipboard and for drag and drop). Note that the format string translation under Windows is not yet finished.

wxPython note: At this time this class is not directly usable from wxPython. Derive a class from *wxPyDataObjectSimple* (p. 243) instead.

wxPerl note: This class is not currently usable from wxPerl; you may use *Wx::PlDataObjectSimple* (p. 243) instead.

Virtual functions to override

Each class derived directly from wxDataObject must override and implement all of its functions which are pure virtual in the base class.

The data objects which only render their data or only set it (i.e. work in only one direction), should return 0 from *GetFormatCount* (p. 242).

Derived from

None

Include files

<wx/dataobj.h>

See also

Clipboard and drag and drop overview (p. 1817), *DnD sample* (p. 1710), *wxFileDataObject* (p. 510), *wxTextDataObject* (p. 1359), *wxBitmapDataObject* (p. 97), *wxCustomDataObject* (p. 232), *wxDropTarget* (p. 471), *wxDropSource* (p. 468), *wxTextDropTarget* (p. 1360), *wxFileDropTarget* (p. 516)

wxDataObject::wxDataObject

wxDataObject()

Constructor.

wxDataObject::~wxDataObject

~wxDataObject()

Destructor.

wxDataObject::GetAllFormats

virtual void GetAllFormats(**wxDataFormat** **formats*, **Direction** *dir = Get*) **const**

Copy all supported formats in the given direction to the array pointed to by *formats*. There is enough space for GetFormatCount(dir) formats in it.

wxPerl note: In wxPerl this method only takes the `dir` parameter. In scalar context it returns the first format, in list context it returns a list containing all the supported formats.

wxDataObject::GetDataHere

virtual bool GetDataHere(**const wxDataFormat&** *format*, **void** **buf*) **const**

The method will write the data of the format *format* in the buffer *buf* and return true on success, false on failure.

wxDataObject::GetDataSize

virtual size_t GetDataSize(**const wxDataFormat&** *format*) **const**

Returns the data size of the given format *format*.

wxDataObject::GetFormatCount

virtual size_t GetFormatCount(**Direction** *dir = Get*) **const**

Returns the number of available formats for rendering or setting the data.

wxDataObject::GetPreferredFormat

virtual wxDataFormat GetPreferredFormat(**Direction** *dir = Get*) **const**

Returns the preferred format for either rendering the data (if *dir* is `Get`, its default value) or for setting it. Usually this will be the native format of the wxDataObject.

wxDataObject::SetData

virtual bool SetData(**const wxDataFormat&** *format*, **size_t** *len*, **const void** **buf*)

Set the data in the format *format* of the length *len* provided in the buffer *buf*.

Returns true on success, false on failure.

wxDataObjectComposite

wxDataObjectComposite is the simplest *wxDataObject* (p. 239) derivation which may be used to support multiple formats. It contains several *wxDataObjectSimple* (p. 243) objects and supports any format supported by at least one of them. Only one of these data objects is*preferred* (the first one if not explicitly changed by using the second parameter of *Add* (p. 243)) and its format determines the preferred format of the composite data object as well.

See *wxDataObject* (p. 239) documentation for the reasons why you might prefer to use wxDataObject directly instead of wxDataObjectComposite for efficiency reasons.

Virtual functions to override

None, this class should be used directly.

Derived from

wxDataObject (p. 239)

Include files

<wx/dataobj.h>

See also

Clipboard and drag and drop overview (p. 1817), *wxDataObject* (p. 239), *wxDataObjectSimple* (p. 243), *wxFileDataObject* (p. 510), *wxTextDataObject* (p. 1359), *wxBitmapDataObject* (p. 97)

wxDataObjectComposite::wxDataObjectComposite

 wxDataObjectComposite()

The default constructor.

wxDataObjectComposite::Add

void Add(**wxDataObjectSimple** **dataObject*,   **bool** *preferred = false*)

Adds the *dataObject* to the list of supported objects and it becomes the preferred object if *preferred* is true.

wxDataObjectSimple

This is the simplest possible implementation of the *wxDataObject* (p. 239) class. The data object of (a class derived from) this class only supports one format, so the number

of virtual functions to be implemented is reduced.

Notice that this is still an abstract base class and cannot be used but should be derived from.

wxPython note: If you wish to create a derived wxDataObjectSimple class in wxPython you should derive the class from wxPyDataObjectSimple in order to get Python-aware capabilities for the various virtual methods.

wxPerl note: In wxPerl, you need to derive your data object class from Wx::PlDataObjectSimple.

Virtual functions to override

The objects supporting rendering the data must override *GetDataSize* (p. 245) and *GetDataHere* (p. 245) while the objects which may be set must override *SetData* (p. 245). Of course, the objects supporting both operations must override all three methods.

Derived from

wxDataObject (p. 239)

Include files

<wx/dataobj.h>

See also

Clipboard and drag and drop overview (p. 1817), *DnD sample* (p. 1710), *wxFileDataObject* (p. 510), *wxTextDataObject* (p. 1359), *wxBitmapDataObject* (p. 97)

wxDataObjectSimple::wxDataObjectSimple

 wxDataObjectSimple(**const wxDataFormat&** *format = wxFormatInvalid*)

Constructor accepts the supported format (none by default) which may also be set later with *SetFormat* (p. 244).

wxDataObjectSimple::GetFormat

const wxDataFormat& GetFormat() **const**

Returns the (one and only one) format supported by this object. It is supposed that the format is supported in both directions.

wxDataObjectSimple::SetFormat

void SetFormat(**const wxDataFormat&** *format*)

Sets the supported format.

wxDataObjectSimple::GetDataSize

virtual size_t GetDataSize() **const**

Gets the size of our data. Must be implemented in the derived class if the object supports rendering its data.

wxDataObjectSimple::GetDataHere

virtual bool GetDataHere(**void** **buf*) **const**

Copy the data to the buffer, return true on success. Must be implemented in the derived class if the object supports rendering its data.

wxPython note: When implementing this method in wxPython, no additional parameters are required and the data should be returned from the method as a string.

wxDataObjectSimple::SetData

virtual bool SetData(**size_t** *len*, **const void** **buf*)

Copy the data from the buffer, return true on success. Must be implemented in the derived class if the object supports setting its data.

wxPython note: When implementing this method in wxPython, the data comes as a single string parameter rather than the two shown here.

wxDataOutputStream

This class provides functions that write binary data types in a portable way. Data can be written in either big-endian or little-endian format, little-endian being the default on all architectures.

If you want to write data to text files (or streams) use *wxTextOutputStream* (p. 1372) instead.

The << operator is overloaded and you can use this class like a standard C++ iostream. See *wxDataInputStream* (p. 236) for its usage and caveats.

See also *wxDataInputStream* (p. 236).

Derived from

None

Include files

<wx/datstrm.h>

wxDataOutputStream::wxDataOutputStream

wxDataOutputStream(**wxOutputStream&** *stream*)

wxDataOutputStream(**wxOutputStream&** *stream*, **wxMBConv&** *conv* = *wxMBConvUTF8*)

Constructs a datastream object from an output stream. Only write methods will be available. The second form is only available in Unicode build of wxWidgets.

Parameters

stream

> The output stream.

conv

> Charset conversion object object used to encoding Unicode strings before writing them to the stream in Unicode mode (see *wxDataOutputStream::WriteString* (p. 247)documentation for detailed description). Note that you must not destroy*conv* before you destroy this wxDataOutputStream instance! It is recommended to use default value (UTF-8).

wxDataOutputStream::~wxDataOutputStream

~wxDataOutputStream()

Destroys the wxDataOutputStream object.

wxDataOutputStream::BigEndianOrdered

void BigEndianOrdered(**bool** *be_order*)

If *be_order* is true, all data will be written in big-endian order, e.g. for reading on a Sparc or from Java-Streams (which always use big-endian order), otherwise data will be written in little-endian order.
wxDataOutputStream::Write8

void Write8(**wxUint8** *i8*)

Writes the single byte *i8* to the stream.

void Write8(**const wxUint8 ****buffer*, **size_t** *size*)

Writes an array of bytes to the stream. The amount of bytes to write is specified with the *size* variable.

wxDataOutputStream::Write16

void Write16(**wxUint16** *i16*)

Writes the 16 bit unsigned integer *i16* to the stream.

void Write16(**const wxUint16 ****buffer*, **size_t** *size*)

Writes an array of 16 bit unsigned integer to the stream. The amount of 16 bit unsigned integer to write is specified with the *size* variable.

wxDataOutputStream::Write32

void Write32(**wxUint32** *i32*)

Writes the 32 bit unsigned integer *i32* to the stream.

void Write32(**const wxUint32 ****buffer*, **size_t** *size*)

Writes an array of 32 bit unsigned integer to the stream. The amount of 32 bit unsigned integer to write is specified with the *size* variable.

wxDataOutputStream::Write64

void Write64(**wxUint64** *i64*)

Writes the 64 bit unsigned integer *i64* to the stream.

void Write64(**const wxUint64 ****buffer*, **size_t** *size*)

Writes an array of 64 bit unsigned integer to the stream. The amount of 64 bit unsigned integer to write is specified with the *size* variable.

wxDataOutputStream::WriteDouble

void WriteDouble(**double** *f*)

Writes the double *f* to the stream using the IEEE format.

void WriteDouble(**const double ****buffer*, **size_t** *size*)

Writes an array of double to the stream. The amount of double to write is specified with the *size* variable.

wxDataOutputStream::WriteString

void WriteString(**const wxString&***string*)

Writes *string* to the stream. Actually, this method writes the size of the string before writing *string* itself.

In ANSI build of wxWidgets, the string is written to the stream in exactly same way it is represented in memory. In Unicode build, however, the string is first converted to multibyte representation with *conv* object passed to stream's constructor (consequently, ANSI application can read data written by Unicode application, as long as they agree on encoding) and this representation is written to the stream. UTF-8 is used by default.

wxDateEvent

This event class holds information about a date change and is used together with *wxDatePickerCtrl* (p. 248). It also serves as a base class for *wxCalendarEvent* (p. 130).

Derived from

wxCommandEvent (p. 180)
wxEvent (p. 483)
wxObject (p. 1012)

Include files

<wx/dateevt.h>

wxDateEvent::GetDate

const wxDateTime& GetDate() **const**

Returns the date.

wxDateEvent::SetDate

void SetDate(**const wxDateTime&** *date*)

Sets the date carried by the event, normally only used by the library internally.

wxDatePickerCtrl

This control allows the user to select a date. Unlike *wxCalendarCtrl* (p. 121), which is a relatively big control, wxDatePickerCtrl is implemented as a small window showing the currently selected date. The control can be edited using the keyboard, and can also display a popup window for more user-friendly date selection, depending on the styles used and the platform, except PalmOS where date is selected using native dialog.

It is only available if `wxUSE_DATEPICKCTRL` is set to 1.

Derived from

wxControl (p. 215)
wxWindow (p. 1490)
wxEvtHandler (p. 486)
wxObject (p. 1012)

Include files

<wx/datectrl.h>

(only available if `wxUSE_DATEPICKCTRL` is set to 1)

Window styles

wxDP_SPIN	Creates a control without month calendar drop down but with spin control-like arrows to change individual date components. This style is not supported by the generic version.
wxDP_DROPDOWN	Creates a control with a month calendar drop down part from which the user can select a date.
wxDP_DEFAULT	Creates a control with default style which is the best supported for the current platform (currently wxDP_SPIN under Windows and wxDP_DROPDOWN elsewhere).
wxDP_ALLOWNONE	With this style, the control allows the user to not enter any valid date at all. Without it -- which is by default -- the control always has some valid date.
wxDP_SHOWCENTURY	Forces display of the century in the default date format. Without this flas the century could be displayed or not depending on the default date representation in the system.

Event handling

EVT_DATE_CHANGED(id, func)	This event fires when the user changes the current selection in the control.

See also

wxCalendarCtrl (p. 121),
wxDateEvent (p. 248)

wxDatePickerCtrl::wxDatePickerCtrl

wxDatePickerCtrl(**wxWindow** **parent*, **wxWindowID** *id*, **const wxDateTime&** *dt* =

wxDefaultDateTime, **const wxPoint&** *pos = wxDefaultPosition*, **const wxSize&** *size = wxDefaultSize*, **long** *style = wxDP_DEFAULT | wxDP_SHOWCENTURY*, **const wxValidator&** *validator = wxDefaultValidator*,**const wxString&** *name = "datectrl"*)

Initializes the object and calls *Create* (p. 250) with all the parameters.

wxDatePickerCtrl::Create

bool Create(**wxWindow ****parent*, **wxWindowID** *id*, **const wxDateTime&** *dt = wxDefaultDateTime*, **const wxPoint&** *pos = wxDefaultPosition*, **const wxSize&** *size = wxDefaultSize*, **long** *style = wxDP_DEFAULT | wxDP_SHOWCENTURY*, **const wxValidator&** *validator = wxDefaultValidator*,**const wxString&** *name = "datectrl"*)

Parameters

parent

> Parent window, must not be non-NULL.

id

> The identifier for the control.

dt

> The initial value of the control, if an invalid date (such as the default value) is used, the control is set to today.

pos

> Initial position.

size

> Initial size. If left at default value, the control chooses its own best size by using the height approximately equal to a text control and width large enough to show the date string fully.

style

> The window style, should be left at 0 as there are no special styles for this control in this version.

validator

> Validator which can be used for additional date checks.

name

> Control name.

Return value

true if the control was successfully created or false if creation failed.

wxDatePickerCtrl::GetRange

bool GetRange(**wxDateTime** ******dt1*, **wxDateTime** **dt2*) **const**

If the control had been previously limited to a range of dates using *SetRange()* (p. 251), returns the lower and upper bounds of this range. If no range is set (or only one of the bounds is set), the *dt1* and/or *dt2* are set to be invalid.

Parameters

dt1

Pointer to the object which receives the lower range limit or becomes invalid if it is not set. May be `NULLL` if the caller is not interested in lower limit

dt2

Same as above but for the upper limit

Return value

`false` if no range limits are currently set, `true` if at least one bound is set.

wxDatePickerCtrl::GetValue

wxDateTime GetValue() **const**

Returns the currently selected. If there is no selection or the selection is outside of the current range, an invalid object is returned.

wxDatePickerCtrl::SetRange

void SetRange(**const wxDateTime&** *dt1*, **const wxDateTime&** *dt2*)

Sets the valid range for the date selection. If *dt1* is valid, it becomes the earliest date (inclusive) accepted by the control. If *dt2* is valid, it becomes the latest possible date.

Remarks

If the current value of the control is outside of the newly set range bounds, the behaviour is undefined.

wxDatePickerCtrl::SetValue

void SetValue(**const wxDateTime&** *dt*)

Changes the current value of the control. The date should be valid and included in the currently selected range, if any.

Calling this method does not result in a date change event.

wxDateSpan

This class is a "logical time span" and is useful for implementing program logic for such things as "add one month to the date" which, in general, doesn't mean to add 60*60*24*31 seconds to it, but to take the same date the next month (to understand that this is indeed different consider adding one month to Feb, 15 -- we want to get Mar, 15, of course).

When adding a month to the date, all lesser components (days, hours, ...) won't be changed unless the resulting date would be invalid: for example, Jan 31 + 1 month will be Feb 28, not (non existing) Feb 31.

Because of this feature, adding and subtracting back again the same wxDateSpan will **not**, in general give back the original date: Feb 28 - 1 month will be Jan 28, not Jan 31!

wxDateSpan objects can be either positive or negative. They may be multiplied by scalars which multiply all deltas by the scalar: i.e.2*(1 month and 1 day) is 2 months and 2 days. They can be added together and with *wxDateTime* (p. 257) or *wxTimeSpan* (p. 1389), but the type of result is different for each case.

Beware about weeks: if you specify both weeks and days, the total number of days added will be 7*weeks + days! See also GetTotalDays() function.

Equality operators are defined for wxDateSpans. Two datespans are equal if and only if they both give the same target date when added to **every** source date. Thus wxDateSpan::Months(1) is not equal to wxDateSpan::Days(30), because they don't give the same date when added to 1 Feb. But wxDateSpan::Days(14) is equal to wxDateSpan::Weeks(2)

Finally, notice that for adding hours, minutes and so on you don't need this class at all: *wxTimeSpan* (p. 1389) will do the job because there are no subtleties associated with those (we don't support leap seconds).

Derived from

No base class

Include files

<wx/datetime.h>

See also

Date classes overview (p. 1726), *wxDateTime* (p. 257)

wxDateSpan::wxDateSpan

 wxDateSpan(**int** *years = 0*, **int** *months = 0*, **int** *weeks = 0*, **int** *days = 0*)

Constructs the date span object for the given number of years, months, weeks and days.

Note that the weeks and days add together if both are given.

wxDateSpan::Add

wxDateSpan Add(**const wxDateSpan&** *other*) **const**

wxDateSpan& Add(**const wxDateSpan&** *other*)

wxDateSpan& operator+=(**const wxDateSpan&** *other*)

Returns the sum of two date spans. The first version returns a new object, the second and third ones modify this object in place.

wxDateSpan::Day

static wxDateSpan Day()

Returns a date span object corresponding to one day.

See also

Days (p. 253)

wxDateSpan::Days

static wxDateSpan Days(**int** *days*)

Returns a date span object corresponding to the given number of days.

See also

Day (p. 253)

wxDateSpan::GetDays

int GetDays() **const**

Returns the number of days (only, that it not counting the weeks component!) in this date span.

See also

GetTotalDays (p. 254)

wxDateSpan::GetMonths

int GetMonths() **const**

Returns the number of the months (not counting the years) in this date span.

wxDateSpan::GetTotalDays

int GetTotalDays() **const**

Returns the combined number of days in this date span, counting both weeks and days. It still doesn't take neither months nor years into the account.

See also

GetWeeks (p. 254), *GetDays* (p. 253)

wxDateSpan::GetWeeks

int GetWeeks() **const**

Returns the number of weeks in this date span.

See also

GetTotalDays (p. 254)

wxDateSpan::GetYears

int GetYears() **const**

Returns the number of years in this date span.

wxDateSpan::Month

static wxDateSpan Month()

Returns a date span object corresponding to one month.

See also

Months (p. 254)

wxDateSpan::Months

static wxDateSpan Months(**int** *mon*)

Returns a date span object corresponding to the given number of months.

See also

Month (p. 254)

wxDateSpan::Multiply

wxDateSpan Multiply(**int** *factor*) **const**

wxDateSpan& Multiply(**int** *factor*)

wxDateSpan& operator*=(**int** *factor*)

Returns the product of the date span by the specified *factor*. The product is computed by multiplying each of the components by the factor.

The first version returns a new object, the second and third ones modify this object in place.

wxDateSpan::Negate

wxDateSpan Negate() **const**

Returns the date span with the opposite sign.

See also

Neg (p. 255)

wxDateSpan::Neg

wxDateSpan& Neg()

wxDateSpan& operator-()

Changes the sign of this date span.

See also

Negate (p. 255)

wxDateSpan::SetDays

wxDateSpan& SetDays(**int** *n*)

Sets the number of days (without modifying any other components) in this date span.

wxDateSpan::SetYears

wxDateSpan& SetYears(**int** *n*)

Sets the number of years (without modifying any other components) in this date span.

wxDateSpan::SetMonths

wxDateSpan& SetMonths(**int** *n*)

Sets the number of months (without modifying any other components) in this date span.

wxDateSpan::SetWeeks

wxDateSpan& SetWeeks(**int** *n*)

Sets the number of weeks (without modifying any other components) in this date span.

wxDateSpan::Subtract

wxDateSpan Subtract(**const wxDateSpan&** *other*) **const**

wxDateSpan& Subtract(**const wxDateSpan&** *other*)

wxDateSpan& operator+=(**const wxDateSpan&** *other*)

Returns the difference of two date spans. The first version returns a new object, the second and third ones modify this object in place.

wxDateSpan::Week

static wxDateSpan Week()

Returns a date span object corresponding to one week.

See also

Weeks (p. 256)

wxDateSpan::Weeks

static wxDateSpan Weeks(**int** *weeks*)

Returns a date span object corresponding to the given number of weeks.

See also

Week (p. 256)

wxDateSpan::Year

static wxDateSpan Year()

Returns a date span object corresponding to one year.

See also

Years (p. 257)

wxDateSpan::Years

static wxDateSpan Years(**int** *years*)

Returns a date span object corresponding to the given number of years.

See also

Year (p. 256)

wxDateSpan::operator==

bool operator==(**wxDateSpan&** *other*) **const**

Returns `true` if this date span is equal to the other one. Two date spans are considered equal if and only if they have the same number of years and months and the same total number of days (counting both days and weeks).

wxDateSpan::operator!=

bool operator!=(**wxDateSpan&** *other*) **const**

Returns `true` if this date span is different from the other one.

See also

operator== (p. 257)

wxDateTime

wxDateTime class represents an absolute moment in the time.

Types

The type `wxDateTime_t` is typedefed as `unsigned short` and is used to contain the number of years, hours, minutes, seconds and milliseconds.

Constants

Global constant `wxDefaultDateTime` and synonym for it `wxInvalidDateTime` are defined. This constant will be different from any valid wxDateTime object.

All the following constants are defined inside wxDateTime class (i.e., to refer to them you should prepend their names with `wxDateTime::`).

Time zone symbolic names:

```
    enum TZ
    {
        // the time in the current time zone
        Local,

        // zones from GMT (= Greenwhich Mean Time): they're guaranteed to
be
        // consequent numbers, so writing something like `GMT0 + offset'
is
        // safe if abs(offset) <= 12

        // underscore stands for minus
        GMT_12, GMT_11, GMT_10, GMT_9, GMT_8, GMT_7,
        GMT_6, GMT_5, GMT_4, GMT_3, GMT_2, GMT_1,
        GMT0,
        GMT1, GMT2, GMT3, GMT4, GMT5, GMT6,
        GMT7, GMT8, GMT9, GMT10, GMT11, GMT12,
        // Note that GMT12 and GMT_12 are not the same: there is a
difference
        // of exactly one day between them

        // some symbolic names for TZ

        // Europe
        WET = GMT0,                         // Western Europe Time
        WEST = GMT1,                        // Western Europe Summer Time
        CET = GMT1,                         // Central Europe Time
        CEST = GMT2,                        // Central Europe Summer Time
        EET = GMT2,                         // Eastern Europe Time
        EEST = GMT3,                        // Eastern Europe Summer Time
        MSK = GMT3,                         // Moscow Time
        MSD = GMT4,                         // Moscow Summer Time

        // US and Canada
        AST = GMT_4,                        // Atlantic Standard Time
        ADT = GMT_3,                        // Atlantic Daylight Time
        EST = GMT_5,                        // Eastern Standard Time
        EDT = GMT_4,                        // Eastern Daylight Saving
Time
        CST = GMT_6,                        // Central Standard Time
        CDT = GMT_5,                        // Central Daylight Saving
Time
        MST = GMT_7,                        // Mountain Standard Time
        MDT = GMT_6,                        // Mountain Daylight Saving
Time
        PST = GMT_8,                        // Pacific Standard Time
        PDT = GMT_7,                        // Pacific Daylight Saving
Time
        HST = GMT_10,                       // Hawaiian Standard Time
        AKST = GMT_9,                       // Alaska Standard Time
        AKDT = GMT_8,                       // Alaska Daylight Saving Time

        // Australia

        A_WST = GMT8,                       // Western Standard Time
        A_CST = GMT12 + 1,                  // Central Standard Time
(+9.5)
        A_EST = GMT10,                      // Eastern Standard Time
        A_ESST = GMT11,                     // Eastern Summer Time

        // Universal Coordinated Time = the new and politically correct
```

```
name
        // for GMT
        UTC = GMT0
    };
```

Month names: Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec and
Inv_Month for an invalid.month value are the values of `wxDateTime::Month`enum.

Likewise, Sun, Mon, Tue, Wed, Thu, Fri, Sat, and Inv_WeekDay are the values
in`wxDateTime::WeekDay` enum.

Finally, Inv_Year is defined to be an invalid value for year parameter.

GetMonthName() (p. 266) and*GetWeekDayName* (p. 267) functions use the followign
flags:

```
enum NameFlags
{
    Name_Full = 0x01,        // return full name
    Name_Abbr = 0x02         // return abbreviated name
};
```

Several functions accept an extra parameter specifying the calendar to use (although
most of them only support now the Gregorian calendar). This parameters is one of the
following values:

```
enum Calendar
{
    Gregorian,  // calendar currently in use in Western countries
    Julian      // calendar in use since -45 until the 1582 (or later)
};
```

Date calculations often depend on the country and wxDateTime allows to set the country
whose conventions should be used using*SetCountry* (p. 268). It takes one of the
following values as parameter:

```
enum Country
{
    Country_Unknown, // no special information for this country
    Country_Default, // set the default country with SetCountry()
method
                     // or use the default country with any other

    Country_WesternEurope_Start,
    Country_EEC = Country_WesternEurope_Start,
    France,
    Germany,
    UK,
    Country_WesternEurope_End = UK,

    Russia,

    USA
};
```

Different parts of the world use different conventions for the week start. In some
countries, the week starts on Sunday, while in others -- on Monday. The ISO standard
doesn't address this issue, so we support both conventions in the functions whose result

depends on it (*GetWeekOfYear* (p. 274) and*GetWeekOfMonth* (p. 274)).

The desired behvaiour may be specified by giving one of the following constants as argument to these functions:

```
enum WeekFlags
{
    Default_First,    // Sunday_First for US, Monday_First for the rest
    Monday_First,     // week starts with a Monday
    Sunday_First      // week starts with a Sunday
};
```

Derived from

No base class

Include files

<wx/datetime.h>

See also

Date classes overview (p. 1726), *wxTimeSpan* (p. 1389), *wxDateSpan* (p. 252), *wxCalendarCtrl* (p. 121)

Static functions

For convenience, all static functions are collected here. These functions either set or return the static variables of wxDateSpan (the country), return the current moment, year, month or number of days in it, or do some general calendar-related actions.

Please note that although several function accept an extra *Calendar*parameter, it is currently ignored as only the Gregorian calendar is supported. Future versions will support other calendars.

wxPython note: These methods are standalone functions named`wxDateTime_<StaticMethodName>` in wxPython.

SetCountry (p. 268)
GetCountry (p. 265)
IsWestEuropeanCountry (p. 267)
GetCurrentYear (p. 265)
ConvertYearToBC (p. 264)
GetCurrentMonth (p. 265)
IsLeapYear (p. 267)
GetCentury (p. 266)
GetNumberOfDays (p. 266)
GetNumberOfDays (p. 266)
GetMonthName (p. 266)
GetWeekDayName (p. 267)
GetAmPmStrings (p. 265)

Constructors, assignment operators and setters

Constructors and various `Set()` methods are collected here. If you construct a date object from separate values for day, month and year, you should use *IsValid* (p. 272) method to check that the values were correct as constructors can not return an error code.

Accessors

Here are the trivial accessors. Other functions, which might have to perform some more complicated calculations to find the answer are under the*Calendar calculations* (p. 263) section.

Date comparison

There are several function to allow date comparison. To supplement them, a few global operators >, < etc taking wxDateTime are defined.

Date arithmetics

These functions carry out *arithmetics* (p. 1728) on the wxDateTime objects. As explained in the overview, either wxTimeSpan or wxDateSpan may be added to wxDateTime, hence all functions are overloaded to accept both arguments.

Also, both `Add()` and `Subtract()` have both const and non-const version. The first one returns a new object which represents the sum/difference of the original one with the argument while the second form modifies the object to which it is applied. The operators -= and += are defined to be equivalent to the second forms of these functions.

Parsing and formatting dates

These functions convert wxDateTime objects to and from text. The conversions to text are mostly trivial: you can either do it using the default date and time representations for the current locale (*FormatDate* (p. 279) and*FormatTime* (p. 279)), using the international standard representation defined by ISO 8601 (*FormatISODate* (p. 279) and*FormatISOTime* (p. 280)) or by specifying any format at all and using *Format* (p. 279) directly.

The conversions from text are more interesting, as there are much more possibilities to care about. The simplest cases can be taken care of with*ParseFormat* (p. 278) which can parse any date in the given (rigid) format. *ParseRfc822Date* (p. 277) is another function for parsing dates in predefined format -- the one of RFC 822 which (still...) defines the format of email messages on the Internet. This format can not be described with `strptime(3)`-like format strings used by*Format* (p. 279), hence the need for a separate function.

But the most interesting functions are*ParseTime* (p. 279),*ParseDate* (p. 278) and*ParseDateTime* (p. 278). They try to parse the date ans time (or only one of them) in 'free' format, i.e. allow them to be specified in any of possible ways. These functions will usually be used to parse the (interactive) user input which is not bound to be in any predefined format. As an example, *ParseDateTime* (p. 278) can parse the strings such as `"tomorrow"`, `"March first"` and even`"next Sunday"`.

ParseRfc822Date (p. 277)
ParseFormat (p. 278)
ParseDateTime (p. 278)
ParseDate (p. 278)
ParseTime (p. 279)
Format (p. 279)
FormatDate (p. 279)
FormatTime (p. 279)
FormatISODate (p. 279)
FormatISOTime (p. 280)

Calendar calculations

The functions in this section perform the basic calendar calculations, mostly related to the week days. They allow to find the given week day in the week with given number (either in the month or in the year) and so on.

All (non-const) functions in this section don't modify the time part of the wxDateTime -- they only work with the date part of it.

SetToWeekDayInSameWeek (p. 280)
GetWeekDayInSameWeek (p. 280)
SetToNextWeekDay (p. 280)
GetNextWeekDay (p. 280)
SetToPrevWeekDay (p. 280)
GetPrevWeekDay (p. 281)
SetToWeekDay (p. 281)

Astronomical/historical functions

Some degree of support for the date units used in astronomy and/or history is provided. You can construct a wxDateTime object from a *JDN* (p. 270) and you may also get its JDN, *MJD* (p. 283) or *Rata Die number* (p. 283) from it.

Time zone and DST support

Please see the *time zone overview* (p. 1729) for more information about time zones. Normally, these functions should be rarely used.

wxDateTime::ConvertYearToBC

static int ConvertYearToBC(**int** *year*)

Converts the year in absolute notation (i.e. a number which can be negative, positive or zero) to the year in BC/AD notation. For the positive years, nothing is done, but the year 0 is year 1 BC and so for other years there is a difference of 1.

This function should be used like this:

```
wxDateTime dt(...);
```

```
        int y = dt.GetYear();
        printf("The year is %d%s", wxDateTime::ConvertYearToBC(y), y > 0 ?
"AD" : "BC");
```

wxDateTime::GetAmPmStrings

static void GetAmPmStrings(**wxString ****am***, **wxString ****pm***)

Returns the translations of the strings AM and PM used for time formatting for the current locale. Either of the pointers may be NULL if the corresponding value is not needed.

wxDateTime::GetBeginDST

static wxDateTime GetBeginDST(**int** *year = Inv_Year*, **Country** *country = Country_Default*)

Get the beginning of DST for the given country in the given year (current one by default). This function suffers from limitations described in*DST overview* (p. 1729).

See also

GetEndDST (p. 266)

wxDateTime::GetCountry

static Country GetCountry()

Returns the current default country. The default country is used for DST calculations, for example.

See also

SetCountry (p. 268)

wxDateTime::GetCurrentYear

static int GetCurrentYear(**Calendar** *cal = Gregorian*)

Get the current year in given calendar (only Gregorian is currently supported).

wxDateTime::GetCurrentMonth

static Month GetCurrentMonth(**Calendar** *cal = Gregorian*)

Get the current month in given calendar (only Gregorian is currently supported).

wxDateTime::GetCentury

static int GetCentury(**int** *year = Inv_Year*)

Get the current century, i.e. first two digits of the year, in given calendar (only Gregorian is currently supported).

wxDateTime::GetEndDST

static wxDateTime GetEndDST(**int** *year = Inv_Year*, **Country** *country = Country_Default*)

Returns the end of DST for the given country in the given year (current one by default).

See also

GetBeginDST (p. 265)

wxDateTime::GetMonthName

static wxString GetMonthName(**Month** *month*, **NameFlags** *flags = Name_Full*)

Gets the full (default) or abbreviated (specify `Name_Abbr` name of the given month.

See also

GetWeekDayName (p. 267)

wxDateTime::GetNumberOfDays

static wxDateTime_t GetNumberOfDays(**int** *year*, **Calendar** *cal = Gregorian*)

static wxDateTime_t GetNumberOfDays(**Month** *month*, **int** *year = Inv_Year*, **Calendar** *cal = Gregorian*)

Returns the number of days in the given year or in the given month of the year.

The only supported value for *cal* parameter is currently `Gregorian`.

wxPython note: These two methods are named `GetNumberOfDaysInYear`and `GetNumberOfDaysInMonth` in wxPython.

wxDateTime::GetTimeNow

static time_t GetTimeNow()

Returns the current time.

wxDateTime::GetTmNow

static struct tm * GetTmNow()

Returns the current time broken down.

wxDateTime::GetWeekDayName

static wxString GetWeekDayName(**WeekDay** *weekday*, **NameFlags** *flags* = *Name_Full*)

Gets the full (default) or abbreviated (specify `Name_Abbr` name of the given week day.

See also

GetMonthName (p. 266)

wxDateTime::IsLeapYear

static bool IsLeapYear(**int** *year* = *Inv_Year*, **Calendar** *cal* = *Gregorian*)

Returns `true` if the *year* is a leap one in the specified calendar.

This functions supports Gregorian and Julian calendars.

wxDateTime::IsWestEuropeanCountry

static bool IsWestEuropeanCountry(**Country** *country* = *Country_Default*)

This function returns `true` if the specified (or default) country is one of Western European ones. It is used internally by wxDateTime to determine the DST convention and date and time formatting rules.

wxDateTime::IsDSTApplicable

static bool IsDSTApplicable(**int** *year* = *Inv_Year*, **Country** *country* = *Country_Default*)

Returns `true` if DST was used n the given year (the current one by default) in the given country.

wxDateTime::Now

static wxDateTime Now()

Returns the object corresponding to the current time.

Example:

```
       wxDateTime now = wxDateTime::Now();
       printf("Current time in Paris:\t%s\n", now.Format("%c",
     wxDateTime::CET).c_str());
```

Note that this function is accurate up to second:*wxDateTime::UNow* (p. 268) should be used for better precision (but it is less efficient and might not be available on all platforms).

See also

Today (p. 268)

wxDateTime::SetCountry

static void SetCountry(Country *country*)

Sets the country to use by default. This setting influences the DST calculations, date formatting and other things.

The possible values for *country* parameter are enumerated in*wxDateTime constants section* (p. 257).

See also

GetCountry (p. 265)

wxDateTime::Today

static wxDateTime Today()

Returns the object corresponding to the midnight of the current day (i.e. the same as *Now()* (p. 267), but the time part is set to 0).

See also

Now (p. 267)

wxDateTime::UNow

static wxDateTime UNow()

Returns the object corresponding to the current time including the milliseconds if a function to get time with such precision is available on the current platform (supported under most Unices and Win32).

See also

Now (p. 267)

wxDateTime::wxDateTime

wxDateTime()

Default constructor. Use one of `Set()` functions to initialize the object later.

wxDateTime::wxDateTime

wxDateTime& wxDateTime(**time_t** *timet*)

Same as *Set* (p. 269).

wxPython note: This constructor is named `wxDateTimeFromTimeT` in wxPython.

wxDateTime::wxDateTime

wxDateTime& wxDateTime(**const struct tm&** *tm*)

Same as *Set* (p. 269)

wxPython note: Unsupported.

wxDateTime::wxDateTime

wxDateTime& wxDateTime(**double** *jdn*)

Same as *Set* (p. 269)

wxPython note: This constructor is named `wxDateTimeFromJDN` in wxPython.

wxDateTime::wxDateTime

wxDateTime& wxDateTime(**wxDateTime_t** *hour*, **wxDateTime_t** *minute = 0*,
wxDateTime_t *second = 0*, **wxDateTime_t** *millisec = 0*)

Same as *Set* (p. 269)

wxPython note: This constructor is named `wxDateTimeFromHMS` in wxPython.

wxDateTime::wxDateTime

wxDateTime& wxDateTime(**wxDateTime_t** *day*, **Month** *month = Inv_Month*, **int**
Inv_Year,**wxDateTime_t** *hour = 0*, **wxDateTime_t** *minute = 0*, **wxDateTime_t** *second =
0*, **wxDateTime_t** *millisec = 0*)

Same as *Set* (p. 271)

wxPython note: This constructor is named `wxDateTimeFromDMY` in wxPython.

wxDateTime::SetToCurrent

wxDateTime& SetToCurrent()

Sets the date and time of to the current values. Same as assigning the result of *Now()* (p. 267) to this object.

wxDateTime::Set

wxDateTime& Set(**time_t** *timet*)

Constructs the object from *timet* value holding the number of seconds since Jan 1, 1970.

wxPython note: This method is named `SetTimeT` in wxPython.

wxDateTime::Set

wxDateTime& Set(**const struct tm&** *tm*)

Sets the date and time from the broken down representation in the standard `tm` structure.

wxPython note: Unsupported.

wxDateTime::Set

wxDateTime& Set(**double** *jdn*)

Sets the date from the so-called *Julian Day Number*.

By definition, the Julian Day Number, usually abbreviated as JDN, of a particular instant is the fractional number of days since 12 hours Universal Coordinated Time (Greenwich mean noon) on January 1 of the year -4712 in the Julian proleptic calendar.

wxPython note: This method is named `SetJDN` in wxPython.

wxDateTime::Set

wxDateTime& Set(**wxDateTime_t** *hour*, **wxDateTime_t** *minute = 0*, **wxDateTime_t** *second = 0*, **wxDateTime_t** *millisec = 0*)

Sets the date to be equal to *Today* (p. 268) and the time from supplied parameters.

wxPython note: This method is named `SetHMS` in wxPython.

wxDateTime::Set

wxDateTime& Set(**wxDateTime_t** *day*, **Month** *month = Inv_Month*, **int** *year = Inv_Year*, **wxDateTime_t** *hour = 0*, **wxDateTime_t** *minute = 0*, **wxDateTime_t** *second = 0*, **wxDateTime_t** *millisec = 0*)

Sets the date and time from the parameters.

wxDateTime::ResetTime

wxDateTime& ResetTime()

Reset time to midnight (00:00:00) without changing the date.

wxDateTime::SetYear

wxDateTime& SetYear(**int** *year*)

Sets the year without changing other date components.

wxDateTime::SetMonth

wxDateTime& SetMonth(**Month** *month*)

Sets the month without changing other date components.

wxDateTime::SetDay

wxDateTime& SetDay(**wxDateTime_t** *day*)

Sets the day without changing other date components.

wxDateTime::SetHour

wxDateTime& SetHour(**wxDateTime_t** *hour*)

Sets the hour without changing other date components.

wxDateTime::SetMinute

wxDateTime& SetMinute(**wxDateTime_t** *minute*)

Sets the minute without changing other date components.

wxDateTime::SetSecond

wxDateTime& SetSecond(**wxDateTime_t** *second*)

Sets the second without changing other date components.

wxDateTime::SetMillisecond

wxDateTime& SetMillisecond(**wxDateTime_t** *millisecond*)

Sets the millisecond without changing other date components.

wxDateTime::operator=

wxDateTime& operator(**time_t** *timet*)

Same as *Set* (p. 270).

wxDateTime::operator=

wxDateTime& operator(**const struct tm&** *tm*)

Same as *Set* (p. 270).

wxDateTime::IsValid

bool IsValid() **const**

Returns `true` if the object represents a valid time moment.

wxDateTime::GetTm

Tm GetTm(**const TimeZone&** *tz = Local*) **const**

Returns broken down representation of the date and time.

wxDateTime::GetTicks

time_t GetTicks() **const**

Returns the number of seconds since Jan 1, 1970. An assert failure will occur if the date is not in the range covered by `time_t` type.

wxDateTime::GetYear

int GetYear(**const TimeZone&** *tz = Local*) **const**

Returns the year in the given timezone (local one by default).

wxDateTime::GetMonth

Month GetMonth(**const TimeZone&** *tz = Local*) **const**

Returns the month in the given timezone (local one by default).

wxDateTime::GetDay

wxDateTime_t GetDay(**const TimeZone&** *tz = Local*) **const**

Returns the day in the given timezone (local one by default).

wxDateTime::GetWeekDay

WeekDay GetWeekDay(**const TimeZone&** *tz = Local*) **const**

Returns the week day in the given timezone (local one by default).

wxDateTime::GetHour

wxDateTime_t GetHour(**const TimeZone&** *tz = Local*) **const**

Returns the hour in the given timezone (local one by default).

wxDateTime::GetMinute

wxDateTime_t GetMinute(**const TimeZone&** *tz = Local*) **const**

Returns the minute in the given timezone (local one by default).

wxDateTime::GetSecond

wxDateTime_t GetSecond(**const TimeZone&** *tz = Local*) **const**

Returns the seconds in the given timezone (local one by default).

wxDateTime::GetMillisecond

wxDateTime_t GetMillisecond(**const TimeZone&** *tz = Local*) **const**

Returns the milliseconds in the given timezone (local one by default).

wxDateTime::GetDayOfYear

wxDateTime_t GetDayOfYear(**const TimeZone&** *tz = Local*) **const**

Returns the day of the year (in 1...366 range) in the given timezone (local one by default).

wxDateTime::GetWeekOfYear

wxDateTime_t GetWeekOfYear(**WeekFlags** *flags = Monday_First*, **const TimeZone&** *tz = Local*) **const**

Returns the number of the week of the year this date is in. The first week of the year is, according to international standards, the one containing Jan 4 or, equivalently, the first week which has Thursday in this year. Both of these definitions are the same as saying that the first week of the year must contain more than half of its days in this year. Accordingly, the week number will always be in 1...53 range (52 for non leap years).

The function depends on the *week start* (p. 257) convention specified by the *flags* argument but its results for Sunday_First are not well-defined as the ISO definition quoted above applies to the weeks starting on Monday only.

wxDateTime::GetWeekOfMonth

wxDateTime_t GetWeekOfMonth(**WeekFlags** *flags = Monday_First*, **const TimeZone&** *tz = Local*) **const**

Returns the ordinal number of the week in the month (in 1...5 range).

As *GetWeekOfYear* (p. 274), this function supports both conventions for the week start. See the description of these *week start* (p. 257) conventions.

wxDateTime::IsWorkDay

bool IsWorkDay(**Country** *country = Country_Default*) **const**

Returns true is this day is not a holiday in the given country.

wxDateTime::IsGregorianDate

bool IsGregorianDate(**GregorianAdoption** *country = Gr_Standard*) **const**

Returns true if the given date is later than the date of adoption of the Gregorian calendar in the given country (and hence the Gregorian calendar calculations make sense for it).

wxDateTime::SetFromDOS

wxDateTime& Set(**unsigned long** *ddt*)

Sets the date from the date and time inDOS
(`http://developer.novell.com/ndk/doc/smscomp/index.html?page=/ndk/doc/smscomp/sms_docs/data/hc2vlu5i.html`)format.

wxDateTime::GetAsDOS

unsigned long GetAsDOS() **const**

Returns the date and time inDOS
(`http://developer.novell.com/ndk/doc/smscomp/index.html?page=/ndk/doc/smscomp/sms_docs/data/hc2vlu5i.html`)format.

wxDateTime::IsEqualTo

bool IsEqualTo(**const wxDateTime&** *datetime*) **const**

Returns `true` if the two dates are strictly identical.

wxDateTime::IsEarlierThan

bool IsEarlierThan(**const wxDateTime&** *datetime*) **const**

Returns `true` if this date precedes the given one.

wxDateTime::IsLaterThan

bool IsLaterThan(**const wxDateTime&** *datetime*) **const**

Returns `true` if this date is later than the given one.

wxDateTime::IsStrictlyBetween

bool IsStrictlyBetween(**const wxDateTime&** *t1*, **const wxDateTime&** *t2*) **const**

Returns `true` if this date lies strictly between the two others,

See also

IsBetween (p. 275)

wxDateTime::IsBetween

bool IsBetween(**const wxDateTime&** *t1*, **const wxDateTime&** *t2*) **const**

Returns `true` if *IsStrictlyBetween* (p. 275)is `true` or if the date is equal to one of the limit values.

See also

IsStrictlyBetween (p. 275)

wxDateTime::IsSameDate

bool IsSameDate(**const wxDateTime&** *dt*) **const**

Returns `true` if the date is the same without comparing the time parts.

wxDateTime::IsSameTime

bool IsSameTime(**const wxDateTime&** *dt*) **const**

Returns `true` if the time is the same (although dates may differ).

wxDateTime::IsEqualUpTo

bool IsEqualUpTo(**const wxDateTime&** *dt*, **const wxTimeSpan&** *ts*) **const**

Returns `true` if the date is equal to another one up to the given time interval, i.e. if the absolute difference between the two dates is less than this interval.

wxDateTime::Add

wxDateTime Add(**const wxTimeSpan&** *diff*) **const**

wxDateTime& Add(**const wxTimeSpan&** *diff*)

wxDateTime& operator+=(**const wxTimeSpan&** *diff*)

Adds the given time span to this object.

wxPython note: This method is named `AddTS` in wxPython.

wxDateTime::Add

wxDateTime Add(**const wxDateSpan&** *diff*) **const**

wxDateTime& Add(**const wxDateSpan&** *diff*)

wxDateTime& operator+=(**const wxDateSpan&** *diff*)

Adds the given date span to this object.

wxPython note: This method is named `AddDS` in wxPython.

wxDateTime::Subtract

wxDateTime Subtract(**const wxTimeSpan&** *diff*) **const**

wxDateTime& Subtract(**const wxTimeSpan&** *diff*)

wxDateTime& operator-=(**const wxTimeSpan&** *diff*)

Subtracts the given time span from this object.

wxPython note: This method is named `SubtractTS` in wxPython.

wxDateTime::Subtract

wxDateTime Subtract(**const wxDateSpan&** *diff*) **const**

wxDateTime& Subtract(**const wxDateSpan&** *diff*)

wxDateTime& operator-=(**const wxDateSpan&** *diff*)

Subtracts the given date span from this object.

wxPython note: This method is named `SubtractDS` in wxPython.

wxDateTime::Subtract

wxTimeSpan Subtract(**const wxDateTime&** *dt*) **const**

Subtracts another date from this one and returns the difference between them as wxTimeSpan.

wxDateTime::ParseRfc822Date

const wxChar * ParseRfc822Date(**const wxChar*** *date*)

Parses the string *date* looking for a date formatted according to the RFC 822 in it. The exact description of this format may, of course, be found in the RFC (section 5), but, briefly, this is the format used in the headers of Internet email messages and one of the most common strings expressing date in this format may be something like `"Sat, 18 Dec 1999 00:48:30 +0100"`.

Returns `NULL` if the conversion failed, otherwise return the pointer to the character immediately following the part of the string which could be parsed. If the entire string contains only the date in RFC 822 format, the returned pointer will be pointing to a `NUL`

character.

This function is intentionally strict, it will return an error for any string which is not RFC 822 compliant. If you need to parse date formatted in more free ways, you should use *ParseDateTime* (p. 278) or*ParseDate* (p. 278) instead.

wxDateTime::ParseFormat

const wxChar * ParseFormat(**const wxChar ****date*, **const wxChar ****format* = *wxDefaultDateTimeFormat*, **const wxDateTime&** *dateDef* = *wxDefaultDateTime*)

This function parses the string *date* according to the given*format*. The system `strptime(3)` function is used whenever available, but even if it is not, this function is still implemented, although support for locale-dependent format specifiers such as "`%c`", "`%x`" or "`%X`" may not be perfect and GNU extensions such as "`%z`" and "`%Z`" are not implemented. This function does handle the month and weekday names in the current locale on all platforms, however.

Please see the description of the ANSI C function `strftime(3)` for the syntax of the format string.

The *dateDef* parameter is used to fill in the fields which could not be determined from the format string. For example, if the format is "`%d`" (the ay of the month), the month and the year are taken from *dateDef*. If it is not specified, *Today* (p. 268) is used as the default date.

Returns `NULL` if the conversion failed, otherwise return the pointer to the character which stopped the scan.

wxDateTime::ParseDateTime

const wxChar * ParseDateTime(**const wxChar ****datetime*)

Parses the string *datetime* containing the date and time in free format. This function tries as hard as it can to interpret the given string as date and time. Unlike *ParseRfc822Date* (p. 277), it will accept anything that may be accepted and will only reject strings which can not be parsed in any way at all.

Returns `NULL` if the conversion failed, otherwise return the pointer to the character which stopped the scan. This method is currently not implemented, so always returns NULL.

wxDateTime::ParseDate

const wxChar * ParseDate(**const wxChar ****date*)

This function is like *ParseDateTime* (p. 278), but it only allows the date to be specified. It is thus less flexible then*ParseDateTime* (p. 278), but also has less chances to misinterpret the user input.

Returns NULL if the conversion failed, otherwise return the pointer to the character which stopped the scan.

wxDateTime::ParseTime

const wxChar * ParseTime(**const wxChar ****time*)

This functions is like *ParseDateTime* (p. 278), but only allows the time to be specified in the input string.

Returns NULL if the conversion failed, otherwise return the pointer to the character which stopped the scan.

wxDateTime::Format

wxString Format(**const wxChar ****format* = *wxDefaultDateTimeFormat*, **const TimeZone&** *tz* = *Local*) **const**

This function does the same as the standard ANSI C strftime(3) function. Please see its description for the meaning of *format* parameter.

It also accepts a few wxWidgets-specific extensions: you can optionally specify the width of the field to follow using printf(3)-like syntax and the format specification %l can be used to get the number of milliseconds.

See also

ParseFormat (p. 278)

wxDateTime::FormatDate

wxString FormatDate() **const**

Identical to calling *Format()* (p. 279) with "%x"argument (which means 'preferred date representation for the current locale').

wxDateTime::FormatTime

wxString FormatTime() **const**

Identical to calling *Format()* (p. 279) with "%X"argument (which means 'preferred time representation for the current locale').

wxDateTime::FormatISODate

wxString FormatISODate() **const**

This function returns the date representation in the ISO 8601 format (YYYY-MM-DD).

wxDateTime::FormatISOTime

wxString FormatISOTime() **const**

This function returns the time representation in the ISO 8601 format (HH:MM:SS).

wxDateTime::SetToWeekDayInSameWeek

wxDateTime& SetToWeekDayInSameWeek(**WeekDay** *weekday*, **WeekFlags***flags* =
Monday_First)

Adjusts the date so that it will still lie in the same week as before, but its week day will be the given one.

Returns the reference to the modified object itself.

wxDateTime::GetWeekDayInSameWeek

wxDateTime GetWeekDayInSameWeek(**WeekDay** *weekday*, **WeekFlags***flags* =
Monday_First) **const**

Returns the copy of this object to which*SetToWeekDayInSameWeek* (p. 280) was applied.

wxDateTime::SetToNextWeekDay

wxDateTime& SetToNextWeekDay(**WeekDay** *weekday*)

Sets the date so that it will be the first *weekday* following the current date.

Returns the reference to the modified object itself.

wxDateTime::GetNextWeekDay

wxDateTime GetNextWeekDay(**WeekDay** *weekday*) **const**

Returns the copy of this object to which*SetToNextWeekDay* (p. 280) was applied.

wxDateTime::SetToPrevWeekDay

wxDateTime& SetToPrevWeekDay(**WeekDay** *weekday*)

Sets the date so that it will be the last *weekday* before the current date.

Returns the reference to the modified object itself.

wxDateTime::GetPrevWeekDay

wxDateTime GetPrevWeekDay(**WeekDay** *weekday*) **const**

Returns the copy of this object to which*SetToPrevWeekDay* (p. 280) was applied.

wxDateTime::SetToWeekDay

bool SetToWeekDay(**WeekDay** *weekday*, **int** *n = 1*, **Month** *month = Inv_Month*, **int** *year = Inv_Year*)

Sets the date to the *n*-th *weekday* in the given month of the given year (the current month and year are used by default). The parameter *n*may be either positive (counting from the beginning of the month) or negative (counting from the end of it).

For example, `SetToWeekDay(2, wxDateTime::Wed)` will set the date to the second Wednesday in the current month and`SetToWeekDay(-1, wxDateTime::Sun)` -- to the last Sunday in it.

Returns `true` if the date was modified successfully, `false`otherwise meaning that the specified date doesn't exist.

wxDateTime::GetWeekDay

wxDateTime GetWeekDay(**WeekDay** *weekday*, **int** *n = 1*, **Month** *month = Inv_Month*, **int** *year = Inv_Year*) **const**

Returns the copy of this object to which*SetToWeekDay* (p. 281) was applied.

wxDateTime::SetToLastWeekDay

bool SetToLastWeekDay(**WeekDay** *weekday*, **Month** *month = Inv_Month*, **int** *year = Inv_Year*)

The effect of calling this function is the same as of calling`SetToWeekDay(-1, weekday, month, year)`. The date will be set to the last*weekday* in the given month and year (the current ones by default).

Always returns `true`.

wxDateTime::GetLastWeekDay

wxDateTime GetLastWeekDay(**WeekDay** *weekday*, **Month** *month = Inv_Month*, **int** *year = Inv_Year*)

Returns the copy of this object to which*SetToLastWeekDay* (p. 281) was applied.

wxDateTime::SetToWeekOfYear

static wxDateTime SetToWeekOfYear(**int** *year*, **wxDateTime_t** *numWeek*, **WeekDay** *weekday* = *Mon*)

Set the date to the given *weekday* in the week number *numWeek* of the given *year* . The number should be in range 1...53.

Note that the returned date may be in a different year than the one passed to this function because both the week 1 and week 52 or 53 (for leap years) contain days from different years. See*GetWeekOfYear* (p. 274) for the explanation of how the year weeks are counted.

wxDateTime::SetToLastMonthDay

wxDateTime& SetToLastMonthDay(**Month** *month* = *Inv_Month*, **int** *year* = *Inv_Year*)

Sets the date to the last day in the specified month (the current one by default).

Returns the reference to the modified object itself.

wxDateTime::GetLastMonthDay

wxDateTime GetLastMonthDay(**Month** *month* = *Inv_Month*, **int** *year* = *Inv_Year*) **const**

Returns the copy of this object to which*SetToLastMonthDay* (p. 282) was applied.

wxDateTime::SetToYearDay

wxDateTime& SetToYearDay(**wxDateTime_t** *yday*)

Sets the date to the day number *yday* in the same year (i.e., unlike the other functions, this one does not use the current year). The day number should be in the range 1...366 for the leap years and 1...365 for the other ones.

Returns the reference to the modified object itself.

wxDateTime::GetYearDay

wxDateTime GetYearDay(**wxDateTime_t** *yday*) **const**

Returns the copy of this object to which*SetToYearDay* (p. 282) was applied.

wxDateTime::GetJulianDayNumber

double GetJulianDayNumber() const

Returns the *JDN* (p. 270) corresponding to this date. Beware of rounding errors!

See also

GetModifiedJulianDayNumber (p. 283)

wxDateTime::GetJDN

double GetJDN() const

Synonym for *GetJulianDayNumber* (p. 283).

wxDateTime::GetModifiedJulianDayNumber

double GetModifiedJulianDayNumber() const

Returns the *Modified Julian Day Number* (MJD) which is, by definition, equal to JDN - 2400000.5. The MJDs are simpler to work with as the integral MJDs correspond to midnights of the dates in the Gregorian calendar and not th noons like JDN. The MJD 0 is Nov 17, 1858.

wxDateTime::GetMJD

double GetMJD() const

Synonym for *GetModifiedJulianDayNumber* (p. 283).

wxDateTime::GetRataDie

double GetRataDie() const

Return the *Rata Die number* of this date.

By definition, the Rata Die number is a date specified as the number of days relative to a base date of December 31 of the year 0. Thus January 1 of the year 1 is Rata Die day 1.

wxDateTime::ToTimezone

wxDateTime ToTimezone(const TimeZone& *tz*, bool *noDST = false*) const

Transform the date to the given time zone. If *noDST* is `true`, no DST adjustments will be made.

Returns the date in the new time zone.

wxDateTime::MakeTimezone

wxDateTime& MakeTimezone(**const TimeZone&** *tz*, **bool** *noDST = false*)

Modifies the object in place to represent the date in another time zone. If *noDST* is `true`, no DST adjustments will be made.

wxDateTime::ToGMT

wxDateTime ToGMT(**bool** *noDST = false*) **const**

This is the same as calling *ToTimezone* (p. 283) with the argument `GMT0`.

wxDateTime::MakeGMT

wxDateTime& MakeGMT(**bool** *noDST = false*)

This is the same as calling *MakeTimezone* (p. 284) with the argument `GMT0`.

wxDateTime::IsDST

int IsDST(**Country** *country = Country_Default*) **const**

Returns `true` if the DST is applied for this date in the given country.

See also

GetBeginDST (p. 265) and *GetEndDST* (p. 266)

wxDateTimeHolidayAuthority

TODO

wxDateTimeWorkDays

TODO

wxDb

A wxDb instance is a connection to an ODBC datasource which may be opened, closed, and re-opened an unlimited number of times. A database connection allows function to

be performed directly on the datasource, as well as allowing access to any tables/views defined in the datasource to which the user has sufficient privileges.

See the *database classes overview* (p. 1820) for an introduction to using the ODBC classes.

Include files

<wx/db.h>

Helper classes and data structures

The following classes and structs are defined in db.cpp/.h for use with the wxDb class.

- *wxDbColFor* (p. 316)

- *wxDbColInf* (p. 317)

- *wxDbTableInf* (p. 360)

- *wxDbInf* (p. 324)

Constants

NOTE: In a future release, all ODBC class constants will be prefaced with 'wx'.

wxDB_PATH_MAX	Maximum path length allowed to be passed to the ODBC driver to indicate where the data file(s) are located.
DB_MAX_COLUMN_NAME_LEN	Maximum supported length for the name of a column
DB_MAX_ERROR_HISTORY	Maximum number of error messages retained in the queue before being overwritten by new errors.
DB_MAX_ERROR_MSG_LEN	Maximum supported length of an error message returned by the ODBC classes
DB_MAX_STATEMENT_LEN	Maximum supported length for a complete SQL statement to be passed to the ODBC driver
DB_MAX_TABLE_NAME_LEN	Maximum supported length for the name of a table
DB_MAX_WHERE_CLAUSE_LEN	Maximum supported WHERE clause length that can be passed to the ODBC driver
DB_TYPE_NAME_LEN	Maximum length of the name of a column's data type

Enumerated types

Enumerated types

*enum **wxDbSqlLogState***

285

sqlLogOFF, sqlLogON

*enum **wxDBMS***

These are the databases currently tested and working with the ODBC classes. A call to *wxDb::Dbms* (p. 295) will return one of these enumerated values listed below.

- DB2

- DBase (IV, V)**

- Firebird

- INFORMIX

- Interbase

- MS SQL Server (v7 - minimal testing)

- MS Access (97, 2000, 2002, and 2003)

- MySQL (2.x and 3.5 - use the 2.5x drivers though)

- Oracle (v7, v8, v8i)

- Pervasive SQL

- PostgreSQL

- Sybase (ASA and ASE)

- XBase Sequiter

- VIRTUOSO

See the remarks in *wxDb::Dbms* (p. 295) for exceptions/issues with each of these database engines.

Public member variables

*SWORD **wxDb::cbErrorMsg***

This member variable is populated as a result of calling *wxDb::GetNextError* (p. 303). Contains the count of bytes in the wxDb::errorMsg string.

*int **wxDb::DB_STATUS***

The last ODBC error/status that occurred on this data connection. Possible codes are:

```
DB_ERR_GENERAL_WARNING                      // SqlState = '01000'
DB_ERR_DISCONNECT_ERROR                     // SqlState = '01002'
DB_ERR_DATA_TRUNCATED                       // SqlState = '01004'
DB_ERR_PRIV_NOT_REVOKED                     // SqlState = '01006'
DB_ERR_INVALID_CONN_STR_ATTR                // SqlState = '01S00'
DB_ERR_ERROR_IN_ROW                         // SqlState = '01S01'
```

```
DB_ERR_OPTION_VALUE_CHANGED                  // SqlState = '01S02'
DB_ERR_NO_ROWS_UPD_OR_DEL                    // SqlState = '01S03'
DB_ERR_MULTI_ROWS_UPD_OR_DEL                 // SqlState = '01S04'
DB_ERR_WRONG_NO_OF_PARAMS                    // SqlState = '07001'
DB_ERR_DATA_TYPE_ATTR_VIOL                   // SqlState = '07006'
DB_ERR_UNABLE_TO_CONNECT                     // SqlState = '08001'
DB_ERR_CONNECTION_IN_USE                     // SqlState = '08002'
DB_ERR_CONNECTION_NOT_OPEN                   // SqlState = '08003'
DB_ERR_REJECTED_CONNECTION                   // SqlState = '08004'
DB_ERR_CONN_FAIL_IN_TRANS                    // SqlState = '08007'
DB_ERR_COMM_LINK_FAILURE                     // SqlState = '08S01'
DB_ERR_INSERT_VALUE_LIST_MISMATCH            // SqlState = '21S01'
DB_ERR_DERIVED_TABLE_MISMATCH                // SqlState = '21S02'
DB_ERR_STRING_RIGHT_TRUNC                    // SqlState = '22001'
DB_ERR_NUMERIC_VALUE_OUT_OF_RNG              // SqlState = '22003'
DB_ERR_ERROR_IN_ASSIGNMENT                   // SqlState = '22005'
DB_ERR_DATETIME_FLD_OVERFLOW                 // SqlState = '22008'
DB_ERR_DIVIDE_BY_ZERO                        // SqlState = '22012'
DB_ERR_STR_DATA_LENGTH_MISMATCH              // SqlState = '22026'
DB_ERR_INTEGRITY_CONSTRAINT_VIOL             // SqlState = '23000'
DB_ERR_INVALID_CURSOR_STATE                  // SqlState = '24000'
DB_ERR_INVALID_TRANS_STATE                   // SqlState = '25000'
DB_ERR_INVALID_AUTH_SPEC                     // SqlState = '28000'
DB_ERR_INVALID_CURSOR_NAME                   // SqlState = '34000'
DB_ERR_SYNTAX_ERROR_OR_ACCESS_VIOL           // SqlState = '37000'
DB_ERR_DUPLICATE_CURSOR_NAME                 // SqlState = '3C000'
DB_ERR_SERIALIZATION_FAILURE                 // SqlState = '40001'
DB_ERR_SYNTAX_ERROR_OR_ACCESS_VIOL2          // SqlState = '42000'
DB_ERR_OPERATION_ABORTED                     // SqlState = '70100'
DB_ERR_UNSUPPORTED_FUNCTION                  // SqlState = 'IM001'
DB_ERR_NO_DATA_SOURCE                        // SqlState = 'IM002'
DB_ERR_DRIVER_LOAD_ERROR                     // SqlState = 'IM003'
DB_ERR_SQLALLOCENV_FAILED                    // SqlState = 'IM004'
DB_ERR_SQLALLOCCONNECT_FAILED                // SqlState = 'IM005'
DB_ERR_SQLSETCONNECTOPTION_FAILED            // SqlState = 'IM006'
DB_ERR_NO_DATA_SOURCE_DLG_PROHIB             // SqlState = 'IM007'
DB_ERR_DIALOG_FAILED                         // SqlState = 'IM008'
DB_ERR_UNABLE_TO_LOAD_TRANSLATION_DLL        // SqlState = 'IM009'
DB_ERR_DATA_SOURCE_NAME_TOO_LONG             // SqlState = 'IM010'
DB_ERR_DRIVER_NAME_TOO_LONG                  // SqlState = 'IM011'
DB_ERR_DRIVER_KEYWORD_SYNTAX_ERROR           // SqlState = 'IM012'
DB_ERR_TRACE_FILE_ERROR                      // SqlState = 'IM013'
DB_ERR_TABLE_OR_VIEW_ALREADY_EXISTS          // SqlState = 'S0001'
DB_ERR_TABLE_NOT_FOUND                       // SqlState = 'S0002'
DB_ERR_INDEX_ALREADY_EXISTS                  // SqlState = 'S0011'
DB_ERR_INDEX_NOT_FOUND                       // SqlState = 'S0012'
DB_ERR_COLUMN_ALREADY_EXISTS                 // SqlState = 'S0021'
DB_ERR_COLUMN_NOT_FOUND                      // SqlState = 'S0022'
DB_ERR_NO_DEFAULT_FOR_COLUMN                 // SqlState = 'S0023'
DB_ERR_GENERAL_ERROR                         // SqlState = 'S1000'
DB_ERR_MEMORY_ALLOCATION_FAILURE             // SqlState = 'S1001'
DB_ERR_INVALID_COLUMN_NUMBER                 // SqlState = 'S1002'
DB_ERR_PROGRAM_TYPE_OUT_OF_RANGE             // SqlState = 'S1003'
DB_ERR_SQL_DATA_TYPE_OUT_OF_RANGE            // SqlState = 'S1004'
DB_ERR_OPERATION_CANCELLED                   // SqlState = 'S1008'
DB_ERR_INVALID_ARGUMENT_VALUE                // SqlState = 'S1009'
DB_ERR_FUNCTION_SEQUENCE_ERROR               // SqlState = 'S1010'
DB_ERR_OPERATION_INVALID_AT_THIS_TIME        // SqlState = 'S1011'
DB_ERR_INVALID_TRANS_OPERATION_CODE          // SqlState = 'S1012'
DB_ERR_NO_CURSOR_NAME_AVAIL                   // SqlState = 'S1015'
DB_ERR_INVALID_STR_OR_BUF_LEN                // SqlState = 'S1090'
DB_ERR_DESCRIPTOR_TYPE_OUT_OF_RANGE          // SqlState = 'S1091'
DB_ERR_OPTION_TYPE_OUT_OF_RANGE              // SqlState = 'S1092'
```

```
DB_ERR_INVALID_PARAM_NO                          // SqlState = 'S1093'
DB_ERR_INVALID_SCALE_VALUE                       // SqlState = 'S1094'
DB_ERR_FUNCTION_TYPE_OUT_OF_RANGE                // SqlState = 'S1095'
DB_ERR_INF_TYPE_OUT_OF_RANGE                     // SqlState = 'S1096'
DB_ERR_COLUMN_TYPE_OUT_OF_RANGE                  // SqlState = 'S1097'
DB_ERR_SCOPE_TYPE_OUT_OF_RANGE                   // SqlState = 'S1098'
DB_ERR_NULLABLE_TYPE_OUT_OF_RANGE                // SqlState = 'S1099'
DB_ERR_UNIQUENESS_OPTION_TYPE_OUT_OF_RANGE // SqlState = 'S1100'
DB_ERR_ACCURACY_OPTION_TYPE_OUT_OF_RANGE   // SqlState = 'S1101'
DB_ERR_DIRECTION_OPTION_OUT_OF_RANGE       // SqlState = 'S1103'
DB_ERR_INVALID_PRECISION_VALUE                   // SqlState = 'S1104'
DB_ERR_INVALID_PARAM_TYPE                        // SqlState = 'S1105'
DB_ERR_FETCH_TYPE_OUT_OF_RANGE                   // SqlState = 'S1106'
DB_ERR_ROW_VALUE_OUT_OF_RANGE                    // SqlState = 'S1107'
DB_ERR_CONCURRENCY_OPTION_OUT_OF_RANGE     // SqlState = 'S1108'
DB_ERR_INVALID_CURSOR_POSITION                   // SqlState = 'S1109'
DB_ERR_INVALID_DRIVER_COMPLETION                 // SqlState = 'S1110'
DB_ERR_INVALID_BOOKMARK_VALUE                    // SqlState = 'S1111'
DB_ERR_DRIVER_NOT_CAPABLE                        // SqlState = 'S1C00'
DB_ERR_TIMEOUT_EXPIRED                           // SqlState = 'S1T00'
```

struct **wxDb::dbInf**

This structure is internal to the wxDb class and contains details of the ODBC datasource that the current instance of the wxDb is connected to in its members. When the datasource is opened, all of the information contained in the dbInf structure is queried from the datasource. This information is used almost exclusively within the ODBC class library. Where there may be a need for particular portions of this information outside of the class library, member functions (e.g.*wxDbTable::IsCursorClosedOnCommit* (p. 345)) have been added for ease of use.

```
wxChar dbmsName[40]       - Name of the dbms product
wxChar dbmsVer[64]        - Version # of the dbms product
wxChar driverName[40]     - Driver name
wxChar odbcVer[60]        - ODBC version of the driver
wxChar drvMgrOdbcVer[60]  - ODBC version of the driver manager
wxChar driverVer[60]      - Driver version
wxChar serverName[80]     - Server Name, typically a connect string
wxChar databaseName[128]  - Database filename
wxChar outerJoins[2]      - Does datasource support outer joins
wxChar procedureSupport[2] - Does datasource support stored
                            procedures
UWORD  maxConnections     - Maximum # of connections datasource
                            supports
UWORD  maxStmts           - Maximum # of HSTMTs per HDBC
UWORD  apiConfLvl         - ODBC API conformance level
UWORD  cliConfLvl         - Is datasource SAG compliant
UWORD  sqlConfLvl         - SQL conformance level
UWORD  cursorCommitBehavior  - How cursors are affected on db commit
UWORD  cursorRollbackBehavior - How cursors are affected on db
                            rollback
UWORD  supportNotNullClause  - Does datasource support NOT NULL
                            clause
wxChar supportIEF[2]      - Integrity Enhancement Facility (Ref.
                            Integrity)
UDWORD txnIsolation       - Transaction isolation level supported by
                            driver
UDWORD txnIsolationOptions - Transaction isolation level options
                            available
UDWORD fetchDirections    - Fetch directions supported
```

```
    UDWORD lockTypes          - Lock types supported in SQLSetPos
    UDWORD posOperations      - Position operations supported in
                                SQLSetPos
    UDWORD posStmts           - Position statements supported
    UDWORD scrollConcurrency  - Scrollable cursor concurrency options
                                supported
    UDWORD scrollOptions      - Scrollable cursor options supported
    UDWORD staticSensitivity  - Can additions/deletions/updates be
                                detected
    UWORD  txnCapable         - Indicates if datasource supports
                                transactions
    UDWORD loginTimeout       - Number seconds to wait for a login
                                request
```

*wxChar **wxDb::errorList**[DB_MAX_ERROR_HISTORY][DB_MAX_ERROR_MSG_LEN]*

The last n ODBC errors that have occurred on this database connection.

*wxChar **wxDb::errorMsg**[SQL_MAX_MESSAGE_LENGTH]*

This member variable is populated as a result of calling *wxDb::GetNextError* (p. 303). It contains the ODBC error message text.

*SDWORD **wxDb::nativeError***

Set by wxDb::DispAllErrors, wxDb::GetNextError, and wxDb::DispNextError. It contains the datasource-specific error code returned by the datasource to the ODBC driver. Used for reporting ODBC errors.

*wxChar **wxDb::sqlState**[20]*

Set by wxDb::TranslateSqlState(). Indicates the error state after a failed ODBC operation. Used for reporting ODBC errors.

Remarks

Default cursor scrolling is defined by wxODBC_FWD_ONLY_CURSORS in setup.h when the wxWidgets library is built. This behavior can be overridden when an instance of a wxDb is created (see *wxDb constructor* (p. 291)). Default setting of this value true, as not all databases/drivers support both types of cursors.

See also

wxDbColFor (p. 316), *wxDbColInf* (p. 317),*wxDbTable* (p. 324), *wxDbTableInf* (p. 360),*wxDbInf* (p. 324)

Associated non-class functions

The following functions are used in conjunction with the wxDb class.

void wxDbCloseConnections()

Remarks

Closes all cached connections that have been made through use of

the*wxDbGetConnection* (p. 289) function.

NOTE: These connections are closed regardless of whether they are in use or not. This function should only be called after the program has finished using the connections and all wxDbTable instances that use any of the connections have been closed.

This function performs a *wxDb::CommitTrans* (p. 293)on the connection before closing it to commit any changes that are still pending, as well as to avoid any function sequence errors upon closing each connection.

int wxDbConnectionsInUse()

Remarks

Returns a count of how many database connections are currently free (not being used) that have been cached through use of the *wxDbGetConnection* (p. 289)function.

bool wxDbFreeConnection(**wxDb** **pDb*)

Remarks

Searches the list of cached database connections connection for one matching the passed in wxDb instance. If found, that cached connection is freed.

Freeing a connection means that it is marked as available (free) in the cache of connections, so that a call to *wxDbGetConnection* (p. 289)is able to return a pointer to the wxDb instance for use. Freeing a connection does NOT close the connection, it only makes the connection available again.

wxDb * **wxDbGetConnection**(**wxDbConnectInf** **pDbConfig,***bool** *FwdOnlyCursors=(bool)wxODBC_FWD_ONLY_CURSORS*)

Remarks

This function is used to request a "new" wxDb instance for use by the program. The wxDb instance returned is also opened (see *wxDb::Open* (p. 307)).

This function (along with wxDbFreeConnection() and wxDbCloseConnection()) maintain a cache of wxDb instances for user/re-use by a program. When a program needs a wxDb instance, it may call this function to obtain a wxDb instance. If there is a wxDb instance in the cache that is currently unused that matches the connection requirements specified in *'pDbConfig'* then that cached connection is marked as no longer being free, and a pointer to the wxDb instance is returned.

If there are no connections available in the cache that meet the requirements given in *'pDbConfig'*, then a new wxDb instance is created to connect to the datasource specified in *'pDbConfig'* using the userID and password given in *'pDbConfig'*.

NOTE: The caching routine also uses the *wxDb::Open* (p. 307)connection datatype copying code. If the call to wxDbGetConnection() requests a connection to a datasource, and there is not one available in the cache, a new connection is created. But when the connection is opened, instead of polling the datasource over again for its datatypes, if a connection to the same datasource (using the same userID/password) has already been done previously, the new connection skips querying the datasource for

its datatypes, and uses the same datatypes determined previously by the other connection(s) for that same datasource. This cuts down greatly on network traffic, database load, and connection creation time.

When the program is done using a connection created through a call to wxDbGetConnection(), the program should call wxDbFreeConnection() to release the wxDb instance back to the cache. DO NOT DELETE THE wxDb INSTANCE! Deleting the wxDb instance returned can cause a crash/memory corruption later in the program when the cache is cleaned up.

When exiting the program, call wxDbCloseConnections() to close all the cached connections created by calls to wxDbGetConnection().

const wxChar * wxDbLogExtendedErrorMsg(**const wxChar ****userText*, **wxDb ****pDb*, **wxChar ****ErrFile*, **int** *ErrLine*)

Writes a message to the wxLog window (stdout usually) when an internal error situation occurs.

bool wxDbSqlLog(**wxDbSqlLogState** *state*, **const wxString &***filename* = *SQL_LOG_FILENAME*)

Remarks

This function sets the sql log state for all open wxDb objects

bool wxDbGetDataSource(**HENV** *henv*, **wxChar ****Dsn*, **SWORD** *DsnMax*, **wxChar ****DsDesc*, **SWORD** *DsDescMax*, **UWORD** *direction* = *SQL_FETCH_NEXT*)

Remarks

This routine queries the ODBC driver manager for a list of available datasources. Repeatedly call this function to obtain all the datasources available through the ODBC driver manager on the current workstation.

```
        wxArrayString strArray;

        while (wxDbGetDataSource(DbConnectInf.GetHenv(), Dsn,
    SQL_MAX_DSN_LENGTH+1, DsDesc, 255))
            strArray.Add(Dsn);
```

wxDb::wxDb

wxDb()

Default constructor.

wxDb(**const HENV &***aHenv*, **bool** *FwdOnlyCursors=(bool)wxODBC_FWD_ONLY_CURSORS*)

Constructor, used to create an ODBC connection to a datasource.

Parameters

aHenv

> Environment handle used for this connection. See*wxDConnectInf::AllocHenv* (p. 320)

FwdOnlyCursors

> Will cursors created for use with this datasource connection only allow forward scrolling cursors.

Remarks

This is the constructor for the wxDb class. The wxDb object must be created and opened before any database activity can occur.

Example

```
wxDbConnectInf ConnectInf;
....Set values for member variables of ConnectInf here

wxDb sampleDB(ConnectInf.GetHenv());
if (!sampleDB.Open(ConnectInf.GetDsn(), ConnectInf.GetUserID(),
                   ConnectInf.GetPassword()))
{
   // Error opening datasource
}
```

See also

wxDbGetConnection (p. 289)

wxDb::Catalog

bool Catalog(**wxChar *** *userID*, **const wxString &***fileName* = *SQL_CATALOG_FILENAME*)

Allows a data "dictionary" of the datasource to be created, dumping pertinent information about all data tables to which the user specified in userID has access.

Parameters

userID

> Database user name to use in accessing the database. All tables to which this user has rights will be evaluated in the catalog.

fileName

> *OPTIONAL*. Name of the text file to create and write the DB catalog to. Default is SQL_CATALOG_FILENAME.

Return value

Returns true if the catalog request was successful, or false if there was some reason that the catalog could not be generated.

Example

```
=============== =============== ================= ========= =======
TABLE NAME      COLUMN NAME     DATA TYPE         PRECISION LENGTH
=============== =============== ================= ========= =======
EMPLOYEE        RECID           (0008)NUMBER             15       8
EMPLOYEE        USER_ID         (0012)VARCHAR2          13      13
EMPLOYEE        FULL_NAME       (0012)VARCHAR2          26      26
EMPLOYEE        PASSWORD        (0012)VARCHAR2          26      26
EMPLOYEE        START_DATE      (0011)DATE              19      16
```

wxDb::Close

void Close()

Closes the database connection.

Remarks

At the end of your program, when you have finished all of your database work, you must close the ODBC connection to the datasource. There are actually four steps involved in doing this as illustrated in the example.

Any wxDbTable instances which use this connection must be deleted before closing the database connection.

Example

```
// Commit any open transactions on the datasource
sampleDB.CommitTrans();

// Delete any remaining wxDbTable objects allocated with new
delete parts;

// Close the wxDb connection when finished with it
sampleDB.Close();
```

wxDb::CommitTrans

bool CommitTrans()

Permanently "commits" changes (insertions/deletions/updates) to the database.

Return value

Returns true if the commit was successful, or false if the commit failed.

Remarks

Transactions begin implicitly as soon as you make a change to the database with an insert/update/delete, or any other direct SQL command that performs one of these

operations against the datasource. At any time thereafter, to save the changes to disk permanently, "commit" them by calling this function.

Calling this member function commits ALL open transactions on this ODBC connection. For example, if three different wxDbTable instances used the same connection to the datasource, committing changes made on one of those wxDbTable instances commits any pending transactions on all three wxDbTable instances.

Until a call to wxDb::CommitTrans() is made, no other user or cursor is able to see any changes made to the row(s) that have been inserted/modified/deleted.

Special Note : *Cursors*

It is important to understand that different database/ODBC driver combinations handle transactions differently. One thing in particular that you must pay attention to is cursors, in regard to transactions. Cursors are what allow you to scroll through records forward and backward and to manipulate records as you scroll through them. When you issue a query, a cursor is created behind the scenes. The cursor keeps track of the query and keeps track of the current record pointer. After you commit or rollback a transaction, the cursor may be closed automatically. This is database dependent, and with some databases this behavior can be controlled through management functions. This means you would need to requery the datasource before you can perform any additional work using this cursor. This is only necessary however if the datasource closes the cursor after a commit or rollback. Use the*wxDbTable::IsCursorClosedOnCommit* (p. 345)member function to determine the datasource's transaction behavior. Note, in many situations it is very inefficient to assume the cursor is closed and always requery. This could put a significant, unnecessary load on datasources that leave the cursors open after a transaction.

wxDb::CreateView

bool CreateView(**const wxString &** *viewName*,**const wxString &** *colList*, **const wxString &***pSqlStmt*)

Creates a SQL VIEW of one or more tables in a single datasource. Note that this function will only work against databases which support views (currently only Oracle as of November 21 2000).

Parameters

viewName

> The name of the view. e.g. PARTS_V

colList

> *OPTIONAL* Pass in a comma delimited list of column names if you wish to explicitly name each column in the result set. If not desired, pass in an empty string and the column names from the associated table(s) will be used.

pSqlStmt

Pointer to the select statement portion of the CREATE VIEW statement. Must be a complete, valid SQL SELECT statement.

Remarks

A 'view' is a logical table that derives columns from one or more other tables or views. Once the view is created, it can be queried exactly like any other table in the database.

NOTE: Views are not available with all datasources. Oracle is one example of a datasource which does support views.

Example

```
// Incomplete code sample
db.CreateView("PARTS_SD1", "PN, PD, QTY",
              "SELECT PART_NUM, PART_DESC, QTY_ON_HAND * 1.1 FROM PARTS
\
              WHERE STORAGE_DEVICE = 1");

// PARTS_SD1 can now be queried just as if it were a data table.
// e.g. SELECT PN, PD, QTY FROM PARTS_SD1
```

wxDb::Dbms

wxDBMS Dbms()

Remarks

The return value will be of the enumerated type wxDBMS. This enumerated type contains a list of all the currently tested and supported databases.

Additional databases may work with these classes, but the databases returned by this function have been tested and confirmed to work with these ODBC classes.

Possible values returned by this function can be viewed in the *Enumerated types* (p. 285) section of wxDb.

There are known issues with conformance to the ODBC standards with several datasources supported by the wxWidgets ODBC classes. Please see the overview for specific details on which datasource have which issues.

Return value

The return value will indicate which of the supported datasources is currently connected to by this connection. In the event that the datasource is not recognized, a value of 'dbmsUNIDENTIFIED' is returned.

wxDb::DispAllErrors

bool DispAllErrors(**HENV** *aHenv*, **HDBC** *aHdbc* = *SQL_NULL_HDBC*, **HSTMT** *aHstmt* = *SQL_NULL_HSTMT*)

Used to log all database errors that occurred as a result of an executed database

command. This logging is automatic and also includes debug logging when compiled in debug mode via *wxLogDebug* (p. 1651). If logging is turned on via *wxDb::SetSqlLogging* (p. 310), then an entry is also logged to the defined log file.

Parameters

aHenv

> Handle to the ODBC environment.

aHdbc

> Handle to the ODBC connection. Pass this in if the ODBC function call that erred required a hdbc or hstmt argument.

aHstmt

> Handle to the ODBC statement being executed against. Pass this in if the ODBC function call that failed required a hstmt argument.

Remarks

This member function will log all of the ODBC error messages for the last ODBC function call that was made. This function is normally used internally within the ODBC class library, but can be used programmatically after calling ODBC functions directly (i.e. SQLFreeEnv()).

Return value

The function always returns false, so a call to this function can be made in the return statement of a code block in the event of a failure to perform an action (see the example below).

See also

wxDb::SetSqlLogging (p. 310), wxDbSqlLog

Example

```
    if (SQLExecDirect(hstmt, (UCHAR FAR *) pSqlStmt, SQL_NTS) !=
SQL_SUCCESS)
        // Display all ODBC errors for this stmt
        return(db.DispAllErrors(db.henv, db.hdbc, hstmt));
```

wxDb::DispNextError

void DispNextError()

Remarks

This function is normally used internally within the ODBC class library. It could be used programmatically after calling ODBC functions directly. This function works in conjunction with *wxDb::GetNextError* (p. 303) when errors (or sometimes informational messages) returned from ODBC need to be analyzed rather than simply displaying

them as an error. GetNextError() retrieves the next ODBC error from the ODBC error queue. The wxDb member variables "sqlState", "nativeError" and "errorMsg" could then be evaluated. To display the error retrieved, DispNextError() could then be called. The combination of GetNextError() and DispNextError() can be used to iteratively step through the errors returned from ODBC evaluating each one in context and displaying the ones you choose.

Example

```
// Drop the table before attempting to create it
sprintf(sqlStmt, "DROP TABLE %s", tableName);
// Execute the drop table statement
if (SQLExecDirect(hstmt,(UCHAR FAR *)sqlStmt,SQL_NTS) != SQL_SUCCESS)
{
    // Check for sqlState = S0002, "Table or view not found".
    // Ignore this error, bomb out on any other error.
    pDb->GetNextError(henv, hdbc, hstmt);
    if (wxStrcmp(pDb->sqlState, "S0002"))
    {
        pDb->DispNextError();   // Displayed error retrieved
        pDb->DispAllErrors(henv, hdbc, hstmt); // Display all other
errors, if any
        pDb->RollbackTrans();   // Rollback the transaction
        CloseCursor();          // Close the cursor
        return(false);          // Return Failure
    }
}
```

wxDb::DropView

bool DropView(const wxString &*viewName*)

Drops the data table view named in 'viewName'.

Parameters

viewName

> Name of the view to be dropped.

Remarks

If the view does not exist, this function will return true. Note that views are not supported with all datasources.

wxDb::ExecSql

bool ExecSql(const wxString &*pSqlStmt*)

Allows a native SQL command to be executed directly against the datasource. In addition to being able to run any standard SQL command, use of this function allows a user to (potentially) utilize features specific to the datasource they are connected to that may not be available through ODBC. The ODBC driver will pass the specified command directly to the datasource.

Parameters

pSqlStmt

> Pointer to the SQL statement to be executed.

Remarks

This member extends the wxDb class and allows you to build and execute ANY VALID SQL statement against the datasource. This allows you to extend the class library by being able to issue any SQL statement that the datasource is capable of processing.

See also

wxDb::GetData (p. 300), *wxDb::GetNext* (p. 303)

wxDb::FwdOnlyCursors

bool IsFwdOnlyCursors()

Older form (pre-2.3/2.4 of wxWidgets) of the*wxDb::IsFwdOnlyCursors* (p. 305). This method is provided for backward compatibility only. The method*wxDb::IsFwdOnlyCursors* (p. 305) should be used in place of this method.

wxDbInf * GetCatalog(**const wxChar ***userID*)

wxDb::GetCatalog

wxDbInf * GetCatalog(**const wxChar ***userID*)

Returns a *wxDbInf* (p. 324) pointer that points to the catalog (datasource) name, schema, number of tables accessible to the current user, and a wxDbTableInf pointer to all data pertaining to all tables in the users catalog.

Parameters

userID

> Owner/Schema of the table. Specify a userID when the datasource you are connected to allows multiple unique tables with the same name to be owned by different users. *userID* is evaluated as follows:

```
userID == NULL  ... UserID is ignored (DEFAULT)
userID == ""    ... UserID set equal to 'this->uid'
userID != ""    ... UserID set equal to 'userID'
```

Remarks

The returned catalog will only contain catalog entries for tables to which the user specified in 'userID' has sufficient privileges. If no user is specified (NULL passed in), a catalog pertaining to all tables in the datasource accessible to the connected user (permissions apply) via this connection will be returned.

wxDb::GetColumnCount

int GetColumnCount(**const wxString &***tableName*, **const wxChar ****userID*)

tableName

> The table name you wish to obtain column information about.

userID

> Name of the user that owns the table(s) (also referred to as schema). Required for some datasources for situations where there may be multiple tables with the same name in the datasource, but owned by different users. *userID* is evaluated in the following manner:

```
userID == NULL  ... UserID is ignored (DEFAULT)
userID == ""    ... UserID set equal to 'this->uid'
userID != ""    ... UserID set equal to 'userID'
```

Return value

Returns a count of how many columns are in the specified table. If an error occurs retrieving the number of columns, this function will return a -1.

wxDb::GetColumns

wxDbColInf * GetColumns(**const wxString &***tableName*, **UWORD ****numCols*, **const wxChar ****userID=NULL*)

wxDbColInf * GetColumns(**wxChar ****tableName[]*, **const wxChar ****userID*)

Parameters

tableName

> The table name you wish to obtain column information about.

numCols

> Pointer to a UWORD which will hold a count of the number of columns returned by this function

tableName[]

> An array of pointers to table names you wish to obtain column information about. The last element of this array must be a NULL string.

userID

> Name of the user that owns the table(s) (also referred to as schema). Required for some datasources for situations where there may be multiple tables with the same name in the datasource, but owned by different users. *userID* is evaluated in the

following manner:

```
userID == NULL  ... UserID is ignored (DEFAULT)
userID == ""    ... UserID set equal to 'this->uid'
userID != ""    ... UserID set equal to 'userID'
```

Return value

This function returns a pointer to an array of *wxDbColInf* (p. 317)structures, allowing you to obtain information regarding the columns of the named table(s). If no columns were found, or an error occurred, this pointer will be NULL.

THE CALLING FUNCTION IS RESPONSIBLE FOR DELETING THE *wxDbColInf* MEMORY WHEN IT IS FINISHED WITH IT.

ALL column bindings associated with this wxDb instance are unbound by this function, including those used by any wxDbTable instances that use this wxDb instance. This function should use its own wxDb instance to avoid undesired unbinding of columns.

See also

wxDbColInf (p. 317)

Example

```
wxChar *tableList[] = {"PARTS", 0};
wxDbColInf *colInf = pDb->GetColumns(tableList);
if (colInf)
{
   // Use the column inf
   .......
   // Destroy the memory
   delete [] colInf;
}
```

wxDb::GetData

bool GetData(**UWORD** *colNumber*, **SWORD** *cType*,**PTR** *pData*, **SDWORD** *maxLen*, **SDWORD FAR *** *cbReturned*)

Used to retrieve result set data without binding column values to memory variables (i.e. not using a wxDbTable instance to access table data).

Parameters

colNumber

Ordinal number of the desired column in the result set to be returned.

cType

The C data type that is to be returned. See a partial list in *wxDbTable::SetColDefs* (p. 352)

pData

Memory buffer which will hold the data returned by the call to this function.

maxLen

Maximum size of the buffer *'pData'* in characters. NOTE: Not UNICODE safe. If this is a numeric field, a value of 0 may be passed for this parameter, as the API knows the size of the expected return value.

cbReturned

Pointer to the buffer containing the length of the actual data returned. If this value comes back as SQL_NULL_DATA, then the*wxDb::GetData* (p. 300) call has failed.

See also

Example

```
SDWORD cb;
ULONG reqQty;
wxString sqlStmt;
sqlStmt = "SELECT SUM(REQUIRED_QTY - PICKED_QTY) FROM ORDER_TABLE
WHERE \
              PART_RECID = 1450 AND REQUIRED_QTY > PICKED_QTY";

// Perform the query
if (!pDb->ExecSql(sqlStmt.c_str()))
{
    // ERROR
    return(0);
}

// Request the first row of the result set
if (!pDb->GetNext())
{
    // ERROR
    return(0);
}

// Read column #1 of the row returned by the call to ::GetNext()
// and return the value in 'reqQty'
if (!pDb->GetData(1, SQL_C_ULONG, &reqQty, 0, &cb))
{
    // ERROR
    return(0);
}

// Check for a NULL result
if (cb == SQL_NULL_DATA)
    return(0);
```

Remarks

When requesting multiple columns to be returned from the result set (for example, the SQL query requested 3 columns be returned), the calls to this function must request the columns in ordinal sequence (1,2,3 or 1,3 or 2,3).

wxDb::GetDatabaseName

const wxChar * GetDatabaseName()

Returns the name of the database engine.

wxDb::GetDatasourceName

const wxString & GetDatasourceName()

Returns the ODBC datasource name.

wxDb::GetHDBC

HDBC GetHDBC()

Returns the ODBC handle to the database connection.

wxDb::GetHENV

HENV GetHENV()

Returns the ODBC environment handle.

wxDb::GetHSTMT

HSTMT GetHSTMT()

Returns the ODBC statement handle associated with this database connection.

wxDb::GetKeyFields

int GetKeyFields(const wxString &*tableName*, **wxDbColInf** **colInf*, **UWORD** *numColumns*)

Used to determine which columns are members of primary or non-primary indexes on the specified table. If a column is a member of a foreign key for some other table, that information is detected also.

This function is primarily for use by the *wxDb::GetColumns* (p. 299) function, but may be called if desired from the client application.

Parameters

tableName

> Name of the table for which the columns will be evaluated as to their inclusion in any indexes.

colInf

> Data structure containing the column definitions (obtained with *wxDb::GetColumns* (p. 299)). This function populates the PkCol, PkTableName, and FkTableName members of the colInf structure.

numColumns

> Number of columns defined in the instance of colInf.

Return value

Currently always returns true.

See also

wxDbColInf (p. 317), *wxDb::GetColumns* (p. 299)

wxDb::GetNext

bool GetNext()

Called after executing a query, this function requests the next row in the result set after the current position of the cursor.

See also

wxDb::ExecSql (p. 297), *wxDb::GetData* (p. 300)

wxDb::GetNextError

bool GetNextError(**HENV** *aHenv*,**HDBC** *aHdbc = SQL_NULL_HDBC*, **HSTMT** *aHstmt = SQL_NULL_HSTMT*)

Parameters

aHenv

> A handle to the ODBC environment.

aHdbc

> *OPTIONAL.* A handle to the ODBC connection. Pass this in if the ODBC function call that failed required a hdbc or hstmt argument.

AHstmt

> *OPTIONAL.*A handle to the ODBC statement being executed against. Pass this in if the ODBC function call that failed requires a hstmt argument.

Example

```
        if (SQLExecDirect(hstmt, (UCHAR FAR *) pSqlStmt, SQL_NTS) !=
```

```
SQL_SUCCESS)
    {
        return(db.GetNextError(db.henv, db.hdbc, hstmt));
    }
```

See also

wxDb::DispNextError (p. 296),*wxDb::DispAllErrors* (p. 295)

wxDb::GetPassword

const wxString & GetPassword()

Returns the password used to establish this connection to the datasource.

wxDb::GetTableCount

int GetTableCount()

Returns the number of wxDbTable() instances currently using this datasource connection.

wxDb::GetUsername

const wxString & GetUsername()

Returns the user name (uid) used to establish this connection to the datasource.

wxDb::Grant

bool Grant(int *privileges*, const wxString &*tableName*,const wxString &*userList* = *"PUBLIC"*)

Use this member function to GRANT privileges to users for accessing tables in the datasource.

Parameters

privileges

Use this argument to select which privileges you want to grant. Pass DB_GRANT_ALL to grant all privileges. To grant individual privileges pass one or more of the following OR'd together:

```
DB_GRANT_SELECT  = 1
DB_GRANT_INSERT  = 2
DB_GRANT_UPDATE  = 4
DB_GRANT_DELETE  = 8
DB_GRANT_ALL     = DB_GRANT_SELECT | DB_GRANT_INSERT |
                   DB_GRANT_UPDATE | DB_GRANT_DELETE
```

tableName

> The name of the table you wish to grant privileges on.

userList

> *OPTIONAL.* A comma delimited list of users to grant the privileges to. If this argument is not passed in, the privileges will be given to the general PUBLIC.

Remarks

Some databases require user names to be specified in all capital letters (i.e. Oracle). This function does not automatically capitalize the user names passed in the comma-separated list. This is the responsibility of the calling routine.

The currently logged in user must have sufficient grantor privileges for this function to be able to successfully grant the indicated privileges.

Example

```
db.Grant(DB_GRANT_SELECT | DB_GRANT_INSERT, "PARTS", "mary, sue");
```

wxDb::IsFwdOnlyCursors

bool IsFwdOnlyCursors()

This setting indicates whether this database connection was created as being capable of using only forward scrolling cursors.

This function does NOT indicate if the ODBC driver or datasource supports backward scrolling cursors. There is no standard way of detecting if the driver or datasource can support backward scrolling cursors.

If a wxDb instance was created as being capable of only forward scrolling cursors, then even if the datasource and ODBC driver support backward scrolling cursors, tables using this database connection would only be able to use forward scrolling cursors.

The default setting of whether a wxDb connection to a database allows forward-only or also backward scrolling cursors is defined in setup.h by the value of wxODBC_FWD_ONLY_CURSORS. This default setting can be overridden when the wxDb connection is initially created (see*wxDb constructor* (p. 291) and *wxDbGetConnection* (p. 289)).

Return value

Returns true if this datasource connection is defined as using only forward scrolling cursors, or false if the connection is defined as being allowed to use backward scrolling cursors and their associated functions (see note above).

Remarks

Added as of wxWidgets v2.4 release, this function is a renamed version of

wxDb::FwdOnlyCursors() to match the normal wxWidgets naming conventions for class member functions.

This function is not available in versions prior to v2.4. You should use *wxDb::FwdOnlyCursors* (p. 298) for wxWidgets versions prior to 2.4.

See also

wxDb constructor (p. 291), *wxDbGetConnection* (p. 289)

wxDb::IsOpen

bool IsOpen()

Indicates whether the database connection to the datasource is currently opened.

Remarks

This function may indicate that the database connection is open, even if the call to *wxDb::Open* (p. 307) may have failed to fully initialize the connection correctly. The connection to the database*is* open and can be used via the direct SQL commands, if this function returns true. Other functions which depend on the*wxDb::Open* (p. 307) to have completed correctly may not function as expected. The return result from *wxDb::Open* (p. 307) is the only way to know if complete initialization of this wxDb connection was successful or not. See *wxDb::Open* (p. 307) for more details on partial failures to open a connection instance.

wxDb::LogError

void LogError(**const wxString &***errMsg* **const wxString &***SQLState=""*)

errMsg

> Free-form text to display describing the error/text to be logged.

SQLState

> *OPTIONAL.* Native SQL state error. Default is 0.

Remarks

Calling this function will enter a log message in the error list maintained for the database connection. This log message is free form and can be anything the programmer wants to enter in the error list.

If SQL logging is turned on, the call to this function will also log the text into the SQL log file.

See also

wxDb::WriteSqlLog (p. 314)

wxDb::ModifyColumn

void ModifyColumn(**const wxString &***tableName* **const wxString &***ColumnName***int** *dataType* **ULONG** *columnLength=0* **const wxString &***optionalParam=""*)

Used to change certain properties of a column such as the length, or whether a column allows NULLs or not.

tableName

Name of the table that the column to be modified is in.

columnName

Name of the column to be modified. NOTE: Name of column cannot be changed with this function.

dataType

Any one of DB_DATA_TYPE_VARCHAR, DB_DATA_TYPE_INTEGER, DB_DATA_TYPE_FLOAT, DB_DATA_TYPE_DATE.

columnLength

New size of the column. Valid only for DB_DATA_TYPE_VARCHAR dataType fields. Default is 0.

optionalParam

Default is "".

Remarks

Cannot be used to modify the precision of a numeric column, therefore 'columnLength' is ignored unless the dataType is DB_DATA_TYPE_VARCHAR.

Some datasources do not allow certain properties of a column to be changed if any rows currently have data stored in that column. Those datasources that do allow columns to be changed with data in the rows many handle truncation and/or expansion in different ways. Please refer to the reference material for the datasource being used for behavioral descriptions.

Example

```
ok = pDb->ModifyColumn("CONTACTS", "ADDRESS2",
                       DB_, colDefs[j].SzDataObj,
                       wxT("NOT NULL"));
```

wxDb::Open

bool Open(**const wxString &***Dsn*, **const wxString &***Uid*,**const wxString &***AuthStr*, **bool** *failOnDataTypeUnsupported*)

bool Open(**const wxString &***inConnectStr*, **bool** *failOnDataTypeUnsupported*)

bool Open(**wxDbConnectInf** **dbConnectInf*, **bool** *failOnDataTypeUnsupported*)

bool Open(**wxDb** **copyDb*)

Opens a connection to the datasource, sets certain behaviors of the datasource to confirm to the accepted behaviors (e.g. cursor position maintained on commits), and queries the datasource for its representations of the basic datatypes to determine the form in which the data going to/from columns in the data tables are to be handled.

The second form of this function, which accepts a "wxDb *" as a parameter, can be used to avoid the overhead (execution time, database load, network traffic) which are needed to determine the data types and representations of data that are necessary for cross-datasource support by these classes.

Normally the first form of the wxDb::Open() function will open the connection and then send a series of queries to the datasource asking it for its representation of data types, and all the features it supports. If one connection to the datasource has already been made previously, the information gathered when that connection was created can just be copied to any new connections to the same datasource by passing a pointer to the first connection in as a parameter to the wxDb::Open() function. Note that this new connection created from the first connections information will use the same Dsn/Uid/AuthStr as the first connection used.

Parameters

Dsn

> datasource name. The name of the ODBC datasource as assigned when the datasource is initially set up through the ODBC data source manager.

Uid

> User ID. The name (ID) of the user you wish to connect as to the datasource. The user name (ID) determines what objects you have access to in the datasource and what datasource privileges you have. Privileges include being able to create new objects, update objects, delete objects and so on. Users and privileges are normally administered by the database administrator.

AuthStr

> The password associated with the Uid.

failOnDataTypeUnsupporte

> As part of connecting to a database, the wxDb::Open() function will query the database to find out the native types that it supports. With some databases, some data types may not be supported, or not sufficiently supported, for use with the wxODBC classes. Normally a program should fail in this case, so as not to try to use a data type that is not supported. This parameter allows the programmer to override the failure if they wish and continue on using the connection.

dbConnectInf

> Contains a DSN, Uid, Password, or a connection string to be used in opening a

new connection to the database. If a connection string is present, then the connection string will be used. If there is no connection string present, then the DSN, Uid, and Password are used.

inConnectStr

A valid ODBC connection string used to connect to a database

copyDb

Already completely configured and opened datasource connection from which all Dsn, Uid, AuthStr, connection string, and data typing information is to be copied from for use by this datasource connection. If 'copyDb' used a connection string to create its connection originally, then the connection being made by this call to wxDb::Open() will use that same connection string.

Remarks

After a wxDb instance is created, it must then be opened. When opening a datasource, there must be three pieces of information passed. The data source name, user name (ID) and the password for the user. No database activity on the datasource can be performed until the connection is opened. This is normally done at program startup and the datasource remains open for the duration of the program/module run.

It is possible to have connections to multiple datasources open at the same time to support distributed database connections by having separate instances of wxDb objects that use either the same or different Dsn/Uid/AuthStr settings.

If this function returns a value of false, it does not necessarily mean that the connection to the datasource was not opened. It may mean that some portion of the initialization of the connection failed (such as a datatype not being able to be determined how the datasource represents it). To determine if the connection to the database failed, use the *wxDb::IsOpen* (p. 306)function after receiving a false result back from this function to determine if the connection was opened or not. If this function returns false, but *wxDb::IsOpen* (p. 306)returns true, then direct SQL commands may be passed to the database connection and can be successfully executed, but use of the datatypes (such as by a wxDbTable instance) that are normally determined during open will not be possible.

> The *Dsn*, *Uid*, and *AuthStr* string pointers that are passed in are copied. NOT the strings themselves, only the pointers. The calling routine must maintain the memory for these three strings for the life of the wxDb instance.

Example

```
wxDb sampleDB(DbConnectInf.GetHenv());
if (!sampleDB.Open("Oracle 7.1 HP/UX", "gtasker", "myPassword"))
{
   if (sampleDb.IsOpen())
   {
     // Connection is open, but the initialization of
     // datatypes and parameter settings failed
   }
   else
   {
```

```
            // Error opening datasource
        }
    }
```

wxDb::RollbackTrans

bool RollbackTrans()

Function to "undo" changes made to the database. After an insert/update/delete, the operation may be "undone" by issuing this command any time before a *wxDb::CommitTrans* (p. 293) is called on the database connection.

Remarks

Transactions begin implicitly as soon as you make a change to the database. The transaction continues until either a commit or rollback is executed. Calling wxDb::RollbackTrans() will result in ALL changes done using this database connection that have not already been committed to be "undone" back to the last commit/rollback that was successfully executed.

Calling this member function rolls back ALL open (uncommitted) transactions on this ODBC connection, including all wxDbTable instances that use this connection.

See also

wxDb::CommitTrans (p. 293) for a special note on cursors

wxDb::SetDebugErrorMessages

void SetDebugErrorMessages(**bool** *state*)

state

> Either true (debug messages are logged) or false (debug messages are not logged).

Remarks

Turns on/off debug error messages from the ODBC class library. When this function is passed true, errors are reported to the user/logged automatically in a text or pop-up dialog when an ODBC error occurs. When passed false, errors are silently handled.

When compiled in release mode (FINAL=1), this setting has no affect.

See also

wxDb constructor (p. 291)

wxDb::SetSqlLogging

bool SetSqlLogging(**wxDbSqlLogState** *state*, **const wxString &***filename* = *SQL_LOG_FILENAME,* **bool** *append = false*)

Parameters

state

> Either sqlLogOFF or sqlLogON (see *enum wxDbSqlLogState* (p. 316)). Turns logging of SQL commands sent to the datasource OFF or ON.

filename

> *OPTIONAL*. Name of the file to which the log text is to be written. Default is SQL_LOG_FILENAME.

append

> *OPTIONAL*. Whether the file is appended to or overwritten. Default is false.

Remarks

When called with *sqlLogON*, all commands sent to the datasource engine are logged to the file specified by *filename*. Logging is done by embedded *wxDb::WriteSqlLog* (p. 314) calls in the database member functions, or may be manually logged by adding calls to *wxDb::WriteSqlLog* (p. 314) in your own source code.

When called with *sqlLogOFF*, the logging file is closed, and any calls to *wxDb::WriteSqlLog* (p. 314) are ignored.

wxDb::SQLColumnName

const wxString SQLColumnName(**const char *** *colName*)

Returns the column name in a form ready for use in SQL statements. In most cases, the column name is returned verbatim. But some databases (e.g. MS Access, SQL Server, MSDE) allow for spaces in column names, which must be specially quoted. For example, if the datasource allows spaces in the column name, the returned string will have the correct enclosing marks around the name to allow it to be properly included in a SQL statement for the DBMS that is currently connected to with this connection.

Parameters

colName

> Native name of the column in the table that is to be evaluated to determine if any special quoting marks needed to be added to it before including the column name in a SQL statement

See also

wxDb::SQLTableName (p. 312)

wxDb::SQLTableName

const wxString SQLTableName(**const char *** *tableName*)

Returns the table name in a form ready for use in SQL statements. In most cases, the table name is returned verbatim. But some databases (e.g. MS Access, SQL Server, MSDE) allow for spaces in table names, which must be specially quoted. For example, if the datasource allows spaces in the table name, the returned string will have the correct enclosing marks around the name to allow it to be properly included in a SQL statement for the data source that is currently connected to with this connection.

Parameters

tableName

> Native name of the table that is to be evaluated to determine if any special quoting marks needed to be added to it before including the table name in a SQL statement

See also

wxDb::SQLColumnName (p. 311)

wxDb::TableExists

bool TableExists(**const wxString &***tableName*, **const wxChar ****userID=NULL*, **const wxString &***path=""*)

Checks the ODBC datasource for the existence of a table. If a *userID*is specified, then the table must be accessible by that user (user must have at least minimal privileges to the table).

Parameters

tableName

> Name of the table to check for the existence of.

userID

> Owner of the table (also referred to as schema). Specify a userID when the datasource you are connected to allows multiple unique tables with the same name to be owned by different users. *userID*is evaluated as follows:

```
userID == NULL  ... UserID is ignored (DEFAULT)
userID == ""    ... UserID set equal to 'this->uid'
userID != ""    ... UserID set equal to 'userID'
```

Remarks

tableName may refer to a table, view, alias or synonym.

This function does not indicate whether or not the user has privileges to query or perform

other functions on the table. Use the *wxDb::TablePrivileges* (p. 313) to determine if the user has sufficient privileges or not.

wxDb::TablePrivileges

bool TablePrivileges(**const wxString &***tableName*, **const wxString &***priv*,**const wxChar ****userID=NULL*, **const wxChar ****schema=NULL*,**const wxString &***path=""*)

Checks the ODBC datasource for the existence of a table. If a *userID*is specified, then the table must be accessible by that user (user must have at least minimal privileges to the table).

Parameters

tableName

>   Name of the table on which to check privileges.*tableName* may refer to a table, view, alias or synonym.

priv

>   The table privilege being evaluated. May be one of the following (or a datasource specific privilege):

```
SELECT      : The connected user is permitted to retrieve data for
              one or more columns of the table.

INSERT      : The connected user is permitted to insert new rows
              containing data for one or more columns into the
              table.

UPDATE      : The connected user is permitted to update the data in
              one or more columns of the table.

DELETE      : The connected user is permitted to delete rows of
              data from the table.

REFERENCES  : Is the connected user permitted to refer to one or
              more columns of the table within a constraint (for
              example, a unique, referential, or table check
              constraint).
```

userID

>   *OPTIONAL.* User for which to determine if the privilege specified to be checked is granted or not. Default is "".*userID* is evaluated as follows:

```
userID == NULL  ... NOT ALLOWED!
userID == ""    ... UserID set equal to 'this->uid'
userID != ""    ... UserID set equal to 'userID'
```

schema

OPTIONAL. Owner of the table. Specify a userID when the datasource you are connected to allows multiple unique tables with the same name to be owned by different users. Specifying the table owner makes determination of the users privileges MUCH faster. Default is NULL. *userID* is evaluated as follows:

```
schema == NULL  ... Any owner (DEFAULT)
schema == ""    ... Owned by 'this->uid'
schema != ""    ... Owned by userID specified in 'schema'
```

path

OPTIONAL. Path to the table. Default is "". Currently unused.

Remarks

The scope of privilege allowed to the connected user by a given table privilege is datasource dependent.

For example, the privilege UPDATE might allow the connected user to update all columns in a table on one datasource, but only those columns for which the grantor (the user that granted the connected user) has the UPDATE privilege on another datasource.

Looking up a user's privileges to a table can be time consuming depending on the datasource and ODBC driver. This time can be minimized by passing a *schema*as a parameter. With some datasources/drivers, the difference can be several seconds of time difference.

wxDb::TranslateSqlState

int TranslateSqlState(**const wxString &***SQLState*)

Converts an ODBC sqlstate to an internal error code.

Parameters

SQLState

State to be converted.

Return value

Returns the internal class DB_ERR code. See *wxDb::DB_STATUS* (p. 284) definition.

wxDb::WriteSqlLog

bool WriteSqlLog(**const wxString &***logMsg*)

Parameters

logMsg

Free form string to be written to the log file.

Remarks

Very useful debugging tool that may be turned on/off during run time (see (see *wxDb::SetSqlLogging* (p. 310) for details on turning logging on/off). The passed in string *logMsg* will be written to a log file if SQL logging is turned on.

Return value

If SQL logging is off when a call to WriteSqlLog() is made, or there is a failure to write the log message to the log file, the function returns false without performing the requested log, otherwise true is returned.

See also

wxDb::SetSqlLogging (p. 310)

wxDbColDataPtr

Pointer to dynamic column definitions for use with a wxDbTable instance. Currently there are no member functions for this class.

See the *database classes overview* (p. 1820) for an introduction to using the ODBC classes.

```
void   *PtrDataObj;
int     SzDataObj;
SWORD   SqlCtype;
```

wxDbColDef

This class is used to hold information about the columns bound to an instance of a wxDbTable object.

Each instance of this class describes one column in the wxDbTable object. When calling the *wxDb constructor* (p. 291), a parameter passed in indicates the number of columns that will be defined for the wxDbTable object. The constructor uses this information to allocate adequate memory for all of the column descriptions in your wxDbTable object. Private member wxDbTable::colDefs is a pointer to this chunk of memory maintained by the wxDbTable class (and can be retrieved using the*wxDbTable::GetColDefs* (p. 339) function). To access the nth column definition of your wxDbTable object, just reference wxDbColDefs element [n - 1].

Typically, *wxDbTable::SetColDefs* (p. 352) is used to populate an array of these data structures for the wxDbTable instance.

Currently there are no accessor functions for this class, so all members are public.

```
wxChar  ColName[DB_MAX_COLUMN_NAME_LEN+1];  // Column Name
int     DbDataType;    - Logical Data Type;
                         e.g. DB_DATA_TYPE_INTEGER
SWORD   SqlCtype;      - C data type; e.g. SQL_C_LONG
```

```
void   *PtrDataObj;   - Address of the data object
int    SzDataObj;     - Size, in bytes, of the data object
bool   KeyField;      - Is column part of the PRIMARY KEY for the
                        table? -- Date fields should NOT be
                        KeyFields
bool   Updateable;    - Column is updateable?
bool   InsertAllowed; - Column included in INSERT statements?
bool   DerivedCol;    - Column is a derived value?
SDWORD CbValue;       - !!!Internal use only!!!
bool   Null;          - NOT FULLY IMPLEMENTED
                        Allows NULL values in Inserts and Updates
```

See also

database classes overview (p. 1820),*wxDbTable::GetColDefs* (p. 339), *wxDb constructor* (p. 291)

Include files

<wx/db.h>

wxDbColDef::Initialize

Simply initializes all member variables to a cleared state. Called by the constructor automatically.

wxDbColFor

Beginning support for handling international formatting specifically on dates and floats.

```
wxString      s_Field;       // Formated String for Output
wxString      s_Format[7];   // Formated Objects - TIMESTAMP has
                                the biggest (7)
wxString      s_Amount[7];   // Formated Objects - amount of
                                things that can be formatted
int           i_Amount[7];   // Formated Objects -
                                   TT MM YYYY HH MM SS m
int           i_Nation;      // 0 = timestamp
                                1 = EU
                                2 = UK
                                3 = International
                                4 = US
int           i_dbDataType;  // conversion of the 'sqlDataType'
                                to the generic data type used by
                                these classes
SWORD         i_sqlDataType;
```

The constructor for this class initializes all the values to zero or NULL.

The destructor does nothing at this time.

Only one function is provided with this class currently.

See the *database classes overview* (p. 1820) for an introduction to using the ODBC classes.

Include files

<wx/db.h>

wxDbColFor::Format

int Format(**int** *Nation*, **int** *dbDataType*,**SWORD** *sqlDataType*, **short** *columnSize*,**short** *decimalDigits*)

Work in progress, and should be inter-related with wxLocale eventually.

wxDbColFor::Initialize

Simply initializes all member variables to a cleared state. Called by the constructor automatically.

wxDbColInf

Used with the *wxDb::GetColumns* (p. 299) functions for obtaining all retrievable information about a column's definition.

```
wxChar        catalog[128+1];
wxChar        schema[128+1];
wxChar        tableName[DB_MAX_TABLE_NAME_LEN+1];
wxChar        colName[DB_MAX_COLUMN_NAME_LEN+1];
SWORD         sqlDataType;
wxChar        typeName[128+1];
SWORD         columnSize;
SWORD         bufferLength;
short         decimalDigits;
short         numPrecRadix;
short         nullable;
wxChar        remarks[254+1];
int           dbDataType;  // conversion of the 'sqlDataType'
                           // to the generic data type used by
                           // these classes
int           PkCol;       // Primary key column
                                  0 = No
                                  1 = First Key
                                  2 = Second Key, etc...
wxChar        PkTableName[DB_MAX_TABLE_NAME_LEN+1];
                           // Tables that use this PKey as a FKey
int           FkCol;       // Foreign key column
                                  0 = No
                                  1 = First Key
                                  2 = Second Key, etc...
wxChar        FkTableName[DB_MAX_TABLE_NAME_LEN+1];
                           // Foreign key table name
wxDbColFor  *pColFor;      // How should this column be formatted
```

The constructor for this class initializes all the values to zero, "", or NULL.

The destructor for this class takes care of deleting the pColFor member if it is non-NULL.

See the *database classes overview* (p. 1820) for an introduction to using the ODBC classes.

Include files

<wx/db.h>

wxDbColInf::Initialize

Simply initializes all member variables to a cleared state. Called by the constructor automatically.

wxDbConnectInf

This class is used for holding the data necessary for connecting to the ODBC datasource. That information includes: SQL environment handle, datasource name, user ID, password and default directory path (used with dBase). Other optional fields held in this class are and file type, both for future functions planned to be added for creating/manipulating datasource definitions.

wxDbConnectInf::wxDbConnectInf

 wxDbConnectInf()

Default constructor.

 wxDbConnectInf(**HENV** *henv*, **const wxString &***dsn*,**const wxString &***userID=""*, **const wxString &***password*,**const wxString &***defaultDir=""*, **const wxString &***description="",***const wxString &***fileType=""*)

Constructor which allows initial settings of all the classes member variables.

See the special note below on the henv parameter for forcing this constructor to create a SQL environment handle automatically, rather than needing to pass one in to the function.

Parameters

henv

> Environment handle used for this connection. See *wxDConnectInf::AllocHenv* (p. 320) for how to create an SQL environment handle. NOTE: Passing in a NULL for this parameter will inform the constructor that it should create its own SQL

environment handle. If NULL is passed for this parameter, the constructor will call *wxDConnectInf::AllocHenv* (p. 320) internally. A flag is set internally also to indicate that the HENV was created by the constructor so that when the default class destructor is called, the destructor will call *wxDConnectInf::FreeHenv* (p. 320) to free the environment handle automatically.

dsn

Name of the datasource to be used in creating wxDb instances for creating connection(s) to a datasource.

userID

OPTIONAL Many datasources allow (or even require) use of a username to determine privileges that connecting user is allowed to have when accessing the datasource or the data tables. Default is "".

password

OPTIONAL Password to be associated with the user ID specified in 'userID'. Default is "".

defaultDir

OPTIONAL Used for datasources which require the path to where the data file is stored to be specified. dBase is one example of the type of datasource which requires this information. Default is "".

description

OPTIONAL **FUTURE USE** Default is "".

fileType

OPTIONAL **FUTURE USE** Default is "".

Remarks

It is strongly recommended that programs use the longer form of the constructor and allow the constructor to create the SQL environment handle automatically, and manage the destruction of the handle.

Example

```
   wxDbConnectInf *DbConnectInf;

 DbConnectInf = new wxDbConnectInf(0,"MY_DSN", "MY_USER", "MY_PASSWORD");

 ....the rest of the program

 delete DbConnectInf;
```

See also

wxDConnectInf::AllocHenv (p. 320),*wxDConnectInf::FreeHenv* (p. 320)

wxDbConnectInf::~wxDbConnectInf

~wxDbConnectInf()

Handles the default destruction of the instance of the class. If the long form of the *wxDConnectInf* (p. 318) was used, then this destructor also takes care of calling *wxDConnectInf::FreeHenv* (p. 320) to free the SQL environment handle.

wxDbConnectInf::AllocHenv

bool AllocHenv()

Allocates a SQL environment handle that will be used to interface with an ODBC datasource.

Remarks

This function can be automatically called by the long from of the*wxDbConnectInf* (p. 318) constructor.

wxDbConnectInf::FreeHenv

void FreeHenv()

Frees the SQL environment handle being managed by the instance of this class.

Remarks

If the SQL environment handle was created using the long form of the *wxDbConnectInf* (p. 318) constructor, then the flag indicating that the HENV should be destroyed when the classes destructor is called is reset to be false, so that any future handles created using the *wxDbConnectInf::AllocHenv* (p. 320) function must be manually released with a call to this function.

wxDbConnectInf::Initialize

Simply initializes all member variables to a cleared state. Called by the constructor automatically.

wxDbConnectInf::GetAuthStr

const wxChar * GetAuthStr()

Accessor function to return the password assigned for this class instance that will be used with the user ID.

Synonymous with *wxDbConnectInf::GetPassword* (p. 321)

wxDbConnectInf::GetDefaultDir

const wxChar * GetDefaultDir()

Accessor function to return the default directory in which the datasource's data table is stored. This directory is only used for file based datasources like dBase. MS-Access does not require this to be set, as the path is set in the ODBC Administrator for MS-Access.

wxDbConnectInf::GetDescription

const wxChar * GetDescription()

Accessor function to return the description assigned for this class instance.

NOTE: Description is a FUTURE USE item and is unused currently.

wxDbConnectInf::GetDsn

const wxChar * GetDsn()

Accessor function to return the datasource name assigned for this class instance.

wxDbConnectInf::GetFileType

const wxChar * GetFileType()

Accessor function to return the filetype of the ODBC datasource assigned for this class instance.

NOTE: FileType is a FUTURE USE item and is unused currently.

wxDbConnectInf::GetHenv

const HENV GetHenv()

Accessor function to return the SQL environment handle being managed by this class instance.

wxDbConnectInf::GetPassword

const wxChar * GetPassword()

Accessor function to return the password assigned for this class instance that will be used with the user ID.

Synonymous with *wxDbConnectInf::GetAuthStr* (p. 320)

wxDbConnectInf::GetUid

const wxChar * GetUid()

Accessor function to return the user ID assigned for this class instance.

wxDbConnectInf::GetUserID

const wxChar * GetUserID()

Accessor function to return the user ID assigned for this class instance.

wxDbConnectInf::SetAuthStr

SetAuthStr(const wxString &authstr)

Accessor function to assign the password for this class instance that will be used with the user ID.

Synonymous with *wxDbConnectInf::SetPassword* (p. 323)

wxDbConnectInf::SetDefaultDir

SetDefaultDir(const wxString &defDir)

Accessor function to assign the default directory in which the datasource's data table is stored. This directory is only used for file based datasources like dBase. MS-Access does not require this to be set, as the path is set in the ODBC Administrator for MS-Access.

wxDbConnectInf::SetDescription

SetDescription(const wxString &desc)

Accessor function to assign the description assigned for this class instance.

NOTE: Description is a FUTURE USE item and is unused currently.

wxDbConnectInf::SetDsn

SetDsn(const wxString &dsn)

Accessor function to assign the datasource name for this class instance.

wxDbConnectInf::SetFileType

SetFileType(const wxString &)

Accessor function to return the filetype of the ODBC datasource assigned for this class instance.

NOTE: FileType is a FUTURE USE item and is unused currently.

wxDbConnectInf::SetHenv

void SetHenv(**const HENV** *henv*)

Accessor function to set the SQL environment handle for this class instance.

wxDbConnectInf::SetPassword

SetPassword(const wxString &password)

Accessor function to assign the password for this class instance that will be used with the user ID.

Synonymous with *wxDbConnectInf::SetAuthStr* (p. 322)

wxDbConnectInf::SetUid

SetUid(const wxString &uid)

Accessor function to set the user ID for this class instance.

wxDbConnectInf::SetUserID

SetUserID(const wxString &userID)

Accessor function to assign the user ID for this class instance.

wxDbIdxDef

Used in creation of non-primary indexes. Currently there are no member functions for this class.

```
wxChar   ColName[DB_MAX_COLUMN_NAME_LEN+1]
                            // Name of column
bool     Ascending       // Is index maintained in
                                ASCENDING sequence?
```

There are no constructors/destructors as of this time, and no member functions.

See the *database classes overview* (p. 1820) for an introduction to using the ODBC classes.

wxDbInf

Contains information regarding the database connection (datasource name, number of tables, etc). A pointer to a wxDbTableInf is included in this class so a program can create a wxDbTableInf array instance to maintain all information about all tables in the datasource to have all the datasource's information in one memory structure.

Primarily, this class is used internally by the wxWidgets ODBC classes.

```
wxChar        catalog[128+1];
wxChar        schema[128+1];  // typically means owner of table(s)
int           numTables;      // How many tables does this
                                 datasource have
wxDbTableInf *pTableInf;      // Equals a new
                                 wxDbTableInf[numTables];
```

The constructor for this class initializes all the values to zero, "", or NULL.

The destructor for this class takes care of deleting the pTableInf member if it is non-NULL.

See the *database classes overview* (p. 1820) for an introduction to using the ODBC classes.

Include files

<wx/db.h>

wxDbInf::Initialize

Simply initializes all member variables to a cleared state. Called by the constructor automatically.

wxDbTable

A wxDbTable instance provides re-usable access to rows of data in a table contained within the associated ODBC datasource

See the *database classes overview* (p. 1820) for an introduction to using the ODBC classes.

Include files

<wx/dbtable.h>

<wx/db.h>

Helper classes and data structures

The following classes and structs are defined in dbtable.cpp/.h for use with the wxDbTable class.

- *wxDbColDef* (p. 315)

- *wxDbColDataPtr* (p. 315)

- *wxDbIdxDef* (p. 323)

Constants

```
wxDB_DEFAULT_CURSOR   Primary cursor normally used for cursor based
                      operations.

wxDB_QUERY_ONLY       Used to indicate whether a table that is opened
                      is for query only, or if insert/update/deletes
                      will be performed on the table.  Less overhead
                      (cursors and memory) are allocated for query
                      only tables, plus read access times are faster
                      with some datasources.

wxDB_ROWID_LEN        [Oracle only] - Used when CanUpdateByRowID()
                      is true.  Optimizes updates so they are faster
                      by updating on the Oracle-specific ROWID column
                      rather than some other index.


wxDB_DISABLE_VIEW     Use to indicate when a database view should not
                      be if a table is normally set up to use a view.
                      [Currently unsupported.]
```

wxDbTable::wxDbTable

 wxDbTable(**wxDb ****pwxDb*, **const wxString &***tblName*, **const UWORD** *numColumns*, **const wxString &***qryTblName = "",* **bool** *qryOnly = !wxDB_QUERY_ONLY*, **const wxString &***tblPath = "")*

Default constructor.

Parameters

pwxDb

>   Pointer to the wxDb instance to be used by this wxDbTable instance.

tblName

>   The name of the table in the RDBMS.

numColumns

> The number of columns in the table. (Do NOT include the ROWID column in the count if using Oracle).

qryTblName

> *OPTIONAL*. The name of the table or view to base your queries on. This argument allows you to specify a table/view other than the base table for this object to base your queries on. This allows you to query on a view for example, but all of the INSERT, UPDATE and DELETES will still be performed on the base table for this wxDbTable object. Basing your queries on a view can provide a substantial performance increase in cases where your queries involve many tables with multiple joins. Default is "".

qryOnly

> *OPTIONAL*. Indicates whether the table will be accessible for query purposes only, or should the table create the necessary cursors to be able to insert, update, and delete data from the table. Default is !wxDB_QUERY_ONLY.

tblPath

> *OPTIONAL*. Some datasources (such as dBase) require a path to where the table is stored on the system. Default is "".

wxDbTable::wxDbTable

virtual ~wxDbTable()

Virtual default destructor.

wxDbTable::BuildDeleteStmt

void BuildDeleteStmt(**wxString &***pSqlStmt*,**int** *typeOfDel*, **const wxString &***pWhereClause=""*)

Constructs the full SQL statement that can be used to delete all rows matching the criteria in the pWhereClause.

Parameters

pSqlStmt

> Pointer to buffer for the SQL statement retrieved. To be sure you have adequate space allocated for the SQL statement, allocate DB_MAX_STATEMENT_LEN bytes.

typeOfDel

> The type of delete statement being performed. Can be one of three values: DB_DEL_KEYFIELDS, DB_DEL_WHERE or DB_DEL_MATCHING

pWhereClause

> *OPTIONAL*. If the typeOfDel is DB_DEL_WHERE, then you must also pass in a SQL WHERE clause in this argument. Default is "".

Remarks

This member function constructs a SQL DELETE statement. This can be used for debugging purposes if you are having problems executing your SQL statement.

WHERE and FROM clauses specified using *wxDbTable::SetWhereClause* (p. 357)and *wxDbTable::SetFromClause* (p. 355) are ignored by this function.

wxDbTable::BuildSelectStmt

void BuildSelectStmt(**wxString &***pSqlStmt*,**int** *typeOfSelect*, **bool** *distinct*)

Constructs the full SQL statement that can be used to select all rows matching the criteria in the pWhereClause. This function is called internally in the wxDbTable class whenever the function *wxDbTable::Query* (p. 347)is called.

NOTE: Only the columns specified in *wxDbTable::SetColDefs* (p. 352)statements are included in the list of columns returned by the SQL statement created by a call to this function.

Parameters

pSqlStmt

> Pointer to storage for the SQL statement retrieved. To be sure you have adequate space allocated for the SQL statement, allocate DB_MAX_STATEMENT_LEN bytes.

typeOfSelect

> The type of select statement being performed. Can be one of four values: DB_SELECT_KEYFIELDS, DB_SELECT_WHERE, DB_SELECT_MATCHING or DB_SELECT_STATEMENT.

distinct

> Whether to select distinct records only.

Remarks

This member function constructs a SQL SELECT statement. This can be used for debugging purposes if you are having problems executing your SQL statement.

WHERE and FROM clauses specified using*wxDbTable::SetWhereClause* (p. 357)and *wxDbTable::SetFromClause* (p. 355) are ignored by this function.

wxDbTable::BuildUpdateStmt

void BuildUpdateStmt(**wxString &***pSqlStmt*, **int** *typeOfUpd*,**const wxString
&***pWhereClause=""*)

Constructs the full SQL statement that can be used to update all rows matching the
criteria in the pWhereClause.

If typeOfUpdate is DB_UPD_KEYFIELDS, then the current values in the bound columns
are used to determine which row(s) in the table are to be updated. The exception to this
is when a datasource supports ROW IDs (Oracle). The ROW ID column is used for
efficiency purposes when available.

NOTE: Only the columns specified in *wxDbTable::SetColDefs* (p. 352)statements are
included in the list of columns updated by the SQL statement created by a call to this
function. Any column definitions that were defined as being non-updateable will be
excluded from the SQL UPDATE statement created by this function.

Parameters

pSqlStmt

> Pointer to storage for the SQL statement retrieved. To be sure you have adequate
> space allocated for the SQL statement, allocate DB_MAX_STATEMENT_LEN
> bytes.

typeOfUpdate

> The type of update statement being performed. Can be one of two values:
> DB_UPD_KEYFIELDS or DB_UPD_WHERE.

pWhereClause

> *OPTIONAL*. If the typeOfUpdate is DB_UPD_WHERE, then you must also pass in
> a SQL WHERE clause in this argument. Default is "".

Remarks

This member function allows you to see what the SQL UPDATE statement looks like that
the ODBC class library builds. This can be used for debugging purposes if you are
having problems executing your SQL statement.

WHERE and FROM clauses specified using *wxDbTable::SetWhereClause* (p. 357)and
wxDbTable::SetFromClause (p. 355) are ignored by this function.

wxDbTable::BuildWhereClause

 void BuildWhereClause(**wxString &***pWhereClause*,**int** *typeOfWhere*, **const wxString
&***qualTableName=""*,**bool** *useLikeComparison=false*)

Constructs the portion of a SQL statement which would follow the word 'WHERE' in a
SQL statement to be passed to the datasource. The returned string does NOT include
the word 'WHERE'.

Parameters

pWhereClause

> Pointer to storage for the SQL statement retrieved. To be sure you have adequate space allocated for the SQL statement, allocate DB_MAX_STATEMENT_LEN bytes.

typeOfWhere

> The type of where clause to generate. Can be one of two values: DB_WHERE_KEYFIELDS or DB_WHERE_MATCHING.

qualTableName

> *OPTIONAL*. Prepended to all base table column names. For use when a FROM clause has been specified with the*wxDbTable::SetFromClause* (p. 355), to clarify which table a column name reference belongs to. Default is "".

useLikeComparison

> *OPTIONAL*. Should the constructed WHERE clause utilize the LIKE comparison operator. If false, then the '=' operator is used. Default is false.

Remarks

This member function allows you to see what the SQL WHERE clause looks like that the ODBC class library builds. This can be used for debugging purposes if you are having problems executing your own SQL statements.

If using 'typeOfWhere' set to DB_WHERE_MATCHING, any bound columns currently containing a NULL value are not included in the WHERE clause's list of columns to use in the comparison.

wxDbTable::CanSelectForUpdate

bool CanSelectForUpdate()

Use this function to determine if the datasource supports SELECT ... FOR UPDATE. When the keywords "FOR UPDATE" are included as part of your SQL SELECT statement, all records *retrieved* (not just queried, but actually retrieved) using*wxDbTable::GetNext* (p. 341), etc) from the result set are locked.

Remarks

Not all datasources support the "FOR UPDATE" clause, so you must use this member function to determine if the datasource currently connected to supports this behavior or not before trying to select using "FOR UPDATE".

If the wxDbTable instance was created with the parameter wxDB_QUERY_ONLY, then this function will return false. For all known databases which do not support the FOR UPDATE clause, this function will return false also.

wxDbTable::CanUpdateByROWID

bool CanUpdateByROWID()

CURRENTLY ONLY POSSIBLE IF USING ORACLE.

--- CURRENTLY DISABLED FOR *ALL* DATASOURCES --- NOV 1 2000 - gt

Every Oracle table has a hidden column named ROWID. This is a pointer to the physical location of the record in the datasource and allows for very fast updates and deletes. The key is to retrieve this ROWID during your query so it is available during an update or delete operation.

Use of the ROWID feature is always handled by the class library except in the case of *wxDbTable::QueryBySqlStmt* (p. 348). Since you are passing in the SQL SELECT statement, it is up to you to include the ROWID column in your query. If you do not, the application will still work, but may not be as optimized. The ROWID is always the last column in the column list in your SQL SELECT statement. The ROWID is not a column in the normal sense and should not be considered part of the column definitions for the wxDbTable object.

Remarks

The decision to include the ROWID in your SQL SELECT statement must be deferred until runtime since it depends on whether you are connected to an Oracle datasource or not.

Example

```
    // Incomplete code sample
    wxDbTable parts;
    .....
    if (parts.CanUpdateByROWID())
    {
        // Note that the ROWID column must always be the last column
selected
        sqlStmt = "SELECT PART_NUM, PART_DESC, ROWID" FROM PARTS";
    }
    else
        sqlStmt = "SELECT PART_NUM, PART_DESC FROM PARTS";
```

wxDbTable::ClearMemberVar

void ClearMemberVar(**UWORD** *colNumber*, **bool** *setToNull=false*)

Same as *wxDbTable::ClearMemberVars* (p. 331) except that this function clears only the specified column of its values, and optionally sets the column to be a NULL column.

colNumber

Column number that is to be cleared. This number (between 0 and (numColumns-1)) is the index of the column definition created using the*wxDbTable::SetColDefs* (p. 352) function.

setToNull

> *OPTIONAL*. Indicates whether the column should be flagged as being a NULL value stored in the bound memory variable. If true, then any value stored in the bound member variable is cleared. Default is false.

wxDbTable::ClearMemberVars

void ClearMemberVars(**bool** *setToNull=false*)

Initializes all bound columns of the wxDbTable instance to zero. In the case of a string, zero is copied to the first byte of the string.

setToNull

> *OPTIONAL*. Indicates whether all columns should be flagged as having a NULL value stored in the bound memory variable. If true, then any value stored in the bound member variable is cleared. Default is false.

Remarks

This is useful before calling functions such as *wxDbTable::QueryMatching* (p. 350) or *wxDbTable::DeleteMatching* (p. 336) since these functions build their WHERE clauses from non-zero columns. To call either *wxDbTable::QueryMatching* (p. 350) or *wxDbTable::DeleteMatching* (p. 336) use this sequence:

```
1)   ClearMemberVars()
2)   Assign columns values you wish to match on
3)   Call wxDbTable::QueryMatching() or wxDbTable::DeleteMatching()
```

wxDbTable::CloseCursor

bool CloseCursor(**HSTMT***cursor*)

Closes the specified cursor associated with the wxDbTable object.

Parameters

cursor

> The cursor to be closed.

Remarks

Typically handled internally by the ODBC class library, but may be used by the programmer if desired.

DO NOT CLOSE THE wxDB_DEFAULT_CURSOR!

wxDbTable::Count

ULONG Count(**const wxString &***args="*"*)

Returns the number of records which would be in the result set using the current query parameters specified in the WHERE and FROM clauses.

Parameters

args

> *OPTIONAL*. This argument allows the use of the DISTINCT keyword against a column name to cause the returned count to only indicate the number of rows in the result set that have a unique value in the specified column. An example is shown below. Default is "*", meaning a count of the total number of rows matching is returned, regardless of uniqueness.

Remarks

This function can be called before or after an actual query to obtain the count of records in the result set. Count() uses its own cursor, so result set cursor positioning is not affected by calls to Count().

WHERE and FROM clauses specified using*wxDbTable::SetWhereClause* (p. 357)and *wxDbTable::SetFromClause* (p. 355) ARE used by this function.

Example

```
USERS TABLE

FIRST_NAME      LAST_NAME
-----------     ----------
John            Doe
Richard         Smith
Michael         Jones
John            Carpenter

// Incomplete code sample
wxDbTable users;
.....
users.SetWhereClause("");

// This Count() will return 4, as there are four users listed above
// that match the query parameters
totalNumberOfUsers = users.Count();

// This Count() will return 3, as there are only 3 unique first names
// in the table above - John, Richard, Michael.
totalNumberOfUniqueFirstNames = users.Count("DISTINCT FIRST_NAME");
```

wxDbTable::CreateIndex

bool CreateIndex(**const wxString &***IndexName*, **bool** *unique*,**UWORD** *numIndexColumns*, **wxDbIdxDef** **pIndexDefs*,**bool** *attemptDrop=true*)

This member function allows you to create secondary (non primary) indexes on your tables. You first create your table, normally specifying a primary index, and then create any secondary indexes on the table. Indexes in relational model are not required. You

do not need indexes to look up records in a table or to join two tables together. In the relational model, indexes, if available, provide a quicker means to look up data in a table. To enjoy the performance benefits of indexes, the indexes must be defined on the appropriate columns and your SQL code must be written in such a way as to take advantage of those indexes.

Parameters

IndexName

>   Name of the Index. Name must be unique within the table space of the datasource.

unique

>   Indicates if this index is unique.

numIndexColumns

>   Number of columns in the index.

pIndexDefs

>   A pointer to an array *wxDbIdxDef* (p. 323) structures.

attemptDrop

>   *OPTIONAL*. Indicates if the function should try to execute a *wxDbTable::DropIndex* (p. 337) on the index name provided before trying to create the index name. Default is true.

Remarks

The first parameter, index name, must be unique and should be given a meaningful name. Common practice is to include the table name as a prefix in the index name (e.g. For table PARTS, you might want to call your index PARTS_Index1). This will allow you to easily view all of the indexes defined for a given table grouped together alphabetically.

The second parameter indicates if the index is unique or not. Uniqueness is enforced at the RDBMS level preventing rows which would have duplicate indexes from being inserted into the table when violating a unique index's uniqueness.

In the third parameter, specify how many columns are in your index. This number must match the number of columns defined in the 'pIndexDefs' parameter.

The fourth parameter specifies which columns make up the index using the*wxDbIdxDef* (p. 323) structure. For each column in the index, you must specify two things, the column name and the sort order (ascending / descending). See the example below to see how to build and pass in the *wxDbIdxDef* (p. 323) structure.

The fifth parameter is provided to handle the differences in datasources as to whether they will automatically overwrite existing indexes with the same name or not. Some datasources require that the existing index must be dropped first, so this is the default behavior.

Some datasources (MySQL, and possibly others) require columns which are to be part of an index to be defined as NOT NULL. When this function is called, if a column is not defined to be NOT NULL, a call to this function will modify the column definition to change any columns included in the index to be NOT NULL. In this situation, if a NULL value already exists in one of the columns that is being modified, creation of the index will fail.

PostGres is unable to handle index definitions which specify whether the index is ascending or descending, and defaults to the system default when the index is created.

It is not necessary to call *wxDb::CommitTrans* (p. 293)after executing this function.

Example

```
// Create a secondary index on the PARTS table
wxDbIdxDef  IndexDef[2];  // 2 columns make up the index

wxStrcpy(IndexDef[0].ColName, "PART_DESC");  // Column 1
IndexDef[0].Ascending = true;

wxStrcpy(IndexDef[1].ColName, "SERIAL_NO");  // Column 2
IndexDef[1].Ascending = false;

// Create a name for the index based on the table's name
wxString indexName;
indexName.Printf("%s_Index1",parts->GetTableName());
parts->CreateIndex(indexName, true, 2, IndexDef);
```

wxDbTable::CreateTable

bool CreateTable(**bool** *attemptDrop=true*)

Creates a table based on the definitions previously defined for this wxDbTable instance.

Parameters

attemptDrop

> *OPTIONAL*. Indicates whether the driver should attempt to drop the table before trying to create it. Some datasources will not allow creation of a table if the table already exists in the table space being used. Default is true.

Remarks

This function creates the table and primary index (if any) in the table space associated with the connected datasource. The owner of these objects will be the user id that was given when *wxDb::Open* (p. 307) was called. The objects will be created in the default schema/table space for that user.

In your derived wxDbTable object constructor, the columns and primary index of the table are described through the *wxDbColDef* (p. 315) structure.*wxDbTable::CreateTable* (p. 334) uses this information to create the table and to add the primary index. See*wxDbTable* (p. 324) ctor and wxDbColDef description for additional information on describing the columns of the table.

It is not necessary to call *wxDb::CommitTrans* (p. 293)after executing this function.

wxDbTable::DB_STATUS

bool DB_STATUS()

Accessor function that returns the wxDb private member variable DB_STATUS for the database connection used by this instance of wxDbTable.

wxDbTable::Delete

bool Delete()

Deletes the row from the table indicated by the current cursor.

Remarks

Use *wxDbTable::GetFirst* (p. 339),*wxDbTable::GetLast* (p. 340),*wxDbTable::GetNext* (p. 341) or*wxDbTable::GetPrev* (p. 342) to position the cursor to a valid record. Once positioned on a record, call this function to delete the row from the table.

A *wxDb::CommitTrans* (p. 293) or*wxDb::RollbackTrans* (p. 310) must be called after use of this function to commit or rollback the deletion.

NOTE: Most datasources have a limited size "rollback" segment. This means that it is only possible to insert/update/delete a finite number of rows without performing a *wxDb::CommitTrans* (p. 293) or*wxDb::RollbackTrans* (p. 310). Size of the rollback segment varies from database to database, and is user configurable in most databases. Therefore it is usually best to try to perform a commit or rollback at relatively small intervals when processing a larger number of actions that insert/update/delete rows in a table.

wxDbTable::DeleteCursor

bool DeleteCursor(HSTMT *hstmtDel)

Allows a program to delete a cursor.

Parameters

hstmtDel

> Handle of the cursor to delete.

Remarks

For default cursors associated with the instance of wxDbTable, it is not necessary to specifically delete the cursors. This is automatically done in the wxDbTable destructor.

NOTE: If the cursor could not be deleted for some reason, an error is logged indicating

the reason. Even if the cursor could not be deleted, the HSTMT that is passed in is deleted, and the pointer is set to NULL.

DO NOT DELETE THE wxDB_DEFAULT_CURSOR!

wxDbTable::DeleteMatching

bool DeleteMatching()

This member function allows you to delete records from your wxDbTable object by specifying the data in the columns to match on.

Remarks

To delete all users with a first name of "JOHN", do the following:

1. Clear all "columns" using wxDbTable::ClearMemberVars().

2. Set the FIRST_NAME column equal to "JOHN".

3. Call wxDbTable::DeleteMatching().

The WHERE clause is built by the ODBC class library based on all non-NULL columns. This allows deletion of records by matching on any column(s) in your wxDbTable instance, without having to write the SQL WHERE clause.

A *wxDb::CommitTrans* (p. 293) or *wxDb::RollbackTrans* (p. 310) must be called after use of this function to commit or rollback the deletion.

NOTE: Row(s) should be locked before deleting them to make sure they are not already in use. This can be achieved by calling *wxDbTable::QueryMatching* (p. 350), and then retrieving the records, locking each as you go (assuming FOR UPDATE is allowed on the datasource). After the row(s) have been successfully locked, call this function.

NOTE: Most datasources have a limited "rollback" segment. This means that it is only possible to insert/update/delete a finite number of rows without performing a *wxDb::CommitTrans* (p. 293) or *wxDb::RollbackTrans* (p. 310). Size of the rollback segment varies from database to database, and is user configurable in most databases. Therefore it is usually best to try to perform a commit or rollback at relatively small intervals when processing a larger number of actions that insert/update/delete rows in a table.

Example

```
// Incomplete code sample to delete all users with a first name
// of "JOHN"
users.ClearMemberVars();
wxStrcpy(users.FirstName,"JOHN");
users.DeleteMatching();
```

wxDbTable::DeleteWhere

bool DeleteWhere(**const wxString &***pWhereClause*)

Deletes all rows from the table which match the criteria specified in the WHERE clause that is passed in.

Parameters

pWhereClause

> SQL WHERE clause. This WHERE clause determines which records will be deleted from the table interfaced through the wxDbTable instance. The WHERE clause passed in must be compliant with the SQL 92 grammar. Do not include the keyword 'WHERE'

Remarks

This is the most powerful form of the wxDbTable delete functions. This function gives access to the full power of SQL. This function can be used to delete records by passing a valid SQL WHERE clause. Sophisticated deletions can be performed based on multiple criteria using the full functionality of the SQL language.

A *wxDb::CommitTrans* (p. 293) must be called after use of this function to commit the deletions.

Note: This function is limited to deleting records from the table associated with this wxDbTable object only. Deletions on joined tables is not possible.

NOTE: Most datasources have a limited size "rollback" segment. This means that it is only possible to insert/update/delete a finite number of rows without performing a *wxDb::CommitTrans* (p. 293) or*wxDb::RollbackTrans* (p. 310). Size of the rollback segment varies from database to database, and is user configurable in most databases. Therefore it is usually best to try to perform a commit or rollback at relatively small intervals when processing a larger number of actions that insert/update/delete rows in a table.

WHERE and FROM clauses specified using *wxDbTable::SetWhereClause* (p. 357)and *wxDbTable::SetFromClause* (p. 355) are ignored by this function.

Example

```
// Delete parts 1 thru 10 from containers 'X', 'Y' and 'Z' that
// are magenta in color
parts.DeleteWhere("(PART_NUMBER BETWEEN 1 AND 10) AND \
                   CONTAINER IN ('X', 'Y', 'Z') AND \
                   UPPER(COLOR) = 'MAGENTA'");
```

wxDbTable::DropIndex

bool DropIndex(**const wxString &***IndexName*)

Allows an index on the associated table to be dropped (deleted) if the user login has sufficient privileges to do so.

Parameters

IndexName

>   Name of the index to be dropped.

Remarks

If the index specified in the 'IndexName' parameter does not exist, an error will be logged, and the function will return a result of false.

It is not necessary to call *wxDb::CommitTrans* (p. 293)after executing this function.

wxDbTable::DropTable

bool DropTable()

Deletes the associated table if the user has sufficient privileges to do so.

Remarks

This function returns true if the table does not exist, but only for supported databases (see *wxDb::Dbms* (p. 295)). If a datasource is not specifically supported, and this function is called, the function will return false.

Most datasources/ODBC drivers will delete any indexes associated with the table automatically, and others may not. Check the documentation for your database to determine the behavior.

It is not necessary to call *wxDb::CommitTrans* (p. 293)after executing this function.

wxDbTable::From

const wxString & From()

void From(**const wxString &***From*)

Accessor function for the private class member wxDbTable::from. Can be used as a synonym for *wxDbTable::GetFromClause* (p. 340)(the first form of this function) or*wxDbTable::SetFromClause* (p. 355) (the second form of this function).

Parameters

From

>   A comma separated list of table names that are to be outer joined with the base table's columns so that the joined table's columns may be returned in the result set or used as a portion of a comparison with the base table's columns. NOTE that the base tables name must NOT be included in the FROM clause, as it is automatically included by the wxDbTable class in constructing query statements.

Return value

The first form of this function returns the current value of the wxDbTable member variable ::from.

The second form of the function has no return value, as it will always set the from clause successfully.

See also

wxDbTable::GetFromClause (p. 340),*wxDbTable::SetFromClause* (p. 355)

wxDbTable::GetColDefs

wxDbColDef * GetColDefs()

Accessor function that returns a pointer to the array of column definitions that are bound to the columns that this wxDbTable instance is associated with.

To determine the number of elements pointed to by the returned*wxDbColDef* (p. 315) pointer, use the*wxDbTable::GetNumberOfColumns* (p. 341) function.

Remarks

These column definitions must not be manually redefined after they have been set.

wxDbTable::GetCursor

HSTMT GetCursor()

Returns the HSTMT value of the current cursor for this wxDbTable object.

Remarks

This function is typically used just before changing to use a different cursor so that after the program is finished using the other cursor, the current cursor can be set back to being the cursor in use.

See also

wxDbTable::SetCursor (p. 354), *wxDbTable::GetNewCursor* (p. 340)

wxDbTable::GetDb

wxDb * GetDb()

Accessor function for the private member variable pDb which is a pointer to the datasource connection that this wxDbTable instance uses.

wxDbTable::GetFirst

bool GetFirst()

Retrieves the FIRST row in the record set as defined by the current query. Before retrieving records, a query must be performed using*wxDbTable::Query* (p. 347),*wxDbTable::QueryOnKeyFields* (p. 351),*wxDbTable::QueryMatching* (p. 350) or*wxDbTable::QueryBySqlStmt* (p. 348).

Remarks

This function can only be used if the datasource connection used by the wxDbTable instance was created with FwdOnlyCursors set to false. If the connection does not allow backward scrolling cursors, this function will return false, and the data contained in the bound columns will be undefined.

See also

wxDb::IsFwdOnlyCursors (p. 305)

wxDbTable::GetFromClause

const wxString & GetFromClause()

Accessor function that returns the current FROM setting assigned with the*wxDbTable::SetFromClause* (p. 355).

See also

wxDbTable::From (p. 338)

wxDbTable::GetLast

bool GetLast()

Retrieves the LAST row in the record set as defined by the current query. Before retrieving records, a query must be performed using*wxDbTable::Query* (p. 347),*wxDbTable::QueryOnKeyFields* (p. 351),*wxDbTable::QueryMatching* (p. 350) or*wxDbTable::QueryBySqlStmt* (p. 348).

Remarks

This function can only be used if the datasource connection used by the wxDbTable instance was created with FwdOnlyCursors set to false. If the connection does not allow backward scrolling cursors, this function will return false, and the data contained in the bound columns will be undefined.

See also

wxDb::IsFwdOnlyCursors (p. 305)

wxDbTable::GetNewCursor

HSTMT * GetNewCursor(**bool** *setCursor=false*,**bool** *bindColumns=true*)

This function will create a new cursor that can be used to access the table being referenced by this wxDbTable instance, or to execute direct SQL commands on without affecting the cursors that are already defined and possibly positioned.

Parameters

setCursor

> *OPTIONAL*. Should this new cursor be set to be the current cursor after successfully creating the new cursor. Default is false.

bindColumns

> *OPTIONAL*. Should this new cursor be bound to all the memory variables that the default cursor is bound to. Default is true.

Remarks

This new cursor must be closed using*wxDbTable::DeleteCursor* (p. 335)by the calling program before the wxDbTable instance is deleted, or both memory and resource leaks will occur.

wxDbTable::GetNext

bool GetNext()

Retrieves the NEXT row in the record set after the current cursor position as defined by the current query. Before retrieving records, a query must be performed using *wxDbTable::Query* (p. 347),*wxDbTable::QueryOnKeyFields* (p. 351),*wxDbTable::QueryMatching* (p. 350) or*wxDbTable::QueryBySqlStmt* (p. 348).

Return value

This function returns false when the current cursor has reached the end of the result set. When false is returned, data in the bound columns is undefined.

Remarks

This function works with both forward and backward scrolling cursors.

See also<wxDbTable::++ (p. 360)>

wxDbTable::GetNumberOfColumns

UWORD GetNumberOfColumns()

Accessor function that returns the number of columns that are statically bound for access by the wxDbTable instance.

wxDbTable::GetOrderByClause

const wxString & GetOrderByClause()

Accessor function that returns the current ORDER BY setting assigned with the *wxDbTable::SetOrderByClause* (p. 356).

See also

wxDbTable::OrderBy (p. 346)

wxDbTable::GetPrev

bool GetPrev()

Retrieves the PREVIOUS row in the record set before the current cursor position as defined by the current query. Before retrieving records, a query must be performed using *wxDbTable::Query* (p. 347),*wxDbTable::QueryOnKeyFields* (p. 351),*wxDbTable::QueryMatching* (p. 350) or*wxDbTable::QueryBySqlStmt* (p. 348).

Return value

This function returns false when the current cursor has reached the beginning of the result set and there are now other rows prior to the cursors current position. When false is returned, data in the bound columns is undefined.

Remarks

This function can only be used if the datasource connection used by the wxDbTable instance was created with FwdOnlyCursors set to false. If the connection does not allow backward scrolling cursors, this function will return false, and the data contained in the bound columns will be undefined.

See also

wxDb::IsFwdOnlyCursors (p. 305),*wxDbTable::--* (p. 360)

wxDbTable::GetQueryTableName

const wxString & GetQueryTableName()

Accessor function that returns the name of the table/view that was indicated as being the table/view to query against when this wxDbTable instance was created.

See also

wxDbTable constructor (p. 325)

wxDbTable::GetRowNum

UWORD GetRowNum()

Returns the ODBC row number for performing positioned updates and deletes.

Remarks

This function is not being used within the ODBC class library and may be a candidate for removal if no use is found for it.

Row number with some datasources/ODBC drivers is the position in the result set, while in others it may be a physical position in the database. Check your database documentation to find out which behavior is supported.

wxDbTable::GetTableName

const wxString & GetTableName()

Accessor function that returns the name of the table that was indicated as being the table that this wxDbTable instance was associated with.

wxDbTable::GetTablePath

const wxString & GetTablePath()

Accessor function that returns the path to the data table that was indicated during creation of this wxDbTable instance.

Remarks

Currently only applicable to dBase and MS-Access datasources.

wxDbTable::GetWhereClause

const wxString & GetWhereClause()

Accessor function that returns the current WHERE setting assigned with the*wxDbTable::SetWhereClause* (p. 357)

See also

wxDbTable::Where (p. 359)

wxDbTable::Insert

int Insert()

Inserts a new record into the table being referenced by this wxDbTable instance. The values in the member variables of the wxDbTable instance are inserted into the columns of the new row in the database.

Return value

```
DB_SUCCESS              Record inserted successfully (value = 1)

DB_FAILURE              Insert failed (value = 0)

DB_ERR_INTEGRITY_CONSTRAINT_VIOL
                        The insert failed due to an integrity
                        constraint violation (duplicate non-unique
                        index entry) is attempted.
```

Remarks

A *wxDb::CommitTrans* (p. 293) or*wxDb::RollbackTrans* (p. 310) must be called after use of this function to commit or rollback the insertion.

Example

```cpp
// Incomplete code snippet
wxStrcpy(parts->PartName, "10");
wxStrcpy(parts->PartDesc, "Part #10");
parts->Qty = 1000;
RETCODE retcode = parts->Insert();
switch(retcode)
{
    case DB_SUCCESS:
        parts->GetDb()->CommitTrans();
        return(true);
    case DB_ERR_INTEGRITY_CONSTRAINT_VIOL:
        // Current data would result in a duplicate key
        // on one or more indexes that do not allow duplicates
        parts->GetDb()->RollbackTrans();
        return(false);
    default:
        // Insert failed for some unexpected reason
        parts->GetDb()->RollbackTrans();
        return(false);
}
```

wxDbTable::IsColNull

bool IsColNull(**UWORD** *colNumber*) const

Used primarily in the ODBC class library to determine if a column value is set to "NULL". Works for all data types supported by the ODBC class library.

Parameters

colNumber

The column number of the bound column as defined by the*wxDbTable::SetColDefs* (p. 352)calls which defined the columns accessible to this wxDbTable instance.

Remarks

NULL column support is currently not fully implemented as of wxWidgets 2.4.

wxDbTable::IsCursorClosedOnCommit

bool IsCursorClosedOnCommit()

Accessor function to return information collected during the opening of the datasource connection that is used by this wxDbTable instance. The result returned by this function indicates whether an implicit closing of the cursor is done after a commit on the database connection.

Return value

Returns true if the cursor associated with this wxDbTable object is closed after a commit or rollback operation. Returns false otherwise.

Remarks

If more than one wxDbTable instance used the same database connection, all cursors which use the database connection are closed on the commit if this function indicates true.

wxDbTable::IsQueryOnly

bool IsQueryOnly()

Accessor function that returns a value indicating if this wxDbTable instance was created to allow only queries to be performed on the bound columns. If this function returns true, then no actions may be performed using this wxDbTable instance that would modify (insert/delete/update) the table's data.

wxDbTable::Open

bool Open(**bool** *checkPrivileges=false*, **bool** *checkTableExists=true*)

Every wxDbTable instance must be opened before it can be used. This function checks for the existence of the requested table, binds columns, creates required cursors, (insert/select and update if connection is not wxDB_QUERY_ONLY) and constructs the insert statement that is to be used for inserting data as a new row in the datasource.

NOTE: To retrieve data into an opened table, the of the table must be bound to the variables in the program via call(s) to *wxDbTable::SetColDefs* (p. 352) before calling Open().

See the *database classes overview* (p. 1820) for an introduction to using the ODBC classes.

Parameters

checkPrivileges

> Indicates whether the Open() function should check whether the current connected user has at least SELECT privileges to access the table to which they are trying to

open. Default is false.

checkTableExists

Indicates whether the Open() function should check whether the table exists in the database or not before opening it. Default is true.

Remarks

If the function returns a false value due to the table not existing, a log entry is recorded for the datasource connection indicating the problem that was detected when checking for table existence. Note that it is usually best for the calling routine to check for the existence of the table and for sufficient user privileges to access the table in the mode (wxDB_QUERY_ONLY or !wxDB_QUERY_ONLY) before trying to open the table for the best possible explanation as to why a table cannot be opened.

Checking the user's privileges on a table can be quite time consuming during the open phase. With most applications, the programmer already knows that the user has sufficient privileges to access the table, so this check is normally not required.

For best performance, open the table, and then use the*wxDb::TablePrivileges* (p. 313) function to check the users privileges. Passing a schema to the TablePrivileges() function can significantly speed up the privileges checks.

See also

wxDb::TableExists (p. 312),*wxDb::TablePrivileges* (p. 313)*wxDbTable::SetColDefs* (p. 352)

wxDbTable::OrderBy

const wxString & OrderBy()

void OrderBy(**const wxString &***OrderBy*)

Accessor function for the private class member wxDbTable::orderBy. Can be used as a synonym for*wxDbTable::GetOrderByClause* (p. 342)(the first form of this function) or*wxDbTable::SetOrderByClause* (p. 356)(the second form of this function).

Parameters

OrderBy

A comma separated list of column names that indicate the alphabetized/numeric sorting sequence that the result set is to be returned in. If a FROM clause has also been specified, each column name specified in the ORDER BY clause should be prefaced with the table name to which the column belongs using DOT notation (TABLE_NAME.COLUMN_NAME).

Return value

The first form of this function returns the current value of the wxDbTable member variable ::orderBy.

The second form of the function has no return value.

See also

wxDbTable::GetOrderByClause (p. 342),*wxDbTable::SetFromClause* (p. 355)

wxDbTable::Query

virtual bool Query(**bool** *forUpdate=false*, **bool** *distinct=false*)

Parameters

forUpdate

> *OPTIONAL*. Gives you the option of locking records as they are retrieved. If the RDBMS is not capable of the FOR UPDATE clause, this argument is ignored. See*wxDbTable::CanSelectForUpdate* (p. 329) for additional information regarding this argument. Default is false.

distinct

> *OPTIONAL*. Allows selection of only distinct values from the query (SELECT DISTINCT ... FROM ...). The notion of DISTINCT applies to all columns returned in the result set, not individual columns. Default is false.

Remarks

This function queries records from the datasource based on the three wxDbTable members: "where", "orderBy", and "from". Use*wxDbTable::SetWhereClause* (p. 357) to filter on records to be retrieved (e.g. All users with a first name of "JOHN"). Use *wxDbTable::SetOrderByClause* (p. 356) to change the sequence in which records are returned in the result set from the datasource (e.g. Ordered by LAST_NAME). Use*wxDbTable::SetFromClause* (p. 355) to allow outer joining of the base table (the one being associated with this instance of wxDbTable) with other tables which share a related field.

After each of these clauses are set/cleared, call wxDbTable::Query() to fetch the result set from the datasource.

This scheme has an advantage if you have to requery your record set frequently in that you only have to set your WHERE, ORDER BY, and FROM clauses once. Then to refresh the record set, simply call wxDbTable::Query() as frequently as needed.

Note that repeated calls to wxDbTable::Query() may tax the database server and make your application sluggish if done too frequently or unnecessarily.

The base table name is automatically prepended to the base column names in the event that the FROM clause has been set (is non-null) using*wxDbTable::SetFromClause* (p. 355).

The cursor for the result set is positioned *before* the first record in the result set after the query. To retrieve the first record, call either*wxDbTable::GetFirst* (p. 339) (only if backward scrolling cursors are available) or*wxDbTable::GetNext* (p. 341). Typically, no

data from the result set is returned to the client driver until a request such as*wxDbTable::GetNext* (p. 341) is performed, so network traffic and database load are not overwhelmed transmitting data until the data is actually requested by the client. This behavior is solely dependent on the ODBC driver though, so refer to the ODBC driver's reference material for information on its behaviors.

Values in the bound columns' memory variables are undefined after executing a call to this function and remain that way until a row in the result set is requested to be returned.

The wxDbTable::Query() function is defined as "virtual" so that it may be overridden for application specific purposes.

Be sure to set the wxDbTable's "where", "orderBy", and "from" member variables to "" if they are not to be used in the query. Otherwise, the results returned may have unexpected results (or no results) due to improper or incorrect query parameters constructed from the uninitialized clauses.

Example

```
// Incomplete code sample
parts->SetWhereClause("DESCRIPTION = 'FOOD'");
parts->SetOrderByClause("EXPIRATION_DATE");
parts->SetFromClause("");
// Query the records based on the where, orderBy and from clauses
// specified above
parts->Query();
// Display all records queried
while(parts->GetNext())
    dispPart(parts);  // user defined function
```

wxDbTable::QueryBySqlStmt

bool QueryBySqlStmt(const wxString &*pSqlStmt*)

Performs a query against the datasource by accepting and passing verbatim the SQL SELECT statement passed to the function.

Parameters

pSqlStmt

Pointer to the SQL SELECT statement to be executed.

Remarks

This is the most powerful form of the query functions available. This member function allows a programmer to write their own custom SQL SELECT statement for requesting data from the datasource. This gives the programmer access to the full power of SQL for performing operations such as scalar functions, aggregate functions, table joins, and sub-queries, as well as datasource specific function calls.

The requirements of the SELECT statement are the following:

  1. Must return the correct number of columns. In the derived wxDbTable

constructor, it is specified how many columns are in the wxDbTable object. The SELECT statement must return exactly that many columns.

2. The columns must be returned in the same sequence as specified when defining the bounds columns *wxDbTable::SetColDefs* (p. 352), and the columns returned must be of the proper data type. For example, if column 3 is defined in the wxDbTable bound column definitions to be a float, the SELECT statement must return a float for column 3 (e.g. PRICE * 1.10 to increase the price by 10

3. The ROWID can be included in your SELECT statement as the **last**column selected, if the datasource supports it. Use wxDbTable::CanUpdateByROWID() to determine if the ROWID can be selected from the datasource. If it can, much better performance can be achieved on updates and deletes by including the ROWID in the SELECT statement.

Even though data can be selected from multiple tables (joins) in your select statement, only the base table associated with this wxDbTable object is automatically updated through the ODBC class library. Data from multiple tables can be selected for display purposes however. Include columns in the wxDbTable object and mark them as non-updateable (See*wxDbColDef* (p. 315) for details). This way columns can be selected and displayed from other tables, but only the base table will be updated automatically when performed through the*wxDbTable::Update* (p. 358) function after using this type of query. To update tables other than the base table, use the*wxDbTable::Update* (p. 358) function passing a SQL statement.

After this function has been called, the cursor is positioned before the first record in the record set. To retrieve the first record, call either *wxDbTable::GetFirst* (p. 339) or*wxDbTable::GetNext* (p. 341).

Example

```
  // Incomplete code samples
wxString sqlStmt;
 sqlStmt = "SELECT * FROM PARTS WHERE STORAGE_DEVICE = 'SD98' \
           AND CONTAINER = 12";
 // Query the records using the SQL SELECT statement above
 parts->QueryBySqlStmt(sqlStmt);
 // Display all records queried
 while(parts->GetNext())
     dispPart(&parts);

 Example SQL statements
 ----------------------

 // Table Join returning 3 columns
 SELECT PART_NUM, part_desc, sd_name
     from parts, storage_devices
     where parts.storage_device_id =
           storage_devices.storage_device_id

 // Aggregate function returning total number of
 // parts in container 99
 SELECT count(*) from PARTS where container = 99

 // Order by clause; ROWID, scalar function
 SELECT PART_NUM, substring(part_desc, 1, 10), qty_on_hand + 1, ROWID
     from parts
```

```
        where warehouse = 10
        order by PART_NUM desc          // descending order

    // Subquery
    SELECT * from parts
        where container in (select container
        from storage_devices
            where device_id = 12)
```

wxDbTable::QueryMatching

virtual bool QueryMatching(**bool** *forUpdate=false*,**bool** *distinct=false*)

QueryMatching allows querying of records from the table associated with the wxDbTable object by matching "columns" to values.

For example: To query the datasource for the row with a PART_NUMBER column value of "32", clear all column variables of the wxDbTable object, set the PartNumber variable that is bound to the PART_NUMBER column in the wxDbTable object to "32", and then call wxDbTable::QueryMatching().

Parameters

forUpdate

> *OPTIONAL*. Gives you the option of locking records as they are queried (SELECT ... FOR UPDATE). If the RDBMS is not capable of the FOR UPDATE clause, this argument is ignored. See*wxDbTable::CanSelectForUpdate* (p. 329) for additional information regarding this argument. Default is false.

distinct

> *OPTIONAL*. Allows selection of only distinct values from the query (SELECT DISTINCT ... FROM ...). The notion of DISTINCT applies to all columns returned in the result set, not individual columns. Default is false.

Remarks

The SQL WHERE clause is built by the ODBC class library based on all non-zero/non-NULL columns in your wxDbTable object. Matches can be on one, many or all of the wxDbTable's columns. The base table name is prepended to the column names in the event that the wxDbTable's FROM clause is non-null.

This function cannot be used to perform queries which will check for columns that are 0 or NULL, as the automatically constructed WHERE clause only will contain comparisons on column member variables that are non-zero/non-NULL.

The primary difference between this function and *wxDbTable::QueryOnKeyFields* (p. 351)is that this function can query on any column(s) in the wxDbTable object. Note however that this may not always be very efficient. Searching on non-indexed columns will always require a full table scan.

The cursor is positioned before the first record in the record set after the query is

performed. To retrieve the first record, the program must call either *wxDbTable::GetFirst* (p. 339) or*wxDbTable::GetNext* (p. 341).

WHERE and FROM clauses specified using *wxDbTable::SetWhereClause* (p. 357)and *wxDbTable::SetFromClause* (p. 355) are ignored by this function.

Example

```
// Incomplete code sample
parts->ClearMemberVars();        // Set all columns to zero
wxStrcpy(parts->PartNumber,"32");   // Set columns to query on
parts->OnHold = true;
parts->QueryMatching();          // Query
// Display all records queried
while(parts->GetNext())
    dispPart(parts);  // Some application defined function
```

wxDbTable::QueryOnKeyFields

bool QueryOnKeyFields(**bool** *forUpdate=false*,**bool** *distinct=false*)

QueryOnKeyFields provides an easy mechanism to query records in the table associated with the wxDbTable object by the primary index column(s). Simply assign the primary index column(s) values and then call this member function to retrieve the record.

Note that since primary indexes are always unique, this function implicitly always returns a single record from the database. The base table name is prepended to the column names in the event that the wxDbTable's FROM clause is non-null.

Parameters

forUpdate

> *OPTIONAL*. Gives you the option of locking records as they are queried (SELECT ... FOR UPDATE). If the RDBMS is not capable of the FOR UPDATE clause, this argument is ignored. See*wxDbTable::CanSelectForUpdate* (p. 329) for additional information regarding this argument. Default is false.

distinct

> *OPTIONAL*. Allows selection of only distinct values from the query (SELECT DISTINCT ... FROM ...). The notion of DISTINCT applies to all columns returned in the result set, not individual columns. Default is false.

Remarks

The cursor is positioned before the first record in the record set after the query is performed. To retrieve the first record, the program must call either *wxDbTable::GetFirst* (p. 339) or*wxDbTable::GetNext* (p. 341).

WHERE and FROM clauses specified using *wxDbTable::SetWhereClause* (p. 357)and *wxDbTable::SetFromClause* (p. 355) are ignored by this function.

Example

```
// Incomplete code sample
wxStrcpy(parts->PartNumber, "32");
parts->QueryOnKeyFields();
// Display all records queried
while(parts->GetNext())
    dispPart(parts);  // Some application defined function
```

wxDbTable::Refresh

bool Refresh()

This function re-reads the bound columns into the memory variables, setting them to the current values stored on the disk.

The cursor position and result set are unaffected by calls to this function. (The one exception is in the case where the record to be refreshed has been deleted by some other user or transaction since it was originally retrieved as part of the result set. For most datasources, the default behavior in this situation is to return the value that was originally queried for the result set, even though it has been deleted from the database. But this is datasource dependent, and should be tested before relying on this behavior.)

Remarks

This routine is only guaranteed to work if the table has a unique primary index defined for it. Otherwise, more than one record may be fetched and there is no guarantee that the correct record will be refreshed. The table's columns are refreshed to reflect the current data in the database.

wxDbTable::SetColDefs

bool SetColDefs(**UWORD** *index*, **const wxString &***fieldName*,**int** *dataType*, **void *pData**, **SWORD** *cType*,**int** *size*, **bool** *keyField = false*, **bool** *updateable = true*,**bool** *insertAllowed = true*, **bool** *derivedColumn = false*)

wxDbColDataPtr * SetColDefs(**wxDbColInf **colInfs*, **UWORD** *numCols*)

Parameters

index

> Column number (0 to n-1, where n is the number of columns specified as being defined for this wxDbTable instance when the wxDbTable constructor was called.

fieldName

> Column name from the associated data table.

dataType

> Logical data type. Valid logical types include:

```
DB_DATA_TYPE_VARCHAR        : strings
DB_DATA_TYPE_INTEGER        : non-floating point numbers
DB_DATA_TYPE_FLOAT          : floating point numbers
DB_DATA_TYPE_DATE           : dates
```

pData

> Pointer to the data object that will hold the column's value when a row of data is returned from the datasource.

cType

> SQL C Type. This defines the data type that the SQL representation of the data is converted to to be stored in *pData*. Other valid types are available also, but these are the most common ones:

```
SQL_C_CHAR      // string - deprecated: use SQL_C_WXCHAR
SQL_C_WXCHAR    // string - Used transparently in unicode or non-
unicode builds
SQL_C_LONG
SQL_C_ULONG
SQL_C_SHORT
SQL_C_USHORT
SQL_C_FLOAT
SQL_C_DOUBLE
SQL_C_NUMERIC
SQL_C_TIMESTAMP

SQL_C_BOOLEAN   // defined in db.h
SQL_C_ENUM      // defined in db.h
```

size

> Maximum size in bytes of the *pData* object.

keyField

> *OPTIONAL*. Indicates if this column is part of the primary index. Default is false.

updateable

> *OPTIONAL*. Are updates allowed on this column? Default is true.

insertAllowed

> *OPTIONAL*. Inserts allowed on this column? Default is true.

derivedColumn

> *OPTIONAL*. Is this a derived column (non-base table column for query only)? Default is false.

colInfs

> Pointer to an array of wxDbColInf instances which contains all the information necessary to create *numCols* column definitions.

numCols

Number of elements of wxDbColInf type that are pointed to by *colInfs*, which are to have column definitions created from them.

Remarks

If *pData* is to hold a string of characters, be sure to include enough space for the NULL terminator in pData and in the byte count of *size*.

Using the first form of this function, if the column definition is not able to be created, a value of false is returned. If the specified index of the column exceeds the number of columns defined in the wxDbTable instance, an assert is thrown and logged (in debug builds) and a false is returned.

A failure to create the column definition in the second form results in a value of NULL being returned.

Both forms of this function provide a shortcut for defining the columns in your wxDbTable object. Use this function in any derived wxDbTable constructor when describing the column/columns in the wxDbTable object.

The second form of this function is primarily used when the*wxDb::GetColumns* (p. 299) function was used to query the datasource for the column definitions, so that the column definitions are already stored in wxDbColInf form. One example use of using*wxDb::GetColumns* (p. 299) then using this function is if a data table existed in one datasource, and the table's column definitions were to be copied over to another datasource or table.

Example

```
// Long way not using this function
wxStrcpy(colDefs[0].ColName, "PART_NUM");
colDefs[0].DbDataType   = DB_DATA_TYPE_VARCHAR;
colDefs[0].PtrDataObj   = PartNumber;
colDefs[0].SqlCtype     = SQL_C_WXCHAR;
colDefs[0].SzDataObj    = PART_NUMBER_LEN;
colDefs[0].KeyField     = true;
colDefs[0].Updateable   = false;
colDefs[0].InsertAllowed= true;
colDefs[0].DerivedCol   = false;

// Shortcut using this function
SetColDefs(0, "PART_NUM", DB_DATA_TYPE_VARCHAR, PartNumber,
           SQL_C_WXCHAR, PART_NUMBER_LEN, true, false, true, false);
```

wxDbTable::SetCursor

void SetCursor(**HSTMT** **hstmtActivate = (void **) wxDB_DEFAULT_CURSOR*)

Parameters

hstmtActivate

OPTIONAL. Pointer to the cursor that is to become the current cursor. Passing no cursor handle will reset the cursor back to the wxDbTable's default (original) cursor

that was created when the wxDbTable instance was first created. Default is wxDB_DEFAULT_CURSOR.

Remarks

When swapping between cursors, the member variables of the wxDbTable object are automatically refreshed with the column values of the row that the current cursor is positioned at (if any). If the cursor is not positioned, then the data in member variables is undefined.

The only way to return back to the cursor that was in use before this function was called is to programmatically determine the current cursor's HSTMT**BEFORE** calling this function using *wxDbTable::GetCursor* (p. 339)and saving a pointer to that cursor.

See also

wxDbTable::GetNewCursor (p. 340),*wxDbTable::GetCursor* (p. 339),*wxDbTable::SetCursor* (p. 354)

wxDbTable::SetFromClause

void SetFromClause(**const wxString &***From*)

Accessor function for setting the private class member wxDbTable::from that indicates what other tables should be outer joined with the wxDbTable's base table for access to the columns in those other tables.

Synonym to this function is one form of *wxDbTable::From* (p. 338)

Parameters

From

> A comma separated list of table names that are to be outer joined with the base table's columns so that the joined table's columns may be returned in the result set or used as a portion of a comparison with the base table's columns. NOTE that the base tables name must NOT be included in the FROM clause, as it is automatically included by the wxDbTable class in constructing query statements.

Remarks

Used by the *wxDbTable::Query* (p. 347) and*wxDbTable::Count* (p. 331) member functions to allow outer joining of records from multiple tables.

Do **not** include the keyword "FROM" when setting the FROM clause.

If using the FROM clause when performing a query, be certain to include in the corresponding WHERE clause a comparison of a column from either the base table or one of the other joined tables to each other joined table to ensure the datasource knows on which column values the tables should be joined on.

Example

```
...
// Base table is the "LOCATION" table, and it is being
// outer joined to the "PART" table via the field "PART_NUMBER"
// that can be related between the two tables.
location->SetWhereClause("LOCATION.PART_NUMBER = PART.PART_NUMBER")
location->SetFromClause("PART");
...
```

See also

wxDbTable::From (p. 338),*wxDbTable::GetFromClause* (p. 340)

wxDbTable::SetColNull

bool SetColNull(**UWORD** *colNumber*, **bool** *set=true*)

bool SetColNull(**const wxString &***colName*,**bool** *set=true*)

Both forms of this function allow a member variable representing a column in the table associated with this wxDbTable object to be set to NULL.

The first form allows the column to be set by the index into the column definitions used to create the wxDbTable instance, while the second allows the actual column name to be specified.

Parameters

colNumber

   Index into the column definitions used when first defining this wxDbTable object.

colName

   Actual data table column name that is to be set to NULL.

set

   Whether the column is set to NULL or not. Passing true sets the column to NULL, passing false sets the column to be non-NULL. Default is true.

Remarks

No database updates are done by this function. It only operates on the member variables in memory. Use and insert or update function to store this value to disk.

wxDbTable::SetOrderByClause

void SetOrderByClause(**const wxString &***OrderBy*)

Accessor function for setting the private class member wxDbTable::orderBy which determines sequence/ordering of the rows returned in the result set of a query.

A synonym to this function is one form of the function *wxDbTable::OrderBy* (p. 346)

Parameters

OrderBy

> A comma separated list of column names that indicate the alphabetized sorting sequence that the result set is to be returned in. If a FROM clause has also been specified, each column name specified in the ORDER BY clause should be prefaced with the table name to which the column belongs using DOT notation (TABLE_NAME.COLUMN_NAME).

Remarks

Do **not** include the keywords "ORDER BY" when setting the ORDER BY clause.

Example

```
...
parts->SetOrderByClause("PART_DESCRIP, QUANTITY");
...

...
location->SetOrderByClause("LOCATION.POSITION, PART.PART_NUMBER);
...
```

See also

wxDbTable::OrderBy (p. 346),*wxDbTable::GetOrderByClause* (p. 342)

wxDbTable::SetQueryTimeout

bool SetQueryTimeout(**UDWORD** *nSeconds*)

Allows a time period to be set as the timeout period for queries.

Parameters

nSeconds

> The number of seconds to wait for the query to complete before timing out.

Remarks

Neither Oracle or Access support this function as of yet. Other databases should be evaluated for support before depending on this function working correctly.

wxDbTable::SetWhereClause

void SetWhereClause(**const wxString &***Where*)

Accessor function for setting the private class member wxDbTable::where that determines which rows are returned in the result set by the datasource.

A synonym to this function is one form of the function *wxDbTable::Where* (p. 359)

Parameters

Where

> SQL "where" clause. This clause can contain any SQL language that is legal in standard where clauses. If a FROM clause has also been specified, each column name specified in the ORDER BY clause should be prefaced with the table name to which the column belongs using DOT notation (TABLE_NAME.COLUMN_NAME).

Remarks

Do **not** include the keywords "WHERE" when setting the WHERE clause.

Example

```
...
// Simple where clause
parts->SetWhereClause("PART_NUMBER = '32'");
...
// Any comparison operators
parts->SetWhereClause("PART_DESCRIP LIKE 'HAMMER%'");
...
// Multiple comparisons, including a function call
parts->Where("QTY > 0 AND {fn UCASE(PART_DESCRIP)} LIKE '%DRILL%'");
...
// Using parameters and multiple logical combinations
parts->Where("((QTY > 10) OR (ON_ORDER > 0)) AND ON_HOLD = 0");
...
// This query uses an outer join (requiring a FROM clause also)
// that joins the PART and LOCATION table on he common field
// PART_NUMBER.
parts->Where("PART.ON_HOLD = 0 AND \
              PART.PART_NUMBER = LOCATION.PART_NUMBER AND \
              LOCATION.PART_NUMBER > 0");
```

See also

wxDbTable::Where (p. 359),*wxDbTable::GetWhereClause* (p. 343)

wxDbTable::Update

bool Update()

bool Update(**const wxString &***pSqlStmt*)

The first form of this function will update the row that the current cursor is currently positioned at with the values in the memory variables that are bound to the columns. The actual SQL statement to perform the update is automatically created by the ODBC class, and then executed.

The second form of the function allows full access through SQL statements for updating records in the database. Write any valid SQL UPDATE statement and submit it to this function for execution. Sophisticated updates can be performed using the full power of the SQL dialect. The full SQL statement must have the exact syntax required by the

driver/datasource for performing the update. This usually is in the form of:

```
UPDATE tablename SET col1=X, col2=Y, ... where ...
```

Parameters

pSqlStmt

>   Pointer to SQL UPDATE statement to be executed.

Remarks

A *wxDb::CommitTrans* (p. 293) or*wxDb::RollbackTrans* (p. 310) must be called after use
of this function to commit or rollback the update.

Example

```
wxString sqlStmt;
  sqlStmt = "update PART set QTY = 0 where PART_NUMBER = '32'";
```

wxDbTable::UpdateWhere

bool UpdateWhere(**const wxString &***pWhereClause*)

Performs updates to the base table of the wxDbTable object, updating only the rows
which match the criteria specified in the *pWhereClause*.

All columns that are bound to member variables for this wxDbTable instance that were
defined with the "updateable" parameter set to true will be updated with the information
currently held in the memory variable.

Parameters

pWhereClause

>   Pointer to a valid SQL WHERE clause. Do not include the keyword 'WHERE'.

Remarks

Care should be used when updating columns that are part of indexes with this function
so as not to violate an unique key constraints.

A *wxDb::CommitTrans* (p. 293) or*wxDb::RollbackTrans* (p. 310) must be called after use
of this function to commit or rollback the update(s).

wxDbTable::Where

const wxString & Where()

void Where(**const wxString&** *Where*)

Accessor function for the private class member wxDbTable::where. Can be used as a

synonym for *wxDbTable::GetWhereClause* (p. 343)(the first form of this function) to return the current where clause or*wxDbTable::SetWhereClause* (p. 357) (the second form of this function) to set the where clause for this table instance.

Parameters

Where

> A valid SQL WHERE clause. Do not include the keyword 'WHERE'.

Return value

The first form of this function returns the current value of the wxDbTable member variable ::where.

The second form of the function has no return value, as it will always set the where clause successfully.

See also

wxDbTable::GetWhereClause (p. 343),*wxDbTable::SetWhereClause* (p. 357)

wxDbTable::operator ++

bool operator ++()

Synonym for *wxDbTable::GetNext* (p. 341)

See also

wxDbTable::GetNext (p. 341)

wxDbTable::operator --

bool operator --()

Synonym for *wxDbTable::GetPrev* (p. 342)

See also

wxDbTable::GetPrev (p. 342)

wxDbTableInf

```
tableName[0]    = 0;
tableType[0]    = 0;
tableRemarks[0] = 0;
numCols         = 0;
pColInf         = NULL;
```

Currently only used by *wxDb::GetCatalog* (p. 298) internally and *wxDbInf* (p. 324) class,

but may be used in future releases for user functions. Contains information describing the table (Name, type, etc). A pointer to a wxDbColInf array instance is included so a program can create a *wxDbColInf* (p. 317) array instance (using *wxDb::GetColumns* (p. 299)) to maintain all information about the columns of a table in one memory structure.

Eventually, accessor functions will be added for this class

See the *database classes overview* (p. 1820) for an introduction to using the ODBC classes.

Include files

<wx/db.h>

wxDbTableInf::Initialize

Simply initializes all member variables to a cleared state. Called by the constructor automatically.

wxDbGridColInfo

This class is used to define columns to be shown, names of the columns, order and type of data, when using *wxdbGridTableBase* (p. 363) to display a Table or query in a *wxGrid* (p. 619)

See the database grid example in *wxDbGridTableBase* (p. 363) for an introduction to using the wxDbGrid classes.

Include files

<wx/dbgrid.h>

wxDbGridColInfo::wxDbGridColInfo

 wxDbGridColInfo(**int** *colNumber*, **wxString** *type*, **wxString** *title*, **wxDbGridColInfo** *next)

Default constructor. See the database grid example in *wxDbGridTableBase* (p. 363) to see two different ways for adding columns.

Parameters

colNumber

> Column number in the *wxDbTable* (p. 324) instance to be used (first column is 0).

type

> Column type ,wxString specifying the grid name for the datatype in this column, or use wxGRID_VALUE_DBAUTO to determine the type automatically from the *wxDbColDef* (p. 315) definition

title

> The column label to be used in the grid display

next

> A pointer to the next wxDbGridColInfo structure if using one-step construction, NULL terminates the list. Use Null also if using two step construction.

See the database grid example in *wxDbGridTableBase* (p. 363) to see two different ways for adding columns.

wxDbGridColInfo::~wxDbGridColInfo

 ~wxDbGridColInfo()

Destructor.

wxDbGridColInfo::AddColInfo

void AddColInfo(**int** *colNumber*,**wxString** *type*, **wxString** *title*)

Use this member function for adding columns. See the database grid example in *wxDbGridTableBase* (p. 363) to see two different ways for adding columns.

It is important to note that this class is merely a specifier to the *wxDbGridTableBase* (p. 363) constructor. Changes made to this datatype after the *wxDbGridTableBase* (p. 363) is called will not have any effect.

Parameters*colNumber*

> Column number in the *wxDbTable* (p. 324) instance to be used (first column is 0).

type

> Column type ,wxString specifying the grid name for the datatype in this column, or use wxGRID_VALUE_DBAUTO to determine the type automatically from the *wxDbColDef* (p. 315) definition

title

> The column label to be used in the grid display

Remarks

As wxDbTable must be defined with to have columns which match those to by a

wxDbGridColInfo info structure as this is the structure which informs the grid of how you want to display your *wxDbTable* (p. 324). If no datatype conversion or the referenced column number does not exist the the behavior is undefined.

See the example at *wxDbGridColInfo::wxDbGridColInfo* (p. 361).

wxDbGridTableBase

You can view a database table in a grid using this class.

If you are deriving your own wxDbTable subclass for your table , then you may consider overriding GetCol() and SetCol() to provide calculated fields. This does work but care should be taken when using wxDbGridTableBase in this way.

The constructor and AssignDbTable() call allows you to specify the ownership if the wxDbTable object pointer. If you tell wxGridTableBase to take ownership , it will delete the passed wxDbTable when an new on is assigned or wxGridTableBase's destructor is called. However no checks for aliasing are done so Assign(table,..,true); Assign(table,..,true); is an error. If you need to requery an table object the preferred way is that the client keeps ownership.

Derived From

wxGridTableBase (p. 678)

Include files

<wx/dbgrid.h>

Example

```
// First step, let's define wxDbTable
int numColumns = 2;
wxDbTable *table = new wxDbTable (db, tblName, numColumns);
int int_var;
wxChar string_name[255];
table->SetColDef (0, "column 0", DB_DATA_TYPE_INTEGER, &int_var,
            SQL_C_LONG, sizeof(int_var), true);
table->SetColDef (1, "column 1", DB_DATA_TYPE_VARCHAR, &string_name,
            SQL_C_LONG, sizeof(string_name), false);

    // now let's define columns in the grid

    // first way to do it
    wxDbGridColInfo *columns;
    columns = new wxDbGridColInfo(0, wxGRID_VALUE_LONG, "first column",
            new wxDbGridColInfo(1, wxGRID_VALUE_STRING, "second column",
            NULL);

    // second way to do it
    wxDbGridColInfo *columns;
    // first column is special
    columns = new wxDbGridColInfo(0, wxGRID_VALUE_LONG, "first column",
NULL);
    // all the rest
    columns->AddColInfo (1, wxGRID_VALUE_STRING, "second column");
```

```
        // second way may be better when columns are not known at compile time

        // now, let's open the table and make a Query()
        table->Open();
        // this step is very important
        table->SetRowMode (wxDbTable::WX_ROW_MODE_QUERY);
        // in the grid we will see only the rows of the result query
        m_dbTable->Query();

        wxDbGridTableBase *dbgrid = new wxDbGridTableBase(table, columns,
    wxUSE_QUERY, true);
        delete columns;  // not needed anymore
        wxGrid *grid = new wxGrid ( ... );
        grid->SetTable(dbgrid, true);
        grid->Fit();
```

Include files

<wx/dbgrid.h>

Helper classes and data structures

wxDbGridTableBase::wxDbGridTableBase

 wxDbGridTableBase(**wxDbTable** **tab*, **wxDbGridColInfo** **ColInfo*, **int** *count = wxUSE_QUERY*, **bool** *takeOwnership = true*)

Constructor.

Parameters

tab

> The database table you want to display. Must be opened and queried before display the grid. See the example *above* (p. 363).

ColInfo

> Columns titles, and other values. See *wxDbGridColInfo* (p. 361).

count

> You can use a query result set (wxUSE_QUERY, to use wxDbTable::Count(wxDbTable::Count() or you can fix the total number of rows (count >= 0) to display, or specify it if you already know the size in avoid calling

takeOwnership

> If true, this class deletes wxDbTable when it stops referring to it, if false application must take care of deleting it.

wxDbGridTableBase::ValidateRow

void ValidateRow(**int** *row*)

It ensures that the row data is fetched from the database, and it the wxDbTable local buffer, the row number passed should be the grid row.

Parameters

row

> Row where validation must be done.

wxDbGridTableBase::UpdateRow

bool UpdateRow(**int** *row*)

If row has changed it forces that row to be written back to the database, however support for detecting whether insert/update is required is currently not in wxDbTable, so this function is currently unsupported.

Parameters

row

> Row you want to update.

wxDbGridTableBase::AssignDbTable

bool AssignDbTable(**wxDbTable** **tab*,**int** *count = wxUSE_QUERY*,**bool** *takeOwnership = true*)

Resets the grid for using with a new database table, but using the same columns definition. This can be useful when re-querying the database and want to see the changes.

Parameters

tab

> Database table you want to assign to the grid.

count

> Number of rows you want to show or wxUSE_QUERY for using a query.

takeOwnership

> If false, user must take care of deleting tab after deleting the wxDbGridTableBase. If true, deletion is made by destructor class.

wxDC

A wxDC is a *device context* onto which graphics and text can be drawn. It is intended to represent a number of output devices in a generic way, so a window can have a device context associated with it, and a printer also has a device context. In this way, the same piece of code may write to a number of different devices, if the device context is used as a parameter.

Derived types of wxDC have documentation for specific features only, so refer to this section for most device context information.

Please note that in addition to the versions of the methods documented here, there are also versions which accept single `wxPoint` parameter instead of two `wxCoord` ones or `wxPoint` and `wxSize` instead of four of them.

Derived from

wxObject (p. 1012)

Include files

<wx/dc.h>

See also

Overview (p. 1791)

wxDC::wxDC

wxDC()

Constructor.

wxDC::~wxDC

~wxDC()

Destructor.

wxDC::BeginDrawing

void BeginDrawing()

Allows optimization of drawing code under MS Windows. Enclose drawing primitives between **BeginDrawing** and **EndDrawing** calls.

Drawing to a wxDialog panel device context outside of a system-generated OnPaint event *requires* this pair of calls to enclose drawing code. This is because a Windows dialog box does not have a retained device context associated with it, and selections such as pen and brush settings would be lost if the device context were obtained and

released for each drawing operation.

wxDC::Blit

bool Blit(**wxCoord** *xdest*, **wxCoord** *ydest*, **wxCoord** *width*, **wxCoord** *height*, **wxDC*** *source*, **wxCoord** *xsrc*, **wxCoord** *ysrc*, **int** *logicalFunc = wxCOPY*, **bool** *useMask = false*, **wxCoord** *xsrcMask = -1*, **wxCoord** *ysrcMask = -1*)

Copy from a source DC to this DC, specifying the destination coordinates, size of area to copy, source DC, source coordinates, logical function, whether to use a bitmap mask, and mask source position.

Parameters

xdest

> Destination device context x position.

ydest

> Destination device context y position.

width

> Width of source area to be copied.

height

> Height of source area to be copied.

source

> Source device context.

xsrc

> Source device context x position.

ysrc

> Source device context y position.

logicalFunc

> Logical function to use: see *wxDC::SetLogicalFunction* (p. 383).

useMask

> If true, Blit does a transparent blit using the mask that is associated with the bitmap selected into the source device context. The Windows implementation does the following if MaskBlt cannot be used:

> > **1.** Creates a temporary bitmap and copies the destination area into it.

2. Copies the source area into the temporary bitmap using the specified logical function.

3. Sets the masked area in the temporary bitmap to BLACK by ANDing the mask bitmap with the temp bitmap with the foreground colour set to WHITE and the bg colour set to BLACK.

4. Sets the unmasked area in the destination area to BLACK by ANDing the mask bitmap with the destination area with the foreground colour set to BLACK and the background colour set to WHITE.

5. ORs the temporary bitmap with the destination area.

6. Deletes the temporary bitmap.

This sequence of operations ensures that the source's transparent area need not be black, and logical functions are supported.

Note: on Windows, blitting with masks can be speeded up considerably by compiling wxWidgets with the wxUSE_DC_CACHE option enabled. You can also influence whether MaskBlt or the explicit mask blitting code above is used, by using *wxSystemOptions* (p. 1315) and setting the **no-maskblt** option to 1.

xsrcMask

Source x position on the mask. If both xsrcMask and ysrcMask are -1, xsrc and ysrc will be assumed for the mask source position. Currently only implemented on Windows.

ysrcMask

Source y position on the mask. If both xsrcMask and ysrcMask are -1, xsrc and ysrc will be assumed for the mask source position. Currently only implemented on Windows.

Remarks

There is partial support for Blit in wxPostScriptDC, under X.

See *wxMemoryDC* (p. 935) for typical usage.

See also

wxMemoryDC (p. 935), *wxBitmap* (p. 79), *wxMask* (p. 906)

wxDC::CalcBoundingBox

void CalcBoundingBox(**wxCoord** *x*, **wxCoord** *y*)

Adds the specified point to the bounding box which can be retrieved with *MinX* (p. 381), *MaxX* (p. 380) and *MinY* (p. 381), *MaxY* (p. 381) functions.

wxDC::Clear

void Clear()

Clears the device context using the current background brush.

wxDC::ComputeScaleAndOrigin

virtual void ComputeScaleAndOrigin()

Performs all necessary computations for given platform and context type after each change of scale and origin parameters. Usually called automatically internally after such changes.

wxDC::CrossHair

void CrossHair(**wxCoord** *x*, **wxCoord** *y*)

Displays a cross hair using the current pen. This is a vertical and horizontal line the height and width of the window, centred on the given point.

wxDC::DestroyClippingRegion

void DestroyClippingRegion()

Destroys the current clipping region so that none of the DC is clipped. See also *wxDC::SetClippingRegion* (p. 382).

wxDC::DeviceToLogicalX

wxCoord DeviceToLogicalX(**wxCoord** *x*)

Convert device X coordinate to logical coordinate, using the current mapping mode.

wxDC::DeviceToLogicalXRel

wxCoord DeviceToLogicalXRel(**wxCoord** *x*)

Convert device X coordinate to relative logical coordinate, using the current mapping mode but ignoring the x axis orientation. Use this function for converting a width, for example.

wxDC::DeviceToLogicalY

wxCoord DeviceToLogicalY(**wxCoord** *y*)

Converts device Y coordinate to logical coordinate, using the current mapping mode.

wxDC::DeviceToLogicalYRel

wxCoord DeviceToLogicalYRel(**wxCoord** *y*)

Convert device Y coordinate to relative logical coordinate, using the current mapping mode but ignoring the y axis orientation. Use this function for converting a height, for example.

wxDC::DrawArc

void DrawArc(**wxCoord** *x1*, **wxCoord** *y1*, **wxCoord** *x2*, **wxCoord** *y2*, **wxCoord** *xc*, **wxCoord** *yc*)

Draws an arc of a circle, centred on (*xc, yc*), with starting point (*x1, y1*) and ending at (*x2, y2*). The current pen is used for the outline and the current brush for filling the shape.

The arc is drawn in an anticlockwise direction from the start point to the end point.

wxDC::DrawBitmap

void DrawBitmap(**const wxBitmap&** *bitmap*, **wxCoord** *x*, **wxCoord** *y*, **bool** *transparent*)

Draw a bitmap on the device context at the specified point. If *transparent* is true and the bitmap has a transparency mask, the bitmap will be drawn transparently.

When drawing a mono-bitmap, the current text foreground colour will be used to draw the foreground of the bitmap (all bits set to 1), and the current text background colour to draw the background (all bits set to 0). See also *SetTextForeground* (p. 385), *SetTextBackground* (p. 385) and *wxMemoryDC* (p. 935).

wxDC::DrawCheckMark

void DrawCheckMark(**wxCoord** *x*, **wxCoord** *y*, **wxCoord** *width*, **wxCoord** *height*)

void DrawCheckMark(**const wxRect &***rect*)

Draws a check mark inside the given rectangle.

wxDC::DrawCircle

void DrawCircle(**wxCoord** *x*, **wxCoord** *y*, **wxCoord** *radius*)

void DrawCircle(**const wxPoint&** *pt*, **wxCoord** *radius*)

Draws a circle with the given centre and radius.

See also

DrawEllipse (p. 371)

wxDC::DrawEllipse

void DrawEllipse(**wxCoord** *x*, **wxCoord** *y*, **wxCoord** *width*, **wxCoord** *height*)

void DrawEllipse(**const wxPoint&** *pt*, **const wxSize&** *size*)

void DrawEllipse(**const wxRect&** *rect*)

Draws an ellipse contained in the rectangle specified either with the given top left corner and the given size or directly. The current pen is used for the outline and the current brush for filling the shape.

See also

DrawCircle (p. 370)

wxDC::DrawEllipticArc

void DrawEllipticArc(**wxCoord** *x*, **wxCoord** *y*, **wxCoord** *width*, **wxCoord** *height*, **double** *start*, **double** *end*)

Draws an arc of an ellipse. The current pen is used for drawing the arc and the current brush is used for drawing the pie.

x and *y* specify the x and y coordinates of the upper-left corner of the rectangle that contains the ellipse.

width and *height* specify the width and height of the rectangle that contains the ellipse.

start and *end* specify the start and end of the arc relative to the three-o'clock position from the center of the rectangle. Angles are specified in degrees (360 is a complete circle). Positive values mean counter-clockwise motion. If *start* is equal to *end*, a complete ellipse will be drawn.

wxDC::DrawIcon

void DrawIcon(**const wxIcon&** *icon*, **wxCoord** *x*, **wxCoord** *y*)

Draw an icon on the display (does nothing if the device context is PostScript). This can be the simplest way of drawing bitmaps on a window.

wxDC::DrawLabel

virtual void DrawLabel(**const wxString&** *text*, **const wxBitmap&** *image*, **const wxRect&** *rect*, **int** *alignment = wxALIGN_LEFT | wxALIGN_TOP,* **int** *indexAccel = -1*, **wxRect** **rectBounding = NULL*)

void DrawLabel(**const wxString&** *text*, **const wxRect&** *rect*, **int** *alignment = wxALIGN_LEFT | wxALIGN_TOP,* **int** *indexAccel = -1*)

Draw optional bitmap and the text into the given rectangle and aligns it as specified by alignment parameter; it also will emphasize the character with the given index if it is != -1 and return the bounding rectangle if required.

wxDC::DrawLine

void DrawLine(**wxCoord** *x1*, **wxCoord** *y1*, **wxCoord** *x2*, **wxCoord** *y2*)

Draws a line from the first point to the second. The current pen is used for drawing the line. Note that the point (x2, y2) is *not* part of the line and is not drawn by this function (this is consistent with the behaviour of many other toolkits).

wxDC::DrawLines

void DrawLines(**int** *n*, **wxPoint** *points[]*, **wxCoord** *xoffset = 0,* **wxCoord** *yoffset = 0*)

void DrawLines(**wxList** **points*, **wxCoord** *xoffset = 0,* **wxCoord** *yoffset = 0*)

Draws lines using an array of *points* of size *n*, or list of pointers to points, adding the optional offset coordinate. The current pen is used for drawing the lines. The programmer is responsible for deleting the list of points.

wxPython note: The wxPython version of this method accepts a Python list of wxPoint objects.

wxPerl note: The wxPerl version of this method accepts   as its first parameter a reference to an array of wxPoint objects.

wxDC::DrawPolygon

void DrawPolygon(**int** *n*, **wxPoint** *points[]*, **wxCoord** *xoffset = 0,* **wxCoord** *yoffset = 0,* **int** *fill_style = wxODDEVEN_RULE*)

void DrawPolygon(**wxList** **points*, **wxCoord** *xoffset = 0,* **wxCoord** *yoffset = 0,* **int** *fill_style = wxODDEVEN_RULE*)

Draws a filled polygon using an array of *points* of size *n*, or list of pointers to points, adding the optional offset coordinate.

The last argument specifies the fill rule: **wxODDEVEN_RULE** (the default) or

wxWINDING_RULE.

The current pen is used for drawing the outline, and the current brush for filling the shape. Using a transparent brush suppresses filling. The programmer is responsible for deleting the list of points.

Note that wxWidgets automatically closes the first and last points.

wxPython note: The wxPython version of this method accepts a Python list of wxPoint objects.

wxPerl note: The wxPerl version of this method accepts   as its first parameter a reference to an array of wxPoint objects.

wxDC::DrawPolyPolygon

void DrawPolyPolygon(**int** *n*, **int** *count[]*, **wxPoint** *points[]*, **wxCoord** *xoffset = 0*, **wxCoord** *yoffset = 0*,
  int *fill_style = wxODDEVEN_RULE*)

Draws two or more filled polygons using an array of *points*, adding the optional offset coordinates.

Notice that for the platforms providing a native implementation of this function (Windows and PostScript-based wxDC currently), this is more efficient than using *DrawPolygon* (p. 372) in a loop.

n specifies the number of polygons to draw, the array *count* of size *n* specifies the number of points in each of the polygons in the *points* array.

The last argument specifies the fill rule: **wxODDEVEN_RULE** (the default) or **wxWINDING_RULE**.

The current pen is used for drawing the outline, and the current brush for filling the shape. Using a transparent brush suppresses filling.

The polygons maybe disjoint or overlapping. Each polygon specified in a call to**DrawPolyPolygon** must be closed. Unlike polygons created by the*DrawPolygon* (p. 372) member function, the polygons created by**DrawPolyPolygon** are not closed automatically.

wxPython note: Not implemented yet

wxPerl note: Not implemented yet

wxDC::DrawPoint

void DrawPoint(**wxCoord** *x*, **wxCoord** *y*)

Draws a point using the color of the current pen. Note that the other properties of the pen are not used, such as width etc..

wxDC::DrawRectangle

void DrawRectangle(**wxCoord** *x*, **wxCoord** *y*, **wxCoord** *width*, **wxCoord** *height*)

Draws a rectangle with the given top left corner, and with the given size. The current pen is used for the outline and the current brush for filling the shape.

wxDC::DrawRotatedText

void DrawRotatedText(**const wxString&** *text*, **wxCoord** *x*, **wxCoord** *y*, **double** *angle*)

Draws the text rotated by *angle* degrees.

NB: Under Win9x only TrueType fonts can be drawn by this function. In particular, a font different from `wxNORMAL_FONT` should be used as the latter is not a TrueType font. `wxSWISS_FONT` is an example of a font which is.

See also

DrawText (p. 375)

wxDC::DrawRoundedRectangle

void DrawRoundedRectangle(**wxCoord** *x*, **wxCoord** *y*, **wxCoord** *width*, **wxCoord** *height*, **double** *radius = 20*)

Draws a rectangle with the given top left corner, and with the given size. The corners are quarter-circles using the given radius. The current pen is used for the outline and the current brush for filling the shape.

If *radius* is positive, the value is assumed to be the radius of the rounded corner. If *radius* is negative, the absolute value is assumed to be the *proportion* of the smallest dimension of the rectangle. This means that the corner can be a sensible size relative to the size of the rectangle, and also avoids the strange effects X produces when the corners are too big for the rectangle.

wxDC::DrawSpline

void DrawSpline(**int** *n*, **wxPoint** *points[]*)

Draws a spline between all given control points, using the current pen.

void DrawSpline(**wxList** **points*)

Draws a spline between all given control points, using the current pen. Doesn't delete the wxList and contents.

void DrawSpline(**wxCoord** *x1*, **wxCoord** *y1*, **wxCoord** *x2*, **wxCoord** *y2*, **wxCoord** *x3*, **wxCoord** *y3*)

Draws a three-point spline using the current pen.

wxPython note: The wxPython version of this method accepts a Python list of wxPoint objects.

wxPerl note: The wxPerl version of this method accepts a reference to an array of wxPoint objects.

wxDC::DrawText

void DrawText(**const wxString&** *text*, **wxCoord** *x*, **wxCoord** *y*)

Draws a text string at the specified point, using the current text font, and the current text foreground and background colours.

The coordinates refer to the top-left corner of the rectangle bounding the string. See *wxDC::GetTextExtent* (p. 379) for how to get the dimensions of a text string, which can be used to position the text more precisely.

NB: under wxGTK the current *logical function* (p. 377) is used by this function but it is ignored by wxMSW. Thus, you should avoid using logical functions with this function in portable programs.

wxDC::EndDoc

void EndDoc()

Ends a document (only relevant when outputting to a printer).

wxDC::EndDrawing

void EndDrawing()

Allows optimization of drawing code under MS Windows. Enclose drawing primitives between **BeginDrawing** and **EndDrawing** calls.

wxDC::EndPage

void EndPage()

Ends a document page (only relevant when outputting to a printer).

wxDC::FloodFill

bool FloodFill(**wxCoord** *x*, **wxCoord** *y*, **const wxColour&** *colour*, **int** *style=wxFLOOD_SURFACE*)

Flood fills the device context starting from the given point, using the *current brush colour*, and using a style:

- wxFLOOD_SURFACE: the flooding occurs until a colour other than the given colour is encountered.

- wxFLOOD_BORDER: the area to be flooded is bounded by the given colour.

Returns false if the operation failed.

Note: The present implementation for non-Windows platforms may fail to find colour borders if the pixels do not match the colour exactly. However the function will still return true.

wxDC::GetBackground

const wxBrush& GetBackground() const

Gets the brush used for painting the background (see *wxDC::SetBackground* (p. 382)).

wxDC::GetBackgroundMode

int GetBackgroundMode() const

Returns the current background mode: `wxSOLID` or `wxTRANSPARENT`.

See also

SetBackgroundMode (p. 382)

wxDC::GetBrush

const wxBrush& GetBrush() const

Gets the current brush (see *wxDC::SetBrush* (p. 382)).

wxDC::GetCharHeight

wxCoord GetCharHeight()

Gets the character height of the currently set font.

wxDC::GetCharWidth

wxCoord GetCharWidth()

Gets the average character width of the currently set font.

wxDC::GetClippingBox

void GetClippingBox(**wxCoord** **x*, **wxCoord** **y*, **wxCoord** **width*, **wxCoord** **height*)

Gets the rectangle surrounding the current clipping region.

wxPython note: No arguments are required and the four values defining the rectangle are returned as a tuple.

wxPerl note: This method takes no arguments and returns a four element list (x, y, width, height)

wxDC::GetFont

const wxFont& GetFont() **const**

Gets the current font (see *wxDC::SetFont* (p. 383)).

wxDC::GetLogicalFunction

int GetLogicalFunction()

Gets the current logical function (see *wxDC::SetLogicalFunction* (p. 383)).

wxDC::GetMapMode

int GetMapMode()

Gets the *mapping mode* for the device context (see *wxDC::SetMapMode* (p. 384)).

wxDC::GetPartialTextExtents

bool GetPartialTextExtents(**const wxString&** *text*, **wxArrayInt&** *widths*) **const**

Fills the *widths* array with the widths from the beginning of *text* to the corresponding character of *text*. The generic version simply builds a running total of the widths of each character using *GetTextExtent* (p. 379), however if the various platforms have a native API function that is faster or more accurate than the generic implementation then it should be used instead.

wxPython note: This method only takes the *text* parameter and returns a Python list of integers.

wxDC::GetPen

const wxPen& GetPen() **const**

Gets the current pen (see *wxDC::SetPen* (p. 384)).

wxDC::GetPixel

bool GetPixel(**wxCoord** *x*, **wxCoord** *y*, **wxColour ****colour*)

Sets *colour* to the colour at the specified location. Not available for wxPostScriptDC or wxMetafileDC.

Note that setting a pixel can be done using *DrawPoint* (p. 373).

wxPython note: For wxPython the wxColour value is returned and is not required as a parameter.

wxPerl note: This method only takes the parameters x and y and returns a Wx::Colour value

wxDC::GetPPI

wxSize GetPPI() **const**

Returns the resolution of the device in pixels per inch.

wxDC::GetSize

void GetSize(**wxCoord ****width*, **wxCoord ****height*) **const**

wxSize GetSize() **const**

This gets the horizontal and vertical resolution in device units. It can be used to scale graphics to fit the page. For example, if *maxX* and *maxY* represent the maximum horizontal and vertical 'pixel' values used in your application, the following code will scale the graphic to fit on the printer page:

```
wxCoord w, h;
dc.GetSize(&w, &h);
double scaleX=(double)(maxX/w);
double scaleY=(double)(maxY/h);
dc.SetUserScale(min(scaleX,scaleY),min(scaleX,scaleY));
```

wxPython note: In place of a single overloaded method name, wxPython implements the following methods:

        GetSize()   Returns a wxSize

        GetSizeTuple()   Returns a 2-tuple (width, height)

wxPerl note: In place of a single overloaded method, wxPerl uses:

        GetSize()   Returns a Wx::Size

GetSizeWH()	Returns a 2-element list (`width, height )`

wxDC::GetSizeMM

void GetSizeMM(**wxCoord** **width*, **wxCoord** **height*) **const**

wxSize GetSizeMM() **const**

Returns the horizontal and vertical resolution in millimetres.

wxDC::GetTextBackground

const wxColour& GetTextBackground() **const**

Gets the current text background colour (see *wxDC::SetTextBackground* (p. 385)).

wxDC::GetTextExtent

void GetTextExtent(**const wxString&** *string*, **wxCoord** **w*, **wxCoord** **h*,
 wxCoord **descent* = *NULL*, **wxCoord** **externalLeading* = *NULL*, **wxFont** **font* =
NULL)

Gets the dimensions of the string using the currently selected font. *string* is the text
string to measure, *w* and *h* are the total width and height respectively, *descent* is the
dimension from the baseline of the font to the bottom of the descender, and
externalLeading is any extra vertical space added to the font by the font designer
(usually is zero).

The optional parameter *font* specifies an alternative to the currently selected font: but
note that this does not yet work under Windows, so you need to set a font for the device
context first.

See also *wxFont* (p. 558), *wxDC::SetFont* (p. 383).

wxPython note: The following methods are implemented in wxPython:

GetTextExtent(string)	Returns a 2-tuple, (width, height)
GetFullTextExtent(string, font=NULL)	Returns a 4-tuple, (width, height, descent, externalLeading)

wxPerl note: In wxPerl this method is implemented as **GetTextExtent(string, font =
undef)** returning a four element array (`width, height, descent,`
`externalLeading )`

wxDC::GetTextForeground

const wxColour& GetTextForeground() **const**

Gets the current text foreground colour (see *wxDC::SetTextForeground* (p. 385)).

wxDC::GetUserScale

void GetUserScale(**double** **x*, **double** **y*)

Gets the current user scale factor (set by *SetUserScale* (p. 385)).

wxPerl note: In wxPerl this method takes no arguments and return a two element array
(x, y)

wxDC::LogicalToDeviceX

wxCoord LogicalToDeviceX(**wxCoord** *x*)

Converts logical X coordinate to device coordinate, using the current mapping mode.

wxDC::LogicalToDeviceXRel

wxCoord LogicalToDeviceXRel(**wxCoord** *x*)

Converts logical X coordinate to relative device coordinate, using the current mapping mode but ignoring the x axis orientation. Use this for converting a width, for example.

wxDC::LogicalToDeviceY

wxCoord LogicalToDeviceY(**wxCoord** *y*)

Converts logical Y coordinate to device coordinate, using the current mapping mode.

wxDC::LogicalToDeviceYRel

wxCoord LogicalToDeviceYRel(**wxCoord** *y*)

Converts logical Y coordinate to relative device coordinate, using the current mapping mode but ignoring the y axis orientation. Use this for converting a height, for example.

wxDC::MaxX

wxCoord MaxX()

Gets the maximum horizontal extent used in drawing commands so far.

wxDC::MaxY

wxCoord MaxY()

Gets the maximum vertical extent used in drawing commands so far.

wxDC::MinX

wxCoord MinX()

Gets the minimum horizontal extent used in drawing commands so far.

wxDC::MinY

wxCoord MinY()

Gets the minimum vertical extent used in drawing commands so far.

wxDC::Ok

bool Ok()

Returns true if the DC is ok to use.

wxDC::ResetBoundingBox

void ResetBoundingBox()

Resets the bounding box: after a call to this function, the bounding box doesn't contain anything.

See also

CalcBoundingBox (p. 368)

wxDC::SetAxisOrientation

void SetAxisOrientation(**bool** *xLeftRight,*   **bool** *yBottomUp*)

Sets the x and y axis orientation (i.e., the direction from lowest to highest values on the axis). The default orientation is x axis from left to right and y axis from top down.

Parameters

xLeftRight

> True to set the x axis orientation to the natural left to right orientation, false to invert it.

yBottomUp

> True to set the y axis orientation to the natural bottom up orientation, false to invert it.

wxDC::SetBackground

void SetBackground(**const wxBrush&** *brush*)

Sets the current background brush for the DC.

wxDC::SetBackgroundMode

void SetBackgroundMode(**int** *mode*)

mode may be one of wxSOLID and wxTRANSPARENT. This setting determines whether text will be drawn with a background colour or not.

wxDC::SetBrush

void SetBrush(**const wxBrush&** *brush*)

Sets the current brush for the DC.

If the argument is wxNullBrush, the current brush is selected out of the device context, and the original brush restored, allowing the current brush to be destroyed safely.

See also *wxBrush* (p. 103).

See also *wxMemoryDC* (p. 935) for the interpretation of colours when drawing into a monochrome bitmap.

wxDC::SetClippingRegion

void SetClippingRegion(**wxCoord** *x*, **wxCoord** *y*, **wxCoord** *width*, **wxCoord** *height*)

void SetClippingRegion(**const wxPoint&** *pt*, **const wxSize&** *sz*)

void SetClippingRegion(**const wxRect&** *rect*)

void SetClippingRegion(**const wxRegion&** *region*)

Sets the clipping region for this device context to the intersection of the given region described by the parameters of this method and the previously set clipping region. You should call *DestroyClippingRegion* (p. 369) if you want to set the clipping region exactly to the region specified.

The clipping region is an area to which drawing is restricted. Possible uses for the clipping region are for clipping text or for speeding up window redraws when only a

known area of the screen is damaged.

See also

wxDC::DestroyClippingRegion (p. 369), *wxRegion* (p. 1111)

wxDC::SetDeviceOrigin

void SetDeviceOrigin(**wxCoord** *x*, **wxCoord** *y*)

Sets the device origin (i.e., the origin in pixels after scaling has been applied).

This function may be useful in Windows printing operations for placing a graphic on a page.

wxDC::SetFont

void SetFont(**const wxFont&** *font*)

Sets the current font for the DC. It must be a valid font, in particular you should not pass wxNullFont to this method.

See also *wxFont* (p. 558).

wxDC::SetLogicalFunction

void SetLogicalFunction(**int** *function*)

Sets the current logical function for the device context. This determines how a source pixel (from a pen or brush colour, or source device context if using *wxDC::Blit* (p. 367)) combines with a destination pixel in the current device context.

The possible values and their meaning in terms of source and destination pixel values are as follows:

```
wxAND                   src AND dst
wxAND_INVERT            (NOT src) AND dst
wxAND_REVERSE           src AND (NOT dst)
wxCLEAR                 0
wxCOPY                  src
wxEQUIV                 (NOT src) XOR dst
wxINVERT                NOT dst
wxNAND                  (NOT src) OR (NOT dst)
wxNOR                   (NOT src) AND (NOT dst)
wxNO_OP                 dst
wxOR                    src OR dst
wxOR_INVERT             (NOT src) OR dst
wxOR_REVERSE            src OR (NOT dst)
wxSET                   1
wxSRC_INVERT            NOT src
wxXOR                   src XOR dst
```

The default is wxCOPY, which simply draws with the current colour. The others combine the current colour and the background using a logical operation. wxINVERT is commonly used for drawing rubber bands or moving outlines, since drawing twice reverts to the original colour.

wxDC::SetMapMode

void SetMapMode(**int** *int*)

The *mapping mode* of the device context defines the unit of measurement used to convert logical units to device units. Note that in X, text drawing isn't handled consistently with the mapping mode; a font is always specified in point size. However, setting the *user scale* (see *wxDC::SetUserScale* (p. 385)) scales the text appropriately. In Windows, scalable TrueType fonts are always used; in X, results depend on availability of fonts, but usually a reasonable match is found.

The coordinate origin is always at the top left of the screen/printer.

Drawing to a Windows printer device context uses the current mapping mode, but mapping mode is currently ignored for PostScript output.

The mapping mode can be one of the following:

wxMM_TWIPS	Each logical unit is 1/20 of a point, or 1/1440 of an inch.
wxMM_POINTS	Each logical unit is a point, or 1/72 of an inch.
wxMM_METRIC	Each logical unit is 1 mm.
wxMM_LOMETRIC	Each logical unit is 1/10 of a mm.
wxMM_TEXT	Each logical unit is 1 pixel.

wxDC::SetPalette

void SetPalette(**const wxPalette&** *palette*)

If this is a window DC or memory DC, assigns the given palette to the window or bitmap associated with the DC. If the argument is wxNullPalette, the current palette is selected out of the device context, and the original palette restored.

See *wxPalette* (p. 1027) for further details.

wxDC::SetPen

void SetPen(**const wxPen&** *pen*)

Sets the current pen for the DC.

If the argument is wxNullPen, the current pen is selected out of the device context, and the original pen restored.

See also *wxMemoryDC* (p. 935) for the interpretation of colours when drawing into a monochrome bitmap.

wxDC::SetTextBackground

void SetTextBackground(**const wxColour&** *colour*)

Sets the current text background colour for the DC.

wxDC::SetTextForeground

void SetTextForeground(**const wxColour&** *colour*)

Sets the current text foreground colour for the DC.

See also *wxMemoryDC* (p. 935) for the interpretation of colours when drawing into a monochrome bitmap.

wxDC::SetUserScale

void SetUserScale(**double** *xScale*, **double** *yScale*)

Sets the user scaling factor, useful for applications which require 'zooming'.

wxDC::StartDoc

bool StartDoc(**const wxString&** *message*)

Starts a document (only relevant when outputting to a printer). Message is a message to show while printing.

wxDC::StartPage

bool StartPage()

Starts a document page (only relevant when outputting to a printer).

wxDCClipper

This is a small helper class which sets the specified DC to its constructor clipping region and then automatically destroys it in its destructor. Using it ensures that an unwanted clipping region is not left set on the DC.

Derived from

No base class

Include files

<wx/dc.h>

See also

wxDC (p. 365)

wxDCClipper::wxDCClipper

wxDCClipper(**wxDC&** *dc,* **wxCoord** *x,***wxCoord** *y,***wxCoord** *w,***wxCoord** *h,*)

wxDCClipper(**wxDC&** *dc,* **const wxRect&** *rect*)

Constructor: sets the clipping region for the given device context to the specified rectangle.

wxDCClipper::~wxDCClipper

~wxDCClipper()

Destructor: destroys the clipping region set in the constructor.

wxDDEClient

A wxDDEClient object represents the client part of a client-server DDE (Dynamic Data Exchange) conversation.

To create a client which can communicate with a suitable server, you need to derive a class from wxDDEConnection and another from wxDDEClient. The custom wxDDEConnection class will intercept communications in a 'conversation' with a server, and the custom wxDDEServer is required so that a user-overridden *wxDDEClient::OnMakeConnection* (p. 387) member can return a wxDDEConnection of the required class, when a connection is made.

This DDE-based implementation is available on Windows only, but a platform-independent, socket-based version of this API is available using *wxTCPClient* (p. 1325).

Derived from

wxClientBase
wxObject (p. 1012)

Include files

<wx/dde.h>

See also

wxDDEServer (p. 391), *wxDDEConnection* (p. 387), *Interprocess communications overview* (p. 1841)

wxDDEClient::wxDDEClient

 wxDDEClient()

Constructs a client object.

wxDDEClient::MakeConnection

wxConnectionBase * MakeConnection(**const wxString&** *host*, **const wxString&** *service*, **const wxString&** *topic*)

Tries to make a connection with a server specified by the host (machine name under UNIX, ignored under Windows), service name (must contain an integer port number under UNIX), and topic string. If the server allows a connection, a wxDDEConnection object will be returned. The type of wxDDEConnection returned can be altered by overriding the *wxDDEClient::OnMakeConnection* (p. 387) member to return your own derived connection object.

wxDDEClient::OnMakeConnection

wxConnectionBase * OnMakeConnection()

The type of *wxDDEConnection* (p. 387) returned from a *wxDDEClient::MakeConnection* (p. 387) call can be altered by deriving the **OnMakeConnection** member to return your own derived connection object. By default, a wxDDEConnection object is returned.

The advantage of deriving your own connection class is that it will enable you to intercept messages initiated by the server, such as *wxDDEConnection::OnAdvise* (p. 209). You may also want to store application-specific data in instances of the new class.

wxDDEClient::ValidHost

bool ValidHost(**const wxString&** *host*)

Returns `true` if this is a valid host name, `false` otherwise. This always returns `true` under MS Windows.

wxDDEConnection

A wxDDEConnection object represents the connection between a client and a server. It can be created by making a connection using a *wxDDEClient* (p. 386) object, or by the acceptance of a connection by a *wxDDEServer* (p. 391) object. The bulk of a DDE (Dynamic Data Exchange) conversation is controlled by calling members in a **wxDDEConnection** object or by overriding its members.

An application should normally derive a new connection class from wxDDEConnection, in order to override the communication event handlers to do something interesting.

This DDE-based implementation is available on Windows only, but a platform-independent, socket-based version of this API is available using *wxTCPConnection* (p. 1327).

Derived from

wxConnectionBase
wxObject (p. 1012)

Include files

<wx/dde.h>

Types

wxIPCFormat is defined as follows:

```
enum wxIPCFormat
{
  wxIPC_INVALID =          0,
  wxIPC_TEXT =             1,  /* CF_TEXT */
  wxIPC_BITMAP =           2,  /* CF_BITMAP */
  wxIPC_METAFILE =         3,  /* CF_METAFILEPICT */
  wxIPC_SYLK =             4,
  wxIPC_DIF =              5,
  wxIPC_TIFF =             6,
  wxIPC_OEMTEXT =          7,  /* CF_OEMTEXT */
  wxIPC_DIB =              8,  /* CF_DIB */
  wxIPC_PALETTE =          9,
  wxIPC_PENDATA =          10,
  wxIPC_RIFF =             11,
  wxIPC_WAVE =             12,
  wxIPC_UNICODETEXT =      13,
  wxIPC_ENHMETAFILE =      14,
  wxIPC_FILENAME =         15, /* CF_HDROP */
  wxIPC_LOCALE =           16,
  wxIPC_PRIVATE =          20
};
```

See also

wxDDEClient (p. 386), *wxDDEServer* (p. 391), *Interprocess communications overview* (p. 1841)

wxDDEConnection::wxDDEConnection

wxDDEConnection()

wxDDEConnection(**char*** *buffer*, **int** *size*)

Constructs a connection object. If no user-defined connection object is to be derived from wxDDEConnection, then the constructor should not be called directly, since the default connection object will be provided on requesting (or accepting) a connection. However, if the user defines his or her own derived connection object, the *wxDDEServer::OnAcceptConnection* (p. 392) and/or *wxDDEClient::OnMakeConnection* (p. 387) members should be replaced by functions which construct the new connection object. If the arguments of the wxDDEConnection constructor are void, then a default buffer is associated with the connection. Otherwise, the programmer must provide a a buffer and size of the buffer for the connection object to use in transactions.

wxDDEConnection::Advise

bool Advise(**const wxString&** *item*, **char*** *data*, **int** *size = -1*, **wxIPCFormat** *format = wxCF_TEXT*)

Called by the server application to advise the client of a change in the data associated with the given item. Causes the client connection's *wxDDEConnection::OnAdvise* (p. 389)member to be called. Returns true if successful.

wxDDEConnection::Execute

bool Execute(**char*** *data*, **int** *size = -1*, **wxIPCFormat** *format = wxCF_TEXT*)

Called by the client application to execute a command on the server. Can also be used to transfer arbitrary data to the server (similar to *wxDDEConnection::Poke* (p. 391) in that respect). Causes the server connection's *wxDDEConnection::OnExecute* (p. 390) member to be called. Returns true if successful.

wxDDEConnection::Disconnect

bool Disconnect()

Called by the client or server application to disconnect from the other program; it causes the *wxDDEConnection::OnDisconnect* (p. 390) message to be sent to the corresponding connection object in the other program. The default behaviour of **OnDisconnect** is to delete the connection, but the calling application must explicitly delete its side of the connection having called **Disconnect**. Returns true if successful.

wxDDEConnection::OnAdvise

virtual bool OnAdvise(**const wxString&** *topic*, **const wxString&** *item*, **char*** *data*, **int** *size*, **wxIPCFormat** *format*)

Message sent to the client application when the server notifies it of a change in the data

associated with the given item.

wxDDEConnection::OnDisconnect

virtual bool OnDisconnect()

Message sent to the client or server application when the other application notifies it to delete the connection. Default behaviour is to delete the connection object.

wxDDEConnection::OnExecute

virtual bool OnExecute(**const wxString&** *topic*, **char*** *data*, **int** *size*, **wxIPCFormat** *format*)

Message sent to the server application when the client notifies it to execute the given data. Note that there is no item associated with this message.

wxDDEConnection::OnPoke

virtual bool OnPoke(**const wxString&** *topic*, **const wxString&** *item*, **char*** *data*, **int** *size*, **wxIPCFormat** *format*)

Message sent to the server application when the client notifies it to accept the given data.

wxDDEConnection::OnRequest

virtual char* OnRequest(**const wxString&** *topic*, **const wxString&** *item*, **int ****size*, **wxIPCFormat** *format*)

Message sent to the server application when the client calls *wxDDEConnection::Request* (p. 391). The server should respond by returning a character string from **OnRequest**, or NULL to indicate no data.

wxDDEConnection::OnStartAdvise

virtual bool OnStartAdvise(**const wxString&** *topic*, **const wxString&** *item*)

Message sent to the server application by the client, when the client wishes to start an 'advise loop' for the given topic and item. The server can refuse to participate by returning false.

wxDDEConnection::OnStopAdvise

virtual bool OnStopAdvise(**const wxString&** *topic*, **const wxString&** *item*)

Message sent to the server application by the client, when the client wishes to stop an 'advise loop' for the given topic and item. The server can refuse to stop the advise loop by returning false, although this doesn't have much meaning in practice.

wxDDEConnection::Poke

bool Poke(**const wxString&** *item*, **char*** *data*, **int** *size = -1*, **wxIPCFormat** *format = wxCF_TEXT*)

Called by the client application to poke data into the server. Can be used to transfer arbitrary data to the server. Causes the server connection's *wxDDEConnection::OnPoke* (p. 390) member to be called. Returns true if successful.

wxDDEConnection::Request

char* Request(**const wxString&** *item*, **int ****size*, **wxIPCFormat** *format = wxIPC_TEXT*)

Called by the client application to request data from the server. Causes the server connection's *wxDDEConnection::OnRequest* (p. 390) member to be called. Returns a character string (actually a pointer to the connection's buffer) if successful, NULL otherwise.

wxDDEConnection::StartAdvise

bool StartAdvise(**const wxString&** *item*)

Called by the client application to ask if an advise loop can be started with the server. Causes the server connection's *wxDDEConnection::OnStartAdvise* (p. 390) member to be called. Returns true if the server okays it, false otherwise.

wxDDEConnection::StopAdvise

bool StopAdvise(**const wxString&** *item*)

Called by the client application to ask if an advise loop can be stopped. Causes the server connection's *wxDDEConnection::OnStopAdvise* (p. 390) member to be called. Returns true if the server okays it, false otherwise.

wxDDEServer

A wxDDEServer object represents the server part of a client-server DDE (Dynamic Data Exchange) conversation.

This DDE-based implementation is available on Windows only, but a platform-independent, socket-based version of this API is available using *wxTCPServer* (p. 1331).

Derived from

wxServerBase

Include files

<wx/dde.h>

See also

wxDDEClient (p. 386), *wxDDEConnection* (p. 387), *IPC overview* (p. 1841)

wxDDEServer::wxDDEServer

wxDDEServer()

Constructs a server object.

wxDDEServer::Create

bool Create(**const wxString&** *service*)

Registers the server using the given service name. Under UNIX, the string must contain an integer id which is used as an Internet port number. false is returned if the call failed (for example, the port number is already in use).

wxDDEServer::OnAcceptConnection

virtual wxConnectionBase * OnAcceptConnection(**const wxString&** *topic*)

When a client calls **MakeConnection**, the server receives the message and this member is called. The application should derive a member to intercept this message and return a connection object of either the standard wxDDEConnection type, or of a user-derived type. If the topic is "STDIO", the application may wish to refuse the connection. Under UNIX, when a server is created the OnAcceptConnection message is always sent for standard input and output, but in the context of DDE messages it doesn't make a lot of sense.

wxDebugContext

A class for performing various debugging and memory tracing operations. Full functionality (such as printing out objects currently allocated) is only present in a debugging build of wxWidgets, i.e. if the __WXDEBUG__ symbol is defined. wxDebugContext and related functions and macros can be compiled out by setting wxUSE_DEBUG_CONTEXT to 0 is setup.h

Derived from

No parent class.

Include files

<wx/memory.h>

See also

Overview (p. 1747)

wxDebugContext::Check

int Check()

Checks the memory blocks for errors, starting from the currently set checkpoint.

Return value

Returns the number of errors, so a value of zero represents success. Returns -1 if an error was detected that prevents further checking.

wxDebugContext::Dump

bool Dump()

Performs a memory dump from the currently set checkpoint, writing to the current debug stream. Calls the **Dump** member function for each wxObject derived instance.

Return value

true if the function succeeded, false otherwise.

wxDebugContext::GetCheckPrevious

bool GetCheckPrevious()

Returns true if the memory allocator checks all previous memory blocks for errors. By default, this is false since it slows down execution considerably.

See also

wxDebugContext::GetDebugMode

bool GetDebugMode()

Returns true if debug mode is on. If debug mode is on, the wxObject new and delete operators store or use information about memory allocation. Otherwise, a straight malloc and free will be performed by these operators.

See also

wxDebugContext::SetDebugMode (p. 396)

wxDebugContext::GetLevel

int GetLevel()

Gets the debug level (default 1). The debug level is used by the wxTraceLevel function and the WXTRACELEVEL macro to specify how detailed the trace information is; setting a different level will only have an effect if trace statements in the application specify a value other than one.

This is obsolete, replaced by *wxLog* (p. 889) functionality.

See also

wxDebugContext::SetLevel (p. 396)

wxDebugContext::GetStream

ostream& GetStream()

Returns the output stream associated with the debug context.

This is obsolete, replaced by *wxLog* (p. 889) functionality.

See also

wxDebugContext::SetStream (p. 397)

wxDebugContext::GetStreamBuf

streambuf* GetStreamBuf()

Returns a pointer to the output stream buffer associated with the debug context. There may not necessarily be a stream buffer if the stream has been set by the user.

This is obsolete, replaced by *wxLog* (p. 889) functionality.

wxDebugContext::HasStream

bool HasStream()

Returns true if there is a stream currently associated with the debug context.

This is obsolete, replaced by *wxLog* (p. 889) functionality.

See also

wxDebugContext::SetStream (p. 397), *wxDebugContext::GetStream* (p. 394)

wxDebugContext::PrintClasses

bool PrintClasses()

Prints a list of the classes declared in this application, giving derivation and whether instances of this class can be dynamically created.

See also

wxDebugContext::PrintStatistics (p. 395)

wxDebugContext::PrintStatistics

bool PrintStatistics(**bool** *detailed = true*)

Performs a statistics analysis from the currently set checkpoint, writing to the current debug stream. The number of object and non-object allocations is printed, together with the total size.

Parameters

detailed

> If true, the function will also print how many objects of each class have been allocated, and the space taken by these class instances.

See also

wxDebugContext::PrintStatistics (p. 395)

wxDebugContext::SetCheckpoint

void SetCheckpoint(**bool** *all = false*)

Sets the current checkpoint: Dump and PrintStatistics operations will be performed from this point on. This allows you to ignore allocations that have been performed up to this point.

Parameters

all

> If true, the checkpoint is reset to include all memory allocations since the program started.

wxDebugContext::SetCheckPrevious

void SetCheckPrevious(**bool** *check*)

Tells the memory allocator to check all previous memory blocks for errors. By default, this is false since it slows down execution considerably.

See also

wxDebugContext::GetCheckPrevious (p. 393)

wxDebugContext::SetDebugMode

void SetDebugMode(**bool** *debug*)

Sets the debug mode on or off. If debug mode is on, the wxObject new and delete operators store or use information about memory allocation. Otherwise, a straight malloc and free will be performed by these operators.

By default, debug mode is on if __WXDEBUG__ is defined. If the application uses this function, it should make sure that all object memory allocated is deallocated with the same value of debug mode. Otherwise, the delete operator might try to look for memory information that does not exist.

See also

wxDebugContext::GetDebugMode (p. 393)

wxDebugContext::SetFile

bool SetFile(**const wxString&** *filename*)

Sets the current debug file and creates a stream. This will delete any existing stream and stream buffer. By default, the debug context stream outputs to the debugger (Windows) or standard error (other platforms).

wxDebugContext::SetLevel

void SetLevel(**int** *level*)

Sets the debug level (default 1). The debug level is used by the wxTraceLevel function and the WXTRACELEVEL macro to specify how detailed the trace information is; setting a different level will only have an effect if trace statements in the application specify a value other than one.

This is obsolete, replaced by *wxLog* (p. 889) functionality.

See also

wxDebugContext::GetLevel (p. 394)

wxDebugContext::SetStandardError

bool SetStandardError()

Sets the debugging stream to be the debugger (Windows) or standard error (other platforms). This is the default setting. The existing stream will be flushed and deleted.

This is obsolete, replaced by *wxLog* (p. 889) functionality.

wxDebugContext::SetStream

void SetStream(**ostream*** *stream*, **streambuf*** *streamBuf = NULL*)

Sets the stream and optionally, stream buffer associated with the debug context. This operation flushes and deletes the existing stream (and stream buffer if any).

This is obsolete, replaced by *wxLog* (p. 889) functionality.

Parameters

stream

      Stream to associate with the debug context. Do not set this to NULL.

streamBuf

      Stream buffer to associate with the debug context.

See also

wxDebugContext::GetStream (p. 394), *wxDebugContext::HasStream* (p. 394)

wxDebugStreamBuf

This class allows you to treat debugging output in a similar (stream-based) fashion on different platforms. Under Windows, an ostream constructed with this buffer outputs to the debugger, or other program that intercepts debugging output. On other platforms, the output goes to standard error (cerr).

This is soon to be obsolete, replaced by *wxLog* (p. 889) functionality.

Derived from

streambuf

Include files

<wx/memory.h>

Example

```
wxDebugStreamBuf streamBuf;
```

```
ostream stream(&streamBuf);

stream << "Hello world!" << endl;
```

See also

Overview (p. 1747)

wxDebugReport

wxDebugReport is used to generate a debug report, containing information about the program current state. It is usually used from *wxApp::OnFatalException()* (p. 37) as shown in the *sample* (p. 1709).

A wxDebugReport object contains one or more files. A few of them can be created by the class itself but more can be created from the outside and then added to the report. Also note that several virtual functions may be overridden to further customize the class behaviour.

Once a report is fully assembled, it can simply be left in the temporary directory so that the user can email it to the developers (in which case you should still use *wxDebugReportCompress* (p. 402) to compress it in a single file) or uploaded to a Web server using *wxDebugReportUpload* (p. 404) (setting up the Web server to accept uploads is your responsibility, of course). Other handlers, for example for automatically emailing the report, can be defined as well but are not currently included in wxWidgets.

Example of use

```
wxDebugReport report;
wxDebugReportPreviewStd preview;

report.AddCurrentContext();  // could also use AddAll()
report.AddCurrentDump();     // to do both at once

if ( preview.Show(report) )
    report.Process();
```

Derived from

No base class

Include files

<wx/debugrpt.h>

Data structures

This enum is used for functions that report either the current state or the state during the last (fatal) exception:

```
enum wxDebugReport::Context
{
    Context_Current,
    Context_Exception
};
```

wxDebugReport::wxDebugReport

wxDebugReport()

The constructor creates a temporary directory where the files that will be included in the report are created. Use *IsOk()* (p. 402) to check for errors.

wxDebugReport::~wxDebugReport

~wxDebugReport()

The destructor normally destroys the temporary directory created in the constructor with all the files it contains. Call *Reset()* (p. 402) to prevent this from happening.

wxDebugReport::AddAll

void AddAll(**Context** *context = Context_Exception*)

Adds all available information to the report. Currently this includes a text (XML) file describing the process context and, under Win32, a minidump file.

wxDebugReport::AddContext

bool AddContext(**Context** *ctx*)

Add an XML file containing the current or exception context and the stack trace.

wxDebugReport::AddCurrentContext

bool AddCurrentContext()

The same as *AddContext(Context_Current)* (p. 399).

wxDebugReport::AddCurrentDump

bool AddCurrentDump()

The same as *AddDump(Context_Current)* (p. 399).

wxDebugReport::AddDump

bool AddDump(**Context** *ctx*)

Adds the minidump file to the debug report.

Minidumps are only available under recent Win32 versions (`dbghlp32.dll` can be installed under older systems to make minidumps available).

wxDebugReport::AddExceptionContext

bool AddExceptionContext()

The same as *AddContext(Context_Exception)* (p. 399).

wxDebugReport::AddExceptionDump

bool AddExceptionDump()

The same as *AddDump(Context_Exception)* (p. 399).

wxDebugReport::AddFile

void AddFile(**const wxString&** *name*, **const wxString&** *description*)

Add another file to the report: the file must already exist on disk. Its name is relative to *GetDirectory()* (p. 401).

description only exists to be shown to the user in the report summary shown by *wxDebugReportPreview* (p. 403).

wxDebugReport::AddText

bool AddText(**const wxString&** *name*, **const wxString&** *text*, **const wxString&** *description*)

This is a convenient wrapper around *AddFile* (p. 400). It creates the file with the given *name* and writes *text* to it, then adds the file to the report.

Returns `true` if file could be added successfully, `false` if an IO error occured.

wxDebugReport::DoAddCustomContext

void DoAddCustomContext(**wxXmlNode *** *nodeRoot*)

This function may be overridden to add arbitrary custom context to the XML context file created by *AddContext* (p. 399). By default, it does nothing.

wxDebugReport::DoAddExceptionInfo

bool DoAddExceptionInfo(**wxXmlNode*** *nodeContext*)

This function may be overridden to modify the contents of the exception tag in the XML context file.

wxDebugReport::DoAddLoadedModules

bool DoAddLoadedModules(**wxXmlNode*** *nodeModules*)

This function may be overridden to modify the contents of the modules tag in the XML context file.

wxDebugReport::DoAddSystemInfo

bool DoAddSystemInfo(**wxXmlNode*** *nodeSystemInfo*)

This function may be overridden to modify the contents of the system tag in the XML context file.

wxDebugReport::GetDirectory

const wxString& GetDirectory() **const**

Returns the name of the temporary directory used for the files in this report.

This method should be used to construct the full name of the files which you wish to add to the report using *AddFile* (p. 400).

wxDebugReport::GetFile

bool GetFile(**size_t** *n*, **wxString*** *name*, **wxString*** *desc*) **const**

Retrieves the name (relative to *GetDirectory()* (p. 401)) and the description of the file with the given index. If *n* is greater than or equal to the number of filse, `false` is returned.

wxDebugReport::GetFilesCount

size_t GetFilesCount() **const**

Gets the current number files in this report.

wxDebugReport::GetReportName

wxString GetReportName() **const**

Gets the name used as a base name for various files, by default *wxApp::GetAppName()* (p. 33) is used.

wxDebugReport::IsOk

bool IsOk() **const**

Returns `true` if the object was successfully initialized. If this method returns `false` the report can't be used.

wxDebugReport::Process

bool Process()

Processes this report: the base class simply notifies the user that the report has been generated. This is usually not enough -- instead you should override this method to do something more useful to you.

wxDebugReport::RemoveFile

void RemoveFile(**const wxString&** *name*)

Removes the file from report: this is used by *wxDebugReportPreview* (p. 403) to allow the user to remove files potentially containing private information from the report.

wxDebugReport::Reset

void Reset()

Resets the directory name we use. The object can't be used any more after this as it becomes uninitialized and invalid.

wxDebugReportCompress

wxDebugReportCompress is a *wxDebugReport* (p. 398) which compresses all the files in this debug report into a single .ZIP file in its *Process()* function.

Derived from

wxDebugReport (p. 398)

Include files

<wx/debugrpt.h>

wxDebugReportCompress::wxDebugReportCompress

wxDebugReportCompress()

Default constructor does nothing special.

wxDebugReportCompress::GetCompressedFileName

const wxString& GetCompressedFileName() **const**

Returns the full path of the compressed file (empty if creation failed).

wxDebugReportPreview

This class presents the debug report to the user and allows him to veto report entirely or remove some parts of it. Although not mandatory, using this class is strongly recommended as data included in the debug report might contain sensitive private information and the user should be notified about it as well as having a possibility to examine the data which had been gathered to check whether this is effectively the case and discard the debug report if it is.

wxDebugReportPreview is an abstract base class, currently the only concrete class deriving from it is *wxDebugReportPreviewStd* (p. 404).

Derived from

No base class

Include files

<wx/debugrpt.h>

wxDebugReportPreview::wxDebugReportPreview

wxDebugReportPreview()

Trivial default constructor.

wxDebugReportPreview::~wxDebugReportPreview

~wxDebugReportPreview()

dtor is trivial as well but should be virtual for a base class

wxDebugReportPreview::Show

bool Show(**wxDebugReport&** *dbgrpt*) **const**

Present the report to the user and allow him to modify it by removing some or all of the files and, potentially, adding some notes. Return `true` if the report should be processed or `false` if the user chose to cancel report generation or removed all files from it.

wxDebugReportPreviewStd

wxDebugReportPreviewStd is a standard debug report preview window. It displays a GUIdialog allowing the user to examine the contents of a debug report, remove files from and add notes to it.

Derived from

wxDebugReportPreview (p. 403)

Include files

<wx/debugrpt.h>

wxDebugReportPreviewStd::wxDebugReportPreviewStd

 wxDebugReportPreviewStd()

Trivial default constructor.

wxDebugReportPreviewStd::Show

bool Show(**wxDebugReport&** *dbgrpt*) **const**

Show the dialog, see *wxDebugReportPreview::Show()* (p. 403) for more information.

wxDebugReportUpload

This class is used to upload a compressed file using HTTP POST request. As this class derives from wxDebugReportCompress, before upload the report is compressed in a single .ZIP file.

Derived from

wxDebugReportCompress (p. 402)

Include files

<wx/debugrpt.h>

wxDebugReportUpload::wxDebugReportUpload

 wxDebugReportUpload(**const wxString&** *url*, **const wxString&** *input*, **const wxString&** *action*, **const wxString&** *curl = _T("curl")*)

This class will upload the compressed file created by its base class to an HTML multipart/form-data form at the specified address. The *url* is the upload page address, *input* is the name of the "`type=file`" control on the form used for the file name and *action* is the value of the form action field. The report is uploaded using `curl` program which should be available, the *curl* parameter may be used to specify the full path to it.

wxDebugReportUpload::OnServerReply

bool OnServerReply(**const wxArrayString&** *WXUNUSED(reply)*)

This function may be overridden in a derived class to show the output from curl: this may be an HTML page or anything else that the server returned. Value returned by this function becomes the return value of *wxDebugReport::Process()* (p. 402).

wxDelegateRendererNative

wxDelegateRendererNative allows reuse of renderers code by forwarding all the *wxRendererNative* (p. 1122) methods to the given object and thus allowing you to only modify some of its methods -- without having to reimplement all of them.

Note that the "normal", inheritance-based approach, doesn't work with the renderers as it is impossible to derive from a class unknown at compile-time and the renderer is only chosen at run-time. So suppose that you want to only add something to the drawing of the tree control buttons but leave all the other methods unchanged -- the only way to do it, considering that the renderer class which you want to customize might not even be written yet when you write your code (it could be written later and loaded from a DLL during run-time), is by using this class.

Except for the constructor, it has exactly the same methods as *wxRendererNative* (p. 1122) and their implementation is trivial: they are simply forwarded to the real renderer. Note that the "real" renderer may, in turn, be a wxDelegateRendererNative as well and that there may be arbitrarily many levels like this -- but at the end of the chain there must be a real renderer which does the drawing.

Derived from

wxRendererNative (p. 1122)

Include files

<wx/renderer.h>

wxDelegateRendererNative::wxDelegateRendererNative

wxDelegateRendererNative()

wxDelegateRendererNative(**wxRendererNative&** *rendererNative*)

The default constructor does the same thing as the other one except that it uses the *generic renderer* (p. 1125) instead of the user-specified *rendererNative*.

In any case, this sets up the delegate renderer object to follow all calls to the specified real renderer.

Note that this object does *not* take ownership of (i.e. won't delete)*rendererNative*.

wxDelegateRendererNative::DrawXXX

DrawXXX(*...*)

This class also provides all the virtual methods of *wxRendererNative* (p. 1122), please refer to that class documentation for the details.

wxDialog

A dialog box is a window with a title bar and sometimes a system menu, which can be moved around the screen. It can contain controls and other windows and is usually used to allow the user to make some choice or to answer a question.

Derived from

wxTopLevelWindow (p. 1419)
wxWindow (p. 1490)
wxEvtHandler (p. 486)
wxObject (p. 1012)

Include files

<wx/dialog.h>

Remarks

There are two kinds of dialog -- *modal* and *modeless*. A modal dialog blocks program flow and user input on other windows until it is dismissed, whereas a modeless dialog behaves more like a frame in that program flow continues, and input in other windows is still possible. To show a modal dialog you should use the *ShowModal* (p. 415) method while to show a dialog modelessly you simply use *Show* (p. 415), just as with frames.

Note that the modal dialog is one of the very few examples of wxWindow-derived objects which may be created on the stack and not on the heap. In other words, although this

code snippet:

```
void AskUser()
{
    MyAskDialog *dlg = new MyAskDialog(...);
    if ( dlg->ShowModal() == wxID_OK )
        ...
    //else: dialog was cancelled or some another button pressed

    dlg->Destroy();
}
```

works, you can also achieve the same result by using a simpler code fragment below:

```
void AskUser()
{
    MyAskDialog dlg(...);
    if ( dlg.ShowModal() == wxID_OK )
        ...

    // no need to call Destroy() here
}
```

An application can define a *wxCloseEvent* (p. 153) handler for the dialog to respond to system close events.

Window styles

wxCAPTION   Puts a caption on the dialog box.

wxDEFAULT_DIALOG_STYLE   Equivalent to a combination of wxCAPTION, wxCLOSE_BOX and wxSYSTEM_MENU (the last one is not used under Unix)

wxRESIZE_BORDER   Display a resizeable frame around the window.

wxSYSTEM_MENU   Display a system menu.

wxCLOSE_BOX   Displays a close box on the frame.

wxMAXIMIZE_BOX   Displays a maximize box on the dialog.

wxMINIMIZE_BOX   Displays a minimize box on the dialog.

wxTHICK_FRAME   Display a thick frame around the window.

wxSTAY_ON_TOP   The dialog stays on top of all other windows (Windows only).

wxNO_3D   Under Windows, specifies that the child controls should not have 3D borders unless specified in the control.

wxDIALOG_NO_PARENT   By default, a dialog created with a NULL parent window will be given the *application's top level window* (p. 34) as parent. Use this style to prevent this from happening and create an orphan dialog. This is not recommended for modal dialogs.

wxDIALOG_EX_CONTEXTHELP   Under Windows, puts a query button on the caption. When pressed, Windows will go into a context-sensitive help mode and wxWidgets will send a wxEVT_HELP event if the user clicked on an application window. *Note* that this is an extended style and must be set by calling *SetExtraStyle* (p. 1533) before Create is called (two-step construction).

wxDIALOG_EX_METAL   On Mac OS X, frames with this style will be shown with a metallic look. This is an *extra* style.

Under Unix or Linux, MWM (the Motif Window Manager) or other window managers recognizing the MHM hints should be running for any of these styles to have an effect.

See also *Generic window styles* (p. 1762).

See also

wxDialog overview (p. 1765), *wxFrame* (p. 579), *Validator overview* (p. 1765)

wxDialog::wxDialog

wxDialog()

Default constructor.

wxDialog(**wxWindow*** *parent*, **wxWindowID** *id*, **const wxString&** *title*, **const wxPoint&** *pos* = *wxDefaultPosition*, **const wxSize&** *size* = *wxDefaultSize*, **long** *style* = *wxDEFAULT_DIALOG_STYLE*, **const wxString&** *name* = *"dialogBox"*)

Constructor.

Parameters

parent

Can be NULL, a frame or another dialog box.

id

An identifier for the dialog. A value of -1 is taken to mean a default.

title

The title of the dialog.

pos

The dialog position. A value of (-1, -1) indicates a default position, chosen by either the windowing system or wxWidgets, depending on platform.

size

The dialog size. A value of (-1, -1) indicates a default size, chosen by either the windowing system or wxWidgets, depending on platform.

style

The window style. See *wxDialog* (p. 406).

name

Used to associate a name with the window, allowing the application user to set Motif resource values for individual dialog boxes.

See also

wxDialog::Create (p. 409)

wxDialog::~wxDialog

~wxDialog()

Destructor. Deletes any child windows before deleting the physical window.

wxDialog::Centre

void Centre(**int** *direction = wxBOTH*)

Centres the dialog box on the display.

Parameters

direction

May be `wxHORIZONTAL`, `wxVERTICAL` or `wxBOTH`.

wxDialog::Create

bool Create(**wxWindow*** *parent*, **wxWindowID** *id*, **const wxString&** *title*, **const wxPoint&** *pos = wxDefaultPosition*, **const wxSize&** *size = wxDefaultSize*, **long** *style = wxDEFAULT_DIALOG_STYLE*, **const wxString&** *name = "dialogBox"*)

Used for two-step dialog box construction. See *wxDialog::wxDialog* (p. 408) for details.

wxDialog::CreateButtonSizer

wxSizer* CreateButtonSizer(**long** *flags*)

Creates a sizer with standard buttons. *flags* is a bit list of the following flags: wxOK, wxCANCEL, wxYES, wxNO, wxHELP, wxNO_DEFAULT.

The sizer lays out the buttons in a manner appropriate to the platform.

This function simply calls *CreateStdDialogButtonSizer* (p. 410).

wxDialog::CreateStdDialogButtonSizer

wxStdDialogButtonSizer* CreateStdDialogButtonSizer(**long** *flags*)

Creates a *wxStdDialogButtonSizer* (p. 1272) with standard buttons. *flags* is a bit list of the following flags: wxOK, wxCANCEL, wxYES, wxNO, wxHELP, wxNO_DEFAULT.

The sizer lays out the buttons in a manner appropriate to the platform.

wxDialog::DoOK

virtual bool DoOK()

This function is called when the titlebar OK button is pressed (PocketPC only). A command event for the identifier returned by GetAffirmativeId is sent by default. You can override this function. If the function returns false, wxWidgets will call Close() for the dialog.

wxDialog::EndModal

void EndModal(**int** *retCode*)

Ends a modal dialog, passing a value to be returned from the *wxDialog::ShowModal* (p. 415) invocation.

Parameters

retCode

> The value that should be returned by **ShowModal**.

See also

wxDialog::ShowModal (p. 415), *wxDialog::GetReturnCode* (p. 411),
wxDialog::SetReturnCode (p. 414)

wxDialog::GetAffirmativeId

int GetAffirmativeId() **const**

Gets the identifier to be used when the user presses an OK button in a PocketPC titlebar.

See also

wxDialog::SetAffirmativeId (p. 413)

wxDialog::GetReturnCode

int GetReturnCode()

Gets the return code for this window.

Remarks

A return code is normally associated with a modal dialog, where *wxDialog::ShowModal* (p. 415) returns a code to the application.

See also

wxDialog::SetReturnCode (p. 414), *wxDialog::ShowModal* (p. 415), *wxDialog::EndModal* (p. 410)

wxDialog::GetTitle

wxString GetTitle() **const**

Returns the title of the dialog box.

wxDialog::GetToolBar

wxToolBar* GetToolBar() **const**

On PocketPC, a dialog is automatically provided with an empty toolbar. GetToolBar allows you to access the toolbar and add tools to it. Removing tools and adding arbitrary controls are not currently supported.

This function is not available on any other platform.

wxDialog::Iconize

void Iconize(**const bool** *iconize*)

Iconizes or restores the dialog. Windows only.

Parameters

iconize

If true, iconizes the dialog box; if false, shows and restores it.

Remarks

Note that in Windows, iconization has no effect since dialog boxes cannot be iconized.

However, applications may need to explicitly restore dialog boxes under Motif which have user-iconizable frames, and under Windows calling `Iconize(false)` will bring the window to the front, as does `Show(true)`.

wxDialog::IsIconized

bool IsIconized() const

Returns true if the dialog box is iconized. Windows only.

Remarks

Always returns false under Windows since dialogs cannot be iconized.

wxDialog::IsModal

bool IsModal() const

Returns true if the dialog box is modal, false otherwise.

wxDialog::OnApply

void OnApply(wxCommandEvent& *event*)

The default handler for the wxID_APPLY identifier.

Remarks

This function calls *wxWindow::Validate* (p. 1547) and *wxWindow::TransferDataToWindow* (p. 1545).

See also

wxDialog::OnOK (p. 413), *wxDialog::OnCancel* (p. 412)

wxDialog::OnCancel

void OnCancel(wxCommandEvent& *event*)

The default handler for the wxID_CANCEL identifier.

Remarks

The function either calls **EndModal(wxID_CANCEL)** if the dialog is modal, or sets the return value to wxID_CANCEL and calls **Show(false)** if the dialog is modeless.

See also

wxDialog::OnOK (p. 413), *wxDialog::OnApply* (p. 412)

wxDialog::OnOK

void OnOK(**wxCommandEvent&** *event*)

The default handler for the wxID_OK identifier.

Remarks

The function calls *wxWindow::Validate* (p. 1547), then *wxWindow::TransferDataFromWindow* (p. 1545). If this returns true, the function either calls **EndModal(wxID_OK)** if the dialog is modal, or sets the return value to wxID_OK and calls **Show(false)** if the dialog is modeless.

See also

wxDialog::OnCancel (p. 412), *wxDialog::OnApply* (p. 412)

wxDialog::OnSysColourChanged

void OnSysColourChanged(**wxSysColourChangedEvent&** *event*)

The default handler for wxEVT_SYS_COLOUR_CHANGED.

Parameters

event

The colour change event.

Remarks

Changes the dialog's colour to conform to the current settings (Windows only). Add an event table entry for your dialog class if you wish the behaviour to be different (such as keeping a user-defined background colour). If you do override this function, call wxEvent::Skip to propagate the notification to child windows and controls.

See also

wxSysColourChangedEvent (p. 1314)

wxDialog::SetAffirmativeId

void SetAffirmativeId(**int** *id*)

Sets the identifier to be used when the user presses an OK button in a PocketPC titlebar. By default, this is wxID_OK.

See also

wxDialog::GetAffirmativeId (p. 410)

wxDialog::SetIcon

void SetIcon(**const wxIcon&** *icon*)

Sets the icon for this dialog.

Parameters

icon

> The icon to associate with this dialog.

See also *wxIcon* (p. 766).

wxDialog::SetIcons

void SetIcons(**const wxIconBundle&** *icons*)

Sets the icons for this dialog.

Parameters

icons

> The icons to associate with this dialog.

See also *wxIconBundle* (p. 774).

wxDialog::SetModal

void SetModal(**const bool** *flag*)

NB: This function is deprecated and doesn't work for all ports, just use *ShowModal* (p. 415) to show a modal dialog instead.

Allows the programmer to specify whether the dialog box is modal (wxDialog::Show blocks control until the dialog is hidden) or modeless (control returns immediately).

Parameters

flag

> If true, the dialog will be modal, otherwise it will be modeless.

wxDialog::SetReturnCode

void SetReturnCode(**int** *retCode*)

Sets the return code for this window.

Parameters

retCode

>   The integer return code, usually a control identifier.

Remarks

A return code is normally associated with a modal dialog, where *wxDialog::ShowModal* (p. 415) returns a code to the application. The function *wxDialog::EndModal* (p. 410) calls **SetReturnCode**.

See also

wxDialog::GetReturnCode (p. 411), *wxDialog::ShowModal* (p. 415), *wxDialog::EndModal* (p. 410)

wxDialog::SetTitle

void SetTitle(**const wxString&** *title*)

Sets the title of the dialog box.

Parameters

title

>   The dialog box title.

wxDialog::Show

bool Show(**const bool** *show*)

Hides or shows the dialog.

Parameters

show

>   If true, the dialog box is shown and brought to the front; otherwise the box is hidden. If false and the dialog is modal, control is returned to the calling program.

Remarks

The preferred way of dismissing a modal dialog is to use *wxDialog::EndModal* (p. 410).

wxDialog::ShowModal

int ShowModal()

Shows a modal dialog. Program flow does not return until the dialog has been dismissed with *wxDialog::EndModal* (p. 410).

Return value

The return value is the value set with *wxDialog::SetReturnCode* (p. 414).

See also

wxDialog::EndModal (p. 410), *wxDialog:GetReturnCode* (p. 411), *wxDialog::SetReturnCode* (p. 414)

wxDialUpEvent

This is the event class for the dialup events sent by *wxDialUpManager* (p. 416).

Derived from

wxEvent (p. 483)
wxObject (p. 1012)

Include files

<wx/dialup.h>

wxDialUpEvent::wxDialUpEvent

 wxDialUpEvent(**bool** *isConnected*, **bool** *isOwnEvent*)

Constructor is only used by *wxDialUpManager* (p. 416).

wxDialUpEvent::IsConnectedEvent

bool IsConnectedEvent() **const**

Is this a `CONNECTED` or `DISCONNECTED` event? In other words, does it notify about transition from offline to online state or vice versa?

wxDialUpEvent::IsOwnEvent

bool IsOwnEvent() **const**

Does this event come from wxDialUpManager::Dial() or from some extrenal process (i.e. does it result from our own attempt to establish the connection)?

wxDialUpManager

This class encapsulates functions dealing with verifying the connection status of the

workstation (connected to the Internet via a direct connection, connected through a modem or not connected at all) and to establish this connection if possible/required (i.e. in the case of the modem).

The program may also wish to be notified about the change in the connection status (for example, to perform some action when the user connects to the network the next time or, on the contrary, to stop receiving data from the net when the user hangs up the modem). For this, you need to use one of the event macros described below.

This class is different from other wxWidgets classes in that there is at most one instance of this class in the program accessed via *wxDialUpManager::Create()* (p. 417) and you can't create the objects of this class directly.

Derived from

No base class

Include files

<wx/dialup.h>

Event table macros

To be notified about the change in the network connection status, use these event handler macros to direct input to member functions that take a *wxDialUpEvent* (p. 416) argument.

EVT_DIALUP_CONNECTED(func)   A connection with the network was established.

EVT_DIALUP_DISCONNECTED(func) The connection with the network was lost.

See also
dialup sample (p. 1709)
wxDialUpEvent (p. 416)

wxDialUpManager::Create

wxDialUpManager* Create()

This function should create and return the object of the platform-specific class derived from wxDialUpManager. You should delete the pointer when you are done with it.

wxDialUpManager::IsOk

bool IsOk() const

Returns `true` if the dialup manager was initialized correctly. If this function returns `false`, no other functions will work neither, so it is a good idea to call this function and check its result before calling any other wxDialUpManager methods

wxDialUpManager::~wxDialUpManager

~wxDialUpManager()

Destructor.

wxDialUpManager::GetISPNames

size_t GetISPNames(**wxArrayString&** *names*) **const**

This function is only implemented under Windows.

Fills the array with the names of all possible values for the first parameter to *Dial()* (p. 418) on this machine and returns their number (may be 0).

wxDialUpManager::Dial

bool Dial(**const wxString&** *nameOfISP* = *wxEmptyString*, **const wxString&** *username* = *wxEmptyString*, **const wxString&** *password* = *wxEmptyString*, **bool** *async* = *true*)

Dial the given ISP, use *username* and *password* to authenticate.

The parameters are only used under Windows currently, for Unix you should use *SetConnectCommand* (p. 420) to customize this functions behaviour.

If no *nameOfISP* is given, the function will select the default one (proposing the user to choose among all connections defined on this machine) and if no username and/or password are given, the function will try to do without them, but will ask the user if really needed.

If *async* parameter is `false`, the function waits until the end of dialing and returns `true` upon successful completion.

If *async* is `true`, the function only initiates the connection and returns immediately - the result is reported via events (an event is sent anyhow, but if dialing failed it will be a DISCONNECTED one).

wxDialUpManager::IsDialing

bool IsDialing() **const**

Returns true if (async) dialing is in progress.

See also

Dial (p. 418)

wxDialUpManager::CancelDialing

bool CancelDialing()

Cancel dialing the number initiated with *Dial* (p. 418) with async parameter equal to `true`.

Note that this won't result in DISCONNECTED event being sent.

See also

IsDialing (p. 418)

wxDialUpManager::HangUp

bool HangUp()

Hang up the currently active dial up connection.

wxDialUpManager::IsAlwaysOnline

bool IsAlwaysOnline() **const**

Returns `true` if the computer has a permanent network connection (i.e. is on a LAN) and so there is no need to use Dial() function to go online.

NB: this functions tries to guess the result and it is not always guaranteed to be correct, so it is better to ask user for confirmation or give him a possibility to override it.

wxDialUpManager::IsOnline

bool IsOnline() **const**

Returns `true` if the computer is connected to the network: under Windows, this just means that a RAS connection exists, under Unix we check that the "well-known host" (as specified by *SetWellKnownHost* (p. 420)) is reachable.

wxDialUpManager::SetOnlineStatus

void SetOnlineStatus(**bool** *isOnline = true*)

Sometimes the built-in logic for determining the online status may fail, so, in general, the user should be allowed to override it. This function allows to forcefully set the online status - whatever our internal algorithm may think about it.

See also

IsOnline (p. 419)

wxDialUpManager::EnableAutoCheckOnlineStatus

bool EnableAutoCheckOnlineStatus(**size_t** *nSeconds = 60*)

Enable automatic checks for the connection status and sending of
`wxEVT_DIALUP_CONNECTED`/`wxEVT_DIALUP_DISCONNECTED` events. The interval
parameter is only for Unix where we do the check manually and specifies how often
should we repeat the check (each minute by default). Under Windows, the notification
about the change of connection status is sent by the system and so we don't do any
polling and this parameter is ignored.

Returns `false` if couldn't set up automatic check for online status.

wxDialUpManager::DisableAutoCheckOnlineStatus

void DisableAutoCheckOnlineStatus()

Disable automatic check for connection status change - notice that
the`wxEVT_DIALUP_XXX` events won't be sent any more neither.

wxDialUpManager::SetWellKnownHost

void SetWellKnownHost(**const wxString&** *hostname*, **int** *portno = 80*)

This method is for Unix only.

Under Unix, the value of well-known host is used to check whether we're connected to
the internet. It is unused under Windows, but this function is always safe to call. The
default value is `www.yahoo.com:80`.

wxDialUpManager::SetConnectCommand

void SetConnectCommand(**const wxString&** *commandDial = wxT("/usr/bin/pon")*,
const wxString& *commandHangup = wxT("/usr/bin/poff")*)

This method is for Unix only.

Sets the commands to start up the network and to hang up again.

See also

Dial (p. 418)

wxDir

wxDir is a portable equivalent of Unix open/read/closedir functions which allow
enumerating of the files in a directory. wxDir allows enumerate files as well as
directories.

wxDir also provides a flexible way to enumerate files recursively using *Traverse* (p. 423) or a simpler *GetAllFiles* (p. 422) function.

Example of use:

```
wxDir dir(wxGetCwd());

if ( !dir.IsOpened() )
{
    // deal with the error here - wxDir would already log an error
message
    // explaining the exact reason of the failure
    return;
}

puts("Enumerating object files in current directory:");

wxString filename;

bool cont = dir.GetFirst(&filename, filespec, flags);
while ( cont )
{
    printf("%s\n", filename.c_str());

    cont = dir.GetNext(&filename);
}
```

Derived from

No base class

Constants

These flags define what kind of filename is included in the list of files enumerated by GetFirst/GetNext.

```
enum
{
    wxDIR_FILES     = 0x0001,       // include files
    wxDIR_DIRS      = 0x0002,       // include directories
    wxDIR_HIDDEN    = 0x0004,       // include hidden files
    wxDIR_DOTDOT    = 0x0008,       // include '.' and '..'

    // by default, enumerate everything except '.' and '..'
    wxDIR_DEFAULT   = wxDIR_FILES | wxDIR_DIRS | wxDIR_HIDDEN
}
```

Include files

<wx/dir.h>

wxDir::wxDir

wxDir()

Default constructor, use *Open()* (p. 423) afterwards.

 wxDir(**const wxString&** *dir*)

Opens the directory for enumeration, use *IsOpened()* (p. 423) to test for errors.

wxDir::~wxDir

 ~wxDir()

Destructor cleans up the associated resources. It is not virtual and so this class is not meant to be used polymorphically.

wxDir::Exists

static bool Exists(**const wxString&** *dir*)

Test for existence of a directory with the given name

wxDir::GetAllFiles

static size_t GetAllFiles(**const wxString&** *dirname*, **wxArrayString** **files*, **const wxString&** *filespec* = *wxEmptyString*, **int** *flags* = *wxDIR_DEFAULT*)

The function appends the names of all the files under directory *dirname* to the array *files* (note that its old content is preserved). Only files matching the *filespec* are taken, with empty spec matching all the files.

The *flags* parameter should always include `wxDIR_FILES` or the array would be unchanged and should include `wxDIR_DIRS` flag to recurse into subdirectories (both flags are included in the value by default).

See also: *Traverse* (p. 423)

wxDir::GetFirst

bool GetFirst(**wxString*** *filename*, **const wxString&** *filespec* = *wxEmptyString*, **int** *flags* = *wxDIR_DEFAULT*) **const**

Start enumerating all files matching *filespec* (or all files if it is empty) and flags, return true on success.

wxDir::GetName

wxString GetName() **const**

Returns the name of the directory itself. The returned string does not have the trailing

path separator (slash or backslash).

wxDir::GetNext

bool GetNext(**wxString*** *filename*) **const**

Continue enumerating files satisfying the criteria specified by the last call to *GetFirst* (p. 422).

wxDir::HasFiles

bool HasFiles(**const wxString&** *filespec = wxEmptyString*)

Returns `true` if the directory contains any files matching the given *filespec*. If *filespec* is empty, look for any files at all. In any case, even hidden files are taken into account.

wxDir::HasSubDirs

bool HasSubDirs(**const wxString&** *dirspec = wxEmptyString*)

Returns `true` if the directory contains any subdirectories (if a non empty *filespec* is given, only check for directories matching it). The hidden subdirectories are taken into account as well.

wxDir::IsOpened

bool IsOpened() **const**

Returns true if the directory was successfully opened by a previous call to *Open* (p. 423).

wxDir::Open

bool Open(**const wxString&** *dir*)

Open the directory for enumerating, returns true on success or false if an error occurred.

wxDir::Traverse

size_t Traverse(**wxDirTraverser&** *sink*, **const wxString&** *filespec = wxEmptyString*, **int** *flags = wxDIR_DEFAULT*)

Enumerate all files and directories under the given directory recursively calling the element of the provided *wxDirTraverser* (p. 426) object for each of them.

More precisely, the function will really recurse into subdirectories if *flags* contains `wxDIR_DIRS` flag. It will ignore the files (but still possibly recurse into subdirectories) if

`wxDIR_FILES` flag is given.

For each found directory, *sink.OnDir()* (p. 427) is called and *sink.OnFile()* (p. 427) is called for every file. Depending on the return value, the enumeration may continue or stop.

The function returns the total number of files found or `(size_t)-1` on error.

See also: *GetAllFiles* (p. 422)

wxDirDialog

This class represents the directory chooser dialog.

Derived from

wxDialog (p. 406)
wxWindow (p. 1490)
wxEvtHandler (p. 486)
wxObject (p. 1012)

Include files

<wx/dirdlg.h> <wx/generic/dirdlgg.h>

Window styles

wxDD_NEW_DIR_BUTTON	Add "Create new directory" button and allow directory names to be editable. On Windows the new directory button is only available with recent versions of the common dialogs.

See also *Generic window styles* (p. 1762).

See also

wxDirDialog overview (p. 1801), *wxFileDialog* (p. 511)

wxDirDialog::wxDirDialog

 wxDirDialog(**wxWindow*** *parent*, **const wxString&** *message = "Choose a directory"*, **const wxString&** *defaultPath = ""*, **long** *style = 0*, **const wxPoint&** *pos = wxDefaultPosition*, **const wxSize&** *size = wxDefaultSize*, **const wxString&** *name = "wxDirCtrl"*)

Constructor. Use *wxDirDialog::ShowModal* (p. 426) to show the dialog.

Parameters

parent

> Parent window.

message

> Message to show on the dialog.

defaultPath

> The default path, or the empty string.

style

> A dialog style, currently unused.

pos

> Dialog position. Ignored under Windows.

size

> Dialog size. Ignored under Windows.

name

> The dialog name, not used.

wxDirDialog::~wxDirDialog

~wxDirDialog()

Destructor.

wxDirDialog::GetPath

wxString GetPath() **const**

Returns the default or user-selected path.

wxDirDialog::GetMessage

wxString GetMessage() **const**

Returns the message that will be displayed on the dialog.

wxDirDialog::GetStyle

long GetStyle() **const**

Returns the dialog style.

wxDirDialog::SetMessage

void SetMessage(**const wxString&** *message*)

Sets the message that will be displayed on the dialog.

wxDirDialog::SetPath

void SetPath(**const wxString&** *path*)

Sets the default path.

wxDirDialog::SetStyle

void SetStyle(**long** *style*)

Sets the dialog style. This is currently unused.

wxDirDialog::ShowModal

int ShowModal()

Shows the dialog, returning wxID_OK if the user pressed OK, and wxID_CANCEL otherwise.

wxDirTraverser

wxDirTraverser is an abstract interface which must be implemented by objects passed to *Traverse* (p. 423) function.

Example of use (this works almost like *GetAllFiles* (p. 422)):

```
class wxDirTraverserSimple : public wxDirTraverser
{
public:
    wxDirTraverserSimple(wxArrayString& files) : m_files(files) { }

    virtual wxDirTraverseResult OnFile(const wxString& filename)
    {
        m_files.Add(filename);
        return wxDIR_CONTINUE;
    }

    virtual wxDirTraverseResult OnDir(const wxString&
WXUNUSED(dirname))
    {
        return wxDIR_CONTINUE;
```

```
        }

    private:
        wxArrayString& m_files;
    };

    // get the names of all files in the array
    wxArrayString files;
    wxDirTraverserSimple traverser(files);

    wxDir dir(dirname);
    dir.Traverse(traverser);
```

Derived from

No base class

Constants

The elements of `wxDirTraverseResult` are the possible return values of the callback functions:

```
enum wxDirTraverseResult
{
    wxDIR_IGNORE = -1,      // ignore this directory but continue with
others
    wxDIR_STOP,             // stop traversing
    wxDIR_CONTINUE          // continue into this directory
};
```

Include files

<wx/dir.h>

wxDirTraverser::OnDir

virtual wxDirTraverseResult OnDir(const wxString& *dirname*)

This function is called for each directory. It may return `wxSIR_STOP` to abort traversing completely, `wxDIR_IGNORE` to skip this directory but continue with others or `wxDIR_CONTINUE` to enumerate all files and subdirectories in this directory.

This is a pure virtual function and must be implemented in the derived class.

wxDirTraverser::OnFile

virtual wxDirTraverseResult OnFile(const wxString& *filename*)

This function is called for each file. It may return `wxDIR_STOP` to abort traversing (for example, if the file being searched is found) or `wxDIR_CONTINUE` to proceed.

This is a pure virtual function and must be implemented in the derived class.

wxOpenErrorTraverser::OnOpenError

virtual wxOpenErrorTraverseResult OnOpenError(**const wxString&** *openerrorname*)

This function is called for each directory which we failed to open for enumerating. It may return `wxSIR_STOP` to abort traversing completely,`wxDIR_IGNORE` to skip this directory but continue with others or `wxDIR_CONTINUE` to retry opening this directory once again.

The base class version always returns `wxDIR_IGNORE`.

wxDisplay

Determines the sizes and locations of displays connected to the system.

Derived from

None

Include files

<wx/display.h>

See also

wxClientDisplayRect (p. 1623), *wxDisplaySize* (p. 1624), *wxDisplaySizeMM* (p. 1624)

wxDisplay::wxDisplay

 wxDisplay(**size_t** *index = 0*)

Constructor, setting up a wxDisplay instance with the specified display.

Parameters

index

> The index of the display to use. This must be non-negative and lower than the value returned by *GetCount()* (p. 429).

wxDisplay::~wxDisplay

void ~wxDisplay()

Destructor.

wxDisplay::ChangeMode

bool ChangeMode(**const wxVideoMode&** *mode* = *wxDefaultVideoMode*)

Changes the video mode of this display to the mode specified in the mode parameter.

If wxDefaultVideoMode is passed in as the mode parameter, the defined behaviour is that wxDisplay will reset the video mode to the default mode used by the display. On Windows, the behavior is normal. However, there are differences on other platforms. On Unix variations using X11 extensions it should behave as defined, but some irregularities may occur.

On wxMac passing in wxDefaultVideoMode as the mode parameter does nothing. This happens because carbon no longer has access to DMUseScreenPrefs, an undocumented function that changed the video mode to the system default by using the system's 'scrn' resource.

wxDisplay::GetCount

static size_t GetCount()

Returns the number of connected displays.

wxDisplay::GetCurrentMode

wxVideoMode GetCurrentMode() **const**

Returns the current video mode that this display is in.

wxDisplay::GetDepth

int GetDepth() **const**

Returns the bit depth of the display whose index was passed to the constructor.

wxDisplay::GetFromPoint

static int GetFromPoint(**const wxPoint&** *pt*)

Returns the index of the display on which the given point lies. Returns -1 if the point is not on any connected display.

Parameters

pt

> The point to locate.

wxDisplay::GetFromWindow

static int GetFromWindow(**wxWindow*** *win*)

Returns the index of the display on which the given window lies.

If the window is on more than one display it gets the display that overlaps the window the most.

Returns -1 if the window is not on any connected display.

Currently wxMSW only.

Parameters

win

The window to locate.

wxDisplay::GetGeometry

wxRect GetGeometry() **const**

Returns the bounding rectangle of the display whose index was passed to the constructor.

wxDisplay::GetModes

wxArrayVideoModes GetModes(**const wxVideoMode&** *mode* = *wxDefaultVideoMode*) **const**

Fills and returns an array with all the video modes that are supported by this display, or video modes that are supported by this display and match the mode parameter (if mode is not wxDefaultVideoMode).

wxDisplay::GetName

wxString GetName() **const**

Returns the display's name. A name is not available on all platforms.

wxDisplay::IsPrimary

bool IsPrimary()

Returns true if the display is the primary display. The primary display is the one whose index is 0.

wxDllLoader

Deprecation note: This class is deprecated since version 2.4 and is not compiled in by default in version 2.6 and will be removed in 2.8. Please use *wxDynamicLibrary* (p. 474) instead.

wxDllLoader is a class providing an interface similar to Unix's `dlopen()`. It is used by the wxLibrary framework and manages the actual loading of shared libraries and the resolving of symbols in them. There are no instances of this class, it simply serves as a namespace for its static member functions.

Please note that class *wxDynamicLibrary* (p. 474) provides alternative, friendlier interface to wxDllLoader.

The terms *DLL* and *shared library/object* will both be used in the documentation to refer to the same thing: a `.dll` file under Windows or `.so` or `.sl` one under Unix.

Example of using this class to dynamically load the `strlen()` function:

```
#if defined(__WXMSW__)
    static const wxChar *LIB_NAME = _T("kernel32");
    static const wxChar *FUNC_NAME = _T("lstrlenA");
#elif defined(__UNIX__)
    static const wxChar *LIB_NAME = _T("/lib/libc-2.0.7.so");
    static const wxChar *FUNC_NAME = _T("strlen");
#endif

    wxDllType dllHandle = wxDllLoader::LoadLibrary(LIB_NAME);
    if ( !dllHandle )
    {
        ... error ...
    }
    else
    {
        typedef int (*strlenType)(char *);
        strlenType pfnStrlen =
(strlenType)wxDllLoader::GetSymbol(dllHandle, FUNC_NAME);
        if ( !pfnStrlen )
        {
            ... error ...
        }
        else
        {
            if ( pfnStrlen("foo") != 3 )
            {
                ... error ...
            }
            else
            {
                ... ok! ...
            }
        }

        wxDllLoader::UnloadLibrary(dllHandle);
    }
```

Derived from

No base class

Include files

<wx/dynlib.h>

Data structures

This header defines a platform-dependent `wxDllType` typedef which stores a handle to a loaded DLLs on the given platform.

wxDllLoader::GetDllExt

static wxString GetDllExt()

Returns the string containing the usual extension for shared libraries for the given systems (including the leading dot if not empty).

For example, this function will return "`.dll`" under Windows or (usually) "`.so`" under Unix.

wxDllLoader::GetProgramHandle

wxDllType GetProgramHandle()

This function returns a valid handle for the main program itself. Notice that the `NULL` return value is valid for some systems (i.e. doesn't mean that the function failed).

NB: This function is Unix specific. It will always fail under Windows or OS/2.

wxDllLoader::GetSymbol

void * GetSymbol(**wxDllType** *dllHandle*, **const wxString&** *name*)

This function resolves a symbol in a loaded DLL, such as a variable or function name.

Returned value will be `NULL` if the symbol was not found in the DLL or if an error occurred.

Parameters

dllHandle

> Valid handle previously returned by *LoadLibrary* (p. 433)

name

> Name of the symbol.

wxDllLoader::LoadLibrary

wxDllType LoadLibrary(**const wxString &** *libname*, **bool*** *success = NULL*)

This function loads a shared library into memory, with *libname* being the name of the library: it may be either the full name including path and (platform-dependent) extension, just the basename (no path and no extension) or a basename with extension. In the last two cases, the library will be searched in all standard locations.

Returns a handle to the loaded DLL. Use *success* parameter to test if it is valid. If the handle is valid, the library must be unloaded later with *UnloadLibrary* (p. 433).

Parameters

libname

Name of the shared object to load.

success

May point to a bool variable which will be set to true or false; may also be `NULL`.

wxDllLoader::UnloadLibrary

void UnloadLibrary(**wxDllType** *dllhandle*)

This function unloads the shared library. The handle *dllhandle* must have been returned by *LoadLibrary* (p. 433) previously.

wxDocChildFrame

The wxDocChildFrame class provides a default frame for displaying documents on separate windows. This class can only be used for SDI (not MDI) child frames.

The class is part of the document/view framework supported by wxWidgets, and cooperates with the *wxView* (p. 1473), *wxDocument* (p. 454), *wxDocManager* (p. 435) and *wxDocTemplate* (p. 448) classes.

See the example application in `samples/docview`.

Derived from

wxFrame (p. 579)
wxWindow (p. 1490)
wxEvtHandler (p. 486)
wxObject (p. 1012)

Include files

<wx/docview.h>

Document/view overview (p. 1802), *wxFrame* (p. 579)

wxDocChildFrame::m_childDocument

wxDocument* m_childDocument

The document associated with the frame.

wxDocChildFrame::m_childView

wxView* m_childView

The view associated with the frame.

wxDocChildFrame::wxDocChildFrame

 wxDocChildFrame(**wxDocument*** *doc*, **wxView*** *view*, **wxFrame*** *parent*, **wxWindowID** *id*, **const wxString&** *title*, **const wxPoint&** *pos = wxDefaultPosition*, **const wxSize&** *size = wxDefaultSize*, **long** *style = wxDEFAULT_FRAME_STYLE*, **const wxString&** *name = "frame"*)

Constructor.

wxDocChildFrame::~wxDocChildFrame

 ~wxDocChildFrame()

Destructor.

wxDocChildFrame::GetDocument

wxDocument* GetDocument() **const**

Returns the document associated with this frame.

wxDocChildFrame::GetView

wxView* GetView() **const**

Returns the view associated with this frame.

wxDocChildFrame::OnActivate

void OnActivate(**wxActivateEvent** *event*)

Sets the currently active view to be the frame's view. You may need to override (but still call) this function in order to set the keyboard focus for your subwindow.

wxDocChildFrame::OnCloseWindow

void OnCloseWindow(**wxCloseEvent&** *event*)

Closes and deletes the current view and document.

wxDocChildFrame::SetDocument

void SetDocument(**wxDocument *****doc*)

Sets the document for this frame.

wxDocChildFrame::SetView

void SetView(**wxView *****view*)

Sets the view for this frame.

wxDocManager

The wxDocManager class is part of the document/view framework supported by wxWidgets, and cooperates with the *wxView* (p. 1473), *wxDocument* (p. 454) and *wxDocTemplate* (p. 448) classes.

Derived from

wxEvtHandler (p. 486)
wxObject (p. 1012)

Include files

<wx/docview.h>

See also

wxDocManager overview (p. 1805), *wxDocument* (p. 454), *wxView* (p. 1473), *wxDocTemplate* (p. 448), *wxFileHistory* (p. 517)

wxDocManager::m_currentView

wxView* m_currentView

The currently active view.

wxDocManager::m_defaultDocumentNameCounter

int m_defaultDocumentNameCounter

Stores the integer to be used for the next default document name.

wxDocManager::m_fileHistory

wxFileHistory* m_fileHistory

A pointer to an instance of *wxFileHistory* (p. 517), which manages the history of recently-visited files on the File menu.

wxDocManager::m_maxDocsOpen

int m_maxDocsOpen

Stores the maximum number of documents that can be opened before existing documents are closed. By default, this is 10,000.

wxDocManager::m_docs

wxList m_docs

A list of all documents.

wxDocManager::m_flags

long m_flags

Stores the flags passed to the constructor.

wxDocManager::m_lastDirectory

The directory last selected by the user when opening a file.

wxFileHistory* m_fileHistory

wxDocManager::m_templates

wxList m_templates

A list of all document templates.

wxDocManager::wxDocManager

 wxDocManager(**long** *flags* = *wxDEFAULT_DOCMAN_FLAGS*, **bool** *initialize* = *true*)

Constructor. Create a document manager instance dynamically near the start of your application before doing any document or view operations.

flags is currently unused.

If *initialize* is true, the *Initialize* (p. 441) function will be called to create a default history list object. If you derive from wxDocManager, you may wish to call the base constructor with false, and then call Initialize in your own constructor, to allow your own Initialize or OnCreateFileHistory functions to be called.

wxDocManager::~wxDocManager

void ~wxDocManager()

Destructor.

wxDocManager::ActivateView

void ActivateView(**wxView*** *doc*, **bool** *activate* = *true*)

Sets the current view.

wxDocManager::AddDocument

void AddDocument(**wxDocument** **doc*)

Adds the document to the list of documents.

wxDocManager::AddFileToHistory

void AddFileToHistory(**const wxString&** *filename*)

Adds a file to the file history list, if we have a pointer to an appropriate file menu.

wxDocManager::AssociateTemplate

void AssociateTemplate(**wxDocTemplate** **temp*)

Adds the template to the document manager's template list.

wxDocManager::CloseDocuments

bool CloseDocuments(**bool** *force* = *true*)

Closes all currently opened documents.

wxDocManager::CreateDocument

wxDocument* CreateDocument(**const wxString&** *path*, **long** *flags*)

Creates a new document in a manner determined by the *flags* parameter, which can be:

- wxDOC_NEW Creates a fresh document.

- wxDOC_SILENT Silently loads the given document file.

If wxDOC_NEW is present, a new document will be created and returned, possibly after asking the user for a template to use if there is more than one document template. If wxDOC_SILENT is present, a new document will be created and the given file loaded into it. If neither of these flags is present, the user will be presented with a file selector for the file to load, and the template to use will be determined by the extension (Windows) or by popping up a template choice list (other platforms).

If the maximum number of documents has been reached, this function will delete the oldest currently loaded document before creating a new one.

wxDocManager::CreateView

wxView* CreateView(**wxDocument****doc*, **long** *flags*)

Creates a new view for the given document. If more than one view is allowed for the document (by virtue of multiple templates mentioning the same document type), a choice of view is presented to the user.

wxDocManager::DisassociateTemplate

void DisassociateTemplate(**wxDocTemplate ****temp*)

Removes the template from the list of templates.

wxDocManager::FileHistoryAddFilesToMenu

void FileHistoryAddFilesToMenu()

Appends the files in the history list, to all menus managed by the file history object.

void FileHistoryAddFilesToMenu(**wxMenu*** *menu*)

Appends the files in the history list, to the given menu only.

wxDocManager::FileHistoryLoad

void FileHistoryLoad(**wxConfigBase&** *config*)

Loads the file history from a config object.

See also

wxConfig (p. 193)

wxDocManager::FileHistoryRemoveMenu

void FileHistoryRemoveMenu(**wxMenu*** *menu*)

Removes the given menu from the list of menus managed by the file history object.

wxDocManager::FileHistorySave

void FileHistorySave(**wxConfigBase&** *resourceFile*)

Saves the file history into a config object. This must be called explicitly by the application.

See also

wxConfig (p. 193)

wxDocManager::FileHistoryUseMenu

void FileHistoryUseMenu(**wxMenu*** *menu*)

Use this menu for appending recently-visited document filenames, for convenient access. Calling this function with a valid menu pointer enables the history list functionality.

Note that you can add multiple menus using this function, to be managed by the file history object.

wxDocManager::FindTemplateForPath

wxDocTemplate * FindTemplateForPath(**const wxString&** *path*)

Given a path, try to find template that matches the extension. This is only an approximate method of finding a template for creating a document.

wxDocManager::GetCurrentDocument

wxDocument * GetCurrentDocument()

Returns the document associated with the currently active view (if any).

wxDocManager::GetCurrentView

wxView * GetCurrentView()

Returns the currently active view

wxDocManager::GetDocuments

wxList& GetDocuments()

Returns a reference to the list of documents.

wxDocManager::GetFileHistory

wxFileHistory * GetFileHistory()

Returns a pointer to file history.

wxDocManager::GetLastDirectory

wxString GetLastDirectory() **const**

Returns the directory last selected by the user when opening a file. Initially empty.

wxDocManager::GetMaxDocsOpen

int GetMaxDocsOpen()

Returns the number of documents that can be open simultaneously.

wxDocManager::GetHistoryFilesCount

size_t GetHistoryFilesCount()

Returns the number of files currently stored in the file history.

wxDocManager::GetTemplates

wxList& GetTemplates()

Returns a reference to the list of associated templates.

wxDocManager::Initialize

bool Initialize()

Initializes data; currently just calls OnCreateFileHistory. Some data cannot always be initialized in the constructor because the programmer must be given the opportunity to override functionality. If OnCreateFileHistory was called from the constructor, an overridden virtual OnCreateFileHistory would not be called due to C++'s 'interesting' constructor semantics. In fact Initialize *is* called from the wxDocManager constructor, but this can be vetoed by passing false to the second argument, allowing the derived class's constructor to call Initialize, possibly calling a different OnCreateFileHistory from the default.

The bottom line: if you're not deriving from Initialize, forget it and construct wxDocManager with no arguments.

wxDocManager::MakeDefaultName

bool MakeDefaultName(**const wxString&** *buf*)

Copies a suitable default name into *buf*. This is implemented by appending an integer counter to the string **unnamed** and incrementing the counter.

wxPerl note: In wxPerl this function must return the modified name rather than just modifying the argument.

wxDocManager::OnCreateFileHistory

wxFileHistory * OnCreateFileHistory()

A hook to allow a derived class to create a different type of file history. Called from *Initialize* (p. 441).

wxDocManager::OnFileClose

void OnFileClose(**wxCommandEvent&** *event*)

Closes and deletes the currently active document.

wxDocManager::OnFileCloseAll

void OnFileCloseAll(**wxCommandEvent&** *event*)

Closes and deletes all the currently opened documents.

wxDocManager::OnFileNew

void OnFileNew(**wxCommandEvent&** *event*)

Creates a document from a list of templates (if more than one template).

wxDocManager::OnFileOpen

void OnFileOpen(**wxCommandEvent&** *event*)

Creates a new document and reads in the selected file.

wxDocManager::OnFileRevert

void OnFileRevert(**wxCommandEvent&** *event*)

Reverts the current document by calling wxDocument::Revert for the current document.

wxDocManager::OnFileSave

void OnFileSave(**wxCommandEvent&** *event*)

Saves the current document by calling wxDocument::Save for the current document.

wxDocManager::OnFileSaveAs

void OnFileSaveAs(**wxCommandEvent&** *event*)

Calls wxDocument::SaveAs for the current document.

wxDocManager::RemoveDocument

void RemoveDocument(**wxDocument** **doc*)

Removes the document from the list of documents.

wxDocManager::SelectDocumentPath

wxDocTemplate * SelectDocumentPath(**wxDocTemplate *****templates*, **int** *noTemplates*, **wxString&** *path*, **long** *flags*, **bool** *save*)

Under Windows, pops up a file selector with a list of filters corresponding to document templates. The wxDocTemplate corresponding to the selected file's extension is returned.

On other platforms, if there is more than one document template a choice list is popped up, followed by a file selector.

This function is used in wxDocManager::CreateDocument.

wxPerl note: In wxPerl `templates` is a reference to a list of templates. If you override this method in your document manager it must return two values, eg:

  (doctemplate, path) = My::DocManager->SelectDocumentPath(...);

wxDocManager::SelectDocumentType

wxDocTemplate * SelectDocumentType(**wxDocTemplate *****templates**, **int** *noTemplates*, **bool** *sort=false*)

Returns a document template by asking the user (if there is more than one template). This function is used in wxDocManager::CreateDocument.

Parameters

templates

>   Pointer to an array of templates from which to choose a desired template.

noTemplates

>   Number of templates being pointed to by the *templates* pointer.

sort

>   If more than one template is passed in in *templates*, then this parameter indicates whether the list of templates that the user will have to choose from is sorted or not when shown the choice box dialog.   Default is false.

wxPerl note: In wxPerl `templates` is a reference to a list of templates.

wxDocManager::SelectViewType

wxDocTemplate * SelectViewType(**wxDocTemplate *****templates**, **int** *noTemplates*, **bool** *sort=false*)

Returns a document template by asking the user (if there is more than one template), displaying a list of valid views. This function is used in wxDocManager::CreateView. The dialog normally will not appear because the array of templates only contains those relevant to the document in question, and often there will only be one such.

Parameters

templates

>   Pointer to an array of templates from which to choose a desired template.

noTemplates

>   Number of templates being pointed to by the *templates* pointer.

sort

> If more than one template is passed in in *templates*, then this parameter indicates whether the list of templates that the user will have to choose from is sorted or not when shown the choice box dialog. Default is false.

wxPerl note: In wxPerl `templates` is a reference to a list of templates.

wxDocManager::SetLastDirectory

void SetLastDirectory(**const wxString&** *dir*)

Sets the directory to be displayed to the user when opening a file. Initially this is empty.

wxDocManager::SetMaxDocsOpen

void SetMaxDocsOpen(**int** *n*)

Sets the maximum number of documents that can be open at a time. By default, this is 10,000. If you set it to 1, existing documents will be saved and deleted when the user tries to open or create a new one (similar to the behaviour of Windows Write, for example). Allowing multiple documents gives behaviour more akin to MS Word and other Multiple Document Interface applications.

wxDocMDIChildFrame

The wxDocMDIChildFrame class provides a default frame for displaying documents on separate windows. This class can only be used for MDI child frames.

The class is part of the document/view framework supported by wxWidgets, and cooperates with the *wxView* (p. 1473), *wxDocument* (p. 454), *wxDocManager* (p. 435) and *wxDocTemplate* (p. 448) classes.

See the example application in `samples/docview`.

Derived from

wxMDIChildFrame (p. 916)
wxFrame (p. 579)
wxWindow (p. 1490)
wxEvtHandler (p. 486)
wxObject (p. 1012)

Include files

<wx/docmdi.h>

See also

Document/view overview (p. 1802), *wxMDIChildFrame* (p. 916)

wxDocMDIChildFrame::m_childDocument

wxDocument* m_childDocument

The document associated with the frame.

wxDocMDIChildFrame::m_childView

wxView* m_childView

The view associated with the frame.

wxDocMDIChildFrame::wxDocMDIChildFrame

wxDocMDIChildFrame(**wxDocument*** *doc*, **wxView*** *view*, **wxFrame*** *parent*, **wxWindowID** *id*, **const wxString&** *title*, **const wxPoint&** *pos* = *wxDefaultPosition*, **const wxSize&** *size* = *wxDefaultSize*, **long** *style* = *wxDEFAULT_FRAME_STYLE*, **const wxString&** *name* = *"frame"*)

Constructor.

wxDocMDIChildFrame::~wxDocMDIChildFrame

~wxDocMDIChildFrame()

Destructor.

wxDocMDIChildFrame::GetDocument

wxDocument* GetDocument() **const**

Returns the document associated with this frame.

wxDocMDIChildFrame::GetView

wxView* GetView() **const**

Returns the view associated with this frame.

wxDocMDIChildFrame::OnActivate

void OnActivate(**wxActivateEvent** *event*)

Sets the currently active view to be the frame's view. You may need to override (but still call) this function in order to set the keyboard focus for your subwindow.

wxDocMDIChildFrame::OnCloseWindow

void OnCloseWindow(**wxCloseEvent&** *event*)

Closes and deletes the current view and document.

wxDocMDIChildFrame::SetDocument

void SetDocument(**wxDocument ****doc*)

Sets the document for this frame.

wxDocMDIChildFrame::SetView

void SetView(**wxView ****view*)

Sets the view for this frame.

wxDocMDIParentFrame

The wxDocMDIParentFrame class provides a default top-level frame for applications using the document/view framework. This class can only be used for MDI parent frames.

It cooperates with the *wxView* (p. 1473), *wxDocument* (p. 454), *wxDocManager* (p. 435) and *wxDocTemplates* (p. 448) classes.

See the example application in `samples/docview`.

Derived from

wxMDIParentFrame (p. 920)
wxFrame (p. 579)
wxWindow (p. 1490)
wxEvtHandler (p. 486)
wxObject (p. 1012)

Include files

<wx/docmdi.h>

See also

Document/view overview (p. 1802), *wxMDIParentFrame* (p. 920)

wxDocMDIParentFrame::wxDocMDIParentFrame

wxDocMDIParentFrame(**wxDocManager*** *manager*, **wxFrame ****parent*, **wxWindowID**

id, **const wxString&** *title*, **const wxPoint&** *pos* = *wxDefaultPosition*, **const wxSize&** *size* = *wxDefaultSize*, **long** *style* = *wxDEFAULT_FRAME_STYLE*, **const wxString&** *name* = *"frame"*)

Constructor.

wxDocMDIParentFrame::~wxDocMDIParentFrame

 ~wxDocMDIParentFrame()

Destructor.

wxDocMDIParentFrame::OnCloseWindow

void OnCloseWindow(**wxCloseEvent&** *event*)

Deletes all views and documents. If no user input cancelled the operation, the frame will be destroyed and the application will exit.

Since understanding how document/view clean-up takes place can be difficult, the implementation of this function is shown below.

```
void wxDocParentFrame::OnCloseWindow(wxCloseEvent& event)
{
  if (m_docManager->Clear(!event.CanVeto()))
  {
    this->Destroy();
  }
  else
    event.Veto();
}
```

wxDocParentFrame

The wxDocParentFrame class provides a default top-level frame for applications using the document/view framework. This class can only be used for SDI (not MDI) parent frames.

It cooperates with the *wxView* (p. 1473), *wxDocument* (p. 454), *wxDocManager* (p. 435) and *wxDocTemplates* (p. 448) classes.

See the example application in `samples/docview`.

Derived from

wxFrame (p. 579)
wxWindow (p. 1490)
wxEvtHandler (p. 486)
wxObject (p. 1012)

Include files

<wx/docview.h>

See also

Document/view overview (p. 1802), *wxFrame* (p. 579)

wxDocParentFrame::wxDocParentFrame

wxDocParentFrame(**wxDocManager*** *manager*, **wxFrame ****parent*, **wxWindowID** *id*, **const wxString&** *title*, **const wxPoint&** *pos* = *wxDefaultPosition*, **const wxSize&** *size* = *wxDefaultSize*, **long** *style* = *wxDEFAULT_FRAME_STYLE*, **const wxString&** *name* = *"frame"*)

Constructor.

wxDocParentFrame::~wxDocParentFrame

~wxDocParentFrame()

Destructor.

wxDocParentFrame::OnCloseWindow

void OnCloseWindow(**wxCloseEvent&** *event*)

Deletes all views and documents. If no user input cancelled the operation, the frame will be destroyed and the application will exit.

Since understanding how document/view clean-up takes place can be difficult, the implementation of this function is shown below.

```
void wxDocParentFrame::OnCloseWindow(wxCloseEvent& event)
{
  if (m_docManager->Clear(!event.CanVeto()))
  {
    this->Destroy();
  }
  else
    event.Veto();
}
```

wxDocTemplate

The wxDocTemplate class is used to model the relationship between a document class and a view class.

Derived from

wxObject (p. 1012)

Include files

<wx/docview.h>

See also

wxDocTemplate overview (p. 1804), *wxDocument* (p. 454), *wxView* (p. 1473)

wxDocTemplate::m_defaultExt

wxString m_defaultExt

The default extension for files of this type.

wxDocTemplate::m_description

wxString m_description

A short description of this template.

wxDocTemplate::m_directory

wxString m_directory

The default directory for files of this type.

wxDocTemplate::m_docClassInfo

wxClassInfo* m_docClassInfo

Run-time class information that allows document instances to be constructed dynamically.

wxDocTemplate::m_docTypeName

wxString m_docTypeName

The named type of the document associated with this template.

wxDocTemplate::m_documentManager

wxDocTemplate* m_documentManager

A pointer to the document manager for which this template was created.

wxDocTemplate::m_fileFilter

wxString m_fileFilter

The file filter (such as `*.txt`) to be used in file selector dialogs.

wxDocTemplate::m_flags

long m_flags

The flags passed to the constructor.

wxDocTemplate::m_viewClassInfo

wxClassInfo* m_viewClassInfo

Run-time class information that allows view instances to be constructed dynamically.

wxDocTemplate::m_viewTypeName

wxString m_viewTypeName

The named type of the view associated with this template.

wxDocTemplate::wxDocTemplate

 wxDocTemplate(**wxDocManager*** *manager*, **const wxString&** *descr*, **const wxString&** *filter*, **const wxString&** *dir*, **const wxString&** *ext*, **const wxString&** *docTypeName*, **const wxString&** *viewTypeName*, **wxClassInfo*** *docClassInfo = NULL*, **wxClassInfo*** *viewClassInfo = NULL*, **long** *flags = wxDEFAULT_TEMPLATE_FLAGS*)

Constructor. Create instances dynamically near the start of your application after creating a wxDocManager instance, and before doing any document or view operations.

manager is the document manager object which manages this template.

descr is a short description of what the template is for. This string will be displayed in the file filter list of Windows file selectors.

filter is an appropriate file filter such as `*.txt`.

dir is the default directory to use for file selectors.

ext is the default file extension (such as txt).

docTypeName is a name that should be unique for a given type of document, used for gathering a list of views relevant to a particular document.

viewTypeName is a name that should be unique for a given view.

docClassInfo is a pointer to the run-time document class information as returned by the CLASSINFO macro, e.g. CLASSINFO(MyDocumentClass). If this is not supplied, you will need to derive a new wxDocTemplate class and override the CreateDocument member to return a new document instance on demand.

viewClassInfo is a pointer to the run-time view class information as returned by the CLASSINFO macro, e.g. CLASSINFO(MyViewClass). If this is not supplied, you will need to derive a new wxDocTemplate class and override the CreateView member to return a new view instance on demand.

flags is a bit list of the following:

- wxTEMPLATE_VISIBLE The template may be displayed to the user in dialogs.

- wxTEMPLATE_INVISIBLE The template may not be displayed to the user in dialogs.

- wxDEFAULT_TEMPLATE_FLAGS Defined as wxTEMPLATE_VISIBLE.

wxPerl note: In wxPerl `docClassInfo` and `viewClassInfo` can be either `Wx::ClassInfo` objects or strings which contain the name of the perl packages which are to be used as `Wx::Document` and`Wx::View` classes (they must have a constructor named `new`):

Wx::DocTemplate->new(docmgr, descr, filter, dir, ext, docTypeName, viewTypeName, docClassInfo, viewClassInfo, flags) will construct document and view objects from the class information

Wx::DocTemplate->new(docmgr, descr, filter, dir, ext, docTypeName, viewTypeName, docClassName, viewClassName, flags) will construct document and view objects from perl packages

Wx::DocTemplate->new(docmgr, descr, filter, dir, ext, docTypeName, viewTypeName)
`Wx::DocTemplate::CreateDocument(`
`)` and`Wx::DocTemplate::CreateView()`
must be overridden

wxDocTemplate::~wxDocTemplate

void ~wxDocTemplate()

Destructor.

wxDocTemplate::CreateDocument

wxDocument * CreateDocument(**const wxString&** *path*, **long** *flags = 0*)

Creates a new instance of the associated document class. If you have not supplied a wxClassInfo parameter to the template constructor, you will need to override this function to return an appropriate document instance.

This function calls wxDocTemplate::InitDocument which in turns calls wxDocument::OnCreate.

wxDocTemplate::CreateView

wxView * CreateView(**wxDocument ***doc*, **long** *flags = 0*)

Creates a new instance of the associated view class. If you have not supplied a wxClassInfo parameter to the template constructor, you will need to override this function to return an appropriate view instance.

wxDocTemplate::GetDefaultExtension

wxString GetDefaultExtension()

Returns the default file extension for the document data, as passed to the document template constructor.

wxDocTemplate::GetDescription

wxString GetDescription()

Returns the text description of this template, as passed to the document template constructor.

wxDocTemplate::GetDirectory

wxString GetDirectory()

Returns the default directory, as passed to the document template constructor.

wxDocTemplate::GetDocumentManager

wxDocManager * GetDocumentManager()

Returns a pointer to the document manager instance for which this template was created.

wxDocTemplate::GetDocumentName

wxString GetDocumentName()

Returns the document type name, as passed to the document template constructor.

wxDocTemplate::GetFileFilter

wxString GetFileFilter()

Returns the file filter, as passed to the document template constructor.

wxDocTemplate::GetFlags

long GetFlags()

Returns the flags, as passed to the document template constructor.

wxDocTemplate::GetViewName

wxString GetViewName()

Returns the view type name, as passed to the document template constructor.

wxDocTemplate::InitDocument

bool InitDocument(**wxDocument*** *doc*, **const wxString&** *path*, **long** *flags = 0*)

Initialises the document, calling wxDocument::OnCreate. This is called from
wxDocTemplate::CreateDocument.

wxDocTemplate::IsVisible

bool IsVisible()

Returns true if the document template can be shown in user dialogs, false otherwise.

wxDocTemplate::SetDefaultExtension

void SetDefaultExtension(**const wxString&** *ext*)

Sets the default file extension.

wxDocTemplate::SetDescription

void SetDescription(**const wxString&** *descr*)

Sets the template description.

wxDocTemplate::SetDirectory

void SetDirectory(**const wxString&** *dir*)

Sets the default directory.

wxDocTemplate::SetDocumentManager

void SetDocumentManager(**wxDocManager** **manager*)

Sets the pointer to the document manager instance for which this template was created. Should not be called by the application.

wxDocTemplate::SetFileFilter

void SetFileFilter(**const wxString&** *filter*)

Sets the file filter.

wxDocTemplate::SetFlags

void SetFlags(**long** *flags*)

Sets the internal document template flags (see the constructor description for more details).

wxDocument

The document class can be used to model an application's file-based data. It is part of the document/view framework supported by wxWidgets, and cooperates with the *wxView* (p. 1473), *wxDocTemplate* (p. 448) and *wxDocManager* (p. 435) classes.

Derived from

wxEvtHandler (p. 486)
wxObject (p. 1012)

Include files

<wx/docview.h>

See also

wxDocument overview (p. 1803), *wxView* (p. 1473), *wxDocTemplate* (p. 448), *wxDocManager* (p. 435)

wxDocument::m_commandProcessor

wxCommandProcessor* m_commandProcessor

A pointer to the command processor associated with this document.

wxDocument::m_documentFile

wxString m_documentFile

Filename associated with this document ("" if none).

wxDocument::m_documentModified

bool m_documentModified

true if the document has been modified, false otherwise.

wxDocument::m_documentTemplate

wxDocTemplate * m_documentTemplate

A pointer to the template from which this document was created.

wxDocument::m_documentTitle

wxString m_documentTitle

Document title. The document title is used for an associated frame (if any), and is usually constructed by the framework from the filename.

wxDocument::m_documentTypeName

wxString m_documentTypeName

The document type name given to the wxDocTemplate constructor, copied to this variable when the document is created. If several document templates are created that use the same document type, this variable is used in wxDocManager::CreateView to collate a list of alternative view types that can be used on this kind of document. Do not change the value of this variable.

wxDocument::m_documentViews

wxList m_documentViews

List of wxView instances associated with this document.

wxDocument::wxDocument

wxDocument()

Constructor. Define your own default constructor to initialize application-specific data.

wxDocument::~wxDocument

~wxDocument()

Destructor. Removes itself from the document manager.

wxDocument::AddView

virtual bool AddView(wxView *view)

If the view is not already in the list of views, adds the view and calls OnChangedViewList.

wxDocument::Close

virtual bool Close()

Closes the document, by calling OnSaveModified and then (if this returned true) OnCloseDocument. This does not normally delete the document object: use DeleteAllViews to do this implicitly.

wxDocument::DeleteAllViews

virtual bool DeleteAllViews()

Calls wxView::Close and deletes each view. Deleting the final view will implicitly delete the document itself, because the wxView destructor calls RemoveView. This in turns calls wxDocument::OnChangedViewList, whose default implemention is to save and delete the document if no views exist.

wxDocument::GetCommandProcessor

wxCommandProcessor* GetCommandProcessor() const

Returns a pointer to the command processor associated with this document.

See *wxCommandProcessor* (p. 185).

wxDocument::GetDocumentTemplate

wxDocTemplate* GetDocumentTemplate() **const**

Gets a pointer to the template that created the document.

wxDocument::GetDocumentManager

wxDocManager* GetDocumentManager() **const**

Gets a pointer to the associated document manager.

wxDocument::GetDocumentName

wxString GetDocumentName() **const**

Gets the document type name for this document. See the comment for *documentTypeName* (p. 455).

wxDocument::GetDocumentWindow

wxWindow* GetDocumentWindow() **const**

Intended to return a suitable window for using as a parent for document-related dialog boxes. By default, uses the frame associated with the first view.

wxDocument::GetFilename

wxString GetFilename() **const**

Gets the filename associated with this document, or "" if none is associated.

wxDocument::GetFirstView

wxView * GetFirstView() **const**

A convenience function to get the first view for a document, because in many cases a document will only have a single view.

See also: *GetViews* (p. 458)

wxDocument::GetPrintableName

virtual void GetPrintableName(**wxString&** *name*) **const**

Copies a suitable document name into the supplied *name* buffer. The default function uses the title, or if there is no title, uses the filename; or if no filename, the string **unnamed**.

wxPerl note: In wxPerl this function must return the modified name rather than just modifying the argument.

wxDocument::GetTitle

wxString GetTitle() **const**

Gets the title for this document. The document title is used for an associated frame (if any), and is usually constructed by the framework from the filename.

wxDocument::GetViews

wxList & GetViews() **const**

Returns the list whose elements are the views on the document.

See also: *GetFirstView* (p. 457)

wxDocument::IsModified

virtual bool IsModified() **const**

Returns true if the document has been modified since the last save, false otherwise. You may need to override this if your document view maintains its own record of being modified (for example if using wxTextWindow to view and edit the document).

See also *Modify* (p. 459).

wxDocument::LoadObject

virtual istream& LoadObject(**istream&** *stream*)

virtual wxInputStream& LoadObject(**wxInputStream&** *stream*)

Override this function and call it from your own LoadObject before streaming your own data. LoadObject is called by the framework automatically when the document contents need to be loaded.

Note that only one of these forms exists, depending on how wxWidgets was configured.

wxDocument::Modify

virtual void Modify(**bool** *modify*)

Call with true to mark the document as modified since the last save, false otherwise. You may need to override this if your document view maintains its own record of being modified (for example if using wxTextWindow to view and edit the document).

See also *IsModified* (p. 458).

wxDocument::OnChangedViewList

virtual void OnChangedViewList()

Called when a view is added to or deleted from this document. The default implementation saves and deletes the document if no views exist (the last one has just been removed).

wxDocument::OnCloseDocument

virtual bool OnCloseDocument()

The default implementation calls DeleteContents (an empty implementation) sets the modified flag to false. Override this to supply additional behaviour when the document is closed with Close.

wxDocument::OnCreate

virtual bool OnCreate(**const wxString&** *path*, **long** *flags*)

Called just after the document object is created to give it a chance to initialize itself. The default implementation uses the template associated with the document to create an initial view. If this function returns false, the document is deleted.

wxDocument::OnCreateCommandProcessor

virtual wxCommandProcessor* OnCreateCommandProcessor()

Override this function if you want a different (or no) command processor to be created when the document is created. By default, it returns an instance of wxCommandProcessor.

See *wxCommandProcessor* (p. 185).

wxDocument::OnNewDocument

virtual bool OnNewDocument()

The default implementation calls OnSaveModified and DeleteContents, makes a default title for the document, and notifies the views that the filename (in fact, the title) has changed.

wxDocument::OnOpenDocument

virtual bool OnOpenDocument(const wxString& *filename*)

Constructs an input file stream for the given filename (which must not be empty), and calls LoadObject. If LoadObject returns true, the document is set to unmodified; otherwise, an error message box is displayed. The document's views are notified that the filename has changed, to give windows an opportunity to update their titles. All of the document's views are then updated.

wxDocument::OnSaveDocument

virtual bool OnSaveDocument(const wxString& *filename*)

Constructs an output file stream for the given filename (which must not be empty), and calls SaveObject. If SaveObject returns true, the document is set to unmodified; otherwise, an error message box is displayed.

wxDocument::OnSaveModified

virtual bool OnSaveModified()

If the document has been modified, prompts the user to ask if the changes should be changed. If the user replies Yes, the Save function is called. If No, the document is marked as unmodified and the function succeeds. If Cancel, the function fails.

wxDocument::RemoveView

virtual bool RemoveView(wxView* *view*)

Removes the view from the document's list of views, and calls OnChangedViewList.

wxDocument::Save

virtual bool Save()

Saves the document by calling OnSaveDocument if there is an associated filename, or SaveAs if there is no filename.

wxDocument::SaveAs

virtual bool SaveAs()

Prompts the user for a file to save to, and then calls OnSaveDocument.

wxDocument::SaveObject

virtual ostream& SaveObject(**ostream&** *stream*)

virtual wxOutputStream& SaveObject(**wxOutputStream&** *stream*)

Override this function and call it from your own SaveObject before streaming your own data. SaveObject is called by the framework automatically when the document contents need to be saved.

Note that only one of these forms exists, depending on how wxWidgets was configured.

wxDocument::SetCommandProcessor

virtual void SetCommandProcessor(**wxCommandProcessor** **processor*)

Sets the command processor to be used for this document. The document will then be responsible for its deletion. Normally you should not call this; override OnCreateCommandProcessor instead.

See *wxCommandProcessor* (p. 185).

wxDocument::SetDocumentName

void SetDocumentName(**const wxString&** *name*)

Sets the document type name for this document. See the comment for *documentTypeName* (p. 455).

wxDocument::SetDocumentTemplate

void SetDocumentTemplate(**wxDocTemplate*** *templ*)

Sets the pointer to the template that created the document. Should only be called by the framework.

wxDocument::SetFilename

void SetFilename(**const wxString&** *filename*, **bool** *notifyViews = false*)

Sets the filename for this document. Usually called by the framework.

If *notifyViews* is true, wxView::OnChangeFilename is called for all views.

wxDocument::SetTitle

void SetTitle(const wxString& *title*)

Sets the title for this document. The document title is used for an associated frame (if any), and is usually constructed by the framework from the filename.

wxDocument::UpdateAllViews

void UpdateAllViews(wxView* *sender = NULL*, wxObject* *hint = NULL*)

Updates all views. If *sender* is non-NULL, does not update this view.

hint represents optional information to allow a view to optimize its update.

wxDragImage

This class is used when you wish to drag an object on the screen, and a simple cursor is not enough.

On Windows, the WIN32 API is used to do achieve smooth dragging. On other platforms, wxGenericDragImage is used. Applications may also prefer to use wxGenericDragImage on Windows, too.

wxPython note: wxPython uses wxGenericDragImage on all platforms, but uses the wxDragImage name.

To use this class, when you wish to start dragging an image, create a wxDragImage object and store it somewhere you can access it as the drag progresses. Call BeginDrag to start, and EndDrag to stop the drag. To move the image, initially call Show and then Move. If you wish to update the screen contents during the drag (for example, highlight an item as in the dragimag sample), first call Hide, update the screen, call Move, and then call Show.

You can drag within one window, or you can use full-screen dragging either across the whole screen, or just restricted to one area of the screen to save resources. If you want the user to drag between two windows, then you will need to use full-screen dragging.

If you wish to draw the image yourself, use wxGenericDragImage and override *wxDragImage::DoDrawImage* (p. 465) and *wxDragImage::GetImageRect* (p. 465).

Please see `samples/dragimag` for an example.

Derived from

wxObject (p. 1012)

Include files

<wx/dragimag.h>
<wx/generic/dragimgg.h>

wxDragImage::wxDragImage

wxDragImage()

Default constructor.

wxDragImage(**const wxBitmap&** *image*, **const wxCursor&** *cursor = wxNullCursor*, **const wxPoint&** *cursorHotspot = wxPoint(0, 0)*)

Constructs a drag image from a bitmap and optional cursor.

wxDragImage(**const wxIcon&** *image*, **const wxCursor&** *cursor = wxNullCursor*, **const wxPoint&** *cursorHotspot = wxPoint(0, 0)*)

Constructs a drag image from an icon and optional cursor.

wxPython note: This constructor is called wxDragIcon in wxPython.

wxDragImage(**const wxString&** *text*, **const wxCursor&** *cursor = wxNullCursor*, **const wxPoint&** *cursorHotspot = wxPoint(0, 0)*)

Constructs a drag image from a text string and optional cursor.

wxPython note: This constructor is called wxDragString in wxPython.

wxDragImage(**const wxTreeCtrl&** *treeCtrl*, **wxTreeItemId&** *id*)

Constructs a drag image from the text in the given tree control item, and optional cursor.

wxPython note: This constructor is called wxDragTreeItem in wxPython.

wxDragImage(**const wxListCtrl&** *treeCtrl*, **long** *id*)

Constructs a drag image from the text in the given tree control item, and optional cursor.

wxPython note: This constructor is called wxDragListItem in wxPython.

wxDragImage(**const wxCursor&** *cursor = wxNullCursor*, **const wxPoint&** *cursorHotspot = wxPoint(0, 0)*)

Constructs a drag image an optional cursor. This constructor is only available for wxGenericDragImage, and can be used when the application supplies *wxDragImage::DoDrawImage* (p. 465) and *wxDragImage::GetImageRect* (p. 465).

Parameters

image

> Icon or bitmap to be used as the drag image. The bitmap can have a mask.

text

Text used to construct a drag image.

cursor

Optional cursor to combine with the image.

hotspot

This parameter is deprecated.

treeCtrl

Tree control for constructing a tree drag image.

listCtrl

List control for constructing a list drag image.

id

Tree or list control item id.

wxDragImage::BeginDrag

bool BeginDrag(**const wxPoint&** *hotspot*, **wxWindow*** *window*, **bool** *fullScreen* = *false*, **wxRect*** *rect* = *NULL*)

Start dragging the image, in a window or full screen.

bool BeginDrag(**const wxPoint&** *hotspot*, **wxWindow*** *window*, **wxWindow*** *boundingWindow*)

Start dragging the image, using the first window to capture the mouse and the second to specify the bounding area. This form is equivalent to using the first form, but more convenient than working out the bounding rectangle explicitly.

You need to then call *wxDragImage::Show* (p. 466) and *wxDragImage::Move* (p. 466) to show the image on the screen.

Call *wxDragImage::EndDrag* (p. 465) when the drag has finished.

Note that this call automatically calls CaptureMouse.

Parameters

hotspot

The location of the drag position relative to the upper-left corner of the image.

window

The window that captures the mouse, and within which the dragging is limited unless *fullScreen* is true.

boundingWindow

> In the second form of the function, specifies the area within which the drag occurs.

fullScreen

> If true, specifies that the drag will be visible over the full screen, or over as much of the screen as is specified by *rect*. Note that the mouse will still be captured in *window*.

rect

> If non-NULL, specifies the rectangle (in screen coordinates) that bounds the dragging operation. Specifying this can make the operation more efficient by cutting down on the area under consideration, and it can also make a visual difference since the drag is clipped to this area.

wxDragImage::DoDrawImage

virtual bool DoDrawImage(**wxDC&** *dc*, **const wxPoint&** *pos*)

Draws the image on the device context with top-left corner at the given position.

This function is only available with wxGenericDragImage, to allow applications to draw their own image instead of using an actual bitmap. If you override this function, you must also override *wxDragImage::GetImageRect* (p. 465).

wxDragImage::EndDrag

bool EndDrag()

Call this when the drag has finished.

Note that this call automatically calls ReleaseMouse.

wxDragImage::GetImageRect

virtual wxRect GetImageRect(**const wxPoint&** *pos*) **const**

Returns the rectangle enclosing the image, assuming that the image is drawn with its top-left corner at the given point.

This function is available in wxGenericDragImage only, and may be overridden (together with *wxDragImage::DoDrawImage* (p. 465)) to provide a virtual drawing capability.

wxDragImage::Hide

bool Hide()

Hides the image. You may wish to call this before updating the window contents (perhaps highlighting an item). Then call *wxDragImage::Move* (p. 466) and *wxDragImage::Show* (p. 466).

wxDragImage::Move

bool Move(**const wxPoint&** *pt*)

Call this to move the image to a new position. The image will only be shown if *wxDragImage::Show* (p. 466) has been called previously (for example at the start of the drag).

pt is the position in client coordinates (relative to the window specified in BeginDrag).

You can move the image either when the image is hidden or shown, but in general dragging will be smoother if you move the image when it is shown.

wxDragImage::Show

bool Show()

Shows the image. Call this at least once when dragging.

wxDragImage::UpdateBackingFromWindow

bool UpdateBackingFromWindow(**wxDC&** *windowDC*, **wxMemoryDC&** *destDC*, **const wxRect&** *sourceRect*, **const wxRect&** *destRect*) **const**

Override this if you wish to draw the window contents to the backing bitmap yourself. This can be desirable if you wish to avoid flicker by not having to redraw the updated window itself just before dragging, which can cause a flicker just as the drag starts. Instead, paint the drag image's backing bitmap to show the appropriate graphic *minus the objects to be dragged*, and leave the window itself to be updated by the drag image. This can provide eerily smooth, flicker-free drag behaviour.

The default implementation copies the window contents to the backing bitmap. A new implementation will normally copy information from another source, such as from its own backing bitmap if it has one, or directly from internal data structures.

This function is available in wxGenericDragImage only.

wxDropFilesEvent

This class is used for drop files events, that is, when files have been dropped onto the window. This functionality is currently only available under Windows. The window must have previously been enabled for dropping by calling *wxWindow::DragAcceptFiles* (p. 1500).

Important note: this is a separate implementation to the more general drag and drop implementation documented *here* (p. 1817). It uses the older, Windows message-based approach of dropping files.

Derived from

wxEvent (p. 483)
wxObject (p. 1012)

Include files

<wx/event.h>

Event table macros

To process a drop files event, use these event handler macros to direct input to a member function that takes a wxDropFilesEvent argument.

EVT_DROP_FILES(func)   Process a wxEVT_DROP_FILES event.

See also
Event handling overview (p. 1751)

wxDropFilesEvent::wxDropFilesEvent

wxDropFilesEvent(**WXTYPE** *id = 0*, **int** *noFiles = 0*, **wxString*** *files = NULL*)

Constructor.

wxDropFilesEvent::m_files

wxString* m_files

An array of filenames.

wxDropFilesEvent::m_noFiles

int m_noFiles

The number of files dropped.

wxDropFilesEvent::m_pos

wxPoint m_pos

The point at which the drop took place.

wxDropFilesEvent::GetFiles

wxString* GetFiles() **const**

Returns an array of filenames.

wxDropFilesEvent::GetNumberOfFiles

int GetNumberOfFiles() **const**

Returns the number of files dropped.

wxDropFilesEvent::GetPosition

wxPoint GetPosition() **const**

Returns the position at which the files were dropped.

Returns an array of filenames.

wxDropSource

This class represents a source for a drag and drop operation.

See *Drag and drop overview* (p. 1817) and *wxDataObject overview* (p. 1819) for more information.

Derived from

None

Include files

<wx/dnd.h>

Types

wxDragResult is defined as follows:

```
enum wxDragResult
{
    wxDragError,   // error prevented the d&d operation from completing
    wxDragNone,    // drag target didn't accept the data
    wxDragCopy,    // the data was successfully copied
    wxDragMove,    // the data was successfully moved (MSW only)
    wxDragLink,    // operation is a drag-link
    wxDragCancel   // the operation was cancelled by user (not an error)
};
```

See also

wxDropTarget (p. 471), *wxTextDropTarget* (p. 1360), *wxFileDropTarget* (p. 516)

wxDropSource::wxDropSource

wxDropSource(**wxWindow*** *win = NULL*,**const wxIconOrCursor&** *iconCopy = wxNullIconOrCursor*, **const wxIconOrCursor&** *iconMove = wxNullIconOrCursor*, **const wxIconOrCursor&** *iconNone = wxNullIconOrCursor*)

wxDropSource(**wxDataObject&** *data*, **wxWindow*** *win = NULL*,**const wxIconOrCursor&** *iconCopy = wxNullIconOrCursor*, **const wxIconOrCursor&** *iconMove = wxNullIconOrCursor*, **const wxIconOrCursor&** *iconNone = wxNullIconOrCursor*)

The constructors for wxDataObject.

If you use the constructor without *data* parameter you must call *SetData* (p. 469) later.

Note that the exact type of *iconCopy* and subsequent parameters differs between wxMSW and wxGTK: these are cursors under Windows but icons for GTK. You should use the macro *wxDROP_ICON* (p. 1624) in portable programs instead of directly using either of these types.

Parameters

win

> The window which initiates the drag and drop operation.

iconCopy

> The icon or cursor used for feedback for copy operation.

iconMove

> The icon or cursor used for feedback for move operation.

iconNone

> The icon or cursor used for feedback when operation can't be done.

win is the window which initiates the drag and drop operation.

wxDropSource::~wxDropSource

virtual ~wxDropSource()

wxDropSource::SetData

void SetData(**wxDataObject&** *data*)

Sets the data *wxDataObject* (p. 239) associated with the drop source. This will not

delete any previously associated data.

wxDropSource::DoDragDrop

virtual wxDragResult DoDragDrop(**int** *flags* = *wxDrag_CopyOnly*)

Do it (call this in response to a mouse button press, for example). This starts the drag-and-drop operation which will terminate when the user releases the mouse.

Parameters

flags

If `wxDrag_AllowMove` is included in the flags, data may be moved and not only copied (default). If `wxDrag_DefaultMove` is specified (which includes the previous flag), this is even the default operation

.

Return value

Returns the operation requested by the user, may be `wxDragCopy`, `wxDragMove`, `wxDragLink`, `wxDragCancel` or `wxDragNone` if an error occurred.

wxDropSource::GetDataObject

wxDataObject * GetDataObject()

Returns the wxDataObject object that has been assigned previously.

wxDropSource::GiveFeedback

virtual bool GiveFeedback(**wxDragResult** *effect*)

Overridable: you may give some custom UI feedback during the drag and drop operation in this function. It is called on each mouse move, so your implementation must not be too slow.

Parameters

effect

The effect to implement. One of `wxDragCopy`, `wxDragMove`, `wxDragLink` and `wxDragNone`.

scrolling

true if the window is scrolling. MSW only.

Return value

Return false if you want default feedback, or true if you implement your own feedback. The return values is ignored under GTK.

wxDropSource::SetCursor

void SetCursor(**wxDragResult** *res*, **const wxCursor&** *cursor*)

Set the icon to use for a certain drag result.

Parameters

res

> The drag result to set the icon for.

cursor

> The ion to show when this drag result occurs.

wxDropTarget

This class represents a target for a drag and drop operation. A *wxDataObject* (p. 239)can be associated with it and by default, this object will be filled with the data from the drag source, if the data formats supported by the data object match the drag source data format.

There are various virtual handler functions defined in this class which may be overridden to give visual feedback or react in a more fine-tuned way, e.g. by not accepting data on the whole window area, but only a small portion of it. The normal sequence of calls is*OnEnter* (p. 473), possibly many times *OnDragOver* (p. 473),*OnDrop* (p. 472) and finally *OnData* (p. 472).

See *Drag and drop overview* (p. 1817) and *wxDataObject overview* (p. 1819)for more information.

Derived from

None

Include files

<wx/dnd.h>

Types

wxDragResult is defined as follows:

```
enum wxDragResult
{
    wxDragError,    // error prevented the d&d operation from completing
    wxDragNone,     // drag target didn't accept the data
    wxDragCopy,     // the data was successfully copied
    wxDragMove,     // the data was successfully moved (MSW only)
```

```
        wxDragLink,     // operation is a drag-link
        wxDragCancel    // the operation was cancelled by user (not an error)
    };
```

See also

wxDropSource (p. 468), *wxTextDropTarget* (p. 1360), *wxFileDropTarget* (p. 516),*wxDataFormat* (p. 234), *wxDataObject* (p. 239)

wxDropTarget::wxDropTarget

 wxDropTarget(**wxDataObject*** *data = NULL*)

Constructor. *data* is the data to be associated with the drop target.

wxDropTarget::~wxDropTarget

 ~wxDropTarget()

Destructor. Deletes the associated data object, if any.

wxDropTarget::GetData

virtual void GetData()

This method may only be called from within *OnData* (p. 472). By default, this method copies the data from the drop source to the *wxDataObject* (p. 239) associated with this drop target, calling its *wxDataObject::SetData* (p. 242) method.

wxDropTarget::OnData

virtual wxDragResult OnData(**wxCoord** *x*, **wxCoord** *y*, **wxDragResult** *def*)

Called after *OnDrop* (p. 472) returns true. By default this will usually *GetData* (p. 472) and will return the suggested default value *def*.

wxDropTarget::OnDrop

virtual bool OnDrop(**wxCoord** *x*, **wxCoord** *y*)

Called when the user drops a data object on the target. Return false to veto the operation.

Parameters

x

The x coordinate of the mouse.

y

The y coordinate of the mouse.

Return value

Return true to accept the data, false to veto the operation.

wxDropTarget::OnEnter

virtual wxDragResult OnEnter(**wxCoord** *x*, **wxCoord** *y*, **wxDragResult** *def*)

Called when the mouse enters the drop target. By default, this calls *OnDragOver* (p. 473).

Parameters

x

The x coordinate of the mouse.

y

The y coordinate of the mouse.

def

Suggested default for return value. Determined by SHIFT or CONTROL key states.

Return value

Returns the desired operation or `wxDragNone`. This is used for optical feedback from the side of the drop source, typically in form of changing the icon.

wxDropTarget::OnDragOver

virtual wxDragResult OnDragOver(**wxCoord** *x*, **wxCoord** *y*, **wxDragResult** *def*)

Called when the mouse is being dragged over the drop target. By default, this calls functions return the suggested return value *def*.

Parameters

x

The x coordinate of the mouse.

y

The y coordinate of the mouse.

def

Suggested value for return value. Determined by SHIFT or CONTROL key states.

Return value

Returns the desired operation or `wxDragNone`. This is used for optical feedback from the side of the drop source, typically in form of changing the icon.

wxDropTarget::OnLeave

virtual void OnLeave()

Called when the mouse leaves the drop target.

wxDropTarget::SetDataObject

void SetDataObject(**wxDataObject*** *data*)

Sets the data *wxDataObject* (p. 239) associated with the drop target and deletes any previously associated data object.

wxDynamicLibrary

wxDynamicLibrary is a class representing dynamically loadable library (Windows DLL, shared library under Unix etc.). Just create an object of this class to load a library and don't worry about unloading it -- it will be done in the objects destructor automatically.

Derived from

No base class.

Include files

<wx/dynlib.h>

(only available if `wxUSE_DYNLIB_CLASS` is set to 1)

wxDynamicLibrary::wxDynamicLibrary

wxDynamicLibrary()

wxDynamicLibrary(**const wxString&** *name*, **int** *flags* = *wxDL_DEFAULT*)

Constructor. Second form calls *Load* (p. 476).

wxDynamicLibrary::CanonicalizeName

wxString CanonicalizeName(**const wxString&** *name*, **wxDynamicLibraryCategory** *cat* = *wxDL_LIBRARY*)

Returns the platform-specific full name for the library called *name*. E.g. it adds a ".dll" extension under Windows and "lib" prefix and ".so", ".sl" or maybe ".dylib" extension under Unix.

The possible values for *cat* are:

wxDL_LIBRARY	normal library
wxDL_MODULE	a loadable module or plugin

See also

CanonicalizePluginName (p. 475)

wxDynamicLibrary::CanonicalizePluginName

wxString CanonicalizePluginName(**const wxString&** *name*, **wxPluginCategory** *cat* = *wxDL_PLUGIN_GUI*)

This function does the same thing as *CanonicalizeName* (p. 475) but for wxWidgets plugins. The only difference is that compiler and version information are added to the name to ensure that the plugin which is going to be loaded will be compatible with the main program.

The possible values for *cat* are:

wxDL_PLUGIN_GUI	plugin which uses GUI classes (default)
wxDL_PLUGIN_BASE	plugin which only uses wxBase

wxDynamicLibrary::Detach

wxDllType Detach()

Detaches this object from its library handle, i.e. the object will not unload the library any longer in its destructor but it is now the callers responsibility to do this using *Unload* (p. 477).

wxDynamicLibrary::GetSymbol

void * GetSymbol(**const wxString&** *name*) **const**

Returns pointer to symbol *name* in the library or NULL if the library contains no such symbol.

See also

wxDYNLIB_FUNCTION (p. 1631)

wxDynamicLibrary::GetProgramHandle

static wxDllType GetProgramHandle()

Return a valid handle for the main program itself or NULL if symbols from the main program can't be loaded on this platform.

wxDynamicLibrary::HasSymbol

bool HasSymbol(const wxString& *name*) const

Returns true if the symbol with the given *name* is present in the dynamic library, false otherwise. Unlike *GetSymbol* (p. 475), this function doesn't log an error message if the symbol is not found.

This function is new since wxWidgets version 2.5.4

wxDynamicLibrary::IsLoaded

bool IsLoaded() const

Returns true if the library was successfully loaded, false otherwise.

wxDynamicLibrary::ListLoaded

static wxDynamicLibraryDetailsArray ListLoaded()

This static method returns an *array* (p. 53) containing the details of all modules loaded into the address space of the current project, the array elements are object of wxDynamicLibraryDetails class. The array will be empty if an error occurred.

This method is currently implemented only under Win32 and Linux and is useful mostly for diagnostics purposes.

wxDynamicLibrary::Load

bool Load(const wxString& *name*, int *flags* = *wxDL_DEFAULT*)

Loads DLL with the given *name* into memory. The *flags* argument can be a combination of the following bits:

wxDL_LAZY	equivalent of RTLD_LAZY under Unix, ignored elsewhere

wxDL_NOW	equivalent of RTLD_NOW under Unix, ignored elsewhere
wxDL_GLOBAL	equivalent of RTLD_GLOBAL under Unix, ignored elsewhere
wxDL_VERBATIM	don't try to append the appropriate extension to the library name (this is done by default).
wxDL_DEFAULT	default flags, same as wxDL_NOW currently

Returns `true` if the library was successfully loaded, `false` otherwise.

wxDynamicLibrary::Unload

void Unload()

static void Unload(**wxDllType** *handle*)

Unloads the library from memory. wxDynamicLibrary object automatically calls this method from its destructor if it had been successfully loaded.

The second version is only used if you need to keep the library in memory during a longer period of time than the scope of the wxDynamicLibrary object. In this case you may call *Detach* (p. 475) and store the handle somewhere and call this static method later to unload it.

wxDynamicLibraryDetails

This class is used for the objects returned by *wxDynamicLibrary::ListLoaded* (p. 476) method and contains the information about a single module loaded into the address space of the current process. A module in this context may be either a dynamic library or the main program itself.

Derived from

No base class.

Include files

<wx/dynlib.h>

(only available if `wxUSE_DYNLIB_CLASS` is set to 1)

wxDynamicLibraryDetails::GetName

wxString GetName() **const**

Returns the base name of this module, e.g. `kernel32.dll` or `libc-2.3.2.so`.

wxDynamicLibraryDetails::GetPath

wxString GetPath() **const**

Returns the full path of this module if available, e.g.
`c:\windows\system32\kernel32.dll` or `/lib/libc-2.3.2.so`.

wxDynamicLibraryDetails::GetAddress

bool GetAddress(**void *****addr*, **size_t** **len*) **const**

Retrieves the load address and the size of this module.

Parameters

addr

the pointer to the location to return load address in, may be`NULL`

len

pointer to the location to return the size of this module in memory in, may be `NULL`

Return value

`true` if the load address and module size were retrieved, `false` if this information is not available.

wxDynamicLibraryDetails::GetVersion

wxString GetVersion() **const**

Returns the version of this module, e.g. `5.2.3790.0` or `2.3.2`. The returned string is empty if the version information is not available.

wxEncodingConverter

This class is capable of converting strings between two 8-bit encodings/charsets. It can also convert from/to Unicode (but only if you compiled wxWidgets with wxUSE_WCHAR_T set to 1). Only limited subset of encodings in supported by wxEncodingConverter:`wxFONTENCODING_ISO8859_1..15`, `wxFONTENCODING_CP1250..1257` and `wxFONTENCODING_KOI8`.

Note

Please use *wxMBConv classes* (p. 1733) instead if possible. *wxCSConv* (p. 225) has

much better support for various encodings than wxEncodingConverter. wxEncodingConverter is useful only if you rely on `wxCONVERT_SUBSTITUTE` mode of operation (see *Init* (p. 479)).

Derived from

wxObject (p. 1012)

Include files

<wx/encconv.h>

See also

wxFontMapper (p. 575), *wxMBConv* (p. 909), *Writing non-English applications* (p. 1737)

wxEncodingConverter::wxEncodingConverter

 wxEncodingConverter()

Constructor.

wxEncodingConverter::Init

bool Init(**wxFontEncoding** *input_enc*, **wxFontEncoding** *output_enc*, **int** *method* = *wxCONVERT_STRICT*)

Initialize conversion. Both output or input encoding may be wxFONTENCODING_UNICODE, but only if wxUSE_ENCODING is set to 1. All subsequent calls to *Convert()* (p. 480) will interpret its argument as a string in *input_enc* encoding and will output string in *output_enc* encoding. You must call this method before calling Convert. You may call it more than once in order to switch to another conversion.*Method* affects behaviour of Convert() in case input character cannot be converted because it does not exist in output encoding:

wxCONVERT_STRICT	follow behaviour of GNU Recode - just copy unconvertible characters to output and don't change them (its integer value will stay the same)
wxCONVERT_SUBSTITUTE	try some (lossy) substitutions - e.g. replace unconvertible latin capitals with acute by ordinary capitals, replace en-dash or em-dash by '-' etc.

Both modes guarantee that output string will have same length as input string.

Return value

false if given conversion is impossible, true otherwise (conversion may be impossible either if you try to convert to Unicode with non-Unicode build of wxWidgets or if input or output encoding is not supported.)

wxEncodingConverter::CanConvert

static bool CanConvert(**wxFontEncoding***encIn*, **wxFontEncoding** *encOut*)

Return true if (any text in) multibyte encoding *encIn* can be converted to another one (*encOut*) losslessly.

Do not call this method with `wxFONTENCODING_UNICODE` as either parameter, it doesn't make sense (always works in one sense and always depends on the text to convert in the other).

wxEncodingConverter::Convert

bool Convert(**const char*** *input*, **char*** *output*) **const**

bool Convert(**const wchar_t*** *input*, **wchar_t*** *output*) **const**

bool Convert(**const char*** *input*, **wchar_t*** *output*) **const**

bool Convert(**const wchar_t*** *input*, **char*** *output*) **const**

Convert input string according to settings passed to*Init* (p. 479) and writes the result to *output*.

bool Convert(**char*** *str*) **const**

bool Convert(**wchar_t*** *str*) **const**

Convert input string according to settings passed to*Init* (p. 479) in-place, i.e. write the result to the same memory area.

All of the versions above return `true` if the conversion was lossless and`false` if at least one of the characters couldn't be converted and was replaced with `'?'` in the output. Note that if `wxCONVERT_SUBSTITUTE` was passed to *Init* (p. 479), substitution is considered lossless operation.

wxString Convert(**const wxString&** *input*) **const**

Convert wxString and return new wxString object.

Notes

You must call *Init* (p. 479) before using this method!

`wchar_t` versions of the method are not available if wxWidgets was compiled with `wxUSE_WCHAR_T` set to 0.

wxEncodingConverter::GetPlatformEquivalents

static wxFontEncodingArray GetPlatformEquivalents(**wxFontEncoding** *enc*, **int** *platform* = *wxPLATFORM_CURRENT*)

Return equivalents for given font that are used under given platform. Supported platforms:

- wxPLATFORM_UNIX

- wxPLATFORM_WINDOWS

- wxPLATFORM_OS2

- wxPLATFORM_MAC

- wxPLATFORM_CURRENT

wxPLATFORM_CURRENT means the platform this binary was compiled for.

Examples:

```
current platform    enc            returned value
-----------------------------------------------
unix            CP1250             {ISO8859_2}
unix          ISO8859_2            {ISO8859_2}
windows       ISO8859_2              {CP1250}
unix            CP1252  {ISO8859_1,ISO8859_15}
```

Equivalence is defined in terms of convertibility: two encodings are equivalent if you can convert text between then without losing information (it may - and will - happen that you lose special chars like quotation marks or em-dashes but you shouldn't lose any diacritics and language-specific characters when converting between equivalent encodings).

Remember that this function does **NOT** check for presence of fonts in system. It only tells you what are most suitable encodings. (It usually returns only one encoding.)

Notes

- Note that argument *enc* itself may be present in the returned array, so that you can, as a side-effect, detect whether the encoding is native for this platform or not.

- *Convert* (p. 480) is not limited to converting between equivalent encodings, it can convert between two arbitrary encodings.

- If *enc* is present in the returned array, then it is **always** the first item of it.

- Please note that the returned array may contain no items at all.

wxEncodingConverter::GetAllEquivalents

static wxFontEncodingArray GetAllEquivalents(**wxFontEncoding** *enc*)

Similar to *GetPlatformEquivalents* (p. 481), but this one will return ALL equivalent encodings, regardless of the platform, and including itself.

This platform's encodings are before others in the array. And again, if *enc* is in the array, it is the very first item in it.

wxEraseEvent

An erase event is sent when a window's background needs to be repainted.

On some platforms, such as GTK+, this event is simulated (simply generated just before the paint event) and may cause flicker. It is therefore recommended that you set the text background colour explicitly in order to prevent flicker. The default background colour under GTK+ is grey.

To intercept this event, use the EVT_ERASE_BACKGROUND macro in an event table definition.

You must call wxEraseEvent::GetDC and use the returned device context if it is non-NULL. If it is NULL, create your own temporary wxClientDC object.

Derived from

wxEvent (p. 483)
wxObject (p. 1012)

Include files

<wx/event.h>

Event table macros

To process an erase event, use this event handler macro to direct input to a member function that takes a wxEraseEvent argument.

EVT_ERASE_BACKGROUND(func)   Process a wxEVT_ERASE_BACKGROUND event.

Remarks
Use the device context returned by *GetDC* (p. 483) to draw on, don't create a wxPaintDC in the event handler.

See also

Event handling overview (p. 1751)

wxEraseEvent::wxEraseEvent

wxEraseEvent(**int** *id = 0*, **wxDC*** *dc = NULL*)

Constructor.

---

wxEraseEvent::GetDC

wxDC* GetDC() **const**

Returns the device context associated with the erase event to draw on.

---

wxEvent

An event is a structure holding information about an event passed to a callback or member function. **wxEvent** used to be a multipurpose event object, and is an abstract base class for other event classes (see below).

For more information about events, see the *Event handling overview* (p. 1751).

wxPerl note: In wxPerl custom event classes should be derived from`Wx::PlEvent` and `Wx::PlCommandEvent.`

Derived from

wxObject (p. 1012)

Include files

<wx/event.h>

See also

wxCommandEvent (p. 180), *wxMouseEvent* (p. 982)

---

wxEvent::wxEvent

wxEvent(**int** *id = 0*, **wxEventType** *eventType = `wxEVT_NULL`*)

Constructor. Should not need to be used directly by an application.

---

wxEvent::m_propagationLevel

int m_propagationLevel

Indicates how many levels the event can propagate. This member is protected and should typically only be set in the constructors of the derived classes. It may be temporarily changed by *StopPropagation* (p. 486) and *ResumePropagation* (p. 485) and tested with *ShouldPropagate* (p. 486).

The initial value is set to either wxEVENT_PROPAGATE_NONE (by default) meaning that the event shouldn't be propagated at all or to wxEVENT_PROPAGATE_MAX (for command events) meaning that it should be propagated as much as necessary.

Any positive number means that the event should be propagated but no more than the given number of times. E.g. the propagation level may be set to 1 to propagate the event to its parent only, but not to its grandparent.

wxEvent::Clone

virtual wxEvent* Clone() const

Returns a copy of the event.

Any event that is posted to the wxWidgets event system for later action (via*wxEvtHandler::AddPendingEvent* (p. 487) or*wxPostEvent* (p. 1638)) must implement this method. All wxWidgets events fully implement this method, but any derived events implemented by the user should also implement this method just in case they (or some event derived from them) are ever posted.

All wxWidgets events implement a copy constructor, so the easiest way of implementing the Clone function is to implement a copy constructor for a new event (call it MyEvent) and then define the Clone function like this:

```
wxEvent *Clone(void) const { return new MyEvent(*this); }
```

wxEvent::GetEventObject

wxObject* GetEventObject()

Returns the object (usually a window) associated with the event, if any.

wxEvent::GetEventType

WXTYPE GetEventType()

Returns the identifier of the given event type, such as wxEVENT_TYPE_BUTTON_COMMAND.

wxEvent::GetId

int GetId() const

Returns the identifier associated with this event, such as a button command id.

wxEvent::GetSkipped

bool GetSkipped() **const**

Returns true if the event handler should be skipped, false otherwise.

wxEvent::GetTimestamp

long GetTimestamp()

Gets the timestamp for the event.

wxEvent::IsCommandEvent

bool IsCommandEvent() **const**

Returns true if the event is or is derived from *wxCommandEvent* (p. 180) else it returns false. Note: Exists only for optimization purposes.

wxEvent::ResumePropagation

void ResumePropagation(**int** *propagationLevel*)

Sets the propagation level to the given value (for example returned from an earlier call to *StopPropagation* (p. 486)).

wxEvent::SetEventObject

void SetEventObject(**wxObject*** *object*)

Sets the originating object.

wxEvent::SetEventType

void SetEventType(**WXTYPE** *typ*)

Sets the event type.

wxEvent::SetId

void SetId(**int** *id*)

Sets the identifier associated with this event, such as a button command id.

wxEvent::SetTimestamp

void SetTimestamp(**long** *timeStamp*)

Sets the timestamp for the event.

wxEvent::ShouldPropagate

bool ShouldPropagate() **const**

Test if this event should be propagated or not, i.e. if the propagation level is currently greater than 0.

wxEvent::Skip

void Skip(**bool** *skip = true*)

Called by an event handler, it controls whether additional event handlers bound to this event will be called after the current event handler returns. Skip(false) (the default setting) will prevent additional event handlers from being called and control will be returned to the sender of the event immediately after the current handler has finished. Skip(True) will cause the event processing system to continue searching for a handler function for this event.

wxEvent::StopPropagation

int StopPropagation()

Stop the event from propagating to its parent window.

Returns the old propagation level value which may be later passed to *ResumePropagation* (p. 485) to allow propagating the event again.

wxEvtHandler

A class that can handle events from the windowing system. wxWindow (and therefore all window classes) are derived from this class.

When events are received, wxEvtHandler invokes the method listed in the event table using itself as the object. When using multiple inheritance it is imperative that the wxEvtHandler(-derived) class be the first class inherited such that the "this" pointer for the overall object will be identical to the "this" pointer for the wxEvtHandler portion.

Derived from

wxObject (p. 1012)

Include files

<wx/event.h>

See also

Event handling overview (p. 1751)

wxEvtHandler::wxEvtHandler

wxEvtHandler()

Constructor.

wxEvtHandler::~wxEvtHandler

~wxEvtHandler()

Destructor. If the handler is part of a chain, the destructor will unlink itself and restore the previous and next handlers so that they point to each other.

wxEvtHandler::AddPendingEvent

void AddPendingEvent(**wxEvent&** *event*)

This function posts an event to be processed later.

Parameters

event

Event to add to process queue.

Remarks

The difference between sending an event (using the *ProcessEvent* (p. 491) method) and posting it is that in the first case the event is processed before the function returns, while in the second case, the function returns immediately and the event will be processed sometime later (usually during the next event loop iteration).

A copy of *event* is made by the function, so the original can be deleted as soon as function returns (it is common that the original is created on the stack). This requires that the *wxEvent::Clone* (p. 484) method be implemented by *event* so that it can be duplicated and stored until it gets processed.

This is also the method to call for inter-thread communication---it will post events safely between different threads which means that this method is thread-safe by using critical sections where needed. In a multi-threaded program, you often need to inform the main GUI thread about the status of other working threads and such notification should be done using this method.

This method automatically wakes up idle handling if the underlying window system is currently idle and thus would not send any idle events. (Waking up idle handling is done calling *::wxWakeUpIdle* (p. 1592).)

wxEvtHandler::Connect

void Connect(**int** *id*, **int** *lastId*, **wxEventType** *eventType*, **wxObjectEventFunction** *function*, **wxObject*** *userData = NULL*, **wxEvtHandler*** *eventSink = NULL*)

void Connect(**int** *id*, **wxEventType** *eventType*, **wxObjectEventFunction** *function*, **wxObject*** *userData = NULL*, **wxEvtHandler*** *eventSink = NULL*)

void Connect(**wxEventType** *eventType*, **wxObjectEventFunction** *function*, **wxObject*** *userData = NULL*, **wxEvtHandler*** *eventSink = NULL*)

Connects the given function dynamically with the event handler, id and event type. This is an alternative to the use of static event tables. See the 'event' or the old 'dynamic' sample for usage.

Parameters

id

> The identifier (or first of the identifier range) to be associated with the event handler function. For the version not taking this argument, it defaults to wxID_ANY.

lastId

> The second part of the identifier range to be associated with the event handler function.

eventType

> The event type to be associated with this event handler.

function

> The event handler function. Note that this function should be explicitly converted to the correct type which can be done using a macro called wxFooHandler for the handler for any wxFooEvent.

userData

> Data to be associated with the event table entry.

eventSink

> Object whose member function should be called. If this is NULL,*this* will be used.

Example

```
frame->Connect( wxID_EXIT,
  wxEVT_COMMAND_MENU_SELECTED,
  wxCommandEventHandler(MyFrame::OnQuit) );
```

wxPerl note: In wxPerl this function takes 4 arguments: id, lastid, type, method; if method is undef, the handler is disconnected.

wxEvtHandler::Disconnect

bool Disconnect(**int** *id* = `wxID_ANY`, **wxEventType** *eventType = wxEVT_NULL*, **wxObjectEventFunction** *function = NULL*, **wxObject*** *userData = NULL*, **wxEvtHandler*** *eventSink = NULL*)

bool Disconnect(**int** *id*, **int** *lastId* = `wxID_ANY`, **wxEventType** *eventType = wxEVT_NULL*, **wxObjectEventFunction** *function = NULL*, **wxObject*** *userData = NULL*, **wxEvtHandler*** *eventSink = NULL*)

Disconnects the given function dynamically from the event handler, using the specified parameters as search criteria and returning true if a matching function has been found and removed. This method can only disconnect functions which have been added using the *wxEvtHandler::Connect* (p. 488) method. There is no way to disconnect functions connected using the (static) event tables.

Parameters

id

> The identifier (or first of the identifier range) associated with the event handler function.

lastId

> The second part of the identifier range associated with the event handler function.

eventType

> The event type associated with this event handler.

function

> The event handler function.

userData

> Data associated with the event table entry.

eventSink

> Object whose member function should be called.

wxPerl note: In wxPerl this function takes 3 arguments: `id, lastid, type`.

wxEvtHandler::GetClientData

void* GetClientData()

Gets user-supplied client data.

Remarks

Normally, any extra data the programmer wishes to associate with the object should be made available by deriving a new class with new data members.

See also

wxEvtHandler::SetClientData (p. 492)

wxEvtHandler::GetClientObject

wxClientData* GetClientObject() **const**

Get a pointer to the user-supplied client data object.

See also

wxEvtHandler::SetClientObject (p. 493),*wxClientData* (p. 148)

wxEvtHandler::GetEvtHandlerEnabled

bool GetEvtHandlerEnabled()

Returns true if the event handler is enabled, false otherwise.

See also

wxEvtHandler::SetEvtHandlerEnabled (p. 493)

wxEvtHandler::GetNextHandler

wxEvtHandler* GetNextHandler()

Gets the pointer to the next handler in the chain.

See also

wxEvtHandler::SetNextHandler (p. 494), *wxEvtHandler::GetPreviousHandler* (p. 490), *wxEvtHandler::SetPreviousHandler* (p. 494), *wxWindow::PushEventHandler* (p. 1522), *wxWindow::PopEventHandler* (p. 1520)

wxEvtHandler::GetPreviousHandler

wxEvtHandler* GetPreviousHandler()

Gets the pointer to the previous handler in the chain.

See also

wxEvtHandler::SetPreviousHandler (p. 494), *wxEvtHandler::GetNextHandler* (p. 490), *wxEvtHandler::SetNextHandler* (p. 494), *wxWindow::PushEventHandler* (p. 1522),

wxWindow::PopEventHandler (p. 1520)

wxEvtHandler::ProcessEvent

virtual bool ProcessEvent(**wxEvent&** *event*)

Processes an event, searching event tables and calling zero or more suitable event handler function(s).

Parameters

event

> Event to process.

Return value

true if a suitable event handler function was found and executed, and the function did not call *wxEvent::Skip* (p. 486).

Remarks

Normally, your application would not call this function: it is called in the wxWidgets implementation to dispatch incoming user interface events to the framework (and application).

However, you might need to call it if implementing new functionality (such as a new control) where you define new event types, as opposed to allowing the user to override virtual functions.

An instance where you might actually override the **ProcessEvent** function is where you want to direct event processing to event handlers not normally noticed by wxWidgets. For example, in the document/view architecture, documents and views are potential event handlers. When an event reaches a frame, **ProcessEvent** will need to be called on the associated document and view in case event handler functions are associated with these objects. The property classes library (wxProperty) also overrides **ProcessEvent** for similar reasons.

The normal order of event table searching is as follows:

1. If the object is disabled (via a call to *wxEvtHandler::SetEvtHandlerEnabled* (p. 493)) the function skips to step (6).

2. If the object is a wxWindow, **ProcessEvent** is recursively called on the window's *wxValidator* (p. 1461). If this returns true, the function exits.

3. **SearchEventTable** is called for this event handler. If this fails, the base class table is tried, and so on until no more tables exist or an appropriate function was found, in which case the function exits.

4. The search is applied down the entire chain of event handlers (usually the chain has a length of one). If this succeeds, the function exits.

5. If the object is a wxWindow and the event is a wxCommandEvent, **ProcessEvent** is recursively applied to the parent window's event handler. If this returns true, the function exits.

6. Finally, **ProcessEvent** is called on the wxApp object.

See also

wxEvtHandler::SearchEventTable (p. 492)

wxEvtHandler::SearchEventTable

virtual bool SearchEventTable(**wxEventTable&** *table*, **wxEvent&** *event*)

Searches the event table, executing an event handler function if an appropriate one is found.

Parameters

table

> Event table to be searched.

event

> Event to be matched against an event table entry.

Return value

true if a suitable event handler function was found and executed, and the function did not call *wxEvent::Skip* (p. 486).

Remarks

This function looks through the object's event table and tries to find an entry that will match the event.

An entry will match if:

1. The event type matches, and

2. the identifier or identifier range matches, or the event table entry's identifier is zero.

If a suitable function is called but calls *wxEvent::Skip* (p. 486), this function will fail, and searching will continue.

See also

wxEvtHandler::ProcessEvent (p. 491)

wxEvtHandler::SetClientData

void SetClientData(**void*** *data*)

Sets user-supplied client data.

Parameters

data

> Data to be associated with the event handler.

Remarks

Normally, any extra data the programmer wishes to associate with the object should be made available by deriving a new class with new data members. You must not call this method and*SetClientObject* (p. 493) on the same class - only one of them.

See also

wxEvtHandler::GetClientData (p. 489)

wxEvtHandler::SetClientObject

void SetClientObject(**wxClientData*** *data*)

Set the client data object. Any previous object will be deleted.

See also

wxEvtHandler::GetClientObject (p. 490),*wxClientData* (p. 148)

wxEvtHandler::SetEvtHandlerEnabled

void SetEvtHandlerEnabled(**bool** *enabled*)

Enables or disables the event handler.

Parameters

enabled

> true if the event handler is to be enabled, false if it is to be disabled.

Remarks

You can use this function to avoid having to remove the event handler from the chain, for example when implementing a dialog editor and changing from edit to test mode.

See also

wxEvtHandler::GetEvtHandlerEnabled (p. 490)

wxEvtHandler::SetNextHandler

void SetNextHandler(**wxEvtHandler*** *handler*)

Sets the pointer to the next handler.

Parameters

handler

  Event handler to be set as the next handler.

See also

wxEvtHandler::GetNextHandler (p. 490), *wxEvtHandler::SetPreviousHandler* (p. 494), *wxEvtHandler::GetPreviousHandler* (p. 490), *wxWindow::PushEventHandler* (p. 1522), *wxWindow::PopEventHandler* (p. 1520)

wxEvtHandler::SetPreviousHandler

void SetPreviousHandler(**wxEvtHandler*** *handler*)

Sets the pointer to the previous handler.

Parameters

handler

  Event handler to be set as the previous handler.

See also

wxEvtHandler::GetPreviousHandler (p. 490), *wxEvtHandler::SetNextHandler* (p. 494), *wxEvtHandler::GetNextHandler* (p. 490), *wxWindow::PushEventHandler* (p. 1522), *wxWindow::PopEventHandler* (p. 1520)

wxFFile

wxFFile implements buffered file I/O. This is a very small class designed to minimize the overhead of using it - in fact, there is hardly any overhead at all, but using it brings you automatic error checking and hides differences between platforms and compilers. It wraps inside it a `FILE *` handle used by standard C IO library (also known as `stdio`).

Derived from

None.

Include files

<wx/ffile.h>

| **wxFromStart** | Count offset from the start of the file |

wxFromCurrent	Count offset from the current position of the file pointer
wxFromEnd	Count offset from the end of the file (backwards)

wxFFile::wxFFile

wxFFile()

Default constructor.

wxFFile(**const char*** *filename*, **const char*** *mode = "r"*)

Opens a file with the given mode. As there is no way to return whether the operation was successful or not from the constructor you should test the return value of *IsOpened* (p. 497) to check that it didn't fail.

wxFFile(**FILE*** *fp*)

Opens a file with the given file pointer, which has already been opened.

Parameters

filename

> The filename.

mode

> The mode in which to open the file using standard C strings. Note that you should use "b" flag if you use binary files under Windows or the results might be unexpected due to automatic newline conversion done for the text files.

fp

> An existing file descriptor, such as stderr.

wxFFile::~wxFFile

~wxFFile()

Destructor will close the file.

NB: it is not virtual so you should *not* derive from wxFFile!

wxFFile::Attach

void Attach(**FILE*** *fp*)

Attaches an existing file pointer to the wxFFile object.

The descriptor should be already opened and it will be closed by wxFFile object.

wxFFile::Close

bool Close()

Closes the file and returns `true` on success.

wxFFile::Detach

void Detach()

Get back a file pointer from wxFFile object -- the caller is responsible for closing the file if this descriptor is opened. *IsOpened()* (p. 497) will return `false` after call to Detach().

wxFFile::fp

FILE * fp() **const**

Returns the file pointer associated with the file.

wxFFile::Eof

bool Eof() **const**

Returns `true` if the an attempt has been made to read *past*the end of the file.

Note that the behaviour of the file descriptor based class*wxFile* (p. 502) is different as *wxFile::Eof* (p. 506)will return `true` here as soon as the last byte of the file has been read.

Also note that this method may only be called for opened files and may crash if the file is not opened.

See also

IsOpened (p. 497)

wxFFile::Error

Returns `true` if an error has occurred on this file, similar to the standard`ferror()` function.

Please note that this method may only be called for opened files and may crash if the file is not opened.

IsOpened (p. 497)

wxFFile::Flush

bool Flush()

Flushes the file and returns `true` on success.

wxFFile::GetKind

wxFileKind GetKind() **const**

Returns the type of the file. Possible return values are:

```
enum wxFileKind
{
  wxFILE_KIND_UNKNOWN,
  wxFILE_KIND_DISK,     // a file supporting seeking to arbitrary offsets
  wxFILE_KIND_TERMINAL, // a tty
  wxFILE_KIND_PIPE      // a pipe
};
```

wxFFile::IsOpened

bool IsOpened() **const**

Returns `true` if the file is opened. Most of the methods of this class may only be used for an opened file.

wxFFile::Length

wxFileOffset Length() **const**

Returns the length of the file.

wxFFile::Open

bool Open(**const char*** *filename*, **const char*** *mode = "r"*)

Opens the file, returning `true` if successful.

Parameters

filename

The filename.

mode

> The mode in which to open the file.

wxFFile::Read

size_t Read(**void*** *buffer*, **size_t** *count*)

Reads the specified number of bytes into a buffer, returning the actual number read.

Parameters

buffer

> A buffer to receive the data.

count

> The number of bytes to read.

Return value

The number of bytes read.

wxFFile::ReadAll

bool ReadAll(**wxString *** *str*, **wxMBConv&** *conv = wxConvUTF8*)

Reads the entire contents of the file into a string.

Parameters

str

> String to read data into.

conv

> Conversion object to use in Unicode build; by default supposes that file contents is encoded in UTF-8.

Return value

`true` if file was read successfully, `false` otherwise.

wxFFile::Seek

bool Seek(**wxFileOffset** *ofs*, **wxSeekMode** *mode = wxFromStart*)

Seeks to the specified position and returns `true` on success.

Parameters

ofs

   Offset to seek to.

mode

   One of **wxFromStart**, **wxFromEnd**, **wxFromCurrent**.

wxFFile::SeekEnd

bool SeekEnd(**wxFileOffset** *ofs = 0*)

Moves the file pointer to the specified number of bytes before the end of the file and returns `true` on success.

Parameters

ofs

   Number of bytes before the end of the file.

wxFFile::Tell

wxFileOffset Tell() **const**

Returns the current position.

wxFFile::Write

size_t Write(**const void*** *buffer*, **size_t** *count*)

Writes the specified number of bytes from a buffer.

Parameters

buffer

   A buffer containing the data.

count

   The number of bytes to write.

Return value

Number of bytes written.

wxFFile::Write

bool Write(**const wxString&** *s*, **wxMBConv&** *conv* = *wxConvUTF8*)

Writes the contents of the string to the file, returns `true` on success.

The second argument is only meaningful in Unicode build of wxWidgets when*conv* is used to convert *s* to multibyte representation.

wxFFileInputStream

This class represents data read in from a file. There are actually two such groups of classes: this one is based on *wxFFile* (p. 494) whereas *wxFileInputStream* (p. 520) is based in the *wxFile* (p. 502) class.

Note that *wxFile* (p. 502) and *wxFFile* (p. 494) differ in one aspect, namely when to report that the end of the file has been reached. This is documented in *wxFile::Eof* (p. 506) and *wxFFile::Eof* (p. 496) and the behaviour of the stream classes reflects this difference, i.e. wxFileInputStream will report wxSTREAM_EOF after having read the last byte whereas wxFFileInputStream will report wxSTREAM_EOF after trying to read *past* the last byte. Related to EOF behavior, note that *SeekI()* (p. 816) can seek beyond the end of the stream (file) and will thus not return *wxInvalidOffset* for that.

Derived from

wxInputStream (p. 814)

Include files

<wx/wfstream.h>

See also

wxBufferedInputStream (p. 113), *wxFFileOutputStream* (p. 501), *wxFileOutputStream* (p. 538)

wxFFileInputStream::wxFFileInputStream

wxFFileInputStream(**const wxString&** *filename*, **const wxChar *** *mode* = *"rb"*)

Opens the specified file using its *filename* name using the specified mode.

wxFFileInputStream(**wxFFile&** *file*)

Initializes a file stream in read-only mode using the file I/O object *file*.

wxFFileInputStream(**FILE *** *fp*)

Initializes a file stream in read-only mode using the specified file pointer *fp*.

wxFFileInputStream::~wxFFileInputStream

~wxFFileInputStream()

Destructor.

wxFFileInputStream::Ok

bool Ok() **const**

Returns true if the stream is initialized and ready.

wxFFileOutputStream

This class represents data written to a file. There are actually two such groups of classes: this one is based on *wxFFile* (p. 494) whereas *wxFileInputStream* (p. 500) is based in the *wxFile* (p. 502) class.

Note that *wxFile* (p. 502) and *wxFFile* (p. 494) differ in one aspect, namely when to report that the end of the file has been reached. This is documented in *wxFile::Eof* (p. 506) and *wxFFile::Eof* (p. 496) and the behaviour of the stream classes reflects this difference, i.e. wxFileInputStream will report wxSTREAM_EOF after having read the last byte whereas wxFFileInputStream will report wxSTREAM_EOF after trying to read *past* the last byte. Related to EOF behavior, note that *SeekO()* (p. 1017) can seek beyond the end of the stream (file) and will thus not return *wxInvalidOffset* for that.

Derived from

wxOutputStream (p. 1016)

Include files

<wx/wfstream.h>

See also

wxBufferedOutputStream (p. 114), *wxFFileInputStream* (p. 500), *wxFileInputStream* (p. 520)

wxFFileOutputStream::wxFFileOutputStream

wxFFileOutputStream(**const wxString&** *filename*, **const wxChar *** *mode="w+b"*)

Opens the file with the given *filename* name in the specified mode.

wxFFileOutputStream(**wxFFile&** *file*)

Initializes a file stream in write-only mode using the file I/O object *file*.

wxFFileOutputStream(**FILE *** *fp*)

Initializes a file stream in write-only mode using the file descriptor *fp*.

wxFFileOutputStream::~wxFFileOutputStream

~wxFFileOutputStream()

Destructor.

wxFFileOutputStream::Ok

bool Ok() **const**

Returns true if the stream is initialized and ready.

wxFFileStream

Derived from

wxFFileOutputStream (p. 501), *wxFFileInputStream* (p. 500)

Include files

<wx/wfstream.h>

See also

wxStreamBuffer (p. 1277)

wxFFileStream::wxFFileStream

wxFFileStream(**const wxString&** *iofileName*)

Initializes a new file stream in read-write mode using the specified *iofilename* name.

wxFile

A wxFile performs raw file I/O. This is a very small class designed to minimize the overhead of using it - in fact, there is hardly any overhead at all, but using it brings you automatic error checking and hides differences between platforms and compilers. wxFile also automatically closes the file in its destructor making it unnecessary to worry about forgetting to do it. wxFile is a wrapper around `file descriptor.` - see also *wxFFile* (p. 494) for a wrapper around `FILE` structure.

`wxFileOffset` is used by the wxFile functions which require offsets as parameter or return them. If the platform supports it, wxFileOffset if a typedef for a native 64 bit integer, else a 32 bit integer is used for wxFileOffset.

Derived from

None.

Include files

<wx/file.h>

Constants

wx/file.h defines the following constants:

```
#define wxS_IRUSR 00400
#define wxS_IWUSR 00200
#define wxS_IXUSR 00100

#define wxS_IRGRP 00040
#define wxS_IWGRP 00020
#define wxS_IXGRP 00010

#define wxS_IROTH 00004
#define wxS_IWOTH 00002
#define wxS_IXOTH 00001

// default mode for the new files: corresponds to umask 022
#define wxS_DEFAULT  (wxS_IRUSR | wxS_IWUSR | wxS_IRGRP | wxS_IWGRP |
wxS_IROTH | wxS_IWOTH)
```

These constants define the file access rights and are used with *wxFile::Create* (p. 505) and *wxFile::Open* (p. 507).

The *OpenMode* enumeration defines the different modes for opening a file, it is defined inside wxFile class so its members should be specified with *wxFile::* scope resolution prefix. It is also used with *wxFile::Access* (p. 505) function.

wxFile::read	Open file for reading or test if it can be opened for reading with Access()
wxFile::write	Open file for writing deleting the contents of the file if it already exists or test if it can be opened for writing with Access()
wxFile::read_write	Open file for reading and writing; can not be used with Access()
wxFile::write_append	Open file for appending: the file is opened for writing, but the old contents of the file is not erased and the file pointer is initially placed at the end of the file; can not be used with Access(). This is the same as **wxFile::write** if the file doesn't exist.

wxFile::write_excl	Open the file securely for writing (Uses O_EXCL | O_CREAT). Will fail if the file already exists, else create and open it atomically. Useful for opening temporary files without being vulnerable to race exploits.

Other constants defined elsewhere but used by wxFile functions are wxInvalidOffset which represents an invalid value of type *wxFileOffset* and is returned by functions returning *wxFileOffset* on error and the seek mode constants used with *Seek()* (p. 508):

wxFromStart	Count offset from the start of the file
wxFromCurrent	Count offset from the current position of the file pointer
wxFromEnd	Count offset from the end of the file (backwards)

wxFile::wxFile

wxFile()

Default constructor.

wxFile(**const char*** *filename*, **wxFile::OpenMode** *mode = wxFile::read*)

Opens a file with the given mode. As there is no way to return whether the operation was successful or not from the constructor you should test the return value of *IsOpened* (p. 507) to check that it didn't fail.

wxFile(**int** *fd*)

Associates the file with the given file descriptor, which has already been opened.

Parameters

filename

> The filename.

mode

> The mode in which to open the file. May be one of **wxFile::read**, **wxFile::write** and **wxFile::read_write**.

fd

> An existing file descriptor (see *Attach()* (p. 505) for the list of predefined descriptors)

wxFile::~wxFile

~wxFile()

Destructor will close the file.

NB: it is not virtual so you should not use wxFile polymorphically.

wxFile::Access

static bool Access(**const char *** *name*, **OpenMode** *mode*)

This function verifies if we may access the given file in specified mode. Only values of wxFile::read or wxFile::write really make sense here.

wxFile::Attach

void Attach(**int** *fd*)

Attaches an existing file descriptor to the wxFile object. Example of predefined file descriptors are 0, 1 and 2 which correspond to stdin, stdout and stderr (and have symbolic names of **wxFile::fd_stdin**, **wxFile::fd_stdout** and **wxFile::fd_stderr**).

The descriptor should be already opened and it will be closed by wxFile object.

wxFile::Close

void Close()

Closes the file.

wxFile::Create

bool Create(**const char*** *filename*, **bool** *overwrite = false*, **int** *access = wxS_DEFAULT*)

Creates a file for writing. If the file already exists, setting **overwrite** to true will ensure it is overwritten.

wxFile::Detach

void Detach()

Get back a file descriptor from wxFile object - the caller is responsible for closing the file if this descriptor is opened. *IsOpened()* (p. 507) will return false after call to Detach().

wxFile::fd

int fd() **const**

Returns the file descriptor associated with the file.

wxFile::Eof

bool Eof() **const**

Returns true if the end of the file has been reached.

Note that the behaviour of the file pointer based class *wxFFile* (p. 494) is different as *wxFFile::Eof* (p. 496) will return true here only if an attempt has been made to read *past* the last byte of the file, while wxFile::Eof() will return true even before such attempt is made if the file pointer is at the last position in the file.

Note also that this function doesn't work on unseekable file descriptors (examples include pipes, terminals and sockets under Unix) and an attempt to use it will result in an error message in such case. So, to read the entire file into memory, you should write a loop which uses *Read* (p. 507) repeatedly and tests its return condition instead of using Eof() as this will not work for special files under Unix.

wxFile::Exists

static bool Exists(**const char*** *filename*)

Returns true if the given name specifies an existing regular file (not a directory or a link)

wxFile::Flush

bool Flush()

Flushes the file descriptor.

Note that wxFile::Flush is not implemented on some Windows compilers due to a missing fsync function, which reduces the usefulness of this function (it can still be called but it will do nothing on unsupported compilers).

wxFile::GetKind

wxFileKind GetKind() **const**

Returns the type of the file. Possible return values are:

```
enum wxFileKind
{
  wxFILE_KIND_UNKNOWN,
  wxFILE_KIND_DISK,     // a file supporting seeking to arbitrary offsets
  wxFILE_KIND_TERMINAL, // a tty
  wxFILE_KIND_PIPE      // a pipe
};
```

wxFile::IsOpened

bool IsOpened() **const**

Returns true if the file has been opened.

wxFile::Length

wxFileOffset Length() **const**

Returns the length of the file.

wxFile::Open

bool Open(**const char*** *filename*, **wxFile::OpenMode** *mode = wxFile::read*)

Opens the file, returning true if successful.

Parameters

filename

> The filename.

mode

> The mode in which to open the file. May be one of **wxFile::read**, **wxFile::write** and **wxFile::read_write**.

wxFile::Read

size_t Read(**void*** *buffer*, **size_t** *count*)

Reads the specified number of bytes into a buffer, returning the actual number read.

Parameters

buffer

> A buffer to receive the data.

count

> The number of bytes to read.

Return value

The number of bytes read, or the symbol **wxInvalidOffset** (-1) if there was an error.

wxFile::Seek

wxFileOffset Seek(**wxFileOffset** *ofs*, **wxSeekMode** *mode = wxFromStart*)

Seeks to the specified position.

Parameters

ofs

> Offset to seek to.

mode

> One of **wxFromStart**, **wxFromEnd**, **wxFromCurrent**.

Return value

The actual offset position achieved, or wxInvalidOffset on failure.

wxFile::SeekEnd

wxFileOffset SeekEnd(**wxFileOffset** *ofs = 0*)

Moves the file pointer to the specified number of bytes before the end of the file.

Parameters

ofs

> Number of bytes before the end of the file.

Return value

The actual offset position achieved, or wxInvalidOffset on failure.

wxFile::Tell

wxFileOffset Tell() **const**

Returns the current position or wxInvalidOffset if file is not opened or if another error occurred.

wxFile::Write

size_t Write(**const void*** *buffer*, **wxFileOffset** *count*)

Writes the specified number of bytes from a buffer.

Parameters

buffer

>   A buffer containing the data.

count

>   The number of bytes to write.

Return value

the number of bytes actually written

wxFile::Write

bool Write(**const wxString&** *s*, **wxMBConv&** *conv = wxConvUTF8*)

Writes the contents of the string to the file, returns true on success.

The second argument is only meaningful in Unicode build of wxWidgets when *conv* is used to convert *s* to multibyte representation.

Note that this method only works with NUL-terminated strings, if you want to write data with embedded NULs to the file you should use the other *Write() overload* (p. 508).

wxFileConfig

wxFileConfig implements *wxConfigBase* (p. 193) interface for storing and retrieving configuration information using plain text files. The files have a simple format reminiscent of Windows INI files with lines of the form key = value defining the keys and lines of special form [group] indicating the start of each group.

This class is used by default for wxConfig on Unix platforms but may also be used explicitly if you want to use files and not the registry even under Windows.

Derived from

wxConfigBase (p. 193)

Include files

<wx/fileconf.h>

wxFileConfig::wxFileConfig

wxFileConfig(**wxInputStream&** *is*, **wxMBConv&** *conv = wxConvUTF8*)

Read the config data from the specified stream instead of the associated file, as usual.

See also

Save (p. 510)

wxFileConfig::Save

bool Save(**wxOutputStream&** *os*, **wxMBConv&** *conv = wxConvUTF8*)

Saves all config data to the given stream, returns `true` if data was saved successfully or `false` on error.

Note the interaction of this function with the internal "dirty flag": the data is saved unconditionally, i.e. even if the object is not dirty. However after saving it successfully, the dirty flag is reset so no changes will be written back to the file this object is associated with until you change its contents again.

See also

Flush (p. 201)

wxFileConfig::SetUmask

void SetUmask(**int** *mode*)

Allows to set the mode to be used for the config file creation. For example, to create a config file which is not readable by other users (useful if it stores some sensitive information, such as passwords), you could use `SetUmask(0077)`.

This function doesn't do anything on non-Unix platforms.

See also

wxCHANGE_UMASK (p. 1602)

wxFileDataObject

wxFileDataObject is a specialization of *wxDataObject* (p. 239) for file names. The program works with it just as if it were a list of absolute file names, but internally it uses the same format as Explorer and other compatible programs under Windows or GNOME/KDE filemanager under Unix which makes it possible to receive files from them using this class.

Warning: Under all non-Windows platforms this class is currently "input-only", i.e. you can receive the files from another application, but copying (or dragging) file(s) from a wxWidgets application is not currently supported. PS: GTK2 should work as well.

Virtual functions to override

None.

wxDataObjectSimple (p. 243)
wxDataObject (p. 239)

Include files

<wx/dataobj.h>

See also

wxDataObject (p. 239), *wxDataObjectSimple* (p. 243), *wxTextDataObject* (p. 1359), *wxBitmapDataObject* (p. 97), *wxDataObject* (p. 239)

wxFileDataObject

wxFileDataObject()

Constructor.

wxFileDataObject::AddFile

virtual void AddFile(const wxString& *file*)

MSW only: adds a file to the file list represented by this data object.

wxFileDataObject::GetFilenames

const wxArrayString& GetFilenames() const

Returns the *array* (p. 65) of file names.

wxFileDialog

This class represents the file chooser dialog.

Derived from

wxDialog (p. 406)
wxWindow (p. 1490)
wxEvtHandler (p. 486)
wxObject (p. 1012)

Include files

<wx/filedlg.h>

See also

wxFileDialog overview (p. 1800), *wxFileSelector* (p. 1615)

Remarks

Pops up a file selector box. In Windows and GTK2.4+, this is the common file selector dialog. In X, this is a file selector box with somewhat less functionality. The path and filename are distinct elements of a full file pathname. If path is "", the current directory will be used. If filename is "", no default filename will be supplied. The wildcard determines what files are displayed in the file selector, and file extension supplies a type extension for the required filename. Flags may be a combination of wxOPEN, wxSAVE, wxOVERWRITE_PROMPT, wxHIDE_READONLY, wxFILE_MUST_EXIST, wxMULTIPLE, wxCHANGE_DIR or 0.

Both the X and Windows versions implement a wildcard filter. Typing a filename containing wildcards (*, ?) in the filename text item, and clicking on Ok, will result in only those files matching the pattern being displayed. The wildcard may be a specification for multiple types of file with a description for each, such as:

```
"BMP and GIF files (*.bmp;*.gif)|*.bmp;*.gif|PNG files (*.png)|*.png"
```

It must be noted that wildcard support in the native Motif file dialog is quite limited: only one alternative is supported, and it is displayed without the descriptive test; "BMP files (*.bmp)|*.bmp" is displayed as "*.bmp", and both "BMP files (*.bmp)|*.bmp|GIF files (*.gif)|*.gif" and "Image files|*.bmp;*.gif" are errors.

wxFileDialog::wxFileDialog

 wxFileDialog(**wxWindow*** *parent*, **const wxString&** *message = "Choose a file"*, **const wxString&** *defaultDir = ""*, **const wxString&** *defaultFile = ""*, **const wxString&** *wildcard = "*.*"*, **long** *style = 0*, **const wxPoint&** *pos = wxDefaultPosition*)

Constructor. Use *wxFileDialog::ShowModal* (p. 516) to show the dialog.

Parameters

parent

> Parent window.

message

> Message to show on the dialog.

defaultDir

> The default directory, or the empty string.

defaultFile

The default filename, or the empty string.

wildcard

A wildcard, such as "*.*" or "BMP files (*.bmp)|*.bmp|GIF files (*.gif)|*.gif".

Note that the native Motif dialog has some limitations with respect to wildcards; see the Remarks section above.

style

A dialog style. A bitlist of:

wxOPEN	This is an open dialog.
wxSAVE	This is a save dialog.
wxOVERWRITE_PROMPT	For save dialog only: prompt for a confirmation if a file will be overwritten.
wxHIDE_READONLY	Do not display read-only files.
wxFILE_MUST_EXIST	The user may only select files that actually exist.
wxMULTIPLE	For open dialog only: allows selecting multiple files.
wxCHANGE_DIR	Change the current working directory to the directory where the file(s) chosen by the user are.

pos

Dialog position. Not implemented.

NB: Previous versions of wxWidgets used wxCHANGE_DIR by default under MS Windows which allowed the program to simply remember the last directory where user selected the files to open/save. This (desired) functionality must be implemented in the program itself now (manually remember the last path used and pass it to the dialog the next time it is called) or by using this flag.

wxFileDialog::~wxFileDialog

 ~wxFileDialog()

Destructor.

wxFileDialog::GetDirectory

wxString GetDirectory() **const**

Returns the default directory.

wxFileDialog::GetFilename

wxString GetFilename() **const**

Returns the default filename.

wxFileDialog::GetFilenames

void GetFilenames(**wxArrayString&** *filenames*) **const**

Fills the array *filenames* with the names of the files chosen. This function should only be used with the dialogs which have `wxMULTIPLE` style, use *GetFilename* (p. 514) for the others.

Note that under Windows, if the user selects shortcuts, the filenames include paths, since the application cannot determine the full path of each referenced file by appending the directory containing the shortcuts to the filename.

wxFileDialog::GetFilterIndex

int GetFilterIndex() **const**

Returns the index into the list of filters supplied, optionally, in the wildcard parameter. Before the dialog is shown, this is the index which will be used when the dialog is first displayed. After the dialog is shown, this is the index selected by the user.

wxFileDialog::GetMessage

wxString GetMessage() **const**

Returns the message that will be displayed on the dialog.

wxFileDialog::GetPath

wxString GetPath() **const**

Returns the full path (directory and filename) of the selected file.

wxFileDialog::GetPaths

void GetPaths(**wxArrayString&** *paths*) **const**

Fills the array *paths* with the full paths of the files chosen. This function should only be used with the dialogs which have `wxMULTIPLE` style, use *GetPath* (p. 514) for the others.

wxFileDialog::GetStyle

long GetStyle() **const**

Returns the dialog style.

wxFileDialog::GetWildcard

wxString GetWildcard() **const**

Returns the file dialog wildcard.

wxFileDialog::SetDirectory

void SetDirectory(**const wxString&** *directory*)

Sets the default directory.

wxFileDialog::SetFilename

void SetFilename(**const wxString&** *setfilename*)

Sets the default filename.

wxFileDialog::SetFilterIndex

void SetFilterIndex(**int** *filterIndex*)

Sets the default filter index, starting from zero.

wxFileDialog::SetMessage

void SetMessage(**const wxString&** *message*)

Sets the message that will be displayed on the dialog.

wxFileDialog::SetPath

void SetPath(**const wxString&** *path*)

Sets the path (the combined directory and filename that will be returned when the dialog is dismissed).

wxFileDialog::SetStyle

void SetStyle(**long** *style*)

Sets the dialog style. See *wxFileDialog::wxFileDialog* (p. 512) for details.

wxFileDialog::SetWildcard

void SetWildcard(**const wxString&** *wildCard*)

Sets the wildcard, which can contain multiple file types, for example:

"BMP files (*.bmp)|*.bmp|GIF files (*.gif)|*.gif"

Note that the native Motif dialog has some limitations with respect to wildcards; see the Remarks section above.

wxFileDialog::ShowModal

int ShowModal()

Shows the dialog, returning wxID_OK if the user pressed OK, and wxID_CANCEL otherwise.

wxFileDropTarget

This is a *drop target* (p. 471) which accepts files (dragged from File Manager or Explorer).

Derived from

wxDropTarget (p. 471)

Include files

<wx/dnd.h>

See also

Drag and drop overview (p. 1817), *wxDropSource* (p. 468), *wxDropTarget* (p. 471), *wxTextDropTarget* (p. 1360)

wxFileDropTarget::wxFileDropTarget

wxFileDropTarget()

Constructor.

wxFileDropTarget::OnDrop

virtual bool OnDrop(**long** *x*, **long** *y*, **const void** **data*, **size_t** *size*)

See *wxDropTarget::OnDrop* (p. 472). This function is implemented appropriately for files, and calls *wxFileDropTarget::OnDropFiles* (p. 517).

wxFileDropTarget::OnDropFiles

virtual bool OnDropFiles(**wxCoord** *x*, **wxCoord** *y*, **const wxArrayString&** *filenames*)

Override this function to receive dropped files.

Parameters

x

The x coordinate of the mouse.

y

The y coordinate of the mouse.

filenames

An array of filenames.

Return value

Return true to accept the data, false to veto the operation.

wxFileHistory

The wxFileHistory encapsulates a user interface convenience, the list of most recently visited files as shown on a menu (usually the File menu).

wxFileHistory can manage one or more file menus. More than one menu may be required in an MDI application, where the file history should appear on each MDI child menu as well as the MDI parent frame.

Derived from

wxObject (p. 1012)

Include files

<wx/docview.h>

See also

wxFileHistory overview (p. 1806), *wxDocManager* (p. 435)

wxFileHistory::m_fileHistory

char** m_fileHistory

A character array of strings corresponding to the most recently opened files.

wxFileHistory::m_fileHistoryN

size_t m_fileHistoryN

The number of files stored in the history array.

wxFileHistory::m_fileMaxFiles

size_t m_fileMaxFiles

The maximum number of files to be stored and displayed on the menu.

wxFileHistory::m_fileMenu

wxMenu* m_fileMenu

The file menu used to display the file history list (if enabled).

wxFileHistory::wxFileHistory

wxFileHistory(**size_t** *maxFiles = 9*, **wxWindowID** *idBase = wxID_FILE1*)

Constructor. Pass the maximum number of files that should be stored and displayed.

idBase defaults to wxID_FILE1 and represents the id given to the first history menu item. Since menu items can't share the same ID you should change idBase (To one of your own defined IDs) when using more than one wxFileHistory in your application.

wxFileHistory::~wxFileHistory

~wxFileHistory()

Destructor.

wxFileHistory::AddFileToHistory

void AddFileToHistory(**const wxString&** *filename*)

Adds a file to the file history list, if the object has a pointer to an appropriate file menu.

wxFileHistory::AddFilesToMenu

void AddFilesToMenu()

Appends the files in the history list, to all menus managed by the file history object.

void AddFilesToMenu(wxMenu* *menu*)

Appends the files in the history list, to the given menu only.

wxFileHistory::GetHistoryFile

wxString GetHistoryFile(size_t *index*) const

Returns the file at this index (zero-based).

wxFileHistory::GetMaxFiles

size_t GetMaxFiles() const

Returns the maximum number of files that can be stored.

wxFileHistory::GetCount

size_t GetCount() const

Returns the number of files currently stored in the file history.

wxFileHistory::Load

void Load(wxConfigBase& *config*)

Loads the file history from the given config object. This function should be called explicitly by the application.

See also

wxConfig (p. 193)

wxFileHistory::RemoveMenu

void RemoveMenu(wxMenu* *menu*)

Removes this menu from the list of those managed by this object.

wxFileHistory::Save

void Save(**wxConfigBase&** *config*)

Saves the file history into the given config object. This must be called explicitly by the application.

See also

wxConfig (p. 193)

wxFileHistory::UseMenu

void UseMenu(**wxMenu*** *menu*)

Adds this menu to the list of those managed by this object.

wxFileInputStream

This class represents data read in from a file. There are actually two such groups of classes: this one is based on *wxFile* (p. 502) whereas *wxFFileInputStream* (p. 500) is based in the *wxFFile* (p. 494) class.

Note that *wxFile* (p. 502) and *wxFFile* (p. 494) differ in one aspect, namely when to report that the end of the file has been reached. This is documented in *wxFile::Eof* (p. 506) and *wxFFile::Eof* (p. 496) and the behaviour of the stream classes reflects this difference, i.e. wxFileInputStream will report wxSTREAM_EOF after having read the last byte whereas wxFFileInputStream will report wxSTREAM_EOF after trying to read *past* the last byte. Related to EOF behavior, note that *SeekI()* (p. 816) can seek beyond the end of the stream (file) and will thus not return *wxInvalidOffset* for that.

Derived from

wxInputStream (p. 814)

Include files

<wx/wfstream.h>

See also

wxBufferedInputStream (p. 113), *wxFileOutputStream* (p. 538), *wxFFileOutputStream* (p. 501)

wxFileInputStream::wxFileInputStream

 wxFileInputStream(**const wxString&** *ifileName*)

Opens the specified file using its *ifilename* name in read-only mode.

wxFileInputStream(**wxFile&** *file*)

Initializes a file stream in read-only mode using the file I/O object *file*.

wxFileInputStream(**int** *fd*)

Initializes a file stream in read-only mode using the specified file descriptor.

wxFileInputStream::~wxFileInputStream

~wxFileInputStream()

Destructor.

wxFileInputStream::Ok

bool Ok() **const**

Returns true if the stream is initialized and ready.

wxFileName

wxFileName encapsulates a file name. This class serves two purposes: first, it provides the functions to split the file names into components and to recombine these components in the full file name which can then be passed to the OS file functions (and *wxWidgets functions* (p. 1599) wrapping them). Second, it includes the functions for working with the files itself. Note that to change the file data you should use *wxFile* (p. 502) class instead. wxFileName provides functions for working with the file attributes.

Derived from

No base class

Include files

<wx/filename.h>

Data structures

Many wxFileName methods accept the path format argument which is by `wxPATH_NATIVE` by default meaning to use the path format native for the current platform.

The path format affects the operation of wxFileName functions in several ways: first and foremost, it defines the path separator character to use, but it also affects other things such as whether the path has the drive part or not.

```
enum wxPathFormat
```

```
{
    wxPATH_NATIVE = 0,         // the path format for the current platform
    wxPATH_UNIX,
    wxPATH_BEOS = wxPATH_UNIX,
    wxPATH_MAC,
    wxPATH_DOS,
    wxPATH_WIN = wxPATH_DOS,
    wxPATH_OS2 = wxPATH_DOS,
    wxPATH_VMS,

    wxPATH_MAX // Not a valid value for specifying path format
}
```

File name format

wxFileName currently supports the file names in the Unix, DOS/Windows, Mac OS and VMS formats. Although these formats are quite different, wxFileName tries to treat them all in the same generic way. It supposes that all file names consist of the following parts: the volume (also known as drive under Windows or device under VMS), the path which is a sequence of directory names separated by the *path separators* (p. 529) and the full filename itself which, in turn, is composed from the base file name and the extension. All of the individual components of the file name may be empty and, for example, the volume name is always empty under Unix, but if they are all empty simultaneously, the filename object is considered to be in an invalid state and *IsOk* (p. 531) returns `false` for it.

File names can be case-sensitive or not, the function *IsCaseSensitive* (p. 531) allows to determine this.

The rules for determining if the file name is absolute or relative also depends on the file name format and the only portable way to answer to this question is to use *IsAbsolute* (p. 531) method. To ensure that the filename is absolute you may use *MakeAbsolute* (p. 533). There is also an inverse function *MakeRelativeTo* (p. 533) which undoes what*Normalize(wxPATH_NORM_DOTS)* (p. 534) does.

Other functions returning information about the file format provided by this class are *GetVolumeSeparator* (p. 530), *IsPathSeparator* (p. 532).

IsRelative (p. 532)

File name construction

TODO.

File tests

Before doing the other tests you should use *IsOk* (p. 531) to verify that the filename is well defined. If it is, *FileExists* (p. 527) can be used to test if a file with such name exists and *DirExists* (p. 526) - if a directory with this name exists.

File names should be compared using *SameAs* (p. 535) method or == (p. 538).

File name components

These functions allow to examine and modify the individual directories of the path:

AppendDir (p. 524)
InsertDir (p. 531)
GetDirCount (p. 527)*PrependDir* (p. 534)
RemoveDir (p. 535)
RemoveLastDir (p. 535)

To change the components of the file name individually you can use the following functions:

GetExt (p. 528)
GetName (p. 529)
GetVolume (p. 530)
HasExt (p. 531)
HasName (p. 531)
HasVolume (p. 531)
SetExt (p. 535)
ClearExt (p. 525)
SetEmptyExt (p. 536)
SetName (p. 536)
SetVolume (p. 536)

Operations

These methods allow to work with the file creation, access and modification times. Note that not all filesystems under all platforms implement these times in the same way. For example, the access time under Windows has a resolution of one day (so it is really the access date and not time). The access time may be updated when the file is executed or not depending on the platform.

GetModificationTime (p. 529)
GetTimes (p. 530)
SetTimes (p. 536)
Touch (p. 537)

Other file system operations functions are:

Mkdir (p. 533)
Rmdir (p. 535)

wxFileName::wxFileName

wxFileName()

Default constructor.

wxFileName(**const wxFileName&** *filename*)

Copy constructor.

wxFileName(**const wxString&** *fullpath*, **wxPathFormat** *format = wxPATH_NATIVE*)

Constructor taking a full filename. If it terminates with a '/', a directory path is constructed (the name will be empty), otherwise a file name and extension are extracted from it.

wxFileName(**const wxString&** *path*, **const wxString&** *name*, **wxPathFormat** *format = wxPATH_NATIVE*)

Constructor from a directory name and a file name.

wxFileName(**const wxString&** *path*, **const wxString&** *name*, **const wxString&** *ext*, **wxPathFormat** *format = wxPATH_NATIVE*)

Constructor from a directory name, base file name and extension.

wxFileName(**const wxString&** *volume*, **const wxString&** *path*, **const wxString&** *name*, **const wxString&** *ext*, **wxPathFormat** *format = wxPATH_NATIVE*)

Constructor from a volume name, a directory name, base file name and extension.

wxFileName::AppendDir

void AppendDir(**const wxString&** *dir*)

Appends a directory component to the path. This component should contain a single directory name level, i.e. not contain any path or volume separators nor should it be empty, otherwise the function does nothing (and generates an assert failure in debug build).

wxFileName::Assign

void Assign(**const wxFileName&** *filepath*)

void Assign(**const wxString&** *fullpath*, **wxPathFormat** *format = wxPATH_NATIVE*)

void Assign(**const wxString&** *volume*, **const wxString&** *path*, **const wxString&** *name*, **const wxString&** *ext*, **bool** *hasExt*, **wxPathFormat** *format = wxPATH_NATIVE*)

void Assign(**const wxString&** *volume*, **const wxString&** *path*, **const wxString&** *name*, **const wxString&** *ext*, **wxPathFormat** *format = wxPATH_NATIVE*)

void Assign(**const wxString&** *path*, **const wxString&** *name*, **wxPathFormat** *format = wxPATH_NATIVE*)

void Assign(**const wxString&** *path*, **const wxString&** *name*, **const wxString&** *ext*, **wxPathFormat** *format = wxPATH_NATIVE*)

Creates the file name from various combinations of data.

wxFileName::AssignCwd

static void AssignCwd(**const wxString&** *volume = wxEmptyString*)

Makes this object refer to the current working directory on the specified volume (or current volume if *volume* is empty).

See also

GetCwd (p. 527)

wxFileName::AssignDir

void AssignDir(**const wxString&** *dir*, **wxPathFormat** *format = wxPATH_NATIVE*)

Sets this file name object to the given directory name. The name and extension will be empty.

wxFileName::AssignHomeDir

void AssignHomeDir()

Sets this file name object to the home directory.

wxFileName::AssignTempFileName

void AssignTempFileName(**const wxString&** *prefix*, **wxFile** **fileTemp = NULL*)

The function calls *CreateTempFileName* (p. 526) to create a temporary file and sets this object to the name of the file. If a temporary file couldn't be created, the object is put into the *invalid* (p. 531) state.

wxFileName::Clear

void Clear()

Reset all components to default, uninitialized state.

wxFileName::ClearExt

void SetClearExt()

Removes the extension from the file name resulting in a file name with no trailing dot.

See also

SetExt (p. 535)*SetEmptyExt* (p. 536)

wxFileName::CreateTempFileName

static wxString CreateTempFileName(**const wxString&** *prefix*, **wxFile ****fileTemp* = *NULL*)

Returns a temporary file name starting with the given *prefix*. If the *prefix* is an absolute path, the temporary file is created in this directory, otherwise it is created in the default system directory for the temporary files or in the current directory.

If the function succeeds, the temporary file is actually created. If *fileTemp* is not NULL, this file will be opened using the name of the temporary file. When possible, this is done in an atomic way ensuring that no race condition occurs between the temporary file name generation and opening it which could often lead to security compromise on the multiuser systems. If *fileTemp* is NULL, the file is only created, but not opened.

Under Unix, the temporary file will have read and write permissions for the owner only to minimize the security problems.

Parameters

prefix

>   Prefix to use for the temporary file name construction

fileTemp

>   The file to open or NULL to just get the name

Return value

The full temporary file name or an empty string on error.

wxFileName::DirExists

bool DirExists() **const**

static bool DirExists(**const wxString&** *dir*)

Returns true if the directory with this name exists.

wxFileName::DirName

static wxFileName DirName(**const wxString&** *dir*, **wxPathFormat** *format* = *wxPATH_NATIVE*)

Returns the object corresponding to the directory with the given name. The *dir* parameter may have trailing path separator or not.

wxFileName::FileExists

bool FileExists() **const**

static bool FileExists(**const wxString&** *file*)

Returns `true` if the file with this name exists.

See also

DirExists (p. 526)

wxFileName::FileName

static wxFileName FileName(**const wxString&** *file*, **wxPathFormat** *format* = *wxPATH_NATIVE*)

Returns the file name object corresponding to the given *file*. This function exists mainly for symmetry with *DirName* (p. 526).

wxFileName::GetCwd

static wxString GetCwd(**const wxString&** *volume = ""*)

Retrieves the value of the current working directory on the specified volume. If the volume is empty, the program's current working directory is returned for the current volume.

Return value

The string containing the current working directory or an empty string on error.

See also

AssignCwd (p. 525)

wxFileName::GetDirCount

size_t GetDirCount() **const**

Returns the number of directories in the file name.

wxFileName::GetDirs

const wxArrayString& GetDirs() **const**

Returns the directories in string array form.

wxFileName::GetExt

wxString GetExt() **const**

Returns the file name extension.

wxFileName::GetForbiddenChars

static wxString GetForbiddenChars(**wxPathFormat** *format = wxPATH_NATIVE*)

Returns the characters that can't be used in filenames and directory names for the specified format.

wxFileName::GetFormat

static wxPathFormat GetFormat(**wxPathFormat** *format = wxPATH_NATIVE*)

Returns the canonical path format for this platform.

wxFileName::GetFullName

wxString GetFullName() **const**

Returns the full name (including extension but excluding directories).

wxFileName::GetFullPath

wxString GetFullPath(**wxPathFormat** *format = wxPATH_NATIVE*) **const**

Returns the full path with name and extension.

wxFileName::GetHomeDir

static wxString GetHomeDir()

Returns the home directory.

wxFileName::GetLongPath

wxString GetLongPath() **const**

Return the long form of the path (returns identity on non-Windows platforms)

wxFileName::GetModificationTime

wxDateTime GetModificationTime() **const**

Returns the last time the file was last modified.

wxFileName::GetName

wxString GetName() **const**

Returns the name part of the filename.

wxFileName::GetPath

wxString GetPath(**int** *flags* = *wxPATH_GET_VOLUME*, **wxPathFormat** *format* = *wxPATH_NATIVE*) **const**

Returns the path part of the filename (without the name or extension). The possible flags values are:

wxPATH_GET_VOLUME   Return the path with the volume (does nothing for the filename formats without volumes), otherwise the path without volume part is returned.

wxPATH_GET_SEPARATOR   Return the path with the trailing separator, if this flag is not given there will be no separator at the end of the path.

wxFileName::GetPathSeparator

static wxChar GetPathSeparator(**wxPathFormat** *format* = *wxPATH_NATIVE*)

Returns the usually used path separator for this format. For all formats but wxPATH_DOS there is only one path separator anyhow, but for DOS there are two of them and the native one, i.e. the backslash is returned by this method.

See also

GetPathSeparators (p. 529)

wxFileName::GetPathSeparators

static wxString GetPathSeparators(**wxPathFormat** *format* = *wxPATH_NATIVE*)

Returns the string containing all the path separators for this format. For all formats but wxPATH_DOS this string contains only one character but for DOS and Windows both '/' and '\' may be used as separators.

wxFileName::GetPathTerminators

static wxString GetPathTerminators(**wxPathFormat** *format* = *wxPATH_NATIVE*)

Returns the string of characters which may terminate the path part. This is the same as *GetPathSeparators* (p. 529) except for VMS path format where] is used at the end of the path part.

wxFileName::GetShortPath

wxString GetShortPath() **const**

Return the short form of the path (returns identity on non-Windows platforms).

wxFileName::GetTimes

bool GetTimes(**wxDateTime*** *dtAccess*, **wxDateTime*** *dtMod*, **wxDateTime*** *dtCreate*) **const**

Returns the last access, last modification and creation times. The last access time is updated whenever the file is read or written (or executed in the case of Windows), last modification time is only changed when the file is written to. Finally, the creation time is indeed the time when the file was created under Windows and the inode change time under Unix (as it is impossible to retrieve the real file creation time there anyhow) which can also be changed by many operations after the file creation.

Any of the pointers may be `NULL` if the corresponding time is not needed.

Return value

`true` on success, `false` if we failed to retrieve the times.

wxFileName::GetVolume

wxString GetVolume() **const**

Returns the string containing the volume for this file name, empty if it doesn't have one or if the file system doesn't support volumes at all (for example, Unix).

wxFileName::GetVolumeSeparator

static wxString GetVolumeSeparator(**wxPathFormat** *format* = *wxPATH_NATIVE*)

Returns the string separating the volume from the path for this format.

wxFileName::HasExt

bool HasExt() **const**

Returns `true` if an extension is present.

wxFileName::HasName

bool HasName() **const**

Returns `true` if a name is present.

wxFileName::HasVolume

bool HasVolume() **const**

Returns `true` if a volume specifier is present.

wxFileName::InsertDir

void InsertDir(**size_t** *before*, **const wxString&** *dir*)

Inserts a directory component before the zero-based position in the directory list. Please see *AppendDir* (p. 524) for important notes.

wxFileName::IsAbsolute

bool IsAbsolute(**wxPathFormat** *format = wxPATH_NATIVE*)

Returns `true` if this filename is absolute.

wxFileName::IsCaseSensitive

static bool IsCaseSensitive(**wxPathFormat** *format = wxPATH_NATIVE*)

Returns `true` if the file names of this type are case-sensitive.

wxFileName::IsOk

bool IsOk() **const**

Returns `true` if the filename is valid, `false` if it is not initialized yet. The assignment

functions and*Clear* (p. 525) may reset the object to the uninitialized, invalid state (the former only do it on failure).

wxFileName::IsPathSeparator

static bool IsPathSeparator(**wxChar** *ch*, **wxPathFormat** *format* = *wxPATH_NATIVE*)

Returns `true` if the char is a path separator for this format.

wxFileName::IsRelative

bool IsRelative(**wxPathFormat** *format* = *wxPATH_NATIVE*)

Returns `true` if this filename is not absolute.

wxFileName::IsDir

bool IsDir() **const**

Returns `true` if this object represents a directory, `false` otherwise (i.e. if it is a file). Note that this method doesn't test whether the directory or file really exists, you should use *DirExists* (p. 526) or *FileExists* (p. 527) for this.

wxFileName::MacFindDefaultTypeAndCreator

static bool MacFindDefaultTypeAndCreator(**const wxString&** *ext*, **wxUint32*** *type*, **wxUint32*** *creator*)

On Mac OS, gets the common type and creator for the given extension.

wxFileName::MacRegisterDefaultTypeAndCreator

static void MacRegisterDefaultTypeAndCreator(**const wxString&** *ext*, **wxUint32** *type*, **wxUint32** *creator*)

On Mac OS, registers application defined extensions and their default type and creator.

wxFileName::MacSetDefaultTypeAndCreator

bool MacSetDefaultTypeAndCreator()

On Mac OS, looks up the appropriate type and creator from the registration and then sets it.

wxFileName::MakeAbsolute

bool MakeAbsolute(**const wxString&** *cwd = wxEmptyString*, **wxPathFormat** *format = wxPATH_NATIVE*)

Make the file name absolute. This is a shortcut for`Normalize (p. 534)(wxPATH_NORM_DOTS | wxPATH_NORM_ABSOLUTE | wxPATH_NORM_TILDE, cwd, format)`.

See also

MakeRelativeTo (p. 533),*Normalize* (p. 534),*IsAbsolute* (p. 531)

wxFileName::MakeRelativeTo

bool MakeRelativeTo(**const wxString&** *pathBase = wxEmptyString*, **wxPathFormat** *format = wxPATH_NATIVE*)

This function tries to put this file name in a form relative to *pathBase*. In other words, it returns the file name which should be used to access this file if the current directory were *pathBase*.

pathBase

> the directory to use as root, current directory is used by default

format

> the file name format, native by default

Return value

`true` if the file name has been changed, `false` if we failed to do anything with it (currently this only happens if the file name is on a volume different from the volume specified by *pathBase*).

See also

Normalize (p. 534)

wxFileName::Mkdir

bool Mkdir(**int** *perm = 0777*, **int** *flags = 0*)

static bool Mkdir(**const wxString&** *dir*, **int** *perm = 0777*, **int** *flags = 0*)

dir

> the directory to create

parm

the permissions for the newly created directory

flags

if the flags contain `wxPATH_MKDIR_FULL` flag, try to create each directory in the path and also don't return an error if the target directory already exists.

Return value

Returns `true` if the directory was successfully created, `false`otherwise.

wxFileName::Normalize

bool Normalize(**int** *flags* = *wxPATH_NORM_ALL*, **const wxString&** *cwd* = *wxEmptyString*, **wxPathFormat** *format* = *wxPATH_NATIVE*)

Normalize the path. With the default flags value, the path will be made absolute, without any ".." and "." and all environment variables will be expanded in it.

flags

The kind of normalization to do with the file name. It can be any or-combination of the following constants:

wxPATH_NORM_ENV_VARS replace env vars with their values

wxPATH_NORM_DOTS   squeeze all .. and . and prepend cwd

wxPATH_NORM_TILDE   Unix only: replace ~ and ~user

wxPATH_NORM_CASE   if filesystem is case insensitive, transform to tolower case

wxPATH_NORM_ABSOLUTEmake the path absolute

wxPATH_NORM_LONG   make the path the long form

wxPATH_NORM_SHORTCUTresolve if it is a shortcut (Windows only)

wxPATH_NORM_ALL   all of previous flags except `wxPATH_NORM_CASE`

cwd

If not empty, this directory will be used instead of current working directory in normalization.

format

The file name format, native by default.

wxFileName::PrependDir

void PrependDir(**const wxString&** *dir*)

Prepends a directory to the file path. Please see *AppendDir* (p. 524) for important notes.

wxFileName::RemoveDir

void RemoveDir(**size_t** *pos*)

Removes the specified directory component from the path.

See also

GetDirCount (p. 527)

wxFileName::RemoveLastDir

void RemoveLastDir()

Removes last directory component from the path.

wxFileName::Rmdir

bool Rmdir()

static bool Rmdir(**const wxString&** *dir*)

Deletes the specified directory from the file system.

wxFileName::SameAs

bool SameAs(**const wxFileName&** *filepath*, **wxPathFormat** *format* = *wxPATH_NATIVE*) **const**

Compares the filename using the rules of this platform.

wxFileName::SetCwd

bool SetCwd()

static bool SetCwd(**const wxString&** *cwd*)

Changes the current working directory.

wxFileName::SetExt

void SetExt(**const wxString&** *ext*)

Sets the extension of the file name. Setting an empty string as the extension will remove the extension resulting in a file name without a trailing dot, unlike a call to *SetEmptyExt* (p. 536).

See also

SetEmptyExt (p. 536)*ClearExt* (p. 525)

wxFileName::SetEmptyExt

void SetEmptyExt()

Sets the extension of the file name to be an empty extension. This is different from having no extension at all as the file name will have a trailing dot after a call to this method.

See also

SetExt (p. 535)*ClearExt* (p. 525)

wxFileName::SetFullName

void SetFullName(const wxString& *fullname*)

The full name is the file name and extension (but without the path).

wxFileName::SetName

void SetName(const wxString& *name*)

Sets the name.

wxFileName::SetTimes

bool SetTimes(const wxDateTime* *dtAccess*, const wxDateTime* *dtMod*, const wxDateTime* *dtCreate*)

Sets the file creation and last access/modification times (any of the pointers may be NULL).

wxFileName::SetVolume

void SetVolume(const wxString& *volume*)

Sets the volume specifier.

wxFileName::SplitPath

static void SplitPath(**const wxString&** *fullpath*, **wxString*** *volume*, **wxString*** *path*, **wxString*** *name*, **wxString*** *ext*, **bool** **hasExt* = *NULL*, **wxPathFormat** *format* = *wxPATH_NATIVE*)

static void SplitPath(**const wxString&** *fullpath*, **wxString*** *volume*, **wxString*** *path*, **wxString*** *name*, **wxString*** *ext*, **wxPathFormat** *format* = *wxPATH_NATIVE*)

static void SplitPath(**const wxString&** *fullpath*, **wxString*** *path*, **wxString*** *name*, **wxString*** *ext*, **wxPathFormat** *format* = *wxPATH_NATIVE*)

This function splits a full file name into components: the volume (with the first version) path (including the volume in the second version), the base name and the extension. Any of the output parameters (*volume*, *path*, *name* or *ext*) may be NULL if you are not interested in the value of a particular component. Also, *fullpath* may be empty on entry.

On return, *path* contains the file path (without the trailing separator), *name* contains the file name and *ext* contains the file extension without leading dot. All three of them may be empty if the corresponding component is. The old contents of the strings pointed to by these parameters will be overwritten in any case (if the pointers are not NULL).

Note that for a filename "foo." the extension is present, as indicated by the trailing dot, but empty. If you need to cope with such cases, you should use *hasExt* instead of relying on testing whether *ext* is empty or not.

wxFileName::SplitVolume

static void SplitVolume(**const wxString&** *fullpath*, **wxString*** *volume*, **wxString*** *path*, **wxPathFormat** *format* = *wxPATH_NATIVE*)

Splits the given *fullpath* into the volume part (which may be empty) and the pure path part, not containing any volume.

See also

SplitPath (p. 537)

wxFileName::Touch

bool Touch()

Sets the access and modification times to the current moment.

wxFileName::operator=

wxFileName& operator operator=(**const wxFileName&** *filename*)

wxFileName& operator operator=(**const wxString&** *filename*)

Assigns the new value to this filename object.

wxFileName::operator==

bool operator operator==(const wxFileName& *filename***) const**

bool operator operator==(const wxString& *filename***) const**

Returns `true` if the filenames are equal. The string *filenames* is interpreted as a path in the native filename format.

wxFileName::operator!=

bool operator operator!=(const wxFileName& *filename***) const**

bool operator operator!=(const wxString& *filename***) const**

Returns `true` if the filenames are different. The string *filenames*is interpreted as a path in the native filename format.

wxFileOutputStream

This class represents data written to a file. There are actually two such groups of classes: this one is based on *wxFile* (p. 502) whereas *wxFFileInputStream* (p. 500) is based in the *wxFFile* (p. 494) class.

Note that *wxFile* (p. 502) and *wxFFile* (p. 494) differ in one aspect, namely when to report that the end of the file has been reached. This is documented in *wxFile::Eof* (p. 506) and *wxFFile::Eof* (p. 496) and the behaviour of the stream classes reflects this difference, i.e. wxFileInputStream will report wxSTREAM_EOF after having read the last byte whereas wxFFileInputStream will report wxSTREAM_EOF after trying to read *past* the last byte. Related to EOF behavior, note that *SeekO()* (p. 1017) can seek beyond the end of the stream (file) and will thus not return *wxInvalidOffset* for that.

Derived from

wxOutputStream (p. 1016)

Include files

<wx/wfstream.h>

See also

wxBufferedOutputStream (p. 114), *wxFileInputStream* (p. 520), *wxFFileInputStream* (p. 500)

wxFileOutputStream::wxFileOutputStream

wxFileOutputStream(**const wxString&** *ofileName*)

Creates a new file with *ofilename* name and initializes the stream in write-only mode.

wxFileOutputStream(**wxFile&** *file*)

Initializes a file stream in write-only mode using the file I/O object *file*.

wxFileOutputStream(**int** *fd*)

Initializes a file stream in write-only mode using the file descriptor *fd*.

wxFileOutputStream::~wxFileOutputStream

~wxFileOutputStream()

Destructor.

wxFileOutputStream::Ok

bool Ok() **const**

Returns true if the stream is initialized and ready.

wxFileStream

Derived from

wxFileOutputStream (p. 538), *wxFileInputStream* (p. 520)

Include files

<wx/wfstream.h>

See also

wxStreamBuffer (p. 1277)

wxFileStream::wxFileStream

wxFileStream(**const wxString&** *iofileName*)

Initializes a new file stream in read-write mode using the specified *iofilename* name.

wxFileSystem

This class provides an interface for opening files on different file systems. It can handle absolute and/or local filenames. It uses a system of *handlers* (p. 542) to provide access to user-defined virtual file systems.

Derived from

wxObject (p. 1012)

Include files

<wx/filesys.h>

See Also

wxFileSystemHandler (p. 542), *wxFSFile* (p. 590), *Overview* (p. 1749)

wxFileSystem::wxFileSystem

wxFileSystem()

Constructor.

wxFileSystem::AddHandler

static void AddHandler(**wxFileSystemHandler** **handler*)

This static function adds new handler into the list of handlers. The *handlers* (p. 542) provide access to virtual FS.

Note

You can call:

```
wxFileSystem::AddHandler(new My_FS_Handler);
```

This is because (a) AddHandler is a static method, and (b) the handlers are deleted in wxFileSystem's destructor so that you don't have to care about it.

wxFileSystem::ChangePathTo

void ChangePathTo(**const wxString&** *location*, **bool** *is_dir = false*)

Sets the current location. *location* parameter passed to *OpenFile* (p. 542) is relative to this path.

Caution! Unless *is_dir* is true the *location* parameter is not the directory name but the

name of the file in this directory. All these commands change the path to "dir/subdir/":

```
ChangePathTo("dir/subdir/xh.htm");
ChangePathTo("dir/subdir", true);
ChangePathTo("dir/subdir/", true);
```

Parameters

location

> the new location. Its meaning depends on the value of *is_dir*

is_dir

> if true *location* is new directory. If false (default) *location* is **file in** the new directory.

Example

```
f = fs -> OpenFile("hello.htm"); // opens file 'hello.htm'
fs -> ChangePathTo("subdir/folder", true);
f = fs -> OpenFile("hello.htm"); // opens file 'subdir/folder/hello.htm'
!!
```

wxFileSystem::GetPath

wxString GetPath()

Returns actual path (set by *ChangePathTo* (p. 540)).

wxFileSystem::FileNameToURL

static wxString FileNameToURL(**wxFileName** *filename*)

Converts filename into URL.

See also

wxFileSystem::URLToFileName (p. 542),*wxFileName* (p. 521)

wxFileSystem::FindFirst

wxString FindFirst(**const wxString&** *wildcard*, **int** *flags = 0*)

Works like *wxFindFirstFile* (p. 1600). Returns name of the first filename (within filesystem's current path) that matches *wildcard*. *flags* may be one of wxFILE (only files), wxDIR (only directories) or 0 (both).

wxFileSystem::FindNext

wxString FindNext()

Returns the next filename that matches parameters passed to *FindFirst* (p. 541).

wxFileSystem::OpenFile

wxFSFile* OpenFile(**const wxString&** *location*)

Opens the file and returns a pointer to a *wxFSFile* (p. 590) object or NULL if failed. It first tries to open the file in relative scope (based on value passed to ChangePathTo() method) and then as an absolute path. Note that the user is responsible for deleting the returned wxFSFile.

wxFileSystem::URLToFileName

static wxFileName URLToFileName(**const wxString&** *url*)

Converts URL into a well-formed filename. The URL must use the `file` protocol.

See also

wxFileSystem::FileNameToURL (p. 541), *wxFileName* (p. 521)

wxFileSystemHandler

Classes derived from wxFileSystemHandler are used to access virtual file systems. Its public interface consists of two methods: *CanOpen* (p. 543) and *OpenFile* (p. 544). It provides additional protected methods to simplify the process of opening the file: GetProtocol, GetLeftLocation, GetRightLocation, GetAnchor, GetMimeTypeFromExt.

Please have a look at *overview* (p. 1749) if you don't know how locations are constructed.

Also consult *list of available handlers* (p. 1749).

wxPerl note: In wxPerl, you need to derive your file system handler class from Wx::PlFileSystemHandler.

Notes

   • The handlers are shared by all instances of wxFileSystem.

   • wxHTML library provides handlers for local files and HTTP or FTP protocol

   • The *location* parameter passed to OpenFile or CanOpen methods is always an **absolute** path. You don't need to check the FS's current path.

Derived from

wxObject (p. 1012)

Include files

<wx/filesys.h>

See also

wxFileSystem (p. 540), *wxFSFile* (p. 590), *Overview* (p. 1749)

wxFileSystemHandler::wxFileSystemHandler

wxFileSystemHandler()

Constructor.

wxFileSystemHandler::CanOpen

virtual bool CanOpen(**const wxString&** *location*)

Returns true if the handler is able to open this file. This function doesn't check whether the file exists or not, it only checks if it knows the protocol. Example:

```
bool MyHand::CanOpen(const wxString& location)
{
    return (GetProtocol(location) == "http");
}
```

Must be overridden in derived handlers.

wxFileSystemHandler::GetAnchor

wxString GetAnchor(**const wxString&** *location*) **const**

Returns the anchor if present in the location. See *wxFSFile* (p. 591) for details.

Example: GetAnchor("index.htm#chapter2") == "chapter2"

Note: the anchor is NOT part of the left location.

wxFileSystemHandler::GetLeftLocation

wxString GetLeftLocation(**const wxString&** *location*) **const**

Returns the left location string extracted from *location*.

Example: GetLeftLocation("file:myzipfile.zip#zip:index.htm") == "file:myzipfile.zip"

wxFileSystemHandler::GetMimeTypeFromExt

wxString GetMimeTypeFromExt(**const wxString&** *location*)

Returns the MIME type based on **extension** of *location*. (While wxFSFile::GetMimeType returns real MIME type - either extension-based or queried from HTTP.)

Example : GetMimeTypeFromExt("index.htm") == "text/html"

wxFileSystemHandler::GetProtocol

wxString GetProtocol(**const wxString&** *location*) **const**

Returns the protocol string extracted from *location*.

Example: GetProtocol("file:myzipfile.zip#zip:index.htm") == "zip"

wxFileSystemHandler::GetRightLocation

wxString GetRightLocation(**const wxString&** *location*) **const**

Returns the right location string extracted from *location*.

Example : GetRightLocation("file:myzipfile.zip#zip:index.htm") == "index.htm"

wxFileSystemHandler::FindFirst

virtual wxString FindFirst(**const wxString&** *wildcard*, **int** *flags = 0*)

Works like *wxFindFirstFile* (p. 1600). Returns name of the first filename (within filesystem's current path) that matches *wildcard*. *flags* may be one of wxFILE (only files), wxDIR (only directories) or 0 (both).

This method is only called if *CanOpen* (p. 543) returns true.

wxFileSystemHandler::FindNext

virtual wxString FindNext()

Returns next filename that matches parameters passed to *FindFirst* (p. 541).

This method is only called if *CanOpen* (p. 543) returns true and FindFirst returned a non-empty string.

wxFileSystemHandler::OpenFile

virtual wxFSFile* OpenFile(**wxFileSystem&** *fs*, **const wxString&** *location*)

Opens the file and returns wxFSFile pointer or NULL if failed.

Must be overridden in derived handlers.

Parameters

fs

> Parent FS (the FS from that OpenFile was called). See ZIP handler for details of how to use it.

location

> The **absolute** location of file.

wxFileType

This class holds information about a given *file type*. File type is the same as MIME type under Unix, but under Windows it corresponds more to an extension than to MIME type (in fact, several extensions may correspond to a file type). This object may be created in several different ways: the program might know the file extension and wish to find out the corresponding MIME type or, conversely, it might want to find the right extension for the file to which it writes the contents of given MIME type. Depending on how it was created some fields may be unknown so the return value of all the accessors **must** be checked: `false` will be returned if the corresponding information couldn't be found.

The objects of this class are never created by the application code but are returned by *wxMimeTypesManager::GetFileTypeFromMimeType* (p. 976) and *wxMimeTypesManager::GetFileTypeFromExtension* (p. 976) methods. But it is your responsibility to delete the returned pointer when you're done with it!

A brief reminder about what the MIME types are (see the RFC 1341 for more information): basically, it is just a pair category/type (for example, "text/plain") where the category is a basic indication of what a file is. Examples of categories are "application", "image", "text", "binary", and type is a precise definition of the document format: "plain" in the example above means just ASCII text without any formatting, while "text/html" is the HTML document source.

A MIME type may have one or more associated extensions: "text/plain" will typically correspond to the extension ".txt", but may as well be associated with ".ini" or ".conf".

Derived from

None

Include files

<wx/mimetype.h>

See also

wxMimeTypesManager (p. 974)

MessageParameters class

One of the most common usages of MIME is to encode an e-mail message. The MIME type of the encoded message is an example of a *message parameter*. These parameters are found in the message headers ("Content-XXX"). At the very least, they must specify the MIME type and the version of MIME used, but almost always they provide additional information about the message such as the original file name or the charset (for the text documents).

These parameters may be useful to the program used to open, edit, view or print the message, so, for example, an e-mail client program will have to pass them to this program. Because wxFileType itself can not know about these parameters, it uses MessageParameters class to query them. The default implementation only requires the caller to provide the file name (always used by the program to be called - it must know which file to open) and the MIME type and supposes that there are no other parameters. If you wish to supply additional parameters, you must derive your own class from MessageParameters and override GetParamValue() function, for example:

```
// provide the message parameters for the MIME type manager
class MailMessageParameters : public wxFileType::MessageParameters
{
public:
   MailMessageParameters(const wxString& filename,
                         const wxString& mimetype)
      : wxFileType::MessageParameters(filename, mimetype)
   {
   }

   virtual wxString GetParamValue(const wxString& name) const
   {
       // parameter names are not case-sensitive
       if ( name.CmpNoCase("charset") == 0 )
          return "US-ASCII";
       else
           return wxFileType::MessageParameters::GetParamValue(name);
   }
};
```

Now you only need to create an object of this class and pass it to, for example, *GetOpenCommand* (p. 548) like this:

```
wxString command;
if ( filetype->GetOpenCommand(&command,
                              MailMessageParamaters("foo.txt",
"text/plain")) )
{
    // the full command for opening the text documents is in 'command'
    // (it might be "notepad foo.txt" under Windows or "cat foo.txt" under
Unix)
}
else
{
    // we don't know how to handle such files...
}
```

Windows: As only the file name is used by the program associated with the given extension anyhow (but no other message parameters), there is no need to ever derive from MessageParameters class for a Windows-only program.

wxFileType::wxFileType

wxFileType()

The default constructor is private because you should never create objects of this type: they are only returned by *wxMimeTypesManager* (p. 974) methods.

wxFileType::~wxFileType

~wxFileType()

The destructor of this class is not virtual, so it should not be derived from.

wxFileType::GetMimeType

bool GetMimeType(**wxString*** *mimeType*)

If the function returns `true`, the string pointed to by *mimeType* is filled with full MIME type specification for this file type: for example, "text/plain".

wxFileType::GetMimeTypes

bool GetMimeType(**wxArrayString&** *mimeTypes*)

Same as *GetMimeType* (p. 547) but returns array of MIME types. This array will contain only one item in most cases but sometimes, notably under Unix with KDE, may contain more MIME types. This happens when one file extension is mapped to different MIME types by KDE, mailcap and mime.types.

wxFileType::GetExtensions

bool GetExtensions(**wxArrayString&** *extensions*)

If the function returns `true`, the array *extensions* is filled with all extensions associated with this file type: for example, it may contain the following two elements for the MIME type "text/html" (notice the absence of the leading dot): "html" and "htm".

Windows: This function is currently not implemented: there is no (efficient) way to retrieve associated extensions from the given MIME type on this platform, so it will only return `true` if the wxFileType object was created by *GetFileTypeFromExtension* (p. 976) function in the first place.

wxFileType::GetIcon

bool GetIcon(**wxIconLocation *** *iconLoc*)

If the function returns `true`, the `iconLoc` is filled with the location of the icon for this

MIME type. A *wxIcon* (p. 766) may be created from *iconLoc* later.

Windows: The function returns the icon shown by Explorer for the files of the specified type.

Mac: This function is not implemented and always returns `false`.

Unix: MIME manager gathers information about icons from GNOME and KDE settings and thus GetIcon's success depends on availability of these desktop environments.

wxFileType::GetDescription

bool GetDescription(**wxString*** *desc*)

If the function returns `true`, the string pointed to by *desc* is filled with a brief description for this file type: for example, "text document" for the "text/plain" MIME type.

wxFileType::GetOpenCommand

bool GetOpenCommand(**wxString*** *command*, **MessageParameters&** *params*)

wxString GetOpenCommand(**const wxString&** *filename*)

With the first version of this method, if the `true` is returned, the string pointed to by *command* is filled with the command which must be executed (see *wxExecute* (p. 1592)) in order to open the file of the given type. In this case, the name of the file as well as any other parameters is retrieved from *MessageParameters* (p. 546) class.

In the second case, only the filename is specified and the command to be used to open this kind of file is returned directly. An empty string is returned to indicate that an error occurred (typically meaning that there is no standard way to open this kind of files).

wxFileType::GetPrintCommand

bool GetPrintCommand(**wxString*** *command*,**MessageParameters&** *params*)

If the function returns `true`, the string pointed to by *command* is filled with the command which must be executed (see *wxExecute* (p. 1592)) in order to print the file of the given type. The name of the file is retrieved from *MessageParameters* (p. 546) class.

wxFileType::ExpandCommand

static wxString ExpandCommand(**const wxString&** *command*, **MessageParameters&** *params*)

This function is primarily intended for GetOpenCommand and GetPrintCommand usage but may be also used by the application directly if, for example, you want to use some non default command to open the file.

The function replaces all occurrences of

format specification	with
%s	the full file name
%t	the MIME type
%{param}	the value of the parameter *param*

using the MessageParameters object you pass to it.

If there is no '%s' in the command string (and the string is not empty), it is assumed that the command reads the data on stdin and so the effect is the same as "< %s" were appended to the string.

Unlike all other functions of this class, there is no error return for this function.

wxFilterInputStream

A filter stream has the capability of a normal stream but it can be placed on top of another stream. So, for example, it can uncompress or decrypt the data which are read from another stream and pass it to the requester.

Derived from

wxInputStream (p. 814)
wxStreamBase (p. 1275)

Include files

<wx/stream.h>

Note

The interface of this class is the same as that of wxInputStream. Only a constructor differs and it is documented below.

wxFilterInputStream::wxFilterInputStream

 wxFilterInputStream(**wxInputStream&** *stream*)

Initializes a "filter" stream.

wxFilterOutputStream

A filter stream has the capability of a normal stream but it can be placed on top of another stream. So, for example, it can compress, encrypt the data which are passed to

it and write them to another stream.

Derived from

wxOutputStream (p. 1016)
wxStreamBase (p. 1275)

Include files

<wx/stream.h>

Note

The use of this class is exactly the same as of wxOutputStream. Only a constructor differs and it is documented below.

wxFilterOutputStream::wxFilterOutputStream

wxFilterOutputStream(**wxOutputStream&** *stream*)

Initializes a "filter" stream.

wxFindDialogEvent

wxFindReplaceDialog events

Derived from

wxCommandEvent (p. 180)

Include files

<wx/fdrepdlg.h>

Event table macros

To process a command event from *wxFindReplaceDialog* (p. 553), use these event handler macros to direct input to member functions that take a wxFindDialogEvent argument. The *id* parameter is the identifier of the find dialog and you may usually specify -1 for it unless you plan to have several find dialogs sending events to the same owner window simultaneously.

EVT_FIND(id, func)	Find button was pressed in the dialog.
EVT_FIND_NEXT(id, func)	Find next button was pressed in the dialog.
EVT_FIND_REPLACE(id, func)	Replace button was pressed in the dialog.
EVT_FIND_REPLACE_ALL(id, func)	Replace all button was pressed in the dialog.

EVT_FIND_CLOSE(id, func)	The dialog is being destroyed, any pointers to it cannot be used any longer.

wxFindDialogEvent::wxFindDialogEvent

wxFindDialogEvent(**wxEventType** *commandType = wxEVT_NULL*, **int** *id = 0*)

Constuctor used by wxWidgets only.

wxFindDialogEvent::GetFlags

int GetFlags() **const**

Get the currently selected flags: this is the combination of `wxFR_DOWN`,`wxFR_WHOLEWORD` and `wxFR_MATCHCASE` flags.

wxFindDialogEvent::GetFindString

wxString GetFindString() **const**

Return the string to find (never empty).

wxFindDialogEvent::GetReplaceString

const wxString& GetReplaceString() **const**

Return the string to replace the search string with (only for replace and replace all events).

wxFindDialogEvent::GetDialog

wxFindReplaceDialog* GetDialog() **const**

Return the pointer to the dialog which generated this event.

wxFindReplaceData

wxFindReplaceData holds the data for *wxFindReplaceDialog* (p. 553). It is used to initialize the dialog with the default values and will keep the last values from the dialog when it is closed. It is also updated each time a *wxFindDialogEvent* (p. 550) is generated so instead of using the wxFindDialogEvent methods you can also directly query this object.

Note that all `SetXXX()` methods may only be called before showing the dialog and

calling them has no effect later.

Include files

```
#include <wx/fdrepdlg.h>
```

Derived from

wxObject (p. 1012)

Data structures

Flags used by *wxFindReplaceData::GetFlags()* (p. 553) and *wxFindDialogEvent::GetFlags()* (p. 551):

```
enum wxFindReplaceFlags
{
    // downward search/replace selected (otherwise - upwards)
    wxFR_DOWN       = 1,

    // whole word search/replace selected
    wxFR_WHOLEWORD  = 2,

    // case sensitive search/replace selected (otherwise - case
insensitive)
    wxFR_MATCHCASE  = 4
}
```

These flags can be specified in *wxFindReplaceDialog ctor* (p. 554) or *Create()* (p. 554):

```
enum wxFindReplaceDialogStyles
{
    // replace dialog (otherwise find dialog)
    wxFR_REPLACEDIALOG = 1,

    // don't allow changing the search direction
    wxFR_NOUPDOWN      = 2,

    // don't allow case sensitive searching
    wxFR_NOMATCHCASE   = 4,

    // don't allow whole word searching
    wxFR_NOWHOLEWORD   = 8
}
```

wxFindReplaceData::wxFindReplaceData

wxFindReplaceData(**wxUint32** *flags = 0*)

Constuctor initializes the flags to default value (0).

wxFindReplaceData::GetFindString

const wxString& GetFindString()

Get the string to find.

wxFindReplaceData::GetReplaceString

const wxString& GetReplaceString()

Get the replacement string.

wxFindReplaceData::GetFlags

int GetFlags() **const**

Get the combination of `wxFindReplaceFlags` values.

wxFindReplaceData::SetFlags

void SetFlags(**wxUint32** *flags*)

Set the flags to use to initialize the controls of the dialog.

wxFindReplaceData::SetFindString

void SetFindString(**const wxString&** *str*)

Set the string to find (used as initial value by the dialog).

wxFindReplaceData::SetReplaceString

void SetReplaceString(**const wxString&** *str*)

Set the replacement string (used as initial value by the dialog).

wxFindReplaceDialog

wxFindReplaceDialog is a standard modeless dialog which is used to allow the user to search for some text (and possibly replace it with something else). The actual searching is supposed to be done in the owner window which is the parent of this dialog. Note that it means that unlike for the other standard dialogs this one **must** have a parent window. Also note that there is no way to use this dialog in a modal way; it is always, by design and implementation, modeless.

Please see the dialogs sample for an example of using it.

Include files

```
#include <wx/fdrepdlg.h>
```

wxDialog (p. 406)

wxFindReplaceDialog::wxFindReplaceDialog

wxFindReplaceDialog()

wxFindReplaceDialog(**wxWindow *** *parent*, **wxFindReplaceData*** *data*, **const wxString&** *title*, **int** *style = 0*)

After using default constructor *Create()* (p. 554) must be called.

The *parent* and *data* parameters must be non-NULL.

wxFindReplaceDialog::~wxFindReplaceDialog

~wxFindReplaceDialog()

Destructor.

wxFindReplaceDialog::Create

bool Create(**wxWindow *** *parent*, **wxFindReplaceData*** *data*, **const wxString&** *title*, **int** *style = 0*)

Creates the dialog; use *Show* (p. 1544) to show it on screen.

The *parent* and *data* parameters must be non-NULL.wxFindReplaceDialog::GetData

const wxFindReplaceData* GetData() **const**

Get the *wxFindReplaceData* (p. 551) object used by this dialog.

wxFlexGridSizer

A flex grid sizer is a sizer which lays out its children in a two-dimensional table with all table fields in one row having the same height and all fields in one column having the same width, but all rows or all columns are not necessarily the same height or width as in the *wxGridSizer* (p. 683).

Since wxWidgets 2.5.0, wxFlexGridSizer can also size items equally in one direction but unequally ("flexibly") in the other. If the sizer is only flexible in one direction (this can be

changed using *SetFlexibleDrection* (p. 557)), it needs to be decided how the sizer should grow in the other ("non flexible") direction in order to fill the available space. The *SetNonFlexibleGrowMode* (p. 557) method serves this purpose.

Derived from

wxGridSizer (p. 683)
wxSizer (p. 1176)
wxObject (p. 1012)

Include files

<wx/sizer.h>

See also

wxSizer (p. 1176), *Sizer overview* (p. 1771)

wxFlexGridSizer::wxFlexGridSizer

wxFlexGridSizer(**int** *rows*, **int** *cols*, **int** *vgap*, **int** *hgap*)

wxFlexGridSizer(**int** *cols*, **int** *vgap = 0*, **int** *hgap = 0*)

Constructor for a wxGridSizer. *rows* and *cols* determine the number of columns and rows in the sizer - if either of the parameters is zero, it will be calculated to form the total number of children in the sizer, thus making the sizer grow dynamically. *vgap* and *hgap* define extra space between all children.

wxFlexGridSizer::AddGrowableCol

void AddGrowableCol(**size_t** *idx*, **int** *proportion = 0*)

Specifies that column *idx* (starting from zero) should be grown if there is extra space available to the sizer.

The *proportion* parameter has the same meaning as the stretch factor for the *sizers* (p. 1771) except that if all proportions are 0, then all columns are resized equally (instead of not being resized at all).

wxFlexGridSizer::AddGrowableRow

void AddGrowableRow(**size_t** *idx*, **int** *proportion = 0*)

Specifies that row idx (starting from zero) should be grown if there is extra space available to the sizer.

See *AddGrowableCol* (p. 555) for the description of *proportion* parameter.

wxFlexGridSizer::GetFlexibleDirection

int GetFlexibleDirections() **const**

Returns a wxOrientation value that specifies whether the sizer flexibly resizes its columns, rows, or both (default).

Return value

One of the following values:

wxVERTICAL	Rows are flexibly sized.
wxHORIZONTAL	Columns are flexibly sized.
wxBOTH	Both rows and columns are flexibly sized (this is the default value).

See also

SetFlexibleDrection (p. 557)

wxFlexGridSizer::GetNonFlexibleGrowMode

int GetNonFlexibleGrowMode() **const**

Returns the value that specifies how the sizer grows in the "non flexible" direction if there is one.

Return value

One of the following values:

wxFLEX_GROWMODE_NONE	Sizer doesn't grow in the non flexible direction.
wxFLEX_GROWMODE_SPECIFIED	Sizer honors growable columns/rows set with*AddGrowableCol* (p. 555) and *AddGrowableRow* (p. 555). In this case equal sizing applies to minimum sizes of columns or rows (this is the default value).
wxFLEX_GROWMODE_ALL	Sizer equally stretches all columns or rows in the non flexible direction, whether they are growable or not in the flexible direction.

See also

SetFlexibleDrection (p. 557), *SetNonFlexibleGrowMode* (p. 557)

wxFlexGridSizer::RemoveGrowableCol

void RemoveGrowableCol(**size_t** *idx*)

Specifies that column idx is no longer growable.

wxFlexGridSizer::RemoveGrowableRow

void RemoveGrowableRow(**size_t** *idx*)

Specifies that row idx is no longer growable.

wxFlexGridSizer::SetFlexibleDirection

void SetFlexibleDirections(**int** *direction*)

Specifies whether the sizer should flexibly resize its columns, rows, or both. Argument direction can be wxVERTICAL, wxHORIZONTAL or wxBOTH (which is the default value). Any other value is ignored. See*GetFlexibleDirection()* (p. 556) for the explanation of these values.

Note that this method does not trigger relayout.

wxFlexGridSizer::SetNonFlexibleGrowMode

void SetNonFlexibleGrowMode(**wxFlexSizerGrowMode** *mode*)

Specifies how the sizer should grow in the non flexible direction if there is one (so*SetFlexibleDirections()* (p. 557) must have been called previously). Argument *mode* can be one of those documented in*GetNonFlexibleGrowMode* (p. 556), please see there for their explanation.

Note that this method does not trigger relayout.

wxFocusEvent

A focus event is sent when a window's focus changes. The window losing focus receives a "kill focus" event while the window gaining it gets a "set focus" one.

Notice that the set focus event happens both when the user gives focus to the window (whether using the mouse or keyboard) and when it is done from the program itself using *SetFocus* (p. 1533).

Derived from

wxEvent (p. 483)
wxObject (p. 1012)

Include files

<wx/event.h>

Event table macros

To process a focus event, use these event handler macros to direct input to a member function that takes a wxFocusEvent argument.

EVT_SET_FOCUS(func)   Process a wxEVT_SET_FOCUS event.

EVT_KILL_FOCUS(func)   Process a wxEVT_KILL_FOCUS event.

See also
Event handling overview (p. 1751)

wxFocusEvent::wxFocusEvent

 wxFocusEvent(**WXTYPE** *eventType = 0*, **int** *id = 0*)

Constructor.

wxFocusEvent::GetWindow

Returns the window associated with this event, that is the window which had the focus before for the `wxEVT_SET_FOCUS` event and the window which is going to receive focus for the `wxEVT_KILL_FOCUS` one.

Warning: the window pointer may be `NULL`!

wxFont

A font is an object which determines the appearance of text. Fonts are used for drawing text to a device context, and setting the appearance of a window's text.

You can retrieve the current system font settings with *wxSystemSettings* (p. 1318).

wxSystemSettings (p. 1318)

Derived from

wxGDIObject (p. 607)
wxObject (p. 1012)

Include files

<wx/font.h>

Constants

The possible values for the *family* parameter of *wxFont constructor* (p. 561) are (the old names are for compatibility only):

```
enum wxFontFamily
{
    wxFONTFAMILY_DEFAULT = wxDEFAULT,
    wxFONTFAMILY_DECORATIVE = wxDECORATIVE,
    wxFONTFAMILY_ROMAN = wxROMAN,
    wxFONTFAMILY_SCRIPT = wxSCRIPT,
    wxFONTFAMILY_SWISS = wxSWISS,
    wxFONTFAMILY_MODERN = wxMODERN,
    wxFONTFAMILY_TELETYPE = wxTELETYPE,
    wxFONTFAMILY_MAX
};
```

The possible values for the *weight* parameter are (the old names are for compatibility only):

```
enum wxFontWeight
{
    wxFONTWEIGHT_NORMAL = wxNORMAL,
    wxFONTWEIGHT_LIGHT = wxLIGHT,
    wxFONTWEIGHT_BOLD = wxBOLD,
    wxFONTWEIGHT_MAX
};
```

The font flags which can be used during the font creation are:

```
enum
{
    // no special flags: font with default weight/slant/anti-aliasing
    wxFONTFLAG_DEFAULT          = 0,

    // slant flags (default: no slant)
    wxFONTFLAG_ITALIC           = 1 << 0,
    wxFONTFLAG_SLANT            = 1 << 1,

    // weight flags (default: medium)
    wxFONTFLAG_LIGHT            = 1 << 2,
    wxFONTFLAG_BOLD             = 1 << 3,

    // anti-aliasing flag: force on or off (default: the current system
default)
    wxFONTFLAG_ANTIALIASED      = 1 << 4,
    wxFONTFLAG_NOT_ANTIALIASED  = 1 << 5,

    // underlined/strikethrough flags (default: no lines)
    wxFONTFLAG_UNDERLINED       = 1 << 6,
    wxFONTFLAG_STRIKETHROUGH    = 1 << 7,
};
```

The known font encodings are:

```
enum wxFontEncoding
{
    wxFONTENCODING_SYSTEM = -1,     // system default
    wxFONTENCODING_DEFAULT,         // current default encoding

    // ISO8859 standard defines a number of single-byte charsets
    wxFONTENCODING_ISO8859_1,       // West European (Latin1)
    wxFONTENCODING_ISO8859_2,       // Central and East European (Latin2)
    wxFONTENCODING_ISO8859_3,       // Esperanto (Latin3)
    wxFONTENCODING_ISO8859_4,       // Baltic (old) (Latin4)
    wxFONTENCODING_ISO8859_5,       // Cyrillic
```

```
    wxFONTENCODING_ISO8859_6,        // Arabic
    wxFONTENCODING_ISO8859_7,        // Greek
    wxFONTENCODING_ISO8859_8,        // Hebrew
    wxFONTENCODING_ISO8859_9,        // Turkish (Latin5)
    wxFONTENCODING_ISO8859_10,       // Variation of Latin4 (Latin6)
    wxFONTENCODING_ISO8859_11,       // Thai
    wxFONTENCODING_ISO8859_12,       // doesn't exist currently, but put it
                                     // here anyhow to make all ISO8859
                                     // consecutive numbers
    wxFONTENCODING_ISO8859_13,       // Baltic (Latin7)
    wxFONTENCODING_ISO8859_14,       // Latin8
    wxFONTENCODING_ISO8859_15,       // Latin9 (a.k.a. Latin0, includes
euro)
    wxFONTENCODING_ISO8859_MAX,

    // Cyrillic charset soup (see
http://czyborra.com/charsets/cyrillic.html)
    wxFONTENCODING_KOI8,             // we don't support any of KOI8
variants
    wxFONTENCODING_ALTERNATIVE,      // same as MS-DOS CP866
    wxFONTENCODING_BULGARIAN,        // used under Linux in Bulgaria

    // what would we do without Microsoft? They have their own encodings
        // for DOS
    wxFONTENCODING_CP437,            // original MS-DOS codepage
    wxFONTENCODING_CP850,            // CP437 merged with Latin1
    wxFONTENCODING_CP852,            // CP437 merged with Latin2
    wxFONTENCODING_CP855,            // another cyrillic encoding
    wxFONTENCODING_CP866,            // and another one
        // and for Windows
    wxFONTENCODING_CP874,            // WinThai
    wxFONTENCODING_CP1250,           // WinLatin2
    wxFONTENCODING_CP1251,           // WinCyrillic
    wxFONTENCODING_CP1252,           // WinLatin1
    wxFONTENCODING_CP1253,           // WinGreek (8859-7)
    wxFONTENCODING_CP1254,           // WinTurkish
    wxFONTENCODING_CP1255,           // WinHebrew
    wxFONTENCODING_CP1256,           // WinArabic
    wxFONTENCODING_CP1257,           // WinBaltic (same as Latin 7)
    wxFONTENCODING_CP12_MAX,

    wxFONTENCODING_UTF7,             // UTF-7 Unicode encoding
    wxFONTENCODING_UTF8,             // UTF-8 Unicode encoding

    wxFONTENCODING_UNICODE,          // Unicode - currently used only by
                                     // wxEncodingConverter class

    wxFONTENCODING_MAX
};
```

Predefined objects

Objects:

wxNullFont

Pointers:

wxNORMAL_FONT
wxSMALL_FONT
wxITALIC_FONT

wxSWISS_FONT

wxFont::wxFont

wxFont()

Default constructor.

wxFont(**int** *pointSize*, **wxFontFamily** *family*, **int** *style*, **wxFontWeight** *weight*, **const bool** *underline = false*, **const wxString&** *faceName = ""*, **wxFontEncoding** *encoding = wxFONTENCODING_DEFAULT*)

wxFont(**int** *pixelSize*, **wxFontFamily** *family*, **int** *style*, **wxFontWeight** *weight*, **const bool** *underline = false*, **const wxString&** *faceName = ""*, **wxFontEncoding** *encoding = wxFONTENCODING_DEFAULT*)

Creates a font object with the specified attributes.

Parameters

pointSize

Size in points.

pixelSize

Size in pixels: this is directly supported only under MSW currently where this constructor can be used directly, under other platforms a font with the closest size to the given one is found using binary search and the static *New* (p. 565) method must be used.

family

Font family, a generic way of referring to fonts without specifying actual facename. One of:

wxFONTFAMILY_DEFAULT Chooses a default font.

wxFONTFAMILY_DECORATIVE   A decorative font.

wxFONTFAMILY_ROMAN A formal, serif font.

wxFONTFAMILY_SCRIPT A handwriting font.

wxFONTFAMILY_SWISS   A sans-serif font.

wxFONTFAMILY_MODERN A fixed pitch font.

wxFONTFAMILY_TELETYPE A teletype font.

style

One of **wxFONTSTYLE_NORMAL**, **wxFONTSTYLE_SLANT** and **wxFONTSTYLE_ITALIC**.

weight

Font weight, sometimes also referred to as font boldness. One of:

wxFONTWEIGHT_NORMAL Normal font.

wxFONTWEIGHT_LIGHT   Light font.

wxFONTWEIGHT_BOLD   Bold font.

underline

The value can be true or false. At present this has an effect on Windows and Motif 2.x only.

faceName

An optional string specifying the actual typeface to be used. If it is an empty string, a default typeface will be chosen based on the family.

encoding

An encoding which may be one of **wxFONTENCODING_SYSTEM**   Default system encoding.

wxFONTENCODING_DEFAULT   Default application encoding: this is the encoding set by calls to *SetDefaultEncoding* (p. 565) and which may be set to, say, KOI8 to create all fonts by default with KOI8 encoding. Initially, the default application encoding is the same as default system encoding.

wxFONTENCODING_ISO8859_1...15 ISO8859 encodings.

wxFONTENCODING_KOI8 The standard Russian encoding for Internet.

wxFONTENCODING_CP1250...1252 Windows encodings similar to ISO8859 (but not identical).

If the specified encoding isn't available, no font is created (see also *font encoding overview* (p. 1793)).

Remarks

If the desired font does not exist, the closest match will be chosen. Under Windows, only scalable TrueType fonts are used.

See also *wxDC::SetFont* (p. 383), *wxDC::DrawText* (p. 375)and *wxDC::GetTextExtent* (p. 379).

wxFont::~wxFont

~wxFont()

Destructor.

Remarks

The destructor may not delete the underlying font object of the native windowing system, since wxFont uses a reference counting system for efficiency.

Although all remaining fonts are deleted when the application exits, the application should try to clean up all fonts itself. This is because wxWidgets cannot know if a pointer to the font object is stored in an application data structure, and there is a risk of double deletion.

wxFont::IsFixedWidth

bool IsFixedWidth() **const**

Returns `true` if the font is a fixed width (or monospaced) font,`false` if it is a proportional one or font is invalid.

wxFont::GetDefaultEncoding

static wxFontEncoding GetDefaultEncoding()

Returns the current application's default encoding.

See also

Font encoding overview (p. 1793),*SetDefaultEncoding* (p. 565)

wxFont::GetFaceName

wxString GetFaceName() **const**

Returns the typeface name associated with the font, or the empty string if there is no typeface information.

See also

wxFont::SetFaceName (p. 566)

wxFont::GetFamily

wxFontFamily GetFamily() const

Gets the font family. See *wxFont::SetFamily* (p. 566) for a list of valid family identifiers.

See also

wxFont::SetFamily (p. 566)

wxFont::GetNativeFontInfoDesc

wxString GetNativeFontInfoDesc() const

Returns the platform-dependent string completely describing this font or an empty string if the font wasn't constructed using the native font description.

See also

wxFont::SetNativeFontInfo (p. 567)

wxFont::GetPointSize

int GetPointSize() const

Gets the point size.

See also

wxFont::SetPointSize (p. 567)

wxFont::GetStyle

int GetStyle() const

Gets the font style. See *wxFont::wxFont* (p. 561) for a list of valid styles.

See also

wxFont::SetStyle (p. 567)

wxFont::GetUnderlined

bool GetUnderlined() const

Returns true if the font is underlined, false otherwise.

See also

wxFont::SetUnderlined (p. 567)

wxFont::GetWeight

wxFontWeight GetWeight() **const**

Gets the font weight. See *wxFont::wxFont* (p. 561) for a list of valid weight identifiers.

See also

wxFont::SetWeight (p. 568)

wxFont::New

static wxFont * New(**int** *pointSize*, **wxFontFamily** *family*, **int** *style*, **wxFontWeight** *weight*, **const bool** *underline = false*, **const wxString&** *faceName = ""*, **wxFontEncoding** *encoding = wxFONTENCODING_DEFAULT*)

static wxFont * New(**int** *pointSize*, **wxFontFamily** *family*, **int** *flags = wxFONTFLAG_DEFAULT*, **const wxString&** *faceName = ""*, **wxFontEncoding** *encoding = wxFONTENCODING_DEFAULT*)

static wxFont * New(**const wxSize&** *pixelSize*, **wxFontFamily** *family*, **int** *style*, **wxFontWeight** *weight*, **const bool** *underline = false*, **const wxString&** *faceName = ""*, **wxFontEncoding** *encoding = wxFONTENCODING_DEFAULT*)

static wxFont * New(**const wxSize&** *pixelSize*, **wxFontFamily** *family*, **int** *flags = wxFONTFLAG_DEFAULT*, **const wxString&** *faceName = ""*, **wxFontEncoding** *encoding = wxFONTENCODING_DEFAULT*)

These functions take the same parameters as *wxFont constructor* (p. 561) and return a new font object allocated on the heap.

Using `New()` is currently the only way to directly create a font with the given size in pixels on platforms other than wxMSW.

wxFont::Ok

bool Ok() **const**

Returns `true` if this object is a valid font, `false` otherwise.

wxFont::SetDefaultEncoding

static void SetDefaultEncoding(**wxFontEncoding** *encoding*)

Sets the default font encoding.

See also

Font encoding overview (p. 1793),*GetDefaultEncoding* (p. 563)

wxFont::SetFaceName

void SetFaceName(**const wxString&** *faceName*)

Sets the facename for the font.

Parameters

faceName

A valid facename, which should be on the end-user's system.

Remarks

To avoid portability problems, don't rely on a specific face, but specify the font family instead or as well. A suitable font will be found on the end-user's system. If both the family and the facename are specified, wxWidgets will first search for the specific face, and then for a font belonging to the same family.

See also

wxFont::GetFaceName (p. 563), *wxFont::SetFamily* (p. 566)

wxFont::SetFamily

void SetFamily(**wxFontFamily** *family*)

Sets the font family.

Parameters

family

One of:

wxFONTFAMILY_DEFAULT Chooses a default font.

wxFONTFAMILY_DECORATIVE   A decorative font.

wxFONTFAMILY_ROMAN   A formal, serif font.

wxFONTFAMILY_SCRIPT   A handwriting font.

wxFONTFAMILY_SWISS   A sans-serif font.

wxFONTFAMILY_MODERN A fixed pitch font.

wxFONTFAMILY_TELETYPE A teletype font.

See also

wxFont::GetFamily (p. 564), *wxFont::SetFaceName* (p. 566)

wxFont::SetNativeFontInfo

void SetNativeFontInfo(**const wxString&** *info*)

Creates the font corresponding to the given native font description string which must have been previously returned by*GetNativeFontInfoDesc* (p. 564). If the string is invalid, font is unchanged.

wxFont::SetPointSize

void SetPointSize(**int** *pointSize*)

Sets the point size.

Parameters

pointSize

> Size in points.

See also

wxFont::GetPointSize (p. 564)

wxFont::SetStyle

void SetStyle(**int** *style*)

Sets the font style.

Parameters

style

> One of **wxFONTSTYLE_NORMAL**, **wxFONTSTYLE_SLANT** and **wxFONTSTYLE_ITALIC**.

See also

wxFont::GetStyle (p. 564)

wxFont::SetUnderlined

void SetUnderlined(**const bool** *underlined*)

Sets underlining.

Parameters

underlining

true to underline, false otherwise.

See also

wxFont::GetUnderlined (p. 564)

wxFont::SetWeight

void SetWeight(**wxFontWeight** *weight*)

Sets the font weight.

Parameters

weight

One of:

wxFONTWEIGHT_NORMAL Normal font.

wxFONTWEIGHT_LIGHT   Light font.

wxFONTWEIGHT_BOLD   Bold font.

See also

wxFont::GetWeight (p. 565)

wxFont::operator =

wxFont& operator =(**const wxFont&** *font*)

Assignment operator, using reference counting. Returns a reference to 'this'.

wxFont::operator ==

bool operator ==(**const wxFont&** *font*)

Equality operator. Two fonts are equal if they contain pointers to the same underlying font data. It does not compare each attribute, so two independently-created fonts using the same parameters will fail the test.

wxFont::operator !=

bool operator !=(**const wxFont&** *font*)

Inequality operator. Two fonts are not equal if they contain pointers to different underlying font data. It does not compare each attribute.

wxFontData

wxFontDialog overview (p. 1799)

This class holds a variety of information related to font dialogs.

Derived from

wxObject (p. 1012)

Include files

<wx/cmndata.h>

See also

Overview (p. 1799), *wxFont* (p. 558), *wxFontDialog* (p. 571)

wxFontData::wxFontData

 wxFontData()

Constructor. Initializes *fontColour* to black, *showHelp* to black, *allowSymbols* to true, *enableEffects* to true, *minSize* to 0 and *maxSize* to 0.

wxFontData::EnableEffects

void EnableEffects(**bool** *enable*)

Enables or disables 'effects' under MS Windows or generic only. This refers to the controls for manipulating colour, strikeout and underline properties.

The default value is true.

wxFontData::GetAllowSymbols

bool GetAllowSymbols()

Under MS Windows, returns a flag determining whether symbol fonts can be selected. Has no effect on other platforms.

The default value is true.

wxFontData::GetColour

wxColour& GetColour()

Gets the colour associated with the font dialog.

The default value is black.

wxFontData::GetChosenFont

wxFont GetChosenFont()

Gets the font chosen by the user if the user pressed OK (wxFontDialog::ShowModal returned wxID_OK).

wxFontData::GetEnableEffects

bool GetEnableEffects()

Determines whether 'effects' are enabled under Windows. This refers to the controls for manipulating colour, strikeout and underline properties.

The default value is true.

wxFontData::GetInitialFont

wxFont GetInitialFont()

Gets the font that will be initially used by the font dialog. This should have previously been set by the application.

wxFontData::GetShowHelp

bool GetShowHelp()

Returns true if the Help button will be shown (Windows only).

The default value is false.

wxFontData::SetAllowSymbols

void SetAllowSymbols(bool *allowSymbols*)

Under MS Windows, determines whether symbol fonts can be selected. Has no effect on other platforms.

The default value is true.

wxFontData::SetChosenFont

void SetChosenFont(**const wxFont&** *font*)

Sets the font that will be returned to the user (for internal use only).

wxFontData::SetColour

void SetColour(**const wxColour&** *colour*)

Sets the colour that will be used for the font foreground colour.

The default colour is black.

wxFontData::SetInitialFont

void SetInitialFont(**const wxFont&***font*)

Sets the font that will be initially used by the font dialog.

wxFontData::SetRange

void SetRange(**int** *min*, **int** *max*)

Sets the valid range for the font point size (Windows only).

The default is 0, 0 (unrestricted range).

wxFontData::SetShowHelp

void SetShowHelp(**bool** *showHelp*)

Determines whether the Help button will be displayed in the font dialog (Windows only).

The default value is false.

wxFontData::operator =

void operator =(**const wxFontData&** *data*)

Assignment operator for the font data.

wxFontDialog

This class represents the font chooser dialog.

Derived from

wxDialog (p. 406)
wxWindow (p. 1490)
wxEvtHandler (p. 486)
wxObject (p. 1012)

Include files

<wx/fontdlg.h>

See also

Overview (p. 1799),
wxFontData (p. 569),
wxGetFontFromUser (p. 1616)

wxFontDialog::wxFontDialog

wxFontDialog()

wxFontDialog(**wxWindow*** *parent*)

wxFontDialog(**wxWindow*** *parent*, **const wxFontData&** *data*)

Constructor. Pass a parent window, and optionally the *font data* (p. 569) object to be used to initialize the dialog controls. If the default constructor is used, *Create()* (p. 572) must be called before the dialog can be shown.

wxFontDialog::Create

bool Create(**wxWindow*** *parent*)

bool Create(**wxWindow*** *parent*, **const wxFontData&** *data*)

Creates the dialog if it the wxFontDialog object had been initialized using the default constructor. Returns `true` on success and `false` if an error occurred.

wxFontDialog::GetFontData

const wxFontData& GetFontData() **const**

wxFontData& GetFontData()

Returns the *font data* (p. 569) associated with the font dialog.

wxFontDialog::ShowModal

int ShowModal()

Shows the dialog, returning `wxID_OK` if the user pressed Ok, and `wxID_CANCEL` otherwise.

If the user cancels the dialog (ShowModal returns `wxID_CANCEL`), no font will be created. If the user presses OK, a new wxFont will be created and stored in the font dialog's wxFontData structure.

wxFontEnumerator

wxFontEnumerator enumerates either all available fonts on the system or only the ones with given attributes - either only fixed-width (suited for use in programs such as terminal emulators and the like) or the fonts available in the given *encoding* (p. 1793).

To do this, you just have to call one of EnumerateXXX() functions - either *EnumerateFacenames* (p. 573) or *EnumerateEncodings* (p. 574) and the corresponding callback (*OnFacename* (p. 574) or *OnFontEncoding* (p. 574)) will be called repeatedly until either all fonts satisfying the specified criteria are exhausted or the callback returns false.

Virtual functions to override

Either *OnFacename* (p. 574) or *OnFontEncoding* (p. 574) should be overridden depending on whether you plan to call *EnumerateFacenames* (p. 573) or *EnumerateEncodings* (p. 574). Of course, if you call both of them, you should override both functions.

Derived from

None

Include files

<wx/fontenum.h>

See also

Font encoding overview (p. 1793), *Font sample* (p. 1711), *wxFont* (p. 558), *wxFontMapper* (p. 575)

wxFontEnumerator::EnumerateFacenames

virtual bool EnumerateFacenames(wxFontEncoding *encoding* = *wxFONTENCODING_SYSTEM*,   **bool** *fixedWidthOnly = false*)

Call *OnFacename* (p. 574) for each font which supports given encoding (only if it is not wxFONTENCODING_SYSTEM) and is of fixed width (if *fixedWidthOnly* is true).

Calling this function with default arguments will result in enumerating all fonts available on the system.

wxFontEnumerator::EnumerateEncodings

virtual bool EnumerateEncodings(**const wxString&** *font = ""*)

Call *OnFontEncoding* (p. 574) for each encoding supported by the given font - or for each encoding supported by at least some font if *font* is not specified.

wxFontEnumerator::GetEncodings

wxArrayString* GetEncodings()

Return array of strings containing all encodings found by *EnumerateEncodings* (p. 574). This is convenience function. It is based on default implementation of *OnFontEncoding* (p. 574) so don't expect it to work if you overwrite that method.

wxFontEnumerator::GetFacenames

wxArrayString* GetFacenames()

Return array of strings containing all facenames found by *EnumerateFacenames* (p. 573). This is convenience function. It is based on default implementation of *OnFacename* (p. 574) so don't expect it to work if you overwrite that method.

wxFontEnumerator::OnFacename

virtual bool OnFacename(**const wxString&** *font*)

Called by *EnumerateFacenames* (p. 573) for each match. Return true to continue enumeration or false to stop it.

wxFontEnumerator::OnFontEncoding

virtual bool OnFontEncoding(**const wxString&** *font*, **const wxString&** *encoding*)

Called by *EnumerateEncodings* (p. 574) for each match. Return true to continue enumeration or false to stop it.

wxFontList

A font list is a list containing all fonts which have been created. There is only one instance of this class: **wxTheFontList**. Use this object to search for a previously created font of the desired type and create it if not already found. In some windowing systems, the font may be a scarce resource, so it is best to reuse old resources if possible. When an application finishes, all fonts will be deleted and their resources freed, eliminating the possibility of 'memory leaks'.

Derived from

wxList (p. 840)
wxObject (p. 1012)

Include files

<wx/gdicmn.h>

See also

wxFont (p. 558)

wxFontList::wxFontList

wxFontList()

Constructor. The application should not construct its own font list: use the object pointer **wxTheFontList**.

wxFontList::AddFont

void AddFont(**wxFont** **font*)

Used by wxWidgets to add a font to the list, called in the font constructor.

wxFontList::FindOrCreateFont

wxFont * FindOrCreateFont(**int** *point_size*, **int** *family*, **int** *style*, **int** *weight*, **bool** *underline = false*, **const wxString&** *facename = NULL*, **wxFontEncoding** *encoding = wxFONTENCODING_DEFAULT*)

Finds a font of the given specification, or creates one and adds it to the list. See the *wxFont constructor* (p. 561) for details of the arguments.

wxFontList::RemoveFont

void RemoveFont(**wxFont** **font*)

Used by wxWidgets to remove a font from the list.

wxFontMapper

wxFontMapper manages user-definable correspondence between logical font names and the fonts present on the machine.

The default implementations of all functions will ask the user if they are not capable of finding the answer themselves and store the answer in a config file (configurable via SetConfigXXX functions). This behaviour may be disabled by giving the value of false to "interactive" parameter.

However, the functions will always consult the config file to allow the user-defined values override the default logic and there is no way to disable this - which shouldn't be ever needed because if "interactive" was never true, the config file is never created anyhow.

In case everything else fails (i.e. there is no record in config file and "interactive" is false or user denied to choose any replacement), the class queries *wxEncodingConverter* (p. 478) for "equivalent" encodings (e.g. iso8859-2 and cp1250) and tries them.

Using wxFontMapper in conjunction with wxMBConv classes

If you need to display text in encoding which is not available at host system (see *IsEncodingAvailable* (p. 578)), you may use these two classes to find font in some similar encoding (see *GetAltForEncoding* (p. 577)) and convert the text to this encoding (*wxMBConv classes* (p. 1733)).

Following code snippet demonstrates it:

```
if (!wxFontMapper::Get()->IsEncodingAvailable(enc, facename))
{
   wxFontEncoding alternative;
   if (wxFontMapper::Get()->GetAltForEncoding(enc, &alternative,
                                             facename, false))
   {
       wxCSConv convFrom(wxFontMapper::Get()->GetEncodingName(enc));
       wxCSConv convTo(wxFontMapper::Get()->GetEncodingName(alternative));
       text = wxString(text.mb_str(convFrom), convTo);
   }
   else
       ...failure (or we may try iso8859-1/7bit ASCII)...
}
...display text...
```

Derived from

No base class

Include files

<wx/fontmap.h>

See also

wxEncodingConverter (p. 478), *Writing non-English applications* (p. 1737)

wxFontMapper::wxFontMapper

wxFontMapper()

Default ctor.

Note

The preferred way of creating a wxFontMapper instance is to call *wxFontMapper::Get* (p. 577).

wxFontMapper::~wxFontMapper

~wxFontMapper()

Virtual dtor for a base class.

wxFontMapper::CharsetToEncoding

wxFontEncoding CharsetToEncoding(**const wxString&** *charset*, **bool** *interactive* = *true*)

Returns the encoding for the given charset (in the form of RFC 2046) or`wxFONTENCODING_SYSTEM` if couldn't decode it.

Be careful when using this function with *interactive* set to `true`(default value) as the function then may show a dialog box to the user which may lead to unexpected reentrancies and may also take a significantly longer time than a simple function call. For these reasons, it is almost always a bad idea to call this function from the event handlers for repeatedly generated events such as `EVT_PAINT`.

wxFontMapper::Get

static wxFontMapper * Get()

Get the current font mapper object. If there is no current object, creates one.

See also

wxFontMapper::Set (p. 579)

wxFontMapper::GetAltForEncoding

bool GetAltForEncoding(**wxFontEncoding** *encoding*, **wxNativeEncodingInfo*** *info*, **const wxString&** *facename* = *wxEmptyString*, **bool** *interactive* = *true*)

bool GetAltForEncoding(**wxFontEncoding** *encoding,* **wxFontEncoding*** *alt_encoding*, **const wxString&** *facename* = *wxEmptyString*, **bool** *interactive* = *true*)

Find an alternative for the given encoding (which is supposed to not be available on this system). If successful, return true and fill info structure with the parameters required to create the font, otherwise return false.

The first form is for wxWidgets' internal use while the second one is better suitable for general use -- it returns wxFontEncoding which can consequently be passed to wxFont constructor.

wxFontMapper::GetEncoding

static wxFontEncoding GetEncoding(**size_t** *n*)

Returns the *n*-th supported encoding. Together with *GetSupportedEncodingsCount()* (p. 578) this method may be used to get all supported encodings.

wxFontMapper::GetEncodingDescription

static wxString GetEncodingDescription(**wxFontEncoding** *encoding*)

Return user-readable string describing the given encoding.

wxFontMapper::GetEncodingFromName

static wxFontEncoding GetEncodingFromName(**const wxString&** *encoding*)

Return the encoding corresponding to the given internal name. This function is the inverse of *GetEncodingName* (p. 578) and is intentionally less general than *CharsetToEncoding* (p. 577), i.e. it doesn't try to make any guesses nor ever asks the user. It is meant just as a way of restoring objects previously serialized using *GetEncodingName* (p. 578).

wxFontMapper::GetEncodingName

static wxString GetEncodingName(**wxFontEncoding** *encoding*)

Return internal string identifier for the encoding (see also *GetEncodingDescription()* (p. 578))

See also

GetEncodingFromName (p. 578)

wxFontMapper::GetSupportedEncodingsCount

static size_t GetSupportedEncodingsCount()

Returns the number of the font encodings supported by this class. Together with *GetEncoding* (p. 578) this method may be used to get all supported encodings.

wxFontMapper::IsEncodingAvailable

bool IsEncodingAvailable(**wxFontEncoding** *encoding*, **const wxString&** *facename* = *wxEmptyString*)

Check whether given encoding is available in given face or not. If no facename is given, find *any* font in this encoding.

wxFontMapper::SetDialogParent

void SetDialogParent(**wxWindow*** *parent*)

The parent window for modal dialogs.

wxFontMapper::SetDialogTitle

void SetDialogTitle(**const wxString&** *title*)

The title for the dialogs (note that default is quite reasonable).

wxFontMapper::Set

static wxFontMapper * Set(**wxFontMapper ****mapper*)

Set the current font mapper object and return previous one (may be NULL). This method is only useful if you want to plug-in an alternative font mapper into wxWidgets.

See also

wxFontMapper::Get (p. 577)

wxFontMapper::SetConfig

void SetConfig(**wxConfigBase*** *config*)

Set the config object to use (may be NULL to use default).

By default, the global one (from wxConfigBase::Get() will be used) and the default root path for the config settings is the string returned by GetDefaultConfigPath().

wxFontMapper::SetConfigPath

void SetConfigPath(**const wxString&** *prefix*)

Set the root config path to use (should be an absolute path).

wxFrame

A frame is a window whose size and position can (usually) be changed by the user. It usually has thick borders and a title bar, and can optionally contain a menu bar, toolbar and status bar. A frame can contain any window that is not a frame or dialog.

A frame that has a status bar and toolbar created via the CreateStatusBar/CreateToolBar functions manages these windows, and adjusts the value returned by GetClientSize to reflect the remaining size available to application windows.

Derived from

wxTopLevelWindow (p. 1419)
wxWindow (p. 1490)
wxEvtHandler (p. 486)
wxObject (p. 1012)

Include files

<wx/frame.h>

Window styles

wxDEFAULT_FRAME_STYLE   Defined as **wxMINIMIZE_BOX | wxMAXIMIZE_BOX | wxRESIZE_BORDER | wxSYSTEM_MENU | wxCAPTION | wxCLOSE_BOX | wxCLIP_CHILDREN**.

wxICONIZE   Display the frame iconized (minimized). Windows only.

wxCAPTION   Puts a caption on the frame.

wxMINIMIZE   Identical to **wxICONIZE**. Windows only.

wxMINIMIZE_BOX   Displays a minimize box on the frame.

wxMAXIMIZE   Displays the frame maximized. Windows only.

wxMAXIMIZE_BOX   Displays a maximize box on the frame.

wxCLOSE_BOX   Displays a close box on the frame.

wxSTAY_ON_TOP   Stay on top of all other windows, see also wxFRAME_FLOAT_ON_PARENT. Windows only.

wxSYSTEM_MENU   Displays a system menu.

wxRESIZE_BORDER   Displays a resizeable border around the window.

wxFRAME_TOOL_WINDOW   Causes a frame with a small titlebar to be created; the frame does not appear in the taskbar under Windows or GTK+.

wxFRAME_NO_TASKBAR   Creates an otherwise normal frame but it does not appear in the taskbar under Windows or GTK+ (note that it will minimize to the desktop window under Windows which

may seem strange to the users and thus it might be better to use this style only without wxMINIMIZE_BOX style). In wxGTK, the flag is respected only if GTK+ is at least version 2.2 and the window manager supports _NET_WM_STATE_SKIP_TASKBAR (`http://freedesktop.org/Standards/wm-spec/1.3/ar01s05.html`) hint. Has no effect under other platforms.

wxFRAME_FLOAT_ON_PARENT   The frame will always be on top of its parent (unlike wxSTAY_ON_TOP). A frame created with this style must have a non-NULL parent.

wxFRAME_EX_CONTEXTHELP   Under Windows, puts a query button on the caption. When pressed, Windows will go into a context-sensitive help mode and wxWidgets will send a wxEVT_HELP event if the user clicked on an application window. *Note* that this is an extended style and must be set by calling *SetExtraStyle* (p. 1533) before Create is called (two-step construction). You cannot use this style together with wxMAXIMIZE_BOX or wxMINIMIZE_BOX, so you should use `wxDEFAULT_FRAME_STYLE & ~ (wxMINIMIZE_BOX | wxMAXIMIZE_BOX)` for the frames having this style (the dialogs don't have a minimize or a maximize box by default)

wxFRAME_SHAPED   Windows with this style are allowed to have their shape changed with the *SetShape* (p. 1423) method.

wxFRAME_EX_METAL   On Mac OS X, frames with this style will be shown with a metallic look. This is an *extra* style.

The default frame style is for normal, resizeable frames. To create a frame which can not be resized by user, you may use the following combination of styles: `wxDEFAULT_FRAME_STYLE & ~ (wxRESIZE_BORDER |  wxRESIZE_BOX | wxMAXIMIZE_BOX)`. See also *window styles overview* (p. 1762).

Default event processing

wxFrame processes the following events:

wxEVT_SIZE (p. 1175)   If the frame has exactly one child window, not counting the status and toolbar, this child is resized to take the entire frame client area. If two or more windows are present, they should be laid out explicitly either by manually handling wxEVT_SIZE or using*sizers* (p. 1771)

wxEVT_MENU_HIGHLIGHT (p. 962)   The default implementation displays the *help string* (p. 966) associated with the selected item in the first pane of the status bar, if there is one.

Remarks

An application should normally define an *wxCloseEvent* (p. 153) handler for the frame to respond to system close events, for example so that related data and subwindows can be cleaned up.

See also

wxMDIParentFrame (p. 920), *wxMDIChildFrame* (p. 916), *wxMiniFrame* (p. 977), *wxDialog* (p. 406)

wxFrame::wxFrame

 wxFrame()

Default constructor.

 wxFrame(**wxWindow*** *parent*, **wxWindowID** *id*, **const wxString&** *title*, **const wxPoint&** *pos* = *wxDefaultPosition*, **const wxSize&** *size* = *wxDefaultSize*, **long** *style* = *wxDEFAULT_FRAME_STYLE*, **const wxString&** *name* = *"frame"*)

Constructor, creating the window.

Parameters

parent

> The window parent. This may be NULL. If it is non-NULL, the frame will always be displayed on top of the parent window on Windows.

id

> The window identifier. It may take a value of -1 to indicate a default value.

title

> The caption to be displayed on the frame's title bar.

pos

> The window position. A value of (-1, -1) indicates a default position, chosen by either the windowing system or wxWidgets, depending on platform.

size

> The window size. A value of (-1, -1) indicates a default size, chosen by either the windowing system or wxWidgets, depending on platform.

style

> The window style. See *wxFrame* (p. 579).

name

The name of the window. This parameter is used to associate a name with the item, allowing the application user to set Motif resource values for individual windows.

Remarks

For Motif, MWM (the Motif Window Manager) should be running for any window styles to work (otherwise all styles take effect).

See also

wxFrame::Create (p. 583)

wxFrame::~wxFrame

void ~wxFrame()

Destructor. Destroys all child windows and menu bar if present.

wxFrame::Centre

void Centre(**int** *direction = wxBOTH*)

Centres the frame on the display.

Parameters

direction

The parameter may be wxHORIZONTAL, wxVERTICAL or wxBOTH.

wxFrame::Create

bool Create(**wxWindow*** *parent*, **wxWindowID** *id*, **const wxString&** *title*, **const wxPoint&** *pos = wxDefaultPosition*, **const wxSize&** *size = wxDefaultSize*, **long** *style = wxDEFAULT_FRAME_STYLE*, **const wxString&** *name = "frame"*)

Used in two-step frame construction. See *wxFrame::wxFrame* (p. 582) for further details.

wxFrame::CreateStatusBar

virtual wxStatusBar* CreateStatusBar(**int** *number = 1*, **long** *style = 0*, **wxWindowID** *id = -1*, **const wxString&** *name = "statusBar"*)

Creates a status bar at the bottom of the frame.

Parameters

number

The number of fields to create. Specify a value greater than 1 to create a multi-field status bar.

style

The status bar style. See *wxStatusBar* (p. 1267) for a list of valid styles.

id

The status bar window identifier. If -1, an identifier will be chosen by wxWidgets.

name

The status bar window name.

Return value

A pointer to the status bar if it was created successfully, NULL otherwise.

Remarks

The width of the status bar is the whole width of the frame (adjusted automatically when resizing), and the height and text size are chosen by the host windowing system.

By default, the status bar is an instance of wxStatusBar. To use a different class, override *wxFrame::OnCreateStatusBar* (p. 586).

Note that you can put controls and other windows on the status bar if you wish.

See also

wxFrame::SetStatusText (p. 589), *wxFrame::OnCreateStatusBar* (p. 586), *wxFrame::GetStatusBar* (p. 585)

wxFrame::CreateToolBar

virtual wxToolBar* CreateToolBar(**long** *style = wxNO_BORDER | wxTB_HORIZONTAL*, **wxWindowID** *id = -1*, **const wxString&** *name = "toolBar"*)

Creates a toolbar at the top or left of the frame.

Parameters

style

The toolbar style. See *wxToolBar* (p. 1401) for a list of valid styles.

id

The toolbar window identifier. If -1, an identifier will be chosen by wxWidgets.

name

The toolbar window name.

Return value

A pointer to the toolbar if it was created successfully, NULL otherwise.

Remarks

By default, the toolbar is an instance of wxToolBar (which is defined to be a suitable toolbar class on each platform, such as wxToolBar95). To use a different class, override *wxFrame::OnCreateToolBar* (p. 587).

When a toolbar has been created with this function, or made known to the frame with *wxFrame::SetToolBar* (p. 590), the frame will manage the toolbar position and adjust the return value from *wxWindow::GetClientSize* (p. 1506) to reflect the available space for application windows.

Under Pocket PC, you should *always* use this function for creating the toolbar to be managed by the frame, so that wxWidgets can use a combined menubar and toolbar. Where you manage your own toolbars, create a wxToolBar as usual.

See also

wxFrame::CreateStatusBar (p. 583), *wxFrame::OnCreateToolBar* (p. 587), *wxFrame::SetToolBar* (p. 590), *wxFrame::GetToolBar* (p. 586)

wxFrame::GetClientAreaOrigin

wxPoint GetClientAreaOrigin() **const**

Returns the origin of the frame client area (in client coordinates). It may be different from (0, 0) if the frame has a toolbar.

wxFrame::GetMenuBar

wxMenuBar* GetMenuBar() **const**

Returns a pointer to the menubar currently associated with the frame (if any).

See also

wxFrame::SetMenuBar (p. 588), *wxMenuBar* (p. 953), *wxMenu* (p. 940)

wxFrame::GetStatusBar

wxStatusBar* GetStatusBar() **const**

Returns a pointer to the status bar currently associated with the frame (if any).

See also

wxFrame::CreateStatusBar (p. 583), *wxStatusBar* (p. 1267)

wxFrame::GetStatusBarPane

int GetStatusBarPane()

Returns the status bar pane used to display menu and toolbar help.

See also

wxFrame::SetStatusBarPane (p. 589)

wxFrame::GetToolBar

wxToolBar* GetToolBar() **const**

Returns a pointer to the toolbar currently associated with the frame (if any).

See also

wxFrame::CreateToolBar (p. 584), *wxToolBar* (p. 1401), *wxFrame::SetToolBar* (p. 590)

wxFrame::OnCreateStatusBar

virtual wxStatusBar* OnCreateStatusBar(**int** *number*, **long** *style*, **wxWindowID** *id*, **const wxString&** *name*)

Virtual function called when a status bar is requested by *wxFrame::CreateStatusBar* (p. 583).

Parameters

number

> The number of fields to create.

style

> The window style. See *wxStatusBar* (p. 1267) for a list of valid styles.

id

> The window identifier. If -1, an identifier will be chosen by wxWidgets.

name

> The window name.

Return value

A status bar object.

Remarks

An application can override this function to return a different kind of status bar. The

default implementation returns an instance of *wxStatusBar* (p. 1267).

See also

wxFrame::CreateStatusBar (p. 583), *wxStatusBar* (p. 1267).

wxFrame::OnCreateToolBar

virtual wxToolBar* OnCreateToolBar(**long** *style*, **wxWindowID** *id*, **const wxString&** *name*)

Virtual function called when a toolbar is requested by *wxFrame::CreateToolBar* (p. 584).

Parameters

style

>The toolbar style. See *wxToolBar* (p. 1401) for a list of valid styles.

id

>The toolbar window identifier. If -1, an identifier will be chosen by wxWidgets.

name

>The toolbar window name.

Return value

A toolbar object.

Remarks

An application can override this function to return a different kind of toolbar. The default implementation returns an instance of *wxToolBar* (p. 1401).

See also

wxFrame::CreateToolBar (p. 584), *wxToolBar* (p. 1401).

wxFrame::ProcessCommand

void ProcessCommand(**int** *id*)

Simulate a menu command.

Parameters

id

>The identifier for a menu item.

wxFrame::SendSizeEvent

void SendSizeEvent()

This function sends a dummy *size event* (p. 1175) to the frame forcing it to reevaluate its children positions. It is sometimes useful to call this function after adding or deleting a children after the frame creation or if a child size changes.

Note that if the frame is using either sizers or constraints for the children layout, it is enough to call *Layout()* (p. 1517) directly and this function should not be used in this case.

wxFrame::SetMenuBar

void SetMenuBar(**wxMenuBar*** *menuBar*)

Tells the frame to show the given menu bar.

Parameters

menuBar

> The menu bar to associate with the frame.

Remarks

If the frame is destroyed, the menu bar and its menus will be destroyed also, so do not delete the menu bar explicitly (except by resetting the frame's menu bar to another frame or NULL).

Under Windows, a size event is generated, so be sure to initialize data members properly before calling **SetMenuBar**.

Note that on some platforms, it is not possible to call this function twice for the same frame object.

See also

wxFrame::GetMenuBar (p. 585), *wxMenuBar* (p. 953), *wxMenu* (p. 940).

wxFrame::SetStatusBar

void SetStatusBar(**wxStatusBar*** *statusBar*)

Associates a status bar with the frame.

See also

wxFrame::CreateStatusBar (p. 583), *wxStatusBar* (p. 1267), *wxFrame::GetStatusBar* (p. 585)

wxFrame::SetStatusBarPane

void SetStatusBarPane(**int** *n*)

Set the status bar pane used to display menu and toolbar help. Using -1 disables help display.

wxFrame::SetStatusText

virtual void SetStatusText(**const wxString&** *text*, **int** *number = 0*)

Sets the status bar text and redraws the status bar.

Parameters

text

> The text for the status field.

number

> The status field (starting from zero).

Remarks

Use an empty string to clear the status bar.

See also

wxFrame::CreateStatusBar (p. 583), *wxStatusBar* (p. 1267)

wxFrame::SetStatusWidths

virtual void SetStatusWidths(**int** *n*, **int ****widths*)

Sets the widths of the fields in the status bar.

Parameters

nThe number of fields in the status bar. It must be the same used in *CreateStatusBar* (p. 583).

widths

> Must contain an array of *n* integers, each of which is a status field width in pixels. A value of -1 indicates that the field is variable width; at least one field must be -1. You should delete this array after calling **SetStatusWidths**.

Remarks

The widths of the variable fields are calculated from the total width of all fields, minus the sum of widths of the non-variable fields, divided by the number of variable fields.

wxPython note: Only a single parameter is required, a Python list of integers.

wxPerl note: In wxPerl this method takes the field widths as parameters.

wxFrame::SetToolBar

void SetToolBar(**wxToolBar*** *toolBar*)

Associates a toolbar with the frame.

See also

wxFrame::CreateToolBar (p. 584), *wxToolBar* (p. 1401), *wxFrame::GetToolBar* (p. 586)

wxFSFile

This class represents a single file opened by *wxFileSystem* (p. 540). It provides more information than wxWindow's input stream (stream, filename, mime type, anchor).

Note: Any pointer returned by a method of wxFSFile is valid only as long as the wxFSFile object exists. For example a call to GetStream() doesn't *create* the stream but only returns the pointer to it. In other words after 10 calls to GetStream() you will obtain ten identical pointers.

Derived from

wxObject (p. 1012)

Include files

<wx/filesys.h>

See Also

wxFileSystemHandler (p. 542), *wxFileSystem* (p. 540), *Overview* (p. 1749)

wxFSFile::wxFSFile

 wxFSFile(**wxInputStream** **stream*, **const wxString&** *loc*, **const wxString&** *mimetype*, **const wxString&** *anchor*, **wxDateTime** *modif*)

Constructor. You probably won't use it. See Notes for details.

Parameters

stream

     The input stream that will be used to access data

location

> The full location (aka filename) of the file

mimetype

> MIME type of this file. Mime type is either extension-based or HTTP Content-Type

anchor

> Anchor. See *GetAnchor()* (p. 591) for details.

If you are not sure of the meaning of these params, see the description of the GetXXXX() functions.

Notes

It is seldom used by the application programmer but you will need it if you are writing your own virtual FS. For example you may need something similar to wxMemoryInputStream, but because wxMemoryInputStream doesn't free the memory when destroyed and thus passing a memory stream pointer into wxFSFile constructor would lead to memory leaks, you can write your own class derived from wxFSFile:

```
class wxMyFSFile : public wxFSFile
{
    private:
        void *m_Mem;
    public:
        wxMyFSFile(.....)
 ~wxMyFSFile() {free(m_Mem);}
      // of course dtor is virtual ;-)
};
```

wxFSFile::GetAnchor

const wxString& GetAnchor() const

Returns anchor (if present). The term of **anchor** can be easily explained using few examples:

```
index.htm#anchor                     /* 'anchor' is anchor */
index/wx001.htm                      /* NO anchor here!    */
archive/main.zip#zip:index.htm#global /* 'global'          */
archive/main.zip#zip:index.htm       /* NO anchor here!    */
```

Usually an anchor is presented only if the MIME type is 'text/html'. But it may have some meaning with other files; for example myanim.avi#200 may refer to position in animation or reality.wrl#MyView may refer to a predefined view in VRML.

wxFSFile::GetLocation

const wxString& GetLocation() const

Returns full location of the file, including path and protocol. Examples :

```
http://www.wxwidgets.org
http://www.ms.mff.cuni.cz/~vsla8348/wxhtml/archive.zip#zip:info.txt
file:/home/vasek/index.htm
relative-file.htm
```

wxFSFile::GetMimeType

const wxString& GetMimeType() **const**

Returns the MIME type of the content of this file. It is either extension-based (see wxMimeTypesManager) or extracted from HTTP protocol Content-Type header.

wxFSFile::GetModificationTime

wxDateTime GetModificationTime() **const**

Returns time when this file was modified.

wxFSFile::GetStream

wxInputStream* GetStream() **const**

Returns pointer to the stream. You can use the returned stream to directly access data. You may suppose that the stream provide Seek and GetSize functionality (even in the case of the HTTP protocol which doesn't provide this by default. wxHtml uses local cache to work around this and to speed up the connection).

wxFTP

wxFTP can be used to establish a connection to an FTP server and perform all the usual operations. Please consult the RFC 959 for more details about the FTP protocol.

To use a commands which doesn't involve file transfer (i.e. directory oriented commands) you just need to call a corresponding member function or use the generic *SendCommand* (p. 594) method. However to actually transfer files you just get or give a stream to or from this class and the actual data are read or written using the usual stream methods.

Example of using wxFTP for file downloading:

```
wxFTP ftp;

// if you don't use these lines anonymous login will be used
ftp.SetUser("user");
ftp.SetPassword("password");

if ( !ftp.Connect("ftp.wxwindows.org") )
{
```

```
        wxLogError("Couldn't connect");
        return;
    }

    ftp.ChDir("/pub");
    wxInputStream *in = ftp.GetInputStream("wxWidgets-4.2.0.tar.gz");
    if ( !in )
    {
        wxLogError("Coudln't get file");
    }
    else
    {
        size_t size = in->GetSize();
        char *data = new char[size];
        if ( !in->Read(data, size) )
        {
            wxLogError("Read error");
        }
        else
        {
            // file data is in the buffer
            ...
        }

        delete [] data;
        delete in;
    }
```

To upload a file you would do (assuming the connection to the server was opened successfully):

```
    wxOutputStream *out = ftp.GetOutputStream("filename");
    if ( out )
    {
        out->Write(...); // your data
        delete out;
    }
```

Constants

wxFTP defines constants corresponding to the two supported transfer modes:

```
enum TransferMode
{
    ASCII,
    BINARY
};
```

Derived from

wxProtocol (p. 1083)

Include files

<wx/protocol/ftp.h>

See also

wxSocketBase (p. 1203)

wxFTP::wxFTP

wxFTP()

Default constructor.

wxFTP::~wxFTP

~wxFTP()

Destructor will close the connection if connected.

wxFTP::Abort

bool Abort()

Aborts the download currently in process, returns `true` if ok, `false` if an error occurred.

wxFTP::CheckCommand

bool CheckCommand(**const wxString&** *command*, **char** *ret*)

Send the specified *command* to the FTP server. *ret* specifies the expected result.

Return value

true if the command has been sent successfully, else false.

wxFTP::SendCommand

char SendCommand(**const wxString&** *command*)

Send the specified *command* to the FTP server and return the first character of the return code.

wxFTP::GetLastResult

const wxString& GetLastResult()

Returns the last command result, i.e. the full server reply for the last command.

wxFTP::ChDir

bool ChDir(**const wxString&** *dir*)

Change the current FTP working directory. Returns true if successful.

wxFTP::MkDir

bool MkDir(**const wxString&** *dir*)

Create the specified directory in the current FTP working directory. Returns true if successful.

wxFTP::RmDir

bool RmDir(**const wxString&** *dir*)

Remove the specified directory from the current FTP working directory. Returns true if successful.

wxFTP::Pwd

wxString Pwd()

Returns the current FTP working directory.

wxFTP::Rename

bool Rename(**const wxString&** *src*, **const wxString&** *dst*)

Rename the specified *src* element to *dst*. Returns true if successful.

wxFTP::RmFile

bool RmFile(**const wxString&** *path*)

Delete the file specified by *path*. Returns true if successful.

wxFTP::SetAscii

bool SetAscii()

Sets the transfer mode to ASCII. It will be used for the next transfer.

wxFTP::SetBinary

bool SetBinary()

Sets the transfer mode to binary (IMAGE). It will be used for the next transfer.

wxFTP::SetPassive

void SetPassive(**bool** *pasv*)

If *pasv* is `true`, passive connection to the FTP server is used. This is the default as it works with practically all firewalls. If the server doesn't support passive move, you may call this function with `false` argument to use active connection.

wxFTP::SetTransferMode

bool SetTransferMode(**TransferMode** *mode*)

Sets the transfer mode to the specified one. It will be used for the next transfer.

If this function is never called, binary transfer mode is used by default.

wxFTP::SetUser

void SetUser(**const wxString&** *user*)

Sets the user name to be sent to the FTP server to be allowed to log in.

Default value

The default value of the user name is "anonymous".

Remark

This parameter can be included in a URL if you want to use the URL manager. For example, you can use: "ftp://a_user:a_password@a.host:service/a_directory/a_file" to specify a user and a password.

wxFTP::SetPassword

void SetPassword(**const wxString&** *passwd*)

Sets the password to be sent to the FTP server to be allowed to log in.

Default value

The default value of the user name is your email address. For example, it could be "username@userhost.domain". This password is built by getting the current user name and the host name of the local machine from the system.

Remark

This parameter can be included in a URL if you want to use the URL manager. For

example, you can use: "ftp://a_user:a_password@a.host:service/a_directory/a_file" to specify a user and a password.

wxFTP::FileExists

bool FileExists(const wxString& *filename*)

Returns `true` if the given remote file exists, `false` otherwise.

wxFTP::GetFileSize

int GetFileSize(const wxString& *filename*)

Returns the file size in bytes or -1 if the file doesn't exist or the size couldn't be determined. Notice that this size can be approximative size only and shouldn't be used for allocating the buffer in which the remote file is copied, for example.

wxFTP::GetDirList

bool GetDirList(wxArrayString& *files*, const wxString& *wildcard* = "")

The GetList function is quite low-level. It returns the list of the files in the current directory. The list can be filtered using the *wildcard* string. If *wildcard* is empty (default), it will return all files in directory.

The form of the list can change from one peer system to another. For example, for a UNIX peer system, it will look like this:

```
-r--r--r--    1 guilhem  lavaux        12738 Jan 16 20:17 cmndata.cpp
-r--r--r--    1 guilhem  lavaux        10866 Jan 24 16:41 config.cpp
-rw-rw-rw-    1 guilhem  lavaux        29967 Dec 21 19:17 cwlex_yy.c
-rw-rw-rw-    1 guilhem  lavaux        14342 Jan 22 19:51 cwy_tab.c
-r--r--r--    1 guilhem  lavaux        13890 Jan 29 19:18 date.cpp
-r--r--r--    1 guilhem  lavaux         3989 Feb  8 19:18 datstrm.cpp
```

But on Windows system, it will look like this:

```
winamp~1 exe    520196 02-25-1999  19:28  winamp204.exe
        1 file(s)            520 196 bytes
```

Return value: true if the file list was successfully retrieved, false otherwise.

See also

GetFilesList (p. 597)

wxFTP::GetFilesList

bool GetFilesList(wxArrayString& *files*, const wxString& *wildcard* = "")

This function returns the computer-parsable list of the files in the current directory (optionally only of the files matching the *wildcard*, all files by default). This list always has the same format and contains one full (including the directory path) file name per line.

Return value: true if the file list was successfully retrieved, false otherwise.

wxFTP::GetOutputStream

wxOutputStream * GetOutputStream(**const wxString&** *file*)

Initializes an output stream to the specified *file*. The returned stream has all but the seek functionality of wxStreams. When the user finishes writing data, he has to delete the stream to close it.

Return value

An initialized write-only stream.

See also

wxOutputStream (p. 1016)

wxFTP::GetInputStream

wxInputStream * GetInputStream(**const wxString&** *path*)

Creates a new input stream on the specified path. You can use all but the seek functionality of wxStream. Seek isn't available on all streams. For example, HTTP or FTP streams do not deal with it. Other functions like Tell are not available for this sort of stream, at present. You will be notified when the EOF is reached by an error.

Return value

Returns NULL if an error occurred (it could be a network failure or the fact that the file doesn't exist).

Returns the initialized stream. You will have to delete it yourself when you don't need it anymore. The destructor closes the DATA stream connection but will leave the COMMAND stream connection opened. It means that you can still send new commands without reconnecting.

Example of a standalone connection (without wxURL)

```
wxFTP ftp;
wxInputStream *in_stream;
char *data;

ftp.Connect("a.host.domain");
ftp.ChDir("a_directory");
in_stream = ftp.GetInputStream("a_file_to_get");

data = new char[in_stream->GetSize()];
```

```
in_stream->Read(data, in_stream->GetSize());
if (in_stream->LastError() != wxStream_NOERROR) {
  // Do something.
}

delete in_stream; /* Close the DATA connection */

ftp.Close(); /* Close the COMMAND connection */
```

See also

wxInputStream (p. 814)

wxGauge

A gauge is a horizontal or vertical bar which shows a quantity (often time). There are no user commands for the gauge.

Derived from

wxControl (p. 215)
wxWindow (p. 1490)
wxEvtHandler (p. 486)
wxObject (p. 1012)

Include files

<wx/gauge.h>

Window styles

wxGA_HORIZONTAL	Creates a horizontal gauge.
wxGA_VERTICAL	Creates a vertical gauge.
wxGA_SMOOTH	Creates smooth progress bar with one pixel wide update step (not supported by all platforms).

See also *window styles overview* (p. 1762).

Event handling

wxGauge is read-only so generates no events.

See also

wxSlider (p. 1192), *wxScrollBar* (p. 1143)

wxGauge::wxGauge

wxGauge()

Default constructor.

 wxGauge(**wxWindow*** *parent*, **wxWindowID** *id*, **int** *range*, **const wxPoint&** *pos* = *wxDefaultPosition*, **const wxSize&** *size* = *wxDefaultSize*, **long** *style* = *wxGA_HORIZONTAL*, **const wxValidator&** *validator* = *wxDefaultValidator*, **const wxString&** *name* = *"gauge"*)

Constructor, creating and showing a gauge.

Parameters

parent

> Window parent.

id

> Window identifier.

range

> Integer range (maximum value) of the gauge.

pos

> Window position.

size

> Window size.

style

> Gauge style. See *wxGauge* (p. 599).

name

> Window name.

See also

wxGauge::Create (p. 600)

wxGauge::~wxGauge

 ~wxGauge()

Destructor, destroying the gauge.

wxGauge::Create

bool Create(**wxWindow*** *parent*, **wxWindowID** *id*, **int** *range*, **const wxPoint&** *pos* = *wxDefaultPosition*, **const wxSize&** *size* = *wxDefaultSize*, **long** *style* =

wxGA_HORIZONTAL, **const wxValidator&** *validator = wxDefaultValidator*, **const wxString&** *name = "gauge"*)

Creates the gauge for two-step construction. See *wxGauge::wxGauge* (p. 599) for further details.

wxGauge::GetBezelFace

int GetBezelFace() **const**

Returns the width of the 3D bezel face.

Remarks

This method is not implemented (returns 0) for most platforms.

See also

wxGauge::SetBezelFace (p. 602)

wxGauge::GetRange

int GetRange() **const**

Returns the maximum position of the gauge.

Remarks

This method is not implemented (doesn't do anything) for most platforms.

See also

wxGauge::SetRange (p. 602)

wxGauge::GetShadowWidth

int GetShadowWidth() **const**

Returns the 3D shadow margin width.

Remarks

This method is not implemented (returns 0) for most platforms.

See also

wxGauge::SetShadowWidth (p. 602)

wxGauge::GetValue

int GetValue() const

Returns the current position of the gauge.

See also

wxGauge::SetValue (p. 603)

wxGauge::IsVertical

bool IsVertical() const

Returns `true` if the gauge is vertical (has `wxGA_VERTICAL` style) and `false` otherwise.

wxGauge::SetBezelFace

void SetBezelFace(int *width*)

Sets the 3D bezel face width.

Remarks

This method is not implemented (doesn't do anything) for most platforms.

See also

wxGauge::GetBezelFace (p. 601)

wxGauge::SetRange

void SetRange(int *range*)

Sets the range (maximum value) of the gauge.

See also

wxGauge::GetRange (p. 601)

wxGauge::SetShadowWidth

void SetShadowWidth(int *width*)

Sets the 3D shadow width.

Remarks

This method is not implemented (doesn't do anything) for most platforms.

wxGauge::SetValue

void SetValue(**int** *pos*)

Sets the position of the gauge.

Parameters

pos

Position for the gauge level.

See also

wxGauge::GetValue (p. 601)

wxGBPosition

This class represents the position of an item in a virtual grid of rows and columns managed by a *wxGridBagSizer* (p. 656).

Derived from

No base class

Include files

<wx/gbsizer.h>

wxGBPosition::wxGBPosition

wxGBPosition()

wxGBPosition(**int** *row*, **int** *col*)

Construct a new wxGBPosition, optionally setting the row and column. The default is (0,0).

wxGBPosition::GetCol

int GetCol() **const**

Get the current column value.

wxGBPosition::GetRow

int GetRow() **const**

Get the current row value.

wxGBPosition::SetCol

void SetCol(**int** *col*)

Set a new column value.

wxGBPosition::SetRow

void SetRow(**int** *row*)

Set a new row value.

wxGBPosition::operator!

bool operator!(**const wxGBPosition&** *p*) **const**

Is the wxGBPosition valid? (An invalid wxGBPosition is (-1,-1).)

wxGBPosition::operator==

bool operator operator==(**const wxGBPosition&** *p*) **const**

Compare equality of two wxGBPositions.

wxGBSizerItem

The wxGBSizerItem class is used by the *wxGridBagSizer* (p. 656) for tracking the items in the sizer. It adds grid position and spanning information to the normal *wxSizerItem* (p. 1187) by adding*wxGBPosition* (p. 603) and *wxGBSpan* (p. 606)attrbibutes. Most of the time you will not need to use a wxGBSizerItem directly in your code, but there are a couple of cases where it is handy.

Derived from

wxSizerItem (p. 1187)

Include files

<wx/gbsizer.h>

wxGBSizerItem::wxGBSizerItem

wxGBSizerItem(**int** *width*, **int** *height*, **const wxGBPosition&** *pos*, **const wxGBSpan&** *span*, **int** *flag*, **int** *border*, **wxObject*** *userData*)

Construct a sizer item for tracking a spacer.

wxGBSizerItem(**wxWindow*** *window*, **const wxGBPosition&** *pos*, **const wxGBSpan&** *span*, **int** *flag*, **int** *border*, **wxObject*** *userData*)

Construct a sizer item for tracking a window.

wxGBSizerItem(**wxSizer*** *sizer*, **const wxGBPosition&** *pos*, **const wxGBSpan&** *span*, **int** *flag*, **int** *border*, **wxObject*** *userData*)

Construct a sizer item for tracking a subsizer.

wxGBSizerItem::GetEndPos

void GetEndPos(**int&** *row*, **int&** *col*)

Get the row and column of the endpoint of this item

wxGBSizerItem::GetPos

wxGBPosition GetPos() **const**

void GetPos(**int&** *row*, **int&** *col*) **const**

Get the grid position of the item.

wxGBSizerItem::GetSpan

wxGBSpan GetSpan() **const**

void GetSpan(**int&** *rowspan*, **int&** *colspan*) **const**

Get the row and column spanning of the item.

wxGBSizerItem::Intersects

bool Intersects(**const wxGBSizerItem&** *other*)

Returns true if this item and the other item instersect

bool Intersects(**const wxGBPosition&** *pos*, **const wxGBSpan&** *span*)

Returns true if the given pos/span would intersect with this item.

wxGBSizerItem::SetPos

bool SetPos(**const wxGBPosition&** *pos*)

If the item is already a member of a sizer then first ensure that there is no other item that would intersect with this one at the new position, then set the new position. Returns true if the change is successful and after the next Layout the item will be moved.

wxGBSizerItem::SetSpan

bool SetSpan(**const wxGBSpan&** *span*)

If the item is already a member of a sizer then first ensure that there is no other item that would intersect with this one with its new spanning size, then set the new spanning. Returns true if the change is successful and after the next Layout the item will be resized.

wxGBSpan

This class is used to hold the row and column spanning attributes of items in a *wxGridBagSizer* (p. 656).

Derived from

No base class

Include files

<wx/gbsizer.h>

wxGBSpan::wxGBSpan

wxGBSpan()

wxGBSpan(**int** *rowspan*, **int** *colspan*)

Construct a new wxGBSpan, optionally setting the rowspan and colspan. The default is (1,1). (Meaning that the item occupies one cell in each direction.

wxGBSpan::GetColspan

int GetColspan() **const**

Get the current colspan value.

wxGBSpan::GetRowspan

int GetRowspan() **const**

Get the current rowspan value.

wxGBSpan::SetColspan

void SetColspan(**int** *colspan*)

Set a new colspan value.

wxGBSpan::SetRowspan

void SetRowspan(**int** *rowspan*)

Set a new rowspan value.

wxGBSpan::operator!

bool operator!(**const wxGBSpan&** *o*) **const**

Is the wxGBSpan valid? (An invalid wxGBSpan is (-1,-1).)

wxGBSpan::operator==

bool operator operator==(**const wxGBSpan&** *o*) **const**

Compare equality of two wxGBSpans.

wxGDIObject

This class allows platforms to implement functionality to optimise GDI objects, such as wxPen, wxBrush and wxFont. On Windows, the underling GDI objects are a scarce resource and are cleaned up when a usage count goes to zero. On some platforms this class may not have any special functionality.

Since the functionality of this class is platform-specific, it is not documented here in detail.

Derived from

wxObject (p. 1012)

Include files

<wx/gdiobj.h>

See also

wxPen (p. 1036), *wxBrush* (p. 103), *wxFont* (p. 558)

wxGDIObject::wxGDIObject

wxGDIObject()

Default constructor.

wxGenericDirCtrl

This control can be used to place a directory listing (with optional files) on an arbitrary window.

The control contains a *wxTreeCtrl* (p. 1424) window representing the directory hierarchy, and optionally, a *wxChoice* (p. 140) window containing a list of filters.

Derived from

wxControl (p. 215)
wxWindow (p. 1490)
wxEvtHandler (p. 486)
wxObject (p. 1012)

Include files

<wx/dirctrl.h>

Window styles

wxDIRCTRL_DIR_ONLY	Only show directories, and not files.
wxDIRCTRL_3D_INTERNAL	Use 3D borders for internal controls.
wxDIRCTRL_SELECT_FIRST	When setting the default path, select the first file in the directory.
wxDIRCTRL_SHOW_FILTERS	Show the drop-down filter list.
wxDIRCTRL_EDIT_LABELS	Allow the folder and file labels to be editable.

See also *Generic window styles* (p. 1762).

Data structures

wxGenericDirCtrl::wxGenericDirCtrl

wxGenericDirCtrl()

Default constructor.

 wxGenericDirCtrl(**wxWindow*** *parent*, **const wxWindowID** *id = -1*, **const wxString&** *dir = wxDirDialogDefaultFolderStr*, **const wxPoint&** *pos = wxDefaultPosition*, **const wxSize&** *size = wxDefaultSize*, **long** *style = wxDIRCTRL_3D_INTERNAL|wxSUNKEN_BORDER*, **const wxString&** *filter = wxEmptyString*, **int** *defaultFilter = 0*, **const wxString&** *name = wxTreeCtrlNameStr*)

Main constructor.

Parameters

parent

> Parent window.

id

> Window identifier.

dir

> Initial folder.

pos

> Position.

size

> Size.

style

> Window style. Please see *wxGenericDirCtrl* (p. 608) for a list of possible styles.

filter

> A filter string, using the same syntax as that for *wxFileDialog* (p. 511). This may be empty if filters are not being used.

> Example: "All files (*.*)|*.*|JPEG files (*.jpg)|*.jpg"

defaultFilter

> The zero-indexed default filter setting.

name

> The window name.

wxGenericDirCtrl::~wxGenericDirCtrl

 ~wxGenericDirCtrl()

Destructor.

wxGenericDirCtrl::Create

bool Create(**wxWindow*** *parent*, **const wxWindowID** *id = -1*, **const wxString&** *dir = wxDirDialogDefaultFolderStr*, **const wxPoint&** *pos = wxDefaultPosition*, **const wxSize&** *size = wxDefaultSize*, **long** *style = wxDIRCTRL_3D_INTERNAL|wxSUNKEN_BORDER*, **const wxString&** *filter = wxEmptyString*, **int** *defaultFilter = 0*, **const wxString&** *name = wxTreeCtrlNameStr*)

Create function for two-step construction. See *wxGenericDirCtrl::wxGenericDirCtrl* (p. 608) for details.

wxGenericDirCtrl::Init

void Init()

Initializes variables.

wxGenericDirCtrl::CollapseTree

void CollapseTree()

Collapses the entire tree.

wxGenericDirCtrl::ExpandPath

bool ExpandPath(**const wxString&** *path*)

Tries to expand as much of the given path as possible, so that the filename or directory is visible in the tree control.

wxGenericDirCtrl::GetDefaultPath

wxString GetDefaultPath() **const**

Gets the default path.

wxGenericDirCtrl::GetPath

wxString GetPath() **const**

Gets the currently-selected directory or filename.

wxGenericDirCtrl::GetFilePath

wxString GetFilePath() **const**

Gets selected filename path only (else empty string).

This function doesn't count a directory as a selection.

wxGenericDirCtrl::GetFilter

wxString GetFilter() **const**

Returns the filter string.

wxGenericDirCtrl::GetFilterIndex

int GetFilterIndex() **const**

Returns the current filter index (zero-based).

wxGenericDirCtrl::GetFilterListCtrl

wxDirFilterListCtrl* GetFilterListCtrl() **const**

Returns a pointer to the filter list control (if present).

wxGenericDirCtrl::GetRootId

wxTreeItemId GetRootId()

Returns the root id for the tree control.

wxGenericDirCtrl::GetTreeCtrl

wxTreeCtrl* GetTreeCtrl() **const**

Returns a pointer to the tree control.

wxGenericDirCtrl::ReCreateTree

void ReCreateTree()

Collapse and expand the tree, thus re-creating it from scratch. May be used to update the displayed directory content.

wxGenericDirCtrl::SetDefaultPath

void SetDefaultPath(**const wxString&** *path*)

Sets the default path.

wxGenericDirCtrl::SetFilter

void SetFilter(**const wxString&** *filter*)

Sets the filter string.

wxGenericDirCtrl::SetFilterIndex

void SetFilterIndex(**int** *n*)

Sets the current filter index (zero-based).

wxGenericDirCtrl::SetPath

void SetPath(**const wxString&** *path*)

Sets the current path.

wxGenericValidator

wxGenericValidator performs data transfer (but not validation or filtering) for the following basic controls: wxButton, wxCheckBox, wxListBox, wxStaticText, wxRadioButton, wxRadioBox, wxChoice, wxComboBox, wxGauge, wxSlider, wxScrollBar, wxSpinButton, wxTextCtrl, wxCheckListBox.

It checks the type of the window and uses an appropriate type for that window. For example, wxButton and wxTextCtrl transfer data to and from a wxString variable; wxListBox uses a wxArrayInt; wxCheckBox uses a bool.

For more information, please see *Validator overview* (p. 1765).

Derived from

wxValidator (p. 1461)
wxEvtHandler (p. 486)
wxObject (p. 1012)

Include files

<wx/valgen.h>

See also

Validator overview (p. 1765), *wxValidator* (p. 1461),*wxTextValidator* (p. 1374)

wxGenericValidator::wxGenericValidator

wxGenericValidator(**const wxGenericValidator&** *validator*)

Copy constructor.

wxGenericValidator(**bool*** *valPtr*)

Constructor taking a bool pointer. This will be used for wxCheckBox and wxRadioButton.

wxGenericValidator(**wxString*** *valPtr*)

Constructor taking a wxString pointer. This will be used for wxButton, wxComboBox, wxStaticText, wxTextCtrl.

wxGenericValidator(**int*** *valPtr*)

Constructor taking an integer pointer. This will be used for wxGauge, wxScrollBar, wxRadioBox, wxSpinButton, wxChoice.

wxGenericValidator(**wxArrayInt*** *valPtr*)

Constructor taking a wxArrayInt pointer. This will be used for wxListBox, wxCheckListBox.

Parameters

validator

Validator to copy.

valPtr

A pointer to a variable that contains the value. This variable should have a lifetime equal to or longer than the validator lifetime (which is usually determined by the lifetime of the window).

wxGenericValidator::~wxGenericValidator

~wxGenericValidator()

Destructor.

wxGenericValidator::Clone

virtual wxValidator* Clone() **const**

Clones the generic validator using the copy constructor.

wxGenericValidator::TransferFromWindow

virtual bool TransferFromWindow()

Transfers the value from the window to the appropriate data type.

wxGenericValidator::TransferToWindow

virtual bool TransferToWindow()

Transfers the value to the window.

wxGLCanvas

wxGLCanvas is a class for displaying OpenGL graphics. There are wrappers for OpenGL on Windows, and GTK+ and Motif.

To use this class, create a wxGLCanvas window, call *wxGLCanvas::SetCurrent* (p. 617) to direct normal OpenGL commands to the window, and then call *wxGLCanvas::SwapBuffers* (p. 617) to show the OpenGL buffer on the window.

To set up the attributes for the rendering context (number of bits for the depth buffer, number of bits for the stencil buffer and so on) you should set up the correct values of the *attribList* parameter. The values that should be set up and their meanings will be described below.

To switch wxGLCanvas support on under Windows, edit setup.h and set wxUSE_GLCANVAS to 1. You may also need to have to add opengl32.lib to the list of libraries your program is linked with. On Unix, pass --with-opengl to configure to compile using OpenGL or Mesa.

Derived from

wxWindow (p. 1490)
wxEvtHandler (p. 486)
wxObject (p. 1012)

Include files

<wx/glcanvas.h>

Window styles

There are no specific window styles for this class.

See also *window styles overview* (p. 1762).

Constants

The generic GL implementation doesn't support many of these options, such as stereo, auxiliary buffers, alpha channel, and accum buffer. Other implementations may support them.

WX_GL_RGBA	Use true colour
WX_GL_BUFFER_SIZE	Bits for buffer if not WX_GL_RGBA
WX_GL_LEVEL	0 for main buffer, >0 for overlay, <0 for underlay
WX_GL_DOUBLEBUFFER	Use doublebuffer
WX_GL_STEREO	Use stereoscopic display
WX_GL_AUX_BUFFERS	Number of auxiliary buffers (not all implementation support this option)
WX_GL_MIN_RED	Use red buffer with most bits (> MIN_RED bits)
WX_GL_MIN_GREEN	Use green buffer with most bits (> MIN_GREEN bits)
WX_GL_MIN_BLUE	Use blue buffer with most bits (> MIN_BLUE bits)
WX_GL_MIN_ALPHA	Use alpha buffer with most bits (> MIN_ALPHA bits)
WX_GL_DEPTH_SIZE	Bits for Z-buffer (0,16,32)
WX_GL_STENCIL_SIZE	Bits for stencil buffer
WX_GL_MIN_ACCUM_RED	Use red accum buffer with most bits (> MIN_ACCUM_RED bits)
WX_GL_MIN_ACCUM_GREEN	Use green buffer with most bits (> MIN_ACCUM_GREEN bits)
WX_GL_MIN_ACCUM_BLUE	Use blue buffer with most bits (> MIN_ACCUM_BLUE bits)
WX_GL_MIN_ACCUM_ALPHA	Use blue buffer with most bits (> MIN_ACCUM_ALPHA bits)

See also

wxGLContext (p. 617)

wxGLCanvas::wxGLCanvas

void wxGLCanvas(**wxWindow*** *parent*, **wxWindowID** *id = -1*, **const wxPoint&** *pos*, **const wxSize&** *size*, **long** *style=0*, **const wxString&** *name="GLCanvas"*, **int*** *attribList = 0*, **const wxPalette&** *palette = wxNullPalette*)

void wxGLCanvas(**wxWindow*** *parent*, **wxGLCanvas*** *sharedCanvas = NULL*,

wxWindowID *id = -1*, **const wxPoint&** *pos*, **const wxSize&** *size*, **long** *style=0*, **const wxString&** *name="GLCanvas"*, **int*** *attribList = 0*, **const wxPalette&** *palette = wxNullPalette*)

void wxGLCanvas(**wxWindow*** *parent*, **wxGLContext*** *sharedContext = NULL*, **wxWindowID** *id = -1*, **const wxPoint&** *pos*, **const wxSize&** *size*, **long** *style=0*, **const wxString&** *name="GLCanvas"*, **int*** *attribList = 0*, **const wxPalette&** *palette = wxNullPalette*)

Constructor.

parent

> Pointer to a parent window.

sharedcontext

> Context to share object resources with.

id

> Window identifier. If -1, will automatically create an identifier.

pos

> Window position. wxDefaultPosition is (-1, -1) which indicates that wxWidgets should generate a default position for the window.

size

> Window size. wxDefaultSize is (-1, -1) which indicates that wxWidgets should generate a default size for the window. If no suitable size can be found, the window will be sized to 20x20 pixels so that the window is visible but obviously not correctly sized.

style

> Window style.

name

> Window name.

attribList

> Array of int. With this parameter you can set the device context attributes associated to this window. This array is zero-terminated: it should be set up with constants described in the table above. If a constant should be followed by a value, put it in the next array position. For example, the WX_GL_DEPTH_SIZE should be followed by the value that indicates the number of bits for the depth buffer, so:

```
attribList[index]= WX_GL_DEPTH_SIZE;
attribList[index+1]=32;
and so on.
```

palette

> If the window has the palette, it should by pass this value. Note: palette and WX_GL_RGBA are mutually exclusive.

wxGLCanvas::GetContext

wxGLContext* GetContext()

Obtains the context that is associated with this canvas.

wxGLCanvas::SetCurrent

void SetCurrent()

Sets this canvas as the current recipient of OpenGL calls. Each canvas contains an OpenGL device context that has been created during the creation of this window. So this call sets the current device context as the target device context for OpenGL operations.

wxGLCanvas::SetColour

void SetColour(**const char*** *colour*)

Sets the current colour for this window, using the wxWidgets colour database to find a named colour.

wxGLCanvas::SwapBuffers

void SwapBuffers()

Displays the previous OpenGL commands on the window.

wxGLContext

wxGLContext is a class for sharing OpenGL data resources, such as display lists, with another *wxGLCanvas* (p. 614).

By sharing data resources, you can prevent code duplication, save memory, and ultimately help optimize your application.

As an example, let's say you want to draw a ball on two different windows that is identical on each one, but the dimensions of one is slightly different than the other one. What you would do is create your display lists in the shared context, and then translate each ball on the individual canvas's context. This way the actual data for the ball is only created once (in the shared context), and you won't have to duplicate your development efforts on the second ball.

Note that some wxGLContext features are extremely platform-specific - its best to check your native platform's glcanvas header (on windows include/wx/msw/glcanvas.h) to see what features your native platform provides.

Derived from

wxObject (p. 1012)

Include files

<wx/glcanvas.h>

See also

wxGLCanvas (p. 614)

wxGLContext::wxGLContext

void wxGLContext(**bool** *isRGB*, **wxGLCanvas*** *win*, **const wxPalette&** *palette* = *wxNullPalette*)

void wxGLContext(**bool** *isRGB*, **wxGLCanvas*** *win*, **const wxPalette&** *palette* = *wxNullPalette*, **const wxGLContext*** *other*)

win

> Canvas to associate this shared context with.

other

> Context to share data resources with.

wxGLContext::GetWindow

const wxWindow* GetWindow()

Obtains the window that is associated with this context.

wxGLContext::SetCurrent

void SetCurrent()

Sets this context as the current recipient of OpenGL calls. Each context contains an OpenGL device context that has been created during the creation of this window. So this call sets the current device context as the target device context for OpenGL operations.

wxGLContext::SetColour

void SetColour(**const char*** *colour*)

Sets the current colour for this context, using the wxWidgets colour database to find a named colour.

wxGLContext::SwapBuffers

void SwapBuffers()

Displays the previous OpenGL commands on the associated window.

wxGrid

wxGrid and its related classes are used for displaying and editing tabular data. They provide a rich set of features for display, editing, and interacting with a variety of data sources. For simple applications, and to help you get started, wxGrid is the only class you need to refer to directly. It will set up default instances of the other classes and manage them for you. For more complex applications you can derive your own classes for custom grid views, grid data tables, cell editors and renderers. The *wxGrid classes overview* (p. 1813) has examples of simple and more complex applications, explains the relationship between the various grid classes and has a summary of the keyboard shortcuts and mouse functions provided by wxGrid.

wxGrid has been greatly expanded and redesigned for wxWidgets 2.2 onwards. If you have been using the old wxGrid class you will probably want to have a look at the *wxGrid classes overview* (p. 1813) to see how things have changed. The new grid classes are reasonably backward-compatible but there are some exceptions. There are also easier ways of doing many things compared to the previous implementation.

Derived from

wxScrolledWindow (p. 1150)
wxWindow (p. 1490)
wxEvtHandler (p. 486)
wxObject (p. 1012)

Include files

<wx/grid.h>

Window styles

There are presently no specific window styles for wxGrid.

Event handling

The event handler for the following functions takes a *wxGridEvent* (p. 667) parameter. The ..._CMD_... variants also take a window identifier.

EVT_GRID_CELL_LEFT_CLICK(func) The user clicked a cell with the left mouse button. Processes a

wxEVT_GRID_CELL_LEFT_CLICK.

EVT_GRID_CELL_RIGHT_CLICK(func) The user clicked a cell with the right mouse button. Processes a wxEVT_GRID_CELL_RIGHT_CLICK.

EVT_GRID_CELL_LEFT_DCLICK(func) The user double-clicked a cell with the left mouse button. Processes a wxEVT_GRID_CELL_LEFT_DCLICK.

EVT_GRID_CELL_RIGHT_DCLICK(func) The user double-clicked a cell with the right mouse button. Processes a wxEVT_GRID_CELL_RIGHT_DCLICK.

EVT_GRID_LABEL_LEFT_CLICK(func) The user clicked a label with the left mouse button. Processes a wxEVT_GRID_LABEL_LEFT_CLICK.

EVT_GRID_LABEL_RIGHT_CLICK(func) The user clicked a label with the right mouse button. Processes a wxEVT_GRID_LABEL_RIGHT_CLICK.

EVT_GRID_LABEL_LEFT_DCLICK(func) The user double-clicked a label with the left mouse button. Processes a wxEVT_GRID_LABEL_LEFT_DCLICK.

EVT_GRID_LABEL_RIGHT_DCLICK(func) The user double-clicked a label with the right mouse button. Processes a wxEVT_GRID_LABEL_RIGHT_DCLICK.

EVT_GRID_CELL_CHANGE(func) The user changed the data in a cell. Processes a wxEVT_GRID_CELL_CHANGE.

EVT_GRID_SELECT_CELL(func) The user moved to, and selected a cell. Processes a wxEVT_GRID_SELECT_CELL.

EVT_GRID_EDITOR_HIDDEN(func) The editor for a cell was hidden. Processes a wxEVT_GRID_EDITOR_HIDDEN.

EVT_GRID_EDITOR_SHOWN(func) The editor for a cell was shown. Processes a wxEVT_GRID_EDITOR_SHOWN.

EVT_GRID_CMD_CELL_LEFT_CLICK(id, func) The user clicked a cell with the left mouse button; variant taking a window identifier. Processes a wxEVT_GRID_CELL_LEFT_CLICK.

EVT_GRID_CMD_CELL_RIGHT_CLICK(id, func) The user clicked a cell with the right mouse button; variant taking a window identifier. Processes a wxEVT_GRID_CELL_RIGHT_CLICK.

EVT_GRID_CMD_CELL_LEFT_DCLICK(id, func) The user double-clicked a cell with the left mouse button; variant taking a window

identifier. Processes a
wxEVT_GRID_CELL_LEFT_DCLICK.

EVT_GRID_CMD_CELL_RIGHT_DCLICK(id, func) The user double-clicked a
cell with the right mouse button; variant taking
a window identifier. Processes a
wxEVT_GRID_CELL_RIGHT_DCLICK.

EVT_GRID_CMD_LABEL_LEFT_CLICK(id, func) The user clicked a label with the left
mouse button; variant taking a window
identifier. Processes a
wxEVT_GRID_LABEL_LEFT_CLICK.

EVT_GRID_CMD_LABEL_RIGHT_CLICK(id, func) The user clicked a label with
the right mouse button; variant taking a window
identifier. Processes a
wxEVT_GRID_LABEL_RIGHT_CLICK.

EVT_GRID_CMD_LABEL_LEFT_DCLICK(id, func) The user double-clicked a
label with the left mouse button; variant taking
a window identifier. Processes a
wxEVT_GRID_LABEL_LEFT_DCLICK.

EVT_GRID_CMD_LABEL_RIGHT_DCLICK(id, func) The user double-clicked a
label with the right mouse button; variant taking
a window identifier. Processes a
wxEVT_GRID_LABEL_RIGHT_DCLICK.

EVT_GRID_CMD_CELL_CHANGE(id, func) The user changed the data in a cell;
variant taking a window identifier. Processes a
wxEVT_GRID_CELL_CHANGE.

EVT_GRID_CMD_SELECT_CELL(id, func) The user moved to, and selected a
cell; variant taking a window identifier.
Processes a wxEVT_GRID_SELECT_CELL.

EVT_GRID_CMD_EDITOR_HIDDEN(id, func) The editor for a cell was hidden;
variant taking a window identifier. Processes a
wxEVT_GRID_EDITOR_HIDDEN.

EVT_GRID_CMD_EDITOR_SHOWN(id, func) The editor for a cell was shown;
variant taking a window identifier. Processes a
wxEVT_GRID_EDITOR_SHOWN.

The event handler for the following functions takes a *wxGridSizeEvent* (p. 673)
parameter. The ..._CMD_... variants also take a window identifier.

EVT_GRID_COL_SIZE(func) The user resized a column by dragging it.
Processes a wxEVT_GRID_COL_SIZE.

EVT_GRID_ROW_SIZE(func) The user resized a row by dragging it.
Processes a wxEVT_GRID_ROW_SIZE.

EVT_GRID_CMD_COL_SIZE(func) The user resized a column by dragging it;

variant taking a window identifier. Processes a
wxEVT_GRID_COL_SIZE.

EVT_GRID_CMD_ROW_SIZE(func)   The user resized a row by dragging it; variant
taking a window identifier. Processes a
wxEVT_GRID_ROW_SIZE.

The event handler for the following functions takes a *wxGridRangeSelectEvent* (p. 671)
parameter. The ..._CMD_... variant also takes a window identifier.
EVT_GRID_RANGE_SELECT(func)   The user selected a group of contiguous cells.
Processes a wxEVT_GRID_RANGE_SELECT.

EVT_GRID_CMD_RANGE_SELECT(id, func)   The user selected a group of
contiguous cells; variant taking a window
identifier. Processes a
wxEVT_GRID_RANGE_SELECT.

The event handler for the following functions takes a *wxGridEditorCreatedEvent* (p. 666)
parameter. The ..._CMD_... variant also takes a window identifier.
EVT_GRID_EDITOR_CREATED(func) The editor for a cell was created. Processes a
wxEVT_GRID_EDITOR_CREATED.

EVT_GRID_CMD_EDITOR_CREATED(id, func)   The editor for a cell was created;
variant taking a window identifier. Processes a
wxEVT_GRID_EDITOR_CREATED.

See also
wxGrid overview (p. 1813)

Constructors and initialization

wxGrid (p. 623)
~wxGrid (p. 623)
CreateGrid (p. 627)
SetTable (p. 652)

Display format

Selection functions

wxGrid::ClearSelection (p. 626)
wxGrid::IsSelection (p. 640)
wxGrid::SelectAll (p. 643)
wxGrid::SelectBlock (p. 643)
wxGrid::SelectCol (p. 643)
wxGrid::SelectRow (p. 643)

wxGrid::wxGrid

wxGrid()

Default constructor

wxGrid(**wxWindow*** *parent*, **wxWindowID** *id*, **const wxPoint&** *pos* = *wxDefaultPosition*, **const wxSize&** *size* = *wxDefaultSize*, **long** *style* = *wxWANTS_CHARS*, **const wxString&** *name* = *wxPanelNameStr*)

Constructor to create a grid object. Call either *wxGrid::CreateGrid* (p. 627) or *wxGrid::SetTable* (p. 652) directly after this to initialize the grid before using it.

wxGrid::~wxGrid

~wxGrid()

Destructor. This will also destroy the associated grid table unless you passed a table object to the grid and specified that the grid should not take ownership of the table (see *wxGrid::SetTable* (p. 652)).

wxGrid::AppendCols

bool AppendCols(**int** *numCols = 1*, **bool** *updateLabels = true*)

Appends one or more new columns to the right of the grid and returns true if successful. The updateLabels argument is not used at present.

If you are using a derived grid table class you will need to override *wxGridTableBase::AppendCols* (p. 681). See *wxGrid::InsertCols* (p. 638) for further information.

wxGrid::AppendRows

bool AppendRows(**int** *numRows = 1*, **bool** *updateLabels = true*)

Appends one or more new rows to the bottom of the grid and returns true if successful. The updateLabels argument is not used at present.

If you are using a derived grid table class you will need to override *wxGridTableBase::AppendRows* (p. 681). See *wxGrid::InsertRows* (p. 639) for further information.

wxGrid::AutoSize

void AutoSize()

Automatically sets the height and width of all rows and columns to fit their contents.

Note
wxGrid sets up arrays to store individual row and column sizes when non-default sizes are used. The memory requirements for this could become prohibitive if your grid is very large.

wxGrid::AutoSizeColOrRow

void AutoSizeColOrRow(**int** *n*, **bool** *setAsMin*, **bool** *column*)

Common part of AutoSizeColumn/Row() or row?

wxGrid::AutoSizeColumn

void AutoSizeColumn(**int** *col*, **bool** *setAsMin = true*)

Automatically sizes the column to fit its contents. If setAsMin is true the calculated width will also be set as the minimal width for the column.

Note
wxGrid sets up arrays to store individual row and column sizes when non-default sizes are used. The memory requirements for this could become prohibitive if your grid is very large.

wxGrid::AutoSizeColumns

void AutoSizeColumns(**bool** *setAsMin = true*)

Automatically sizes all columns to fit their contents. If setAsMin is true the calculated widths will also be set as the minimal widths for the columns.

Note
wxGrid sets up arrays to store individual row and column sizes when non-default sizes are used. The memory requirements for this could become prohibitive if your grid is very large.

wxGrid::AutoSizeRow

void AutoSizeRow(**int** *row*, **bool** *setAsMin = true*)

Automatically sizes the row to fit its contents. If setAsMin is true the calculated height will also be set as the minimal height for the row.

Note
wxGrid sets up arrays to store individual row and column sizes when non-default sizes

are used. The memory requirements for this could become prohibitive if your grid is very large.

wxGrid::AutoSizeRows

void AutoSizeRows(**bool** *setAsMin = true*)

Automatically sizes all rows to fit their contents. If setAsMin is true the calculated heights will also be set as the minimal heights for the rows.

Note
wxGrid sets up arrays to store individual row and column sizes when non-default sizes are used. The memory requirements for this could become prohibitive if your grid is very large.

wxGrid::BeginBatch

void BeginBatch()

Increments the grid's batch count. When the count is greater than zero repainting of the grid is suppressed. Each call to BeginBatch must be matched by a later call to *wxGrid::EndBatch* (p. 629). Code that does a lot of grid modification can be enclosed between BeginBatch and EndBatch calls to avoid screen flicker. The final EndBatch will cause the grid to be repainted.

wxGrid::BlockToDeviceRect

wxRect BlockToDeviceRect(**const wxGridCellCoords &** *topLeft*, **const wxGridCellCoords &** *bottomRight*)

This function returns the rectangle that encloses the block of cells limited by TopLeft and BottomRight cell in device coords and clipped to the client size of the grid window.

wxGrid::CanDragColSize

bool CanDragColSize()

Returns true if columns can be resized by dragging with the mouse. Columns can be resized by dragging the edges of their labels. If grid line dragging is enabled they can also be resized by dragging the right edge of the column in the grid cell area (see *wxGrid::EnableDragGridSize* (p. 628)).

wxGrid::CanDragRowSize

bool CanDragRowSize()

Returns true if rows can be resized by dragging with the mouse. Rows can be resized by

dragging the edges of their labels. If grid line dragging is enabled they can also be resized by dragging the lower edge of the row in the grid cell area (see *wxGrid::EnableDragGridSize* (p. 628)).

wxGrid::CanDragGridSize

bool CanDragGridSize()

Return true if the dragging of grid lines to resize rows and columns is enabled or false otherwise.

wxGrid::CanEnableCellControl

bool CanEnableCellControl() **const**

Returns true if the in-place edit control for the current grid cell can be used and false otherwise (e.g. if the current cell is read-only).

wxGrid::CanHaveAttributes

bool CanHaveAttributes()

Do we have some place to store attributes in?

wxGrid::CellToRect

wxRect CellToRect(**int** *row*, **int** *col*)

wxRect CellToRect(**const wxGridCellCoords&** *coords*)

Return the rectangle corresponding to the grid cell's size and position in logical coordinates.

wxGrid::ClearGrid

void ClearGrid()

Clears all data in the underlying grid table and repaints the grid. The table is not deleted by this function. If you are using a derived table class then you need to override *wxGridTableBase::Clear* (p. 681) for this function to have any effect.

wxGrid::ClearSelection

void ClearSelection()

Deselects all cells that are currently selected.

wxGrid::CreateGrid

bool CreateGrid(**int** *numRows*, **int** *numCols*, **wxGrid::wxGridSelectionModes** *selmode* = *wxGrid::wxGridSelectCells*)

Creates a grid with the specified initial number of rows and columns. Call this directly after the grid constructor. When you use this function wxGrid will create and manage a simple table of string values for you. All of the grid data will be stored in memory.

For applications with more complex data types or relationships, or for dealing with very large datasets, you should derive your own grid table class and pass a table object to the grid with *wxGrid::SetTable* (p. 652).

wxGrid::DeleteCols

bool DeleteCols(**int** *pos = 0*, **int** *numCols = 1*, **bool** *updateLabels = true*)

Deletes one or more columns from a grid starting at the specified position and returns true if successful. The updateLabels argument is not used at present.

If you are using a derived grid table class you will need to override *wxGridTableBase::DeleteCols* (p. 681). See *wxGrid::InsertCols* (p. 638) for further information.

wxGrid::DeleteRows

bool DeleteRows(**int** *pos = 0*, **int** *numRows = 1*, **bool** *updateLabels = true*)

Deletes one or more rows from a grid starting at the specified position and returns true if successful. The updateLabels argument is not used at present.

If you are using a derived grid table class you will need to override *wxGridTableBase::DeleteRows* (p. 681). See *wxGrid::InsertRows* (p. 639) for further information.

wxGrid::DisableCellEditControl

void DisableCellEditControl()

Disables in-place editing of grid cells. Equivalent to calling EnableCellEditControl(false).

wxGrid::DisableDragColSize

void DisableDragColSize()

Disables column sizing by dragging with the mouse. Equivalent to passing false to *wxGrid::EnableDragColSize* (p. 628).

wxGrid::DisableDragGridSize

void DisableDragGridSize()

Disable mouse dragging of grid lines to resize rows and columns. Equivalent to passing false to *wxGrid::EnableDragGridSize* (p. 628)

wxGrid::DisableDragRowSize

void DisableDragRowSize()

Disables row sizing by dragging with the mouse. Equivalent to passing false to *wxGrid::EnableDragRowSize* (p. 628).

wxGrid::EnableCellEditControl

void EnableCellEditControl(**bool** *enable = true*)

Enables or disables in-place editing of grid cell data. The grid will issue either a wxEVT_GRID_EDITOR_SHOWN or wxEVT_GRID_EDITOR_HIDDEN event.

wxGrid::EnableDragColSize

void EnableDragColSize(**bool** *enable = true*)

Enables or disables column sizing by dragging with the mouse.

wxGrid::EnableDragGridSize

void EnableDragGridSize(**bool** *enable = true*)

Enables or disables row and column resizing by dragging gridlines with the mouse.

wxGrid::EnableDragRowSize

void EnableDragRowSize(**bool** *enable = true*)

Enables or disables row sizing by dragging with the mouse.

wxGrid::EnableEditing

void EnableEditing(**bool** *edit*)

If the edit argument is false this function sets the whole grid as read-only. If the argument is true the grid is set to the default state where cells may be editable. In the default state you can set single grid cells and whole rows and columns to be editable or

read-only via *wxGridCellAttribute::SetReadOnly* (p. 655). For single cells you can also use the shortcut function *wxGrid::SetReadOnly* (p. 650).

For more information about controlling grid cell attributes see the *wxGridCellAttr* (p. 653) cell attribute class and the *wxGrid classes overview* (p. 1813).

wxGrid::EnableGridLines

void EnableGridLines(**bool** *enable = true*)

Turns the drawing of grid lines on or off.

wxGrid::EndBatch

void EndBatch()

Decrements the grid's batch count. When the count is greater than zero repainting of the grid is suppressed. Each previous call to *wxGrid::BeginBatch* (p. 625) must be matched by a later call to EndBatch. Code that does a lot of grid modification can be enclosed between BeginBatch and EndBatch calls to avoid screen flicker. The final EndBatch will cause the grid to be repainted.

wxGrid::Fit

void Fit()

Overridden wxWindow method.

wxGrid::ForceRefresh

void ForceRefresh()

Causes immediate repainting of the grid. Use this instead of the usual wxWindow::Refresh.

wxGrid::GetBatchCount

int GetBatchCount()

Returns the number of times that *wxGrid::BeginBatch* (p. 625) has been called without (yet) matching calls to *wxGrid::EndBatch* (p. 629). While the grid's batch count is greater than zero the display will not be updated.

wxGrid::GetCellAlignment

void GetCellAlignment(**int** *row*, **int** *col*, **int*** *horiz*, **int*** *vert*)

Sets the arguments to the horizontal and vertical text alignment values for the grid cell at the specified location.

Horizontal alignment will be one of wxALIGN_LEFT, wxALIGN_CENTRE or wxALIGN_RIGHT.
Vertical alignment will be one of wxALIGN_TOP, wxALIGN_CENTRE or wxALIGN_BOTTOM.

wxPerl note: This method only takes the parameters `row` and `col` and returns a 2-element list (`horiz, vert`).

wxGrid::GetCellBackgroundColour

wxColour GetCellBackgroundColour(**int** *row*, **int** *col*)

Returns the background colour of the cell at the specified location.

wxGrid::GetCellEditor

wxGridCellEditor* GetCellEditor(**int** *row*, **int** *col*)

Returns a pointer to the editor for the cell at the specified location.

See *wxGridCellEditor* (p. 661) and the *wxGrid overview* (p. 1813) for more information about cell editors and renderers.

wxGrid::GetCellFont

wxFont GetCellFont(**int** *row*, **int** *col*)

Returns the font for text in the grid cell at the specified location.

wxGrid::GetCellRenderer

wxGridCellRenderer* GetCellRenderer(**int** *row*, **int** *col*)

Returns a pointer to the renderer for the grid cell at the specified location.

See *wxGridCellRenderer* (p. 677) and the *wxGrid overview* (p. 1813) for more information about cell editors and renderers.

wxGrid::GetCellTextColour

wxColour GetCellTextColour(**int** *row*, **int** *col*)

Returns the text colour for the grid cell at the specified location.

wxGrid::GetCellValue

wxString GetCellValue(**int** *row*, **int** *col*)

wxString GetCellValue(**const wxGridCellCoords&***coords*)

Returns the string contained in the cell at the specified location. For simple applications where a grid object automatically uses a default grid table of string values you use this function together with *wxGrid::SetCellValue* (p. 645) to access cell values.

For more complex applications where you have derived your own grid table class that contains various data types (e.g. numeric, boolean or user-defined custom types) then you only use this function for those cells that contain string values.

See *wxGridTableBase::CanGetValueAs* (p. 680)and the *wxGrid overview* (p. 1813) for more information.

wxGrid::GetColLeft

int GetColLeft(**int** *col*) **const**

wxGrid::GetColLabelAlignment

void GetColLabelAlignment(**int*** *horiz*, **int*** *vert*)

Sets the arguments to the current column label alignment values.

Horizontal alignment will be one of wxALIGN_LEFT, wxALIGN_CENTRE or wxALIGN_RIGHT.
Vertical alignment will be one of wxALIGN_TOP, wxALIGN_CENTRE or wxALIGN_BOTTOM.

wxPerl note: This method takes no parameters and returns a 2-element list (`horiz, vert`).

wxGrid::GetColLabelSize

int GetColLabelSize()

Returns the current height of the column labels.

wxGrid::GetColLabelValue

wxString GetColLabelValue(**int** *col*)

Returns the specified column label. The default grid table class provides column labels of the form A,B...Z,AA,AB...ZZ,AAA... If you are using a custom grid table you can override *wxGridTableBase::GetColLabelValue* (p. 682) to provide your own labels.

wxGrid::GetColMinimalAcceptableWidth

int GetColMinimalAcceptableWidth()

This returns the value of the lowest column width that can be handled correctly. See member *SetColMinimalAcceptableWidth* (p. 647) for details.

wxGrid::GetColMinimalWidth

int GetColMinimalWidth(**int** *col*) **const**

Get the minimal width of the given column/row.

wxGrid::GetColRight

int GetColRight(**int** *col*) **const**

wxGrid::GetColSize

int GetColSize(**int** *col*)

Returns the width of the specified column.

wxGrid::GetDefaultCellAlignment

void GetDefaultCellAlignment(**int*** *horiz*, **int*** *vert*)

Sets the arguments to the current default horizontal and vertical text alignment values.

Horizontal alignment will be one of wxALIGN_LEFT, wxALIGN_CENTRE or wxALIGN_RIGHT.
Vertical alignment will be one of wxALIGN_TOP, wxALIGN_CENTRE or wxALIGN_BOTTOM.

wxGrid::GetDefaultCellBackgroundColour

wxColour GetDefaultCellBackgroundColour()

Returns the current default background colour for grid cells.

wxGrid::GetDefaultCellFont

wxFont GetDefaultCellFont()

Returns the current default font for grid cell text.

wxGrid::GetDefaultCellTextColour

wxColour GetDefaultCellTextColour()

Returns the current default colour for grid cell text.

wxGrid::GetDefaultColLabelSize

int GetDefaultColLabelSize()

Returns the default height for column labels.

wxGrid::GetDefaultColSize

int GetDefaultColSize()

Returns the current default width for grid columns.

wxGrid::GetDefaultEditor

wxGridCellEditor* GetDefaultEditor() **const**

Returns a pointer to the current default grid cell editor.

See *wxGridCellEditor* (p. 661) and the *wxGrid overview* (p. 1813) for more information about cell editors and renderers.

wxGrid::GetDefaultEditorForCell

wxGridCellEditor* GetDefaultEditorForCell(**int** *row*, **int** *col*) **const**

wxGridCellEditor* GetDefaultEditorForCell(**const wxGridCellCoords&** *c*) **const**

wxGrid::GetDefaultEditorForType

wxGridCellEditor* GetDefaultEditorForType(**const wxString&** *typeName*) **const**

wxGrid::GetDefaultRenderer

wxGridCellRenderer* GetDefaultRenderer() **const**

Returns a pointer to the current default grid cell renderer.

See *wxGridCellRenderer* (p. 677) and the *wxGrid overview* (p. 1813) for more information about cell editors and renderers.

wxGrid::GetDefaultRendererForCell

wxGridCellRenderer* GetDefaultRendererForCell(**int** *row*, **int** *col*) **const**

wxGrid::GetDefaultRendererForType

wxGridCellRenderer* GetDefaultRendererForType(**const wxString&** *typeName*) **const**

wxGrid::GetDefaultRowLabelSize

int GetDefaultRowLabelSize()

Returns the default width for the row labels.

wxGrid::GetDefaultRowSize

int GetDefaultRowSize()

Returns the current default height for grid rows.

wxGrid::GetGridCursorCol

int GetGridCursorCol()

Returns the current grid cell column position.

wxGrid::GetGridCursorRow

int GetGridCursorRow()

Returns the current grid cell row position.

wxGrid::GetGridLineColour

wxColour GetGridLineColour()

Returns the colour used for grid lines.

wxGrid::GridLinesEnabled

bool GridLinesEnabled()

Returns true if drawing of grid lines is turned on, false otherwise.

wxGrid::GetLabelBackgroundColour

wxColour GetLabelBackgroundColour()

Returns the colour used for the background of row and column labels.

wxGrid::GetLabelFont

wxFont GetLabelFont()

Returns the font used for row and column labels.

wxGrid::GetLabelTextColour

wxColour GetLabelTextColour()

Returns the colour used for row and column label text.

wxGrid::GetNumberCols

int GetNumberCols()

Returns the total number of grid columns (actually the number of columns in the underlying grid table).

wxGrid::GetNumberRows

int GetNumberRows()

Returns the total number of grid rows (actually the number of rows in the underlying grid table).

wxGrid::GetOrCreateCellAttr

wxGridCellAttr* GetOrCreateCellAttr(**int** *row*, **int** *col*) **const**

wxGrid::GetRowMinimalAcceptableHeight

int GetRowMinimalAcceptableHeight()

This returns the value of the lowest row width that can be handled correctly. See member *SetRowMinimalAcceptableHeight* (p. 651) for details.

wxGrid::GetRowMinimalHeight

int GetRowMinimalHeight(**int** *col*) **const**

wxGrid::GetRowLabelAlignment

void GetRowLabelAlignment(**int*** *horiz*, **int*** *vert*)

Sets the arguments to the current row label alignment values.

Horizontal alignment will be one of wxLEFT, wxCENTRE or wxRIGHT.
Vertical alignment will be one of wxTOP, wxCENTRE or wxBOTTOM.

wxPerl note: This method takes no parameters and returns a 2-element list (horiz, vert).

wxGrid::GetRowLabelSize

int GetRowLabelSize()

Returns the current width of the row labels.

wxGrid::GetRowLabelValue

wxString GetRowLabelValue(**int** *row*)

Returns the specified row label. The default grid table class provides numeric row labels. If you are using a custom grid table you can override *wxGridTableBase::GetRowLabelValue* (p. 682) to provide your own labels.

wxGrid::GetRowSize

int GetRowSize(**int** *row*)

Returns the height of the specified row.

wxGrid::GetScrollLinesX

int GetScrollLinesX() **const**

Returns the number of pixels per horizontal scroll increment. The default is 15.

wxGrid::GetScrollLinesY

int GetScrollLinesY() **const**

Returns the number of pixels per vertical scroll increment. The default is 15.

wxGrid::GetSelectionMode

wxGrid::wxGridSelectionModes GetSelectionMode() **const**

Returns the current selection mode, see *wxGrid::SetSelectionMode* (p. 652).

wxGrid::GetSelectedCells

wxGridCellCoordsArray GetSelectedCells() **const**

Returns an array of singly selected cells.

wxGrid::GetSelectedCols

wxArrayInt GetSelectedCols() **const**

Returns an array of selected cols.

wxGrid::GetSelectedRows

wxArrayInt GetSelectedRows() **const**

Returns an array of selected rows.

wxGrid::GetSelectionBackground

wxColour GetSelectionBackground() **const**

Access or update the selection fore/back colours

wxGrid::GetSelectionBlockTopLeft

wxGridCellCoordsArray GetSelectionBlockTopLeft() **const**

Returns an array of the top left corners of blocks of selected cells, see
wxGrid::GetSelectionBlockBottomRight (p. 637).

wxGrid::GetSelectionBlockBottomRight

wxGridCellCoordsArray GetSelectionBlockBottomRight() **const**

Returns an array of the bottom right corners of blocks of selected cells, see
wxGrid::GetSelectionBlockTopLeft (p. 637).

wxGrid::GetSelectionForeground

wxColour GetSelectionForeground() **const**

wxGrid::GetTable

wxGridTableBase * GetTable() **const**

Returns a base pointer to the current table object.

wxGrid::GetViewWidth

int GetViewWidth()

Returned number of whole cols visible.

wxGrid::HideCellEditControl

void HideCellEditControl()

Hides the in-place cell edit control.

wxGrid::InitColWidths

void InitColWidths()

Init the m_colWidths/Rights arrays

wxGrid::InitRowHeights

void InitRowHeights()

NB: *never* access m_row/col arrays directly because they are created on demand, *always* use accessor functions instead!

Init the m_rowHeights/Bottoms arrays with default values.

wxGrid::InsertCols

bool InsertCols(**int** *pos = 0*, **int** *numCols = 1*, **bool** *updateLabels = true*)

Inserts one or more new columns into a grid with the first new column at the specified position and returns true if successful. The updateLabels argument is not used at present.

The sequence of actions begins with the grid object requesting the underlying grid table to insert new columns. If this is successful the table notifies the grid and the grid updates the display. For a default grid (one where you have called *wxGrid::CreateGrid* (p. 627))

this process is automatic. If you are using a custom grid table (specified with *wxGrid::SetTable* (p. 652)) then you must override *wxGridTableBase::InsertCols* (p. 681) in your derived table class.

wxGrid::InsertRows

bool InsertRows(**int** *pos = 0*, **int** *numRows = 1*, **bool** *updateLabels = true*)

Inserts one or more new rows into a grid with the first new row at the specified position and returns true if successful. The updateLabels argument is not used at present.

The sequence of actions begins with the grid object requesting the underlying grid table to insert new rows. If this is successful the table notifies the grid and the grid updates the display. For a default grid (one where you have called *wxGrid::CreateGrid* (p. 627)) this process is automatic. If you are using a custom grid table (specified with *wxGrid::SetTable* (p. 652)) then you must override *wxGridTableBase::InsertRows* (p. 681) in your derived table class.

wxGrid::IsCellEditControlEnabled

bool IsCellEditControlEnabled() **const**

Returns true if the in-place edit control is currently enabled.

wxGrid::IsCurrentCellReadOnly

bool IsCurrentCellReadOnly() **const**

Returns true if the current cell has been set to read-only (see *wxGrid::SetReadOnly* (p. 650)).

wxGrid::IsEditable

bool IsEditable()

Returns false if the whole grid has been set as read-only or true otherwise. See *wxGrid::EnableEditing* (p. 628) for more information about controlling the editing status of grid cells.

wxGrid::IsInSelection

bool IsInSelection(**int** *row*, **int** *col*) **const**

bool IsInSelection(**const wxGridCellCoords&** *coords*) **const**

Is this cell currently selected.

wxGrid::IsReadOnly

bool IsReadOnly(**int** *row*, **int** *col*) **const**

Returns true if the cell at the specified location can't be edited. See also *wxGrid::IsReadOnly* (p. 640).

wxGrid::IsSelection

bool IsSelection()

Returns true if there are currently rows, columns or blocks of cells selected.

wxGrid::IsVisible

bool IsVisible(**int** *row*, **int** *col*, **bool** *wholeCellVisible = true*)

bool IsVisible(**const wxGridCellCoords&** *coords*, **bool** *wholeCellVisible = true*)

Returns true if a cell is either wholly visible (the default) or at least partially visible in the grid window.

wxGrid::MakeCellVisible

void MakeCellVisible(**int** *row*, **int** *col*)

void MakeCellVisible(**const wxGridCellCoords&** *coords*)

Brings the specified cell into the visible grid cell area with minimal scrolling. Does nothing if the cell is already visible.

wxGrid::MoveCursorDown

bool MoveCursorDown(**bool** *expandSelection*)

Moves the grid cursor down by one row. If a block of cells was previously selected it will expand if the argument is true or be cleared if the argument is false.

Keyboard
This function is called for Down cursor key presses or Shift+Down to expand a selection.

wxGrid::MoveCursorLeft

bool MoveCursorLeft(**bool** *expandSelection*)

Moves the grid cursor left by one column. If a block of cells was previously selected it will expand if the argument is true or be cleared if the argument is false.

Keyboard
This function is called for Left cursor key presses or Shift+Left to expand a selection.

wxGrid::MoveCursorRight

bool MoveCursorRight(**bool** *expandSelection*)

Moves the grid cursor right by one column. If a block of cells was previously selected it will expand if the argument is true or be cleared if the argument is false.

Keyboard
This function is called for Right cursor key presses or Shift+Right to expand a selection.

wxGrid::MoveCursorUp

bool MoveCursorUp(**bool** *expandSelection*)

Moves the grid cursor up by one row. If a block of cells was previously selected it will expand if the argument is true or be cleared if the argument is false.

Keyboard
This function is called for Up cursor key presses or Shift+Up to expand a selection.

wxGrid::MoveCursorDownBlock

bool MoveCursorDownBlock(**bool** *expandSelection*)

Moves the grid cursor down in the current column such that it skips to the beginning or end of a block of non-empty cells. If a block of cells was previously selected it will expand if the argument is true or be cleared if the argument is false.

Keyboard
This function is called for the Ctrl+Down key combination. Shift+Ctrl+Down expands a selection.

wxGrid::MoveCursorLeftBlock

bool MoveCursorLeftBlock(**bool** *expandSelection*)

Moves the grid cursor left in the current row such that it skips to the beginning or end of a block of non-empty cells. If a block of cells was previously selected it will expand if the argument is true or be cleared if the argument is false.

Keyboard
This function is called for the Ctrl+Left key combination. Shift+Ctrl+left expands a selection.

wxGrid::MoveCursorRightBlock

bool MoveCursorRightBlock(**bool** *expandSelection*)

Moves the grid cursor right in the current row such that it skips to the beginning or end of a block of non-empty cells. If a block of cells was previously selected it will expand if the argument is true or be cleared if the argument is false.

Keyboard
This function is called for the Ctrl+Right key combination. Shift+Ctrl+Right expands a selection.

wxGrid::MoveCursorUpBlock

bool MoveCursorUpBlock(**bool** *expandSelection*)

Moves the grid cursor up in the current column such that it skips to the beginning or end of a block of non-empty cells. If a block of cells was previously selected it will expand if the argument is true or be cleared if the argument is false.

Keyboard
This function is called for the Ctrl+Up key combination. Shift+Ctrl+Up expands a selection.

wxGrid::MovePageDown

bool MovePageDown()

Moves the grid cursor down by some number of rows so that the previous bottom visible row becomes the top visible row.

Keyboard
This function is called for PgDn keypresses.

wxGrid::MovePageUp

bool MovePageUp()

Moves the grid cursor up by some number of rows so that the previous top visible row becomes the bottom visible row.

Keyboard
This function is called for PgUp keypresses.

wxGrid::RegisterDataType

void RegisterDataType(**const wxString&** *typeName*, **wxGridCellRenderer*** *renderer*, **wxGridCellEditor*** *editor*)

Methods for a registry for mapping data types to Renderers/Editors

wxGrid::SaveEditControlValue

void SaveEditControlValue()

Sets the value of the current grid cell to the current in-place edit control value. This is called automatically when the grid cursor moves from the current cell to a new cell. It is also a good idea to call this function when closing a grid since any edits to the final cell location will not be saved otherwise.

wxGrid::SelectAll

void SelectAll()

Selects all cells in the grid.

wxGrid::SelectBlock

void SelectBlock(int *topRow*, int *leftCol*, int *bottomRow*, int *rightCol*, bool *addToSelected = false*)

void SelectBlock(const wxGridCellCoords& *topLeft*, const wxGridCellCoords& *bottomRight*, bool *addToSelected = false*)

Selects a rectangular block of cells. If addToSelected is false then any existing selection will be deselected; if true the column will be added to the existing selection.

wxGrid::SelectCol

void SelectCol(int *col*, bool *addToSelected = false*)

Selects the specified column. If addToSelected is false then any existing selection will be deselected; if true the column will be added to the existing selection.

wxGrid::SelectionToDeviceRect

wxRect SelectionToDeviceRect()

This function returns the rectangle that encloses the selected cells in device coords and clipped to the client size of the grid window.

wxGrid::SelectRow

void SelectRow(int *row*, bool *addToSelected = false*)

Selects the specified row. If addToSelected is false then any existing selection will be deselected; if true the row will be added to the existing selection.

wxGrid::SetCellAlignment

void SetCellAlignment(**int** *row*, **int** *col*, **int** *horiz,* **int** *vert*)

void SetCellAlignment(**int** *align*, **int** *row*, **int** *col*)

Sets the horizontal and vertical alignment for grid cell text at the specified location.

Horizontal alignment should be one of wxALIGN_LEFT, wxALIGN_CENTRE or wxALIGN_RIGHT.
Vertical alignment should be one of wxALIGN_TOP, wxALIGN_CENTRE or wxALIGN_BOTTOM.

wxGrid::SetCellBackgroundColour

void SetCellBackgroundColour(**int** *row*, **int** *col*, **const wxColour&** *colour*)

wxGrid::SetCellEditor

void SetCellEditor(**int** *row*, **int** *col*, **wxGridCellEditor*** *editor*)

Sets the editor for the grid cell at the specified location. The grid will take ownership of the pointer.

See *wxGridCellEditor* (p. 661) and the *wxGrid overview* (p. 1813) for more information about cell editors and renderers.

wxGrid::SetCellFont

void SetCellFont(**int** *row*, **int** *col*, **const wxFont&** *font*)

Sets the font for text in the grid cell at the specified location.

wxGrid::SetCellRenderer

void SetCellRenderer(**int** *row*, **int** *col*, **wxGridCellRenderer*** *renderer*)

Sets the renderer for the grid cell at the specified location. The grid will take ownership of the pointer.

See *wxGridCellRenderer* (p. 677) and the *wxGrid overview* (p. 1813) for more information about cell editors and renderers.

wxGrid::SetCellTextColour

void SetCellTextColour(**int** *row*, **int** *col*, **const wxColour&** *colour*)

void SetCellTextColour(**const wxColour&** *val*, **int** *row*, **int** *col*)

void SetCellTextColour(**const wxColour&** *colour*)

Sets the text colour for the grid cell at the specified location.

wxGrid::SetCellValue

void SetCellValue(**int** *row*, **int** *col*, **const wxString&** *s*)

void SetCellValue(**const wxGridCellCoords&** *coords*, **const wxString&** *s*)

void SetCellValue(**const wxString&** *val*, **int** *row*, **int** *col*)

Sets the string value for the cell at the specified location. For simple applications where a grid object automatically uses a default grid table of string values you use this function together with *wxGrid::GetCellValue* (p. 631) to access cell values.

For more complex applications where you have derived your own grid table class that contains various data types (e.g. numeric, boolean or user-defined custom types) then you only use this function for those cells that contain string values.

The last form is for backward compatibility only.

See *wxGridTableBase::CanSetValueAs* (p. 680)and the *wxGrid overview* (p. 1813) for more information.

wxGrid::SetColAttr

void SetColAttr(**int** *col*, **wxGridCellAttr*** *attr*)

Sets the cell attributes for all cells in the specified column.

For more information about controlling grid cell attributes see the *wxGridCellAttr* (p. 653) cell attribute class and the *wxGrid classes overview* (p. 1813).

wxGrid::SetColFormatBool

void SetColFormatBool(**int** *col*)

Sets the specified column to display boolean values. wxGrid displays boolean values with a checkbox.

wxGrid::SetColFormatNumber

void SetColFormatNumber(**int** *col*)

Sets the specified column to display integer values.

wxGrid::SetColFormatFloat

void SetColFormatFloat(**int** *col*, **int** *width = -1*, **int** *precision = -1*)

Sets the specified column to display floating point values with the given width and precision.

wxGrid::SetColFormatCustom

void SetColFormatCustom(**int** *col*, **const wxString&** *typeName*)

Sets the specified column to display data in a custom format. See the *wxGrid overview* (p. 1813) for more information on working with custom data types.

wxGrid::SetColLabelAlignment

void SetColLabelAlignment(**int** *horiz*, **int** *vert*)

Sets the horizontal and vertical alignment of column label text.

Horizontal alignment should be one of wxALIGN_LEFT, wxALIGN_CENTRE or wxALIGN_RIGHT.

Vertical alignment should be one of wxALIGN_TOP, wxALIGN_CENTRE or wxALIGN_BOTTOM.

wxGrid::SetColLabelSize

void SetColLabelSize(**int** *height*)

Sets the height of the column labels.

wxGrid::SetColLabelValue

void SetColLabelValue(**int** *col*, **const wxString&** *value*)

Set the value for the given column label. If you are using a derived grid table you must override *wxGridTableBase::SetColLabelValue* (p. 682)for this to have any effect.

wxGrid::SetColMinimalWidth

void SetColMinimalWidth(**int** *col*, **int** *width*)

Sets the minimal width for the specified column. This should normally be called when creating the grid because it will not resize a column that is already narrower than the minimal width. The width argument must be higher than the minimimal acceptable column width, see*wxGrid::GetColMinimalAcceptableWidth* (p. 632).

wxGrid::SetColMinimalAcceptableWidth

void SetColMinimalAcceptableWidth(**int** *width*)

This modifies the minimum column width that can be handled correctly. Specifying a low value here allows smaller grid cells to be dealt with correctly. Specifying a value here which is much smaller than the actual minimum size will incur a performance penalty in the functions which perform grid cell index lookup on the basis of screen coordinates. This should normally be called when creating the grid because it will not resize existing columns with sizes smaller than the value specified here.

wxGrid::SetColSize

void SetColSize(**int** *col*, **int** *width*)

Sets the width of the specified column.

This function does not refresh the grid. If you are calling it outside of a BeginBatch / EndBatch block you can use *wxGrid::ForceRefresh* (p. 629) to see the changes.

Automatically sizes the column to fit its contents. If setAsMin is true the calculated width will also be set as the minimal width for the column.

Note
wxGrid sets up arrays to store individual row and column sizes when non-default sizes are used. The memory requirements for this could become prohibitive if your grid is very large.

wxGrid::SetDefaultCellAlignment

void SetDefaultCellAlignment(**int** *horiz*, **int** *vert*)

Sets the default horizontal and vertical alignment for grid cell text.

Horizontal alignment should be one of wxALIGN_LEFT, wxALIGN_CENTRE or wxALIGN_RIGHT.

Vertical alignment should be one of wxALIGN_TOP, wxALIGN_CENTRE or wxALIGN_BOTTOM.

wxGrid::SetDefaultCellBackgroundColour

void SetDefaultCellBackgroundColour(**const wxColour&** *colour*)

Sets the default background colour for grid cells.

wxGrid::SetDefaultCellFont

void SetDefaultCellFont(**const wxFont&** *font*)

Sets the default font to be used for grid cell text.

wxGrid::SetDefaultCellTextColour

void SetDefaultCellTextColour(**const wxColour&** *colour*)

Sets the current default colour for grid cell text.

wxGrid::SetDefaultEditor

void SetDefaultEditor(**wxGridCellEditor*** *editor*)

Sets the default editor for grid cells. The grid will take ownership of the pointer.

See *wxGridCellEditor* (p. 661) and the *wxGrid overview* (p. 1813) for more information about cell editors and renderers.

wxGrid::SetDefaultRenderer

void SetDefaultRenderer(**wxGridCellRenderer*** *renderer*)

Sets the default renderer for grid cells. The grid will take ownership of the pointer.

See *wxGridCellRenderer* (p. 677) and the *wxGrid overview* (p. 1813) for more information about cell editors and renderers.

wxGrid::SetDefaultColSize

void SetDefaultColSize(**int** *width*, **bool** *resizeExistingCols = false*)

Sets the default width for columns in the grid. This will only affect columns subsequently added to the grid unless resizeExistingCols is true.

wxGrid::SetDefaultRowSize

void SetDefaultRowSize(**int** *height*, **bool** *resizeExistingRows = false*)

Sets the default height for rows in the grid. This will only affect rows subsequently added to the grid unless resizeExistingRows is true.

wxGrid::SetGridCursor

void SetGridCursor(**int** *row*, **int** *col*)

Set the grid cursor to the specified cell. This function calls *wxGrid::MakeCellVisible* (p. 640).

wxGrid::SetGridLineColour

void SetGridLineColour(**const wxColour&***colour*)

Sets the colour used to draw grid lines.

wxGrid::SetLabelBackgroundColour

void SetLabelBackgroundColour(**const wxColour&** *colour*)

Sets the background colour for row and column labels.

wxGrid::SetLabelFont

void SetLabelFont(**const wxFont&** *font*)

Sets the font for row and column labels.

wxGrid::SetLabelTextColour

void SetLabelTextColour(**const wxColour&** *colour*)

Sets the colour for row and column label text.

wxGrid::SetMargins

void SetMargins(**int** *extraWidth*, **int** *extraHeight*)

A grid may occupy more space than needed for its rows/columns. This function allows to set how big this extra space is

wxGrid::SetOrCalcColumnSizes

int SetOrCalcColumnSizes(**bool** *calcOnly*, **bool** *setAsMin = true*)

Common part of AutoSizeColumn/Row() and GetBestSize()

wxGrid::SetOrCalcRowSizes

int SetOrCalcRowSizes(**bool** *calcOnly*, **bool** *setAsMin = true*)

wxGrid::SetReadOnly

void SetReadOnly(**int** *row*, **int** *col*, **bool** *isReadOnly = true*)

Makes the cell at the specified location read-only or editable. See also *wxGrid::IsReadOnly* (p. 640).

wxGrid::SetRowAttr

void SetRowAttr(**int** *row*, **wxGridCellAttr*** *attr*)

Sets the cell attributes for all cells in the specified row. See the *wxGridCellAttr* (p. 653) class for more information about controlling cell attributes.

wxGrid::SetRowLabelAlignment

void SetRowLabelAlignment(**int** *horiz*, **int** *vert*)

Sets the horizontal and vertical alignment of row label text.

Horizontal alignment should be one of wxALIGN_LEFT, wxALIGN_CENTRE or wxALIGN_RIGHT.

Vertical alignment should be one of wxALIGN_TOP, wxALIGN_CENTRE or wxALIGN_BOTTOM.

wxGrid::SetRowLabelSize

void SetRowLabelSize(**int** *width*)

Sets the width of the row labels.

wxGrid::SetRowLabelValue

void SetRowLabelValue(**int** *row*, **const wxString&** *value*)

Set the value for the given row label. If you are using a derived grid table you must override *wxGridTableBase::SetRowLabelValue* (p. 682) for this to have any effect.

wxGrid::SetRowMinimalHeight

void SetRowMinimalHeight(**int** *row*, **int** *height*)

Sets the minimal height for the specified row. This should normally be called when

creating the grid because it will not resize a row that is already shorter than the minimal height. The height argument must be higher than the minimimal acceptable row height, see*wxGrid::GetRowMinimalAcceptableHeight* (p. 635).

wxGrid::SetRowMinimalAcceptableHeight

void SetRowMinimalAcceptableHeight(**int** *height*)

This modifies the minimum row width that can be handled correctly. Specifying a low value here allows smaller grid cells to be dealt with correctly. Specifying a value here which is much smaller than the actual minimum size will incur a performance penalty in the functions which perform grid cell index lookup on the basis of screen coordinates. This should normally be called when creating the grid because it will not resize existing rows with sizes smaller than the value specified here.

wxGrid::SetRowSize

void SetRowSize(**int** *row*, **int** *height*)

Sets the height of the specified row.

This function does not refresh the grid. If you are calling it outside of a BeginBatch / EndBatch block you can use *wxGrid::ForceRefresh* (p. 629) to see the changes.

Automatically sizes the column to fit its contents. If setAsMin is true the calculated width will also be set as the minimal width for the column.

Note

wxGrid sets up arrays to store individual row and column sizes when non-default sizes are used. The memory requirements for this could become prohibitive if your grid is very large.

wxGrid::SetScrollLinesX

void SetScrollLinesX(**int** *x*)

Sets the number of pixels per horizontal scroll increment. The default is 15. Sometimes wxGrid has trouble setting the scrollbars correctly due to rounding errors: setting this to 1 can help.

wxGrid::SetScrollLinesY

void SetScrollLinesY(**int** *y*)

Sets the number of pixels per vertical scroll increment. The default is 15. Sometimes wxGrid has trouble setting the scrollbars correctly due to rounding errors: setting this to 1 can help.

wxGrid::SetSelectionBackground

void SetSelectionBackground(**const wxColour&** *c*)

wxGrid::SetSelectionForeground

void SetSelectionForeground(**const wxColour&** *c*)

wxGrid::SetSelectionMode

void SetSelectionMode(**wxGrid::wxGridSelectionModes** *selmode*)

Set the selection behaviour of the grid.

Parameters

wxGrid::wxGridSelectCells

> The default mode where individual cells are selected.

wxGrid::wxGridSelectRows

> Selections will consist of whole rows.

wxGrid::wxGridSelectColumns

> Selections will consist of whole columns.

wxGrid::SetTable

bool SetTable(**wxGridTableBase*** *table*, **bool** *takeOwnership* = *false*, **wxGrid::wxGridSelectionModes** *selmode* = *wxGrid::wxGridSelectCells*)

Passes a pointer to a custom grid table to be used by the grid. This should be called after the grid constructor and before using the grid object. If takeOwnership is set to true then the table will be deleted by the wxGrid destructor.

Use this function instead of *wxGrid::CreateGrid* (p. 627) when your application involves complex or non-string data or data sets that are too large to fit wholly in memory.

wxGrid::ShowCellEditControl

void ShowCellEditControl()

Displays the in-place cell edit control for the current cell.

wxGrid::XToCol

int XToCol(int *x*)

Returns the grid column that corresponds to the logical x coordinate.
Returns wxNOT_FOUND if there is no column at the x position.

wxGrid::XToEdgeOfCol

int XToEdgeOfCol(int *x*)

Returns the column whose right hand edge is close to the given logical x position. If no column edge is near to this position wxNOT_FOUND is returned.

wxGrid::YToEdgeOfRow

int YToEdgeOfRow(int *y*)

Returns the row whose bottom edge is close to the given logical y position. If no row edge is near to this position wxNOT_FOUND is returned.

wxGrid::YToRow

int YToRow(int *y*)

Returns the grid row that corresponds to the logical y coordinate. Returns wxNOT_FOUND if there is no row at the y position.

wxGridCellAttr

This class can be used to alter the cells' appearance in the grid by changing their colour/font/... from default. An object of this class may be returned by wxGridTable::GetAttr().

Derived from

No base class

Include files

<wx/grid.h>

wxGridCellAttr::wxGridCellAttr

wxGridCellAttr()

Default constructor.   **wxGridCellAttr(const wxColour&** *colText*, **const**

wxColour& *colBack*, **const wxFont&** *font*, **int** *hAlign*, **int** *vAlign*)

VZ: considering the number of members wxGridCellAttr has now, this ctor seems to be pretty useless... may be we should just remove it?

wxGridCellAttr::Clone

wxGridCellAttr* Clone() **const**

Creates a new copy of this object.

wxGridCellAttr::IncRef

void IncRef()

This class is ref counted: it is created with ref count of 1, so calling DecRef() once will delete it. Calling IncRef() allows to lock it until the matching DecRef() is called

wxGridCellAttr::DecRef

void DecRef()

wxGridCellAttr::SetTextColour

void SetTextColour(**const wxColour&** *colText*)

Sets the text colour.

wxGridCellAttr::SetBackgroundColour

void SetBackgroundColour(**const wxColour&** *colBack*)

Sets the background colour.

wxGridCellAttr::SetFont

void SetFont(**const wxFont&** *font*)

Sets the font.

wxGridCellAttr::SetAlignment

void SetAlignment(**int** *hAlign*, **int** *vAlign*)

Sets the alignment.

wxGridCellAttr::SetReadOnly

void SetReadOnly(**bool** *isReadOnly* = *true*)

wxGridCellAttr::SetRenderer

void SetRenderer(**wxGridCellRenderer*** *renderer*)

takes ownership of the pointer

wxGridCellAttr::SetEditor

void SetEditor(**wxGridCellEditor*** *editor*)

wxGridCellAttr::HasTextColour

bool HasTextColour() **const**

accessors

wxGridCellAttr::HasBackgroundColour

bool HasBackgroundColour() **const**

wxGridCellAttr::HasFont

bool HasFont() **const**

wxGridCellAttr::HasAlignment

bool HasAlignment() **const**

wxGridCellAttr::HasRenderer

bool HasRenderer() **const**

wxGridCellAttr::HasEditor

bool HasEditor() **const**

wxGridCellAttr::GetTextColour

const wxColour& GetTextColour() **const**

wxGridCellAttr::GetBackgroundColour

const wxColour& GetBackgroundColour() **const**

wxGridCellAttr::GetFont

const wxFont& GetFont() **const**

wxGridCellAttr::GetAlignment

void GetAlignment(**int*** *hAlign*, **int*** *vAlign*) **const**

wxPerl note: This method takes no parameters and returns a 2-element list (`hAlign,` `vAlign ).`

wxGridCellAttr::GetRenderer

wxGridCellRenderer* GetRenderer(**wxGrid*** *grid*, **int** *row*, **int** *col*) **const**

wxGridCellAttr::GetEditor

wxGridCellEditor* GetEditor(**wxGrid*** *grid*, **int** *row*, **int** *col*) **const**

wxGridCellAttr::IsReadOnly

bool IsReadOnly() **const**

wxGridCellAttr::SetDefAttr

void SetDefAttr(**wxGridCellAttr*** *defAttr*)

wxGridBagSizer

A *wxSizer* (p. 1176) that can lay out items in a virtual grid like a *wxFlexGridSizer* (p. 554) but in this case explicit positioning of the items is allowed using *wxGBPosition* (p. 603), and items can optionally span more than one row and/or column using *wxGBSpan* (p. 606).

Derived from

wxFlexGridSizer (p. 554)
wxGridSizer (p. 683)
wxSizer (p. 1176)
wxObject (p. 1012)

Include files

<wx/gbsizer.h>

wxGridBagSizer::wxGridBagSizer

wxGridBagSizer(**int** *vgap = 0*, **int** *hgap = 0*)

Constructor, with optional parameters to specify the gap between the rows and columns.

wxGridBagSizer::Add

wxSizerItem* Add(**wxWindow*** *window*, **const wxGBPosition&** *pos*, **const wxGBSpan&** *span = wxDefaultSpan*, **int** *flag = 0*, **int** *border = 0*, **wxObject*** *userData = NULL*)

wxSizerItem* Add(**wxSizer*** *sizer*, **const wxGBPosition&** *pos*, **const wxGBSpan&** *span = wxDefaultSpan*, **int** *flag = 0*, **int** *border = 0*, **wxObject*** *userData = NULL*)

wxSizerItem* Add(**int** *width*, **int** *height*, **const wxGBPosition&** *pos*, **const wxGBSpan&** *span = wxDefaultSpan*, **int** *flag = 0*, **int** *border = 0*, **wxObject*** *userData = NULL*)

wxSizerItem* Add(**wxGBSizerItem*** *item*)

The Add methods return a valid pointer if the item was successfully placed at the given position, NULL if something was already there.

wxGridBagSizer::CalcMin

wxSize CalcMin()

Called when the managed size of the sizer is needed or when layout needs done.

wxGridBagSizer::CheckForIntersection

bool CheckForIntersection(**wxGBSizerItem*** *item*, **wxGBSizerItem*** *excludeItem = NULL*)

bool CheckForIntersection(**const wxGBPosition&** *pos*, **const wxGBSpan&** *span*, **wxGBSizerItem*** *excludeItem = NULL*)

Look at all items and see if any intersect (or would overlap) the given item. Returns true if so, false if there would be no overlap. If an excludeItem is given then it will not be checked for intersection, for example it may be the item we are checking the position of.

wxGridBagSizer::FindItem

wxGBSizerItem* FindItem(**wxWindow*** *window*)

wxGBSizerItem* FindItem(**wxSizer*** *sizer*)

Find the sizer item for the given window or subsizer, returns NULL if not found. (non-recursive)

wxGridBagSizer::FindItemAtPoint

wxGBSizerItem* FindItemAtPoint(**const wxPoint&** *pt*)

Return the sizer item located at the point given in pt, or NULL if there is no item at that point. The (x,y) coordinates in pt correspond to the client coordinates of the window using the sizer for layout. (non-recursive)

wxGridBagSizer::FindItemAtPosition

wxGBSizerItem* FindItemAtPosition(**const wxGBPosition&** *pos*)

Return the sizer item for the given grid cell, or NULL if there is no item at that position. (non-recursive)

wxGridBagSizer::FindItemWithData

wxGBSizerItem* FindItemWithData(**const wxObject*** *userData*)

Return the sizer item that has a matching user data (it only compares pointer values) or NULL if not found. (non-recursive)

wxGridBagSizer::GetCellSize

wxSize GetCellSize(**int** *row*, **int** *col*) **const**

Get the size of the specified cell, including hgap and vgap. Only valid after a Layout.

wxGridBagSizer::GetEmptyCellSize

wxSize GetEmptyCellSize() **const**

Get the size used for cells in the grid with no item.

wxGridBagSizer::GetItemPosition

wxGBPosition GetItemPosition(**wxWindow*** *window*)

wxGBPosition GetItemPosition(**wxSizer*** *sizer*)

wxGBPosition GetItemPosition(**size_t** *index*)

Get the grid position of the specified item.

wxGridBagSizer::GetItemSpan

wxGBSpan GetItemSpan(**wxWindow*** *window*)

wxGBSpan GetItemSpan(**wxSizer*** *sizer*)

wxGBSpan GetItemSpan(**size_t** *index*)

Get the row/col spanning of the specified item

wxGridBagSizer::RecalcSizes

void RecalcSizes()

Called when the managed size of the sizer is needed or when layout needs done.

wxGridBagSizer::SetEmptyCellSize

void SetEmptyCellSize(**const wxSize&** *sz*)

Set the size used for cells in the grid with no item.

wxGridBagSizer::SetItemPosition

bool SetItemPosition(**wxWindow*** *window*, **const wxGBPosition&** *pos*)

bool SetItemPosition(**wxSizer*** *sizer*, **const wxGBPosition&** *pos*)

bool SetItemPosition(**size_t** *index*, **const wxGBPosition&** *pos*)

Set the grid position of the specified item. Returns true on success. If the move is not allowed (because an item is already there) then false is returned.

wxGridBagSizer::SetItemSpan

bool SetItemSpan(**wxWindow*** *window*, **const wxGBSpan&** *span*)

bool SetItemSpan(**wxSizer*** *sizer*, **const wxGBSpan&** *span*)

bool SetItemSpan(**size_t** *index*, **const wxGBSpan&** *span*)

Set the row/col spanning of the specified item. Returns true on success. If the move is not allowed (because an item is already there) then false is returned.

wxGridCellBoolEditor

The editor for boolean data.

Derived from

wxGridCellEditor (p. 661)

See also

wxGridCellEditor (p. 661), *wxGridCellFloatEditor* (p. 663), *wxGridCellNumberEditor* (p. 664), *wxGridCellTextEditor* (p. 665), *wxGridCellChoiceEditor* (p. 660)

Include files

<wx/grid.h>

wxGridCellBoolEditor::wxGridCellBoolEditor

wxGridCellBoolEditor()

Default constructor.

wxGridCellChoiceEditor

The editor for string data allowing to choose from a list of strings.

Derived from

wxGridCellEditor (p. 661)

See also

wxGridCellEditor (p. 661), *wxGridCellFloatEditor* (p. 663), *wxGridCellBoolEditor* (p. 660), *wxGridCellTextEditor* (p. 665), *wxGridCellNumberEditor* (p. 664)

wxGridCellChoiceEditor::wxGridCellChoiceEditor

wxGridCellChoiceEditor(**size_t** *count = 0*, **const wxString** *choices[] = NULL*, **bool** *allowOthers = false*)

wxGridCellChoiceEditor(**const wxArrayString&** *choices*, **bool** *allowOthers = false*)

count

Number of strings from which the user can choose.

choices

An array of strings from which the user can choose.

allowOthers

If allowOthers if true, the user can type a string not in choices array.

wxGridCellChoiceEditor::SetParameters

void SetParameters(**const wxString&** *params*)

Parameters string format is "item1[,item2[...,itemN]]"

wxGridCellEditor

This class is responsible for providing and manipulating the in-place edit controls for the grid. Instances of wxGridCellEditor (actually, instances of derived classes since it is an abstract class) can be associated with the cell attributes for individual cells, rows, columns, or even for the entire grid.

Derived from

wxGridCellWorker

See also

wxGridCellTextEditor (p. 665), *wxGridCellFloatEditor* (p. 663), *wxGridCellBoolEditor* (p. 660), *wxGridCellNumberEditor* (p. 664), *wxGridCellChoiceEditor* (p. 660)

Include files

<wx/grid.h>

wxGridCellEditor::wxGridCellEditor

wxGridCellEditor()

wxGridCellEditor::IsCreated

bool IsCreated()

wxGridCellEditor::Create

void Create(**wxWindow*** *parent*, **wxWindowID** *id*, **wxEvtHandler*** *evtHandler*)

Creates the actual edit control.

wxGridCellEditor::SetSize

void SetSize(**const wxRect&** *rect*)

Size and position the edit control.

wxGridCellEditor::Show

void Show(**bool** *show*, **wxGridCellAttr*** *attr* = *NULL*)

Show or hide the edit control, use the specified attributes to set colours/fonts for it.

wxGridCellEditor::PaintBackground

void PaintBackground(**const wxRect&** *rectCell*, **wxGridCellAttr*** *attr*)

Draws the part of the cell not occupied by the control: the base class version just fills it with background colour from the attribute.

wxGridCellEditor::BeginEdit

void BeginEdit(**int** *row*, **int** *col*, **wxGrid*** *grid*)

Fetch the value from the table and prepare the edit control to begin editing. Set the focus to the edit control.

wxGridCellEditor::EndEdit

bool EndEdit(**int** *row*, **int** *col*, **wxGrid*** *grid*)

Complete the editing of the current cell. Returns true if the value has changed. If necessary, the control may be destroyed.

wxGridCellEditor::Reset

void Reset()

Reset the value in the control back to its starting value.

wxGridCellEditor::StartingKey

void StartingKey(**wxKeyEvent&** *event*)

If the editor is enabled by pressing keys on the grid, this will be called to let the editor do something about that first key if desired.

wxGridCellEditor::StartingClick

void StartingClick()

If the editor is enabled by clicking on the cell, this method will be called.

wxGridCellEditor::HandleReturn

void HandleReturn(**wxKeyEvent&** *event*)

Some types of controls on some platforms may need some help with the Return key.

wxGridCellEditor::Destroy

void Destroy()

Final cleanup.

wxGridCellEditor::Clone

wxGridCellEditor* Clone() **const**

Create a new object which is the copy of this one.

wxGridCellEditor::~wxGridCellEditor

~wxGridCellEditor()

The dtor is private because only DecRef() can delete us.

wxGridCellFloatEditor

The editor for floating point numbers data.

Derived from

wxGridCellTextEditor (p. 665)
wxGridCellEditor (p. 661)

See also

wxGridCellEditor (p. 661), *wxGridCellNumberEditor* (p. 664), *wxGridCellBoolEditor* (p. 660), *wxGridCellTextEditor* (p. 665), *wxGridCellChoiceEditor* (p. 660)

Include files

<wx/grid.h>

wxGridCellFloatEditor::wxGridCellFloatEditor

wxGridCellFloatEditor(**int** *width = -1*, **int** *precision = -1*)

width

> Minimum number of characters to be shown.

precision

> Number of digits after the decimal dot.

wxGridCellFloatEditor::SetParameters

void SetParameters(**const wxString&** *params*)

Parameters string format is "width,precision"

wxGridCellNumberEditor

The editor for numeric integer data.

Derived from

wxGridCellTextEditor (p. 665)
wxGridCellEditor (p. 661)

See also

wxGridCellEditor (p. 661), *wxGridCellFloatEditor* (p. 663), *wxGridCellBoolEditor* (p. 660), *wxGridCellTextEditor* (p. 665), *wxGridCellChoiceEditor* (p. 660)

Include files

<wx/grid.h>

wxGridCellNumberEditor::wxGridCellNumberEditor

wxGridCellNumberEditor(**int** *min = -1*, **int** *max = -1*)

Allows to specify the range for acceptable data; if min == max == -1, no range checking is done

wxGridCellNumberEditor::GetString

wxString GetString() **const**

String representation of the value.

wxGridCellNumberEditor::HasRange

bool HasRange() **const**

If the return value is true, the editor uses a wxSpinCtrl to get user input, otherwise it uses a wxTextCtrl.

wxGridCellNumberEditor::SetParameters

void SetParameters(**const wxString&** *params*)

Parameters string format is "min,max".

wxGridCellTextEditor

The editor for string/text data.

Derived from

wxGridCellEditor (p. 661)

See also

wxGridCellEditor (p. 661), *wxGridCellFloatEditor* (p. 663), *wxGridCellBoolEditor* (p. 660), *wxGridCellNumberEditor* (p. 664), *wxGridCellChoiceEditor* (p. 660)

Include files

<wx/grid.h>

wxGridCellTextEditor::wxGridCellTextEditor

wxGridCellTextEditor()

Default constructor.

wxGridCellTextEditor::SetParameters

void SetParameters(**const wxString&** *params*)

The parameters string format is "n" where n is a number representing the maximum width.

wxGridEditorCreatedEvent

Derived from

wxCommandEvent (p. 180)
wxEvent (p. 483)
wxObject (p. 1012)

Event handling

The event handler for the following functions takes a *wxGridEditorCreatedEvent* (p. 666) parameter. The ..._CMD_... variants also take a window identifier.

EVT_GRID_EDITOR_CREATED(func) The editor for a cell was created. Processes a wxEVT_GRID_EDITOR_CREATED.

EVT_GRID_CMD_EDITOR_CREATED(id, func)   The editor for a cell was created; variant taking a window identifier. Processes a wxEVT_GRID_EDITOR_CREATED.

Include files
<wx/grid.h>

wxGridEditorCreatedEvent::wxGridEditorCreatedEvent

wxGridEditorCreatedEvent()

Default constructor.

wxGridEditorCreatedEvent(**int** *id*, **wxEventType** *type*, **wxObject*** *obj*, **int** *row*, **int** *col*, **wxControl*** *ctrl*)

wxGridEditorCreatedEvent::GetCol

int GetCol()

Returns the column at which the event occurred.

wxGridEditorCreatedEvent::GetControl

wxControl* GetControl()

Returns the edit control.

wxGridEditorCreatedEvent::GetRow

int GetRow()

Returns the row at which the event occurred.

wxGridEditorCreatedEvent::SetCol

void SetCol(**int** *col*)

Sets the column at which the event occurred.

wxGridEditorCreatedEvent::SetControl

void SetControl(**wxControl*** *ctrl*)

Sets the edit control.

wxGridEditorCreatedEvent::SetRow

void SetRow(**int** *row*)

Sets the row at which the event occurred.

wxGridEvent

This event class contains information about various grid events.

Derived from

wxNotifyEvent (p. 1011)
wxCommandEvent (p. 180)
wxEvent (p. 483)
wxObject (p. 1012)

Include files

<wx/grid.h>

Event handling

The event handler for the following functions takes a *wxGridEvent* (p. 667) parameter. The ..._CMD_... variants also take a window identifier.

EVT_GRID_CELL_LEFT_CLICK(func) The user clicked a cell with the left mouse button. Processes a wxEVT_GRID_CELL_LEFT_CLICK.

EVT_GRID_CELL_RIGHT_CLICK(func) The user clicked a cell with the right mouse button. Processes a wxEVT_GRID_CELL_RIGHT_CLICK.

EVT_GRID_CELL_LEFT_DCLICK(func) The user double-clicked a cell with the left mouse button. Processes a wxEVT_GRID_CELL_LEFT_DCLICK.

EVT_GRID_CELL_RIGHT_DCLICK(func) The user double-clicked a cell with the right mouse button. Processes a wxEVT_GRID_CELL_RIGHT_DCLICK.

EVT_GRID_LABEL_LEFT_CLICK(func) The user clicked a label with the left mouse button. Processes a wxEVT_GRID_LABEL_LEFT_CLICK.

EVT_GRID_LABEL_RIGHT_CLICK(func) The user clicked a label with the right mouse button. Processes a wxEVT_GRID_LABEL_RIGHT_CLICK.

EVT_GRID_LABEL_LEFT_DCLICK(func) The user double-clicked a label with the left mouse button. Processes a wxEVT_GRID_LABEL_LEFT_DCLICK.

EVT_GRID_LABEL_RIGHT_DCLICK(func) The user double-clicked a label with the right mouse button. Processes a wxEVT_GRID_LABEL_RIGHT_DCLICK.

EVT_GRID_CELL_CHANGE(func) The user changed the data in a cell. Processes a wxEVT_GRID_CELL_CHANGE.

EVT_GRID_SELECT_CELL(func) The user moved to, and selected a cell. Processes a wxEVT_GRID_SELECT_CELL.

EVT_GRID_EDITOR_HIDDEN(func) The editor for a cell was hidden. Processes a wxEVT_GRID_EDITOR_HIDDEN.

EVT_GRID_EDITOR_SHOWN(func) The editor for a cell was shown. Processes a wxEVT_GRID_EDITOR_SHOWN.

EVT_GRID_CMD_CELL_LEFT_CLICK(id, func) The user clicked a cell with the left mouse button; variant taking a window identifier. Processes a wxEVT_GRID_CELL_LEFT_CLICK.

EVT_GRID_CMD_CELL_RIGHT_CLICK(id, func) The user clicked a cell with the right mouse button; variant taking a window identifier. Processes a wxEVT_GRID_CELL_RIGHT_CLICK.

EVT_GRID_CMD_CELL_LEFT_DCLICK(id, func) The user double-clicked a cell with the left mouse button; variant taking a window identifier. Processes a wxEVT_GRID_CELL_LEFT_DCLICK.

EVT_GRID_CMD_CELL_RIGHT_DCLICK(id, func) The user double-clicked a cell with the right mouse button; variant taking a window identifier. Processes a wxEVT_GRID_CELL_RIGHT_DCLICK.

EVT_GRID_CMD_LABEL_LEFT_CLICK(id, func) The user clicked a label with the left mouse button; variant taking a window identifier. Processes a wxEVT_GRID_LABEL_LEFT_CLICK.

EVT_GRID_CMD_LABEL_RIGHT_CLICK(id, func) The user clicked a label with the right mouse button; variant taking a window identifier. Processes a wxEVT_GRID_LABEL_RIGHT_CLICK.

EVT_GRID_CMD_LABEL_LEFT_DCLICK(id, func) The user double-clicked a label with the left mouse button; variant taking a window identifier. Processes a wxEVT_GRID_LABEL_LEFT_DCLICK.

EVT_GRID_CMD_LABEL_RIGHT_DCLICK(id, func) The user double-clicked a label with the right mouse button; variant taking a window identifier. Processes a wxEVT_GRID_LABEL_RIGHT_DCLICK.

EVT_GRID_CMD_CELL_CHANGE(id, func) The user changed the data in a cell; variant taking a window identifier. Processes a wxEVT_GRID_CELL_CHANGE.

EVT_GRID_CMD_SELECT_CELL(id, func) The user moved to, and selected a cell; variant taking a window identifier. Processes a wxEVT_GRID_SELECT_CELL.

EVT_GRID_CMD_EDITOR_HIDDEN(id, func) The editor for a cell was hidden; variant taking a window identifier. Processes a wxEVT_GRID_EDITOR_HIDDEN.

EVT_GRID_CMD_EDITOR_SHOWN(id, func) The editor for a cell was shown; variant taking a window identifier. Processes a wxEVT_GRID_EDITOR_SHOWN.

wxGridEvent::wxGridEvent

wxGridEvent()

Default constructor.

wxGridEvent(**int** *id*, **wxEventType** *type*, **wxObject*** *obj*, **int** *row = -1*, **int** *col = -1*, **int** *x = -1*, **int** *y = -1*, **bool** *sel = true*, **bool** *control = false*, **bool** *shift = false*, **bool** *alt = false*, **bool** *meta = false*)

Parameters

wxGridEvent::AltDown

bool AltDown()

Returns true if the Alt key was down at the time of the event.

wxGridEvent::ControlDown

bool ControlDown()

Returns true if the Control key was down at the time of the event.

wxGridEvent::GetCol

int GetCol()

Column at which the event occurred.

wxGridEvent::GetPosition

wxPoint GetPosition()

Position in pixels at which the event occurred.

wxGridEvent::GetRow

int GetRow()

Row at which the event occurred.

wxGridEvent::MetaDown

bool MetaDown()

Returns true if the Meta key was down at the time of the event.

wxGridEvent::Selecting

bool Selecting()

Returns true if the user deselected a cell, false if the user deselected a cell.

wxGridEvent::ShiftDown

bool ShiftDown()

Returns true if the Shift key was down at the time of the event.

wxGridRangeSelectEvent

Derived from

wxNotifyEvent (p. 1011)
wxCommandEvent (p. 180)
wxEvent (p. 483)
wxObject (p. 1012)

Event handling

The event handler for the following functions takes a *wxGridRangeSelectEvent* (p. 671) parameter. The ..._CMD_... variants also take a window identifier.

EVT_GRID_RANGE_SELECT(func) The user selected a group of contiguous cells. Processes a wxEVT_GRID_RANGE_SELECT.

EVT_GRID_CMD_RANGE_SELECT(func) The user selected a group of contiguous cells; variant taking a window identifier. Processes a wxEVT_GRID_RANGE_SELECT.

Include files
<wx/grid.h>

wxGridRangeSelectEvent::wxGridRangeSelectEvent

 wxGridRangeSelectEvent()

Default constructor.

 wxGridRangeSelectEvent(**int** *id,* **wxEventType** *type*, **wxObject*** *obj*, **const wxGridCellCoords&** *topLeft*, **const wxGridCellCoords&** *bottomRight*, **bool** *sel = true*,

bool *control = false*, **bool** *shift = false*, **bool** *alt = false*, **bool** *meta = false*)

wxGridRangeSelectEvent::AltDown

bool AltDown()

Returns true if the Alt key was down at the time of the event.

wxGridRangeSelectEvent::ControlDown

bool ControlDown()

Returns true if the Control key was down at the time of the event.

wxGridRangeSelectEvent::GetBottomRightCoords

wxGridCellCoords GetBottomRightCoords()

Top left corner of the rectangular area that was (de)selected.

wxGridRangeSelectEvent::GetBottomRow

int GetBottomRow()

Bottom row of the rectangular area that was (de)selected.

wxGridRangeSelectEvent::GetLeftCol

int GetLeftCol()

Left column of the rectangular area that was (de)selected.

wxGridRangeSelectEvent::GetRightCol

int GetRightCol()

Right column of the rectangular area that was (de)selected.

wxGridRangeSelectEvent::GetTopLeftCoords

wxGridCellCoords GetTopLeftCoords()

Top left corner of the rectangular area that was (de)selected.

wxGridRangeSelectEvent::GetTopRow

int GetTopRow()

Top row of the rectangular area that was (de)selected.

wxGridRangeSelectEvent::MetaDown

bool MetaDown()

Returns true if the Meta key was down at the time of the event.

wxGridRangeSelectEvent::Selecting

bool Selecting()

Returns true if the area was selected, false otherwise.

wxGridRangeSelectEvent::ShiftDown

bool ShiftDown()

Returns true if the Shift key was down at the time of the event.

wxGridSizeEvent

This event class contains information about a row/column resize event.

Derived from

wxNotifyEvent (p. 1011)
wxCommandEvent (p. 180)
wxEvent (p. 483)
wxObject (p. 1012)

Include files

<wx/grid.h>

Event handling

The event handler for the following functions takes a *wxGridSizeEvent* (p. 673) parameter. The ..._CMD_... variants also take a window identifier.

EVT_GRID_COL_SIZE(func)	The user resized a column by dragging it. Processes a wxEVT_GRID_COL_SIZE.
EVT_GRID_ROW_SIZE(func)	The user resized a row by dragging it. Processes a wxEVT_GRID_ROW_SIZE.

EVT_GRID_CMD_COL_SIZE(func)	The user resized a column by dragging it; variant taking a window identifier. Processes a wxEVT_GRID_COL_SIZE.
EVT_GRID_CMD_ROW_SIZE(func)	The user resized a row by dragging it; variant taking a window identifier. Processes a wxEVT_GRID_ROW_SIZE.

wxGridSizeEvent::wxGridSizeEvent

wxGridSizeEvent()

Default constructor.

wxGridSizeEvent(**int** *id*, **wxEventType** *type*, **wxObject*** *obj*, **int** *rowOrCol = -1*, **int** *x = -1*, **int** *y = -1*, **bool** *control = false*, **bool** *shift = false*, **bool** *alt = false*, **bool** *meta = false*)

wxGridSizeEvent::AltDown

bool AltDown()

Returns true if the Alt key was down at the time of the event.

wxGridSizeEvent::ControlDown

bool ControlDown()

Returns true if the Control key was down at the time of the event.

wxGridSizeEvent::GetPosition

wxPoint GetPosition()

Position in pixels at which the event occurred.

wxGridSizeEvent::GetRowOrCol

int GetRowOrCol()

Row or column at that was resized.

wxGridSizeEvent::MetaDown

bool MetaDown()

Returns true if the Meta key was down at the time of the event.

wxGridSizeEvent::ShiftDown

bool ShiftDown()

Returns true if the Shift key was down at the time of the event.

wxGridCellBoolRenderer

This class may be used to format boolean data in a cell. for string cells.

Derived from

wxGridCellRenderer (p. 677)

See also

wxGridCellRenderer (p. 677), *wxGridCellStringRenderer* (p. 678),
wxGridCellFloatRenderer (p. 675), *wxGridCellNumberRenderer* (p. 677)

Include files

<wx/grid.h>

wxGridCellBoolRenderer::wxGridCellBoolRenderer

 wxGridCellBoolRenderer()

Default constructor

wxGridCellFloatRenderer

This class may be used to format floating point data in a cell.

Derived from

wxGridCellStringRenderer (p. 678)
wxGridCellRenderer (p. 677)

See also

wxGridCellRenderer (p. 677), *wxGridCellNumberRenderer* (p. 677),
wxGridCellStringRenderer (p. 678), *wxGridCellBoolRenderer* (p. 675)

Include files

<wx/grid.h>

wxGridCellFloatRenderer::wxGridCellFloatRenderer

wxGridCellFloatRenderer(**int** *width = -1*, **int** *precision = -1*)

width

Minimum number of characters to be shown.

precision

Number of digits after the decimal dot.

wxGridCellFloatRenderer::GetPrecision

int GetPrecision() **const**

Returns the precision (see *wxGridCellFloatRenderer* (p. 676)).

wxGridCellFloatRenderer::GetWidth

int GetWidth() **const**

Returns the width (see *wxGridCellFloatRenderer* (p. 676)).

wxGridCellFloatRenderer::SetParameters

void SetParameters(**const wxString&** *params*)

Parameters string format is "width[,precision]".

wxGridCellFloatRenderer::SetPrecision

void SetPrecision(**int** *precision*)

Sets the precision (see *wxGridCellFloatRenderer* (p. 676)).

wxGridCellFloatRenderer::SetWidth

void SetWidth(**int** *width*)

Sets the width (see *wxGridCellFloatRenderer* (p. 676))

wxGridCellNumberRenderer

This class may be used to format integer data in a cell.

Derived from

wxGridellStringRenderer (p. 678)
wxGridCellRenderer (p. 677)

See also

wxGridCellRenderer (p. 677), *wxGridCellStringRenderer* (p. 678),
wxGridCellFloatRenderer (p. 675), *wxGridCellBoolRenderer* (p. 675)

Include files

<wx/grid.h>

wxGridCellNumberRenderer::wxGridCellNumberRenderer

wxGridCellNumberRenderer()

Default constructor

wxGridCellRenderer

This class is responsible for actually drawing the cell in the grid. You may pass it to the
wxGridCellAttr (below) to change the format of one given cell or to
wxGrid::SetDefaultRenderer() to change the view of all cells. This is an abstract class,
and you will normally use one of the predefined derived classes or derive your own class
from it.

Derived from

wxGridCellWorker

See also

wxGridCellStringRenderer (p. 678), *wxGridCellNumberRenderer* (p. 677),
wxGridCellFloatRenderer (p. 675), *wxGridCellBoolRenderer* (p. 675)

Include files

<wx/grid.h>

wxGridCellRenderer::Draw

void Draw(**wxGrid&** *grid*, **wxGridCellAttr&** *attr*, **wxDC&** *dc*, **const wxRect&** *rect*, **int** *row*, **int** *col*, **bool** *isSelected*)

Draw the given cell on the provided DC inside the given rectangle using the style specified by the attribute and the default or selected state corresponding to the isSelected value.

This pure virtual function has a default implementation which will prepare the DC using the given attribute: it will draw the rectangle with the background colour from attr and set the text colour and font.

wxGridCellRenderer::GetBestSize

wxSize GetBestSize(**wxGrid&** *grid*, **wxGridCellAttr&** *attr*, **wxDC&** *dc*, **int** *row*, **int** *col*)

Get the preferred size of the cell for its contents.

wxGridCellRenderer::Clone

wxGridCellRenderer* Clone() **const**

wxGridCellStringRenderer

This class may be used to format string data in a cell; it is the default for string cells.

Derived from

wxGridCellRenderer (p. 677)

See also

wxGridCellRenderer (p. 677), *wxGridCellNumberRenderer* (p. 677), *wxGridCellFloatRenderer* (p. 675), *wxGridCellBoolRenderer* (p. 675)

Include files

<wx/grid.h>

wxGridCellStringRenderer::wxGridCellStringRenderer

wxGridCellStringRenderer()

Default constructor

wxGridTableBase

Grid table classes.

Derived from

wxObject (p. 1012)

Include files

<wx/grid.h>

wxGridTableBase::wxGridTableBase

wxGridTableBase()

wxGridTableBase::~wxGridTableBase

~wxGridTableBase()

wxGridTableBase::GetNumberRows

int GetNumberRows()

You must override these functions in a derived table class.

wxGridTableBase::GetNumberCols

int GetNumberCols()

wxGridTableBase::IsEmptyCell

bool IsEmptyCell(**int** *row*, **int** *col*)

wxGridTableBase::GetValue

wxString GetValue(**int** *row*, **int** *col*)

wxGridTableBase::SetValue

void SetValue(**int** *row*, **int** *col*, **const wxString&** *value*)

wxGridTableBase::GetTypeName

wxString GetTypeName(**int** *row*, **int** *col*)

Data type determination and value access.

wxGridTableBase::CanGetValueAs

bool CanGetValueAs(**int** *row*, **int** *col*, **const wxString&** *typeName*)

wxGridTableBase::CanSetValueAs

bool CanSetValueAs(**int** *row*, **int** *col*, **const wxString&** *typeName*)

wxGridTableBase::GetValueAsLong

long GetValueAsLong(**int** *row*, **int** *col*)

wxGridTableBase::GetValueAsDouble

double GetValueAsDouble(**int** *row*, **int** *col*)

wxGridTableBase::GetValueAsBool

bool GetValueAsBool(**int** *row*, **int** *col*)

wxGridTableBase::SetValueAsLong

void SetValueAsLong(**int** *row*, **int** *col*, **long** *value*)

wxGridTableBase::SetValueAsDouble

void SetValueAsDouble(**int** *row*, **int** *col*, **double** *value*)

wxGridTableBase::SetValueAsBool

void SetValueAsBool(**int** *row*, **int** *col*, **bool** *value*)

wxGridTableBase::GetValueAsCustom

void* GetValueAsCustom(**int** *row*, **int** *col*, **const wxString&** *typeName*)

For user defined types

wxGridTableBase::SetValueAsCustom

void SetValueAsCustom(**int** *row*, **int** *col*, **const wxString&** *typeName*, **void*** *value*)

wxGridTableBase::SetView

void SetView(**wxGrid*** *grid*)

Overriding these is optional

wxGridTableBase::GetView

wxGrid * GetView() **const**

wxGridTableBase::Clear

void Clear()

wxGridTableBase::InsertRows

bool InsertRows(**size_t** *pos = 0*, **size_t** *numRows = 1*)

wxGridTableBase::AppendRows

bool AppendRows(**size_t** *numRows = 1*)

wxGridTableBase::DeleteRows

bool DeleteRows(**size_t** *pos = 0*, **size_t** *numRows = 1*)

wxGridTableBase::InsertCols

bool InsertCols(**size_t** *pos = 0*, **size_t** *numCols = 1*)

wxGridTableBase::AppendCols

bool AppendCols(**size_t** *numCols = 1*)

wxGridTableBase::DeleteCols

bool DeleteCols(**size_t** *pos = 0*, **size_t** *numCols = 1*)

wxGridTableBase::GetRowLabelValue

wxString GetRowLabelValue(**int** *row*)

wxGridTableBase::GetColLabelValue

wxString GetColLabelValue(**int** *col*)

wxGridTableBase::SetRowLabelValue

void SetRowLabelValue(**int** *WXUNUSED(row)*, **const** *wxString&*)

wxGridTableBase::SetColLabelValue

void SetColLabelValue(**int** *WXUNUSED(col)*, **const** *wxString&*)

wxGridTableBase::SetAttrProvider

void SetAttrProvider(**wxGridCellAttrProvider*** *attrProvider*)

Attribute handling give us the attr provider to use - we take ownership of the pointer

wxGridTableBase::GetAttrProvider

wxGridCellAttrProvider* GetAttrProvider() **const**

get the currently used attr provider (may be NULL)

wxGridTableBase::CanHaveAttributes

bool CanHaveAttributes()

Does this table allow attributes? Default implementation creates a wxGridCellAttrProvider if necessary.

wxGridTableBase::UpdateAttrRows

void UpdateAttrRows(**size_t** *pos*, **int** *numRows*)

change row/col number in attribute if needed

wxGridTableBase::UpdateAttrCols

void UpdateAttrCols(**size_t** *pos*, **int** *numCols*)

wxGridTableBase::GetAttr

wxGridCellAttr* GetAttr(**int** *row*, **int** *col*)

by default forwarded to wxGridCellAttrProvider if any. May be overridden to handle attributes directly in the table.

wxGridTableBase::SetAttr

void SetAttr(**wxGridCellAttr*** *attr*, **int** *row*, **int** *col*)

these functions take ownership of the pointer

wxGridTableBase::SetRowAttr

void SetRowAttr(**wxGridCellAttr*** *attr*, **int** *row*)

wxGridTableBase::SetColAttr

void SetColAttr(**wxGridCellAttr*** *attr*, **int** *col*)

wxGridSizer

A grid sizer is a sizer which lays out its children in a two-dimensional table with all table fields having the same size, i.e. the width of each field is the width of the widest child, the height of each field is the height of the tallest child.

Derived from

wxSizer (p. 1176)
wxObject (p. 1012)

Include files

<wx/sizer.h>

See also

wxSizer (p. 1176), *Sizer overview* (p. 1771)

wxGridSizer::wxGridSizer

wxGridSizer(**int** *rows*, **int** *cols*, **int** *vgap*, **int** *hgap*)

wxGridSizer(**int** *cols*, **int** *vgap = 0*, **int** *hgap = 0*)

Constructor for a wxGridSizer. *rows* and *cols* determine the number of columns and rows in the sizer - if either of the parameters is zero, it will be calculated to form the total number of children in the sizer, thus making the sizer grow dynamically. *vgap* and *hgap* define extra space between all children.

wxGridSizer::GetCols

int GetCols()

Returns the number of columns in the sizer.

wxGridSizer::GetHGap

int GetHGap()

Returns the horizontal gap (in pixels) between cells in the sizer.

wxGridSizer::GetRows

int GetRows()

Returns the number of rows in the sizer.

wxGridSizer::GetVGap

int GetVGap()

Returns the vertical gap (in pixels) between the cells in the sizer.

wxGridSizer::SetCols

void SetCols(**int** *cols*)

Sets the number of columns in the sizer.

wxGridSizer::SetHGap

void SetHGap(**int** *gap*)

Sets the horizontal gap (in pixels) between cells in the sizer.

wxGridSizer::SetRows

void SetRows(**int** *rows*)

Sets the number of rows in the sizer.

wxGridSizer::SetVGap

void SetVGap(**int** *gap*)

Sets the vertical gap (in pixels) between the cells in the sizer.

wxHashMap

This is a simple, type-safe, and reasonably efficient hash map class, whose interface is a subset of the interface of STL containers. In particular, the interface is modeled after std::map, and the various, non standard, std::hash_map.

Example

```
class MyClass { /* ... */ };

// declare a hash map with string keys and int values
WX_DECLARE_STRING_HASH_MAP( int, MyHash5 );
// same, with int keys and MyClass* values
WX_DECLARE_HASH_MAP( int, MyClass*, wxIntegerHash, wxIntegerEqual,
MyHash1 );
// same, with wxString keys and int values
WX_DECLARE_STRING_HASH_MAP( int, MyHash3 );
// same, with wxString keys and values
WX_DECLARE_STRING_HASH_MAP( wxString, MyHash2 );

MyHash1 h1;
MyHash2 h2;

// store and retrieve values
h1[1] = new MyClass( 1 );
h1[10000000] = NULL;
h1[50000] = new MyClass( 2 );
h2["Bill"] = "ABC";
wxString tmp = h2["Bill"];
// since element with key "Joe" is not present, this will return
// the default value, which is an empty string in the case of wxString
MyClass tmp2 = h2["Joe"];

// iterate over all the elements in the class
MyHash2::iterator it;
for( it = h2.begin(); it != h2.end(); ++it )
{
    wxString key = it->first, value = it->second;
    // do something useful with key and value
}
```

Declaring new hash table types

```
WX_DECLARE_STRING_HASH_MAP( VALUE_T,     // type of the values
                                CLASSNAME ); // name of the class
```

Declares a hash map class named CLASSNAME, with `wxString` keys and VALUE_T values.

```
WX_DECLARE_VOIDPTR_HASH_MAP( VALUE_T,     // type of the values
                             CLASSNAME ); // name of the class
```

Declares a hash map class named CLASSNAME, with `void*` keys and VALUE_T values.

```
WX_DECLARE_HASH_MAP( KEY_T,      // type of the keys
                     VALUE_T,    // type of the values
                     HASH_T,     // hasher
                     KEY_EQ_T,   // key equality predicate
                     CLASSNAME); // name of the class
```

The HASH_T and KEY_EQ_T are the types used for the hashing function and key comparison. wxWidgets provides three predefined hashing functions: `wxIntegerHash`for integer types (`int`, `long`, `short`, and their unsigned counterparts), `wxStringHash` for strings (`wxString`, `wxChar*`, `char*`), and`wxPointerHash` for any kind of pointer. Similarly three equality predicates:`wxIntegerEqual`, `wxStringEqual`, `wxPointerEqual` are provided.

Using this you could declare a hash map mapping `int` values to `wxString` like this:

```
WX_DECLARE_HASH_MAP( int,
                     wxString,
                     wxIntegerHash,
                     wxIntegerEqual,
                     MyHash );

// using an user-defined class for keys
class MyKey { /* ... */ };

// hashing function
class MyKeyHash
{
public:
    MyKeyHash() { }

    unsigned long operator()( const MyKey& k ) const
        { /* compute the hash */ }

    MyKeyHash& operator=(const MyKeyHash&) { return *this; }
};

// comparison operator
class MyKeyEqual
{
public:
    MyKeyEqual() { }
    bool operator()( const MyKey& a, const MyKey& b ) const
        { /* compare for equality */ }

    MyKeyEqual& operator=(const MyKeyEqual&) { return *this; }
};

WX_DECLARE_HASH_MAP( MyKey,      // type of the keys
                     SOME_TYPE,  // any type you like
                     MyKeyHash,  // hasher
```

```
                        MyKeyEqual, // key equality predicate
                        CLASSNAME); // name of the class
```

In the documentation below you should replace wxHashMap with the name you used in the class declaration.

wxHashMap::key_type	Type of the hash keys
wxHashMap::mapped_type	Type of the values stored in the hash map
wxHashMap::value_type	Equivalent to `struct { key_type first; mapped_type second };`
wxHashMap::iterator	Used to enumerate all the elements in a hash map; it is similar to a `value_type*`
wxHashMap::const_iterator	Used to enumerate all the elements in a constant hash map; it is similar to a `const value_type*`
wxHashMap::size_type	Used for sizes
wxHashMap::Insert_Result	The return value for *insert()* (p. 689)

Iterators

An iterator is similar to a pointer, and so you can use the usual pointer operations: `++it` (and `it++`) to move to the next element, `*it` to access the element pointed to, `it->first` (`it->second`) to access the key (value) of the element pointed to. Hash maps provide forward only iterators, this means that you can't use `--it`, `it + 3`, `it1 - it2`.

Include files

<wx/hashmap.h>

wxHashMap::wxHashMap

wxHashMap(**size_type** *size = 10*)

The size parameter is just a hint, the table will resize automatically to preserve performance.

wxHashMap(**const wxHashMap&** *map*)

Copy constructor.

wxHashMap::begin

const_iterator begin() **const**

iterator begin()

Returns an iterator pointing at the first element of the hash map. Please remember that hash maps do not guarantee ordering.

wxHashMap::clear

void clear()

Removes all elements from the hash map.

wxHashMap::count

size_type count(**const key_type&** *key*) **const**

Counts the number of elements with the given key present in the map. This function returns only 0 or 1.

wxHashMap::empty

bool empty() **const**

Returns true if the hash map does not contain any elements, false otherwise.

wxHashMap::end

const_iterator end() **const**

iterator end()

Returns an iterator pointing at the one-after-the-last element of the hash map. Please remember that hash maps do not guarantee ordering.

wxHashMap::erase

size_type erase(**const key_type&** *key*)

Erases the element with the given key, and returns the number of elements erased (either 0 or 1).

void erase(**iterator** *it*)

void erase(**const_iterator** *it*)

Erases the element pointed to by the iterator. After the deletion the iterator is no longer valid and must not be used.

wxHashMap::find

iterator find(**const key_type&** *key*)

const_iterator find(**const key_type&** *key*) **const**

If an element with the given key is present, the functions returns an iterator pointing at that element, otherwise an invalid iterator is returned (i.e. hashmap.find(non_existent_key) == hashmap.end()).

wxHashMap::insert

Insert_Result insert(**const value_type&** *v*)

Inserts the given value in the hash map. The return value is equivalent to a `std::pair<wxHashMap::iterator, bool>`; the iterator points to the inserted element, the boolean value is `true` if `v` was actually inserted.

wxHashMap::operator[]

mapped_type& operator[](**const key_type&** *key*)

Use the key as an array subscript. The only difference is that if the given key is not present in the hash map, an element with the default `value_type()` is inserted in the table.

wxHashMap::size

size_type size() **const**

Returns the number of elements in the map.

wxHashSet

This is a simple, type-safe, and reasonably efficient hash set class, whose interface is a subset of the interface of STL containers. In particular, the interface is modeled after std::set, and the various, non standard, std::hash_map.

Example

```
class MyClass { /* ... */ };

// same, with MyClass* keys (only uses pointer equality!)
WX_DECLARE_HASH_SET( MyClass*, wxPointerHash, wxPointerEqual, MySet1
);
// same, with int keys
WX_DECLARE_HASH_SET( int, wxIntegerHash, wxIntegerEqual, MySet2 );
// declare a hash set with string keys
WX_DECLARE_HASH_SET( wxString, wxStringHash, wxStringEqual, MySet3 );
```

```
MySet1 h1;
MySet2 h1;
MySet3 h3;

// store and retrieve values
h1.insert( new MyClass( 1 ) );

h3.insert( "foo" );
h3.insert( "bar" );
h3.insert( "baz" );

int size = h3.size(); // now is three
bool has_foo = h3.find( "foo" ) != h3.end();

h3.insert( "bar" ); // still has size three

// iterate over all the elements in the class
MySet3::iterator it;
for( it = h3.begin(); it != h3.end(); ++it )
{
    wxString key = *it;
    // do something useful with key
}
```

Declaring new hash set types

```
WX_DECLARE_HASH_SET( KEY_T,      // type of the keys
                     HASH_T,     // hasher
                     KEY_EQ_T,   // key equality predicate
                     CLASSNAME); // name of the class
```

The HASH_T and KEY_EQ_T are the types used for the hashing function and key comparison. wxWidgets provides three predefined hashing functions: `wxIntegerHash`for integer types (`int`, `long`, `short`, and their unsigned counterparts), `wxStringHash` for strings (`wxString`, `wxChar*`, `char*`), and`wxPointerHash` for any kind of pointer. Similarly three equality predicates:`wxIntegerEqual`, `wxStringEqual`, `wxPointerEqual` are provided.

Using this you could declare a hash set using `int` values like this:

```
WX_DECLARE_HASH_SET( int,
                     wxIntegerHash,
                     wxIntegerEqual,
                     MySet );

// using an user-defined class for keys
class MyKey { /* ... */ };

// hashing function
class MyKeyHash
{
public:
    MyKeyHash() { }

    unsigned long operator()( const MyKey& k ) const
        { /* compute the hash */ }

    MyKeyHash& operator=(const MyKeyHash&) { return *this; }
};
```

```
// comparison operator
class MyKeyEqual
{
public:
    MyKeyEqual() { }
    bool operator()( const MyKey& a, const MyKey& b ) const
        { /* compare for equality */ }

    MyKeyEqual& operator=(const MyKeyEqual&) { return *this; }
};

WX_DECLARE_HASH_SET( MyKey,      // type of the keys
                     MyKeyHash,  // hasher
                     MyKeyEqual, // key equality predicate
                     CLASSNAME); // name of the class
```

In the documentation below you should replace wxHashSet with the name you used in the class declaration.

wxHashSet::key_type	Type of the hash keys
wxHashSet::mapped_type	Type of hash keys
wxHashSet::value_type	Type of hash keys
wxHashSet::iterator	Used to enumerate all the elements in a hash set; it is similar to a `value_type*`
wxHashSet::const_iterator	Used to enumerate all the elements in a constant hash set; it is similar to a `const value_type*`
wxHashSet::size_type	Used for sizes
wxHashSet::Insert_Result	The return value for *insert()* (p. 693)

Iterators

An iterator is similar to a pointer, and so you can use the usual pointer operations: `++it` (and `it++`) to move to the next element, `*it` to access the element pointed to, `*it` to access the value of the element pointed to. Hash sets provide forward only iterators, this means that you can't use `--it`, `it + 3`, `it1 - it2`.

Include files

<wx/hashset.h>

wxHashSet::wxHashSet

wxHashSet(**size_type** *size = 10*)

CHAPTER 7

The size parameter is just a hint, the table will resize automatically to preserve performance.

wxHashSet(**const wxHashSet&** *set*)

Copy constructor.

wxHashSet::begin

const_iterator begin() **const**

iterator begin()

Returns an iterator pointing at the first element of the hash set. Please remember that hash sets do not guarantee ordering.

wxHashSet::clear

void clear()

Removes all elements from the hash set.

wxHashSet::count

size_type count(**const key_type&** *key*) **const**

Counts the number of elements with the given key present in the set. This function returns only 0 or 1.

wxHashSet::empty

bool empty() **const**

Returns true if the hash set does not contain any elements, false otherwise.

wxHashSet::end

const_iterator end() **const**

iterator end()

Returns an iterator pointing at the one-after-the-last element of the hash set. Please remember that hash sets do not guarantee ordering.

wxHashSet::erase

size_type erase(**const key_type&** *key*)

Erases the element with the given key, and returns the number of elements erased (either 0 or 1).

void erase(**iterator** *it*)

void erase(**const_iterator** *it*)

Erases the element pointed to by the iterator. After the deletion the iterator is no longer valid and must not be used.

wxHashSet::find

iterator find(**const key_type&** *key*)

const_iterator find(**const key_type&** *key*) **const**

If an element with the given key is present, the functions returns an iterator pointing at that element, otherwise an invalid iterator is returned (i.e. hashset.find(non_existent_key) == hashset.end()).

wxHashSet::insert

Insert_Result insert(**const value_type&** *v*)

Inserts the given value in the hash set. The return value is equivalent to a std::pair<wxHashMap::iterator, bool>; the iterator points to the inserted element, the boolean value is true if v was actually inserted.

wxHashSet::size

size_type size() **const**

Returns the number of elements in the set.

wxHashTable

Please note that this class is retained for backward compatibility reasons; you should use *wxHashMap* (p. 685).

This class provides hash table functionality for wxWidgets, and for an application if it wishes. Data can be hashed on an integer or string key.

Derived from

wxObject (p. 1012)

Include files

<wx/hash.h>

Example

Below is an example of using a hash table.

```
wxHashTable table(wxKEY_STRING);

wxPoint *point = new wxPoint(100, 200);
table.Put("point 1", point);

....

wxPoint *found_point = (wxPoint *)table.Get("point 1");
```

A hash table is implemented as an array of pointers to lists. When no data has been stored, the hash table takes only a little more space than this array (default size is 1000). When a data item is added, an integer is constructed from the integer or string key that is within the bounds of the array. If the array element is NULL, a new (keyed) list is created for the element. Then the data object is appended to the list, storing the key in case other data objects need to be stored in the list also (when a 'collision' occurs).

Retrieval involves recalculating the array index from the key, and searching along the keyed list for the data object whose stored key matches the passed key. Obviously this is quicker when there are fewer collisions, so hashing will become inefficient if the number of items to be stored greatly exceeds the size of the hash table.

See also

wxList (p. 840)

wxHashTable::wxHashTable

 wxHashTable(**unsigned int** *key_type*, **int** *size = 1000*)

Constructor. *key_type* is one of wxKEY_INTEGER, or wxKEY_STRING, and indicates what sort of keying is required. *size* is optional.

wxHashTable::~wxHashTable

 ~wxHashTable()

Destroys the hash table.

wxHashTable::BeginFind

void BeginFind()

The counterpart of *Next*. If the application wishes to iterate through all the data in the hash table, it can call *BeginFind* and then loop on *Next*.

wxHashTable::Clear

void Clear()

Clears the hash table of all nodes (but as usual, doesn't delete user data).

wxHashTable::Delete

wxObject * Delete(**long** *key*)

wxObject * Delete(**const wxString&** *key*)

Deletes entry in hash table and returns the user's data (if found).

wxHashTable::DeleteContents

void DeleteContents(**bool** *flag*)

If set to true data stored in hash table will be deleted when hash table object is destroyed.

wxHashTable::Get

wxObject * Get(**long** *key*)

wxObject * Get(**const char*** *key*)

Gets data from the hash table, using an integer or string key (depending on which has table constructor was used).

wxHashTable::MakeKey

long MakeKey(**const wxString&** *string*)

Makes an integer key out of a string. An application may wish to make a key explicitly (for instance when combining two data values to form a key).

wxHashTable::Next

wxHashTable::Node * Next()

If the application wishes to iterate through all the data in the hash table, it can call *BeginFind* and then loop on *Next*. This function returns a **wxHashTable::Node** pointer

(or NULL if there are no more nodes). The return value is functionally equivalent to **wxNode** but might not be implemented as a **wxNode**. The user will probably only wish to use the**GetData** method to retrieve the data; the node may also be deleted.

wxHashTable::Put

void Put(**long** *key*, **wxObject ***object*)

void Put(**const char*** *key*, **wxObject ***object*)

Inserts data into the hash table, using an integer or string key (depending on which has table constructor was used). The key string is copied and stored by the hash table implementation.

wxHashTable::GetCount

size_t GetCount() **const**

Returns the number of elements in the hash table.

wxHelpController

This is a family of classes by which applications may invoke a help viewer to provide on-line help.

A help controller allows an application to display help, at the contents or at a particular topic, and shut the help program down on termination. This avoids proliferation of many instances of the help viewer whenever the user requests a different topic via the application's menus or buttons.

Typically, an application will create a help controller instance when it starts, and immediately call **Initialize** to associate a filename with it. The help viewer will only get run, however, just before the first call to display something.

Most help controller classes actually derive from wxHelpControllerBase and have names of the form wxXXXHelpController or wxHelpControllerXXX. An appropriate class is aliased to the name wxHelpController for each platform, as follows:

- On desktop Windows, wxBestHelpController is used (MS HTML Help or WinHelp).

- On Windows CE, wxWinceHelpController is used.

- On all other platforms, wxHtmlHelpController is used if wxHTML is compiled into wxWidgets; otherwise wxExtHelpController is used (for invoking an external browser).

The remaining help controller classes need to be named explicitly by an application that wishes to make use of them.

There are currently the following help controller classes defined:

- wxWinHelpController, for controlling Windows Help.

- wxCHMHelpController, for controlling MS HTML Help. To use this, you need to set wxUSE_MS_HTML_HELP to 1 in setup.h and have htmlhelp.h header from Microsoft's HTML Help kit (you don't need VC++ specific htmlhelp.lib because wxWidgets loads necessary DLL at runtime and so it works with all compilers).

- wxBestHelpController, for controlling MS HTML Help or, if Microsoft's runtime is not available, *wxHtmlHelpController* (p. 722). You need to provide**both** CHM and HTB versions of the help file. For 32bit Windows only.

- wxExtHelpController, for controlling external browsers under Unix. The default browser is Netscape Navigator. The 'help' sample shows its use.

- wxWinceHelpController, for controlling a simple `.htm` help controller for Windows CE applications.

- *wxHtmlHelpController* (p. 722), a sophisticated help controller using *wxHTML* (p. 1845), in a similar style to the Microsoft HTML Help viewer and using some of the same files. Although it has an API compatible with other help controllers, it has more advanced features, so it is recommended that you use the specific API for this class instead. Note that if you use .zip or .htb formats for your books, you must add this line to your application initialization:
  `wxFileSystem::AddHandler(new wxZipFSHandler);`or nothing will be shown in your help window.

Derived from

wxHelpControllerBase
wxObject (p. 1012)

Include files

<wx/help.h> (wxWidgets chooses the appropriate help controller class)
<wx/helpbase.h> (wxHelpControllerBase class)
<wx/helpwin.h> (Windows Help controller)
<wx/msw/helpchm.h> (MS HTML Help controller)
<wx/generic/helpext.h> (external HTML browser controller)
<wx/html/helpctrl.h> (wxHTML based help controller: wxHtmlHelpController)

See also

wxHtmlHelpController (p. 722), *wxHTML* (p. 1845)

wxHelpController::wxHelpController

wxHelpController()

Constructs a help instance object, but does not invoke the help viewer.

wxHelpController::~wxHelpController

~wxHelpController()

Destroys the help instance, closing down the viewer if it is running.

wxHelpController::Initialize

virtual void Initialize(**const wxString&** *file*)

virtual void Initialize(**const wxString&** *file*, **int** *server*)

Initializes the help instance with a help filename, and optionally a server socket number if using wxHelp (now obsolete). Does not invoke the help viewer. This must be called directly after the help instance object is created and before any attempts to communicate with the viewer.

You may omit the file extension and a suitable one will be chosen. For wxHtmlHelpController, the extensions zip, htb and hhp will be appended while searching for a suitable file. For WinHelp, the hlp extension is appended.

wxHelpController::DisplayBlock

virtual bool DisplayBlock(**long** *blockNo*)

If the help viewer is not running, runs it and displays the file at the given block number.

WinHelp: Refers to the context number.

MS HTML Help: Refers to the context number.

External HTML help: the same as for *wxHelpController::DisplaySection* (p. 699).

wxHtmlHelpController: sectionNo is an identifier as specified in the `.hhc` file. See *Help files format* (p. 1847).

This function is for backward compatibility only, and applications should use *wxHelpController* (p. 699) instead.

wxHelpController::DisplayContents

virtual bool DisplayContents()

If the help viewer is not running, runs it and displays the contents.

wxHelpController::DisplayContextPopup

virtual bool DisplayContextPopup(**int** *contextId*)

Displays the section as a popup window using a context id.

Returns false if unsuccessful or not implemented.

wxHelpController::DisplaySection

virtual bool DisplaySection(**const wxString&** *section*)

If the help viewer is not running, runs it and displays the given section.

The interpretation of *section* differs between help viewers. For most viewers, this call is equivalent to KeywordSearch. For MS HTML Help, wxHTML help and external HTML help, if *section* has a .htm or .html extension, that HTML file will be displayed; otherwise a keyword search is done.

virtual bool DisplaySection(**int** *sectionNo*)

If the help viewer is not running, runs it and displays the given section.

WinHelp, MS HTML Help sectionNo is a context id.

External HTML help: wxExtHelpController implements *sectionNo* as an id in a map file, which is of the form:

```
0  wx.html              ; Index
1  wx34.html#classref   ; Class reference
2  wx204.html           ; Function reference
```

wxHtmlHelpController: sectionNo is an identifier as specified in the `.hhc` file. See *Help files format* (p. 1847).

See also the help sample for notes on how to specify section numbers for various help file formats.

wxHelpController::DisplayTextPopup

virtual bool DisplayTextPopup(**const wxString&** *text*, **const wxPoint&** *pos*)

Displays the text in a popup window, if possible.

Returns false if unsuccessful or not implemented.

wxHelpController::GetFrameParameters

virtual wxFrame * GetFrameParameters(**const wxSize *** *size = NULL*, **const wxPoint *** *pos = NULL*, **bool ****newFrameEachTime = NULL*)

wxHtmlHelpController returns the frame, size and position.

For all other help controllers, this function does nothing and just returns NULL.

Parameters

viewer

> This defaults to "netscape" for wxExtHelpController.

flags

> This defaults to wxHELP_NETSCAPE for wxExtHelpController, indicating that the viewer is a variant of Netscape Navigator.

wxHelpController::KeywordSearch

virtual bool KeywordSearch(**const wxString&** *keyWord*, **wxHelpSearchMode** *mode = wxHELP_SEARCH_ALL*)

If the help viewer is not running, runs it, and searches for sections matching the given keyword. If one match is found, the file is displayed at this section. The optional parameter allows the search the index (wxHELP_SEARCH_INDEX) but this currently only supported by the wxHtmlHelpController.

WinHelp, MS HTML Help: If more than one match is found, the first topic is displayed.

External HTML help, simple wxHTML help: If more than one match is found, a choice of topics is displayed.

wxHtmlHelpController: see *wxHtmlHelpController::KeywordSearch* (p. 726).

wxHelpController::LoadFile

virtual bool LoadFile(**const wxString&** *file = ""*)

If the help viewer is not running, runs it and loads the given file. If the filename is not supplied or is empty, the file specified in **Initialize** is used. If the viewer is already displaying the specified file, it will not be reloaded. This member function may be used before each display call in case the user has opened another file.

wxHtmlHelpController ignores this call.

wxHelpController::OnQuit

virtual bool OnQuit()

Overridable member called when this application's viewer is quit by the user.

This does not work for all help controllers.

wxHelpController::SetFrameParameters

virtual void SetFrameParameters(**const wxString &** *title*, **const wxSize &** *size*, **const wxPoint &** *pos* = *wxDefaultPosition*, **bool** *newFrameEachTime = false*)

For wxHtmlHelpController, the title is set (again with %s indicating the page title) and also the size and position of the frame if the frame is already open. *newFrameEachTime* is ignored.

For all other help controllers this function has no effect.

wxHelpController::SetViewer

virtual void SetViewer(**const wxString&** *viewer*, **long** *flags*)

Sets detailed viewer information. So far this is only relevant to wxExtHelpController.

Some examples of usage:

```
m_help.SetViewer("kdehelp");
m_help.SetViewer("gnome-help-browser");
m_help.SetViewer("netscape", wxHELP_NETSCAPE);
```

wxHelpController::Quit

virtual bool Quit()

If the viewer is running, quits it by disconnecting.

For Windows Help, the viewer will only close if no other application is using it.

wxHelpControllerHelpProvider

wxHelpControllerHelpProvider is an implementation of wxHelpProvider which supports both context identifiers and plain text help strings. If the help text is an integer, it is passed to wxHelpController::DisplayContextPopup. Otherwise, it shows the string in a tooltip as per wxSimpleHelpProvider. If you use this with a wxCHMHelpController instance on windows, it will use the native style of tip window instead of *wxTipWindow* (p. 1397).

You can use the convenience function **wxContextId** to convert an integer context id to a string for passing to *wxWindow::SetHelpText* (p. 1535).

Derived from

wxSimpleHelpProvider (p. 1168)
wxHelpProvider (p. 703)

Include files

<wx/cshelp.h>

See also

wxHelpProvider (p. 703), *wxSimpleHelpProvider* (p. 1168), *wxContextHelp* (p. 212), *wxWindow::SetHelpText* (p. 1535), *wxWindow::GetHelpText* (p. 1509)

wxHelpControllerHelpProvider::wxHelpControllerHelpProvider

wxHelpControllerHelpProvider(**wxHelpControllerBase*** *hc = NULL*)

Note that the instance doesn't own the help controller. The help controller should be deleted separately.

wxHelpControllerHelpProvider::SetHelpController

void SetHelpController(**wxHelpControllerBase*** *hc*)

Sets the help controller associated with this help provider.

wxHelpControllerHelpProvider::GetHelpController

wxHelpControllerBase* GetHelpController() **const**

Returns the help controller associated with this help provider.

wxHelpEvent

A help event is sent when the user has requested context-sensitive help. This can either be caused by the application requesting context-sensitive help mode via *wxContextHelp* (p. 212), or (on MS Windows) by the system generating a WM_HELP message when the user pressed F1 or clicked on the query button in a dialog caption.

A help event is sent to the window that the user clicked on, and is propagated up the window hierarchy until the event is processed or there are no more event handlers. The application should call wxEvent::GetId to check the identity of the clicked-on window, and then either show some suitable help or call wxEvent::Skip if the identifier is unrecognised. Calling Skip is important because it allows wxWidgets to generate further events for ancestors of the clicked-on window. Otherwise it would be impossible to show help for container windows, since processing would stop after the first window found.

Derived from

wxCommandEvent (p. 180)
wxEvent (p. 483)
wxObject (p. 1012)

Include files

<wx/event.h>

Event table macros

To process an activate event, use these event handler macros to direct input to a member function that takes a wxHelpEvent argument.

EVT_HELP(id, func)	Process a wxEVT_HELP event.
EVT_HELP_RANGE(id1, id2, func)	Process a wxEVT_HELP event for a range of ids.

See also
wxContextHelp (p. 212), *wxDialog* (p. 406), *Event handling overview* (p. 1751)

wxHelpEvent::wxHelpEvent

 wxHelpEvent(**WXTYPE** *eventType = 0*, **wxWindowID** *id = 0*, **const wxPoint&** *point*)

Constructor.

wxHelpEvent::GetPosition

const wxPoint& GetPosition() **const**

Returns the left-click position of the mouse, in screen coordinates. This allows the application to position the help appropriately.

wxHelpEvent::SetPosition

void SetPosition(**const wxPoint&** *pt*)

Sets the left-click position of the mouse, in screen coordinates.

wxHelpProvider

wxHelpProvider is an abstract class used by a program implementing context-sensitive help to show the help text for the given window.

The current help provider must be explicitly set by the application using wxHelpProvider::Set().

Derived from

No base class

Include files

<wx/cshelp.h>

See also

wxContextHelp (p. 212), *wxContextHelpButton* (p. 213), *wxSimpleHelpProvider* (p. 1168), *wxHelpControllerHelpProvider* (p. 701), *wxWindow::SetHelpText* (p. 1535), *wxWindow::GetHelpText* (p. 1509)

wxHelpProvider::~wxHelpProvider

 ~wxHelpProvider()

Virtual destructor for any base class.

wxHelpProvider::AddHelp

void AddHelp(**wxWindowBase*** *window*, **const wxString&** *text*)

Associates the text with the given window or id. Although all help providers have these functions to allow making *wxWindow::SetHelpText* (p. 1535) work, not all of them implement the functions.

wxHelpProvider::Get

wxHelpProvider* Get()

Unlike some other classes, the help provider is not created on demand. This must be explicitly done by the application.

wxHelpProvider::GetHelp

wxString GetHelp(**const wxWindowBase*** *window*)

Gets the help string for this window. Its interpretation is dependent on the help provider except that empty string always means that no help is associated with the window.

void AddHelp(**wxWindowID** *id*, **const wxString&** *text*)

This version associates the given text with all windows with this id. May be used to set the same help string for all Cancel buttons in the application, for example.

wxHelpProvider::RemoveHelp

void RemoveHelp(**wxWindowBase*** *window*)

Removes the association between the window pointer and the help text. This is called by the wxWindow destructor. Without this, the table of help strings will fill up and when window pointers are reused, the wrong help string will be found.

wxHelpProvider::Set

wxHelpProvider* Set(**wxHelpProvider*** *helpProvider*)

Get/set the current, application-wide help provider. Returns the previous one.

wxHelpProvider::ShowHelp

bool ShowHelp(**wxWindowBase*** *window*)

Shows help for the given window. Uses *GetHelp* (p. 704) internally if applicable.

Returns true if it was done, or false if no help was available for this window.

wxHtmlCell

Internal data structure. It represents fragments of parsed HTML page, the so-called **cell** - a word, picture, table, horizontal line and so on. It is used by *wxHtmlWindow* (p. 749) and *wxHtmlWinParser* (p. 758) to represent HTML page in memory.

You can divide cells into two groups : *visible* cells with non-zero width and height and *helper* cells (usually with zero width and height) that perform special actions such as color or font change.

Derived from

wxObject (p. 1012)

Include files

<wx/html/htmlcell.h>

See Also

Cells Overview (p. 1848),*wxHtmlContainerCell* (p. 711)

wxHtmlCell::wxHtmlCell

wxHtmlCell()

Constructor.

wxHtmlCell::AdjustPagebreak

virtual bool AdjustPagebreak(**int *** *pagebreak*)

This method is used to adjust pagebreak position. The parameter is variable that contains y-coordinate of page break (= horizontal line that should not be crossed by words, images etc.). If this cell cannot be divided into two pieces (each one on another page) then it moves the pagebreak few pixels up.

Returns true if pagebreak was modified, false otherwise

```
Usage: while (container->AdjustPagebreak(&p)) {}
```

wxHtmlCell::Draw

virtual void Draw(**wxDC&** *dc*, **int** *x*, **int** *y*, **int** *view_y1*, **int** *view_y2*)

Renders the cell.

Parameters

dc

Device context to which the cell is to be drawn

x,y

Coordinates of parent's upper left corner (origin). You must add this to m_PosX,m_PosY when passing coordinates to dc's methods Example : `dc -> DrawText("hello", x + m_PosX, y + m_PosY)`

view_y1

y-coord of the first line visible in window. This is used to optimize rendering speed

view_y2

y-coord of the last line visible in window. This is used to optimize rendering speed

wxHtmlCell::DrawInvisible

virtual void DrawInvisible(**wxDC&** *dc*, **int** *x*, **int** *y*)

This method is called instead of *Draw* (p. 706) when the cell is certainly out of the screen (and thus invisible). This is not nonsense - some tags (like *wxHtmlColourCell* (p. 710)or font setter) must be drawn even if they are invisible!

Parameters

dc

Device context to which the cell is to be drawn

x,y

> Coordinates of parent's upper left corner. You must add this to m_PosX,m_PosY when passing coordinates to dc's methods Example : `dc -> DrawText("hello", x + m_PosX, y + m_PosY)`

wxHtmlCell::Find

virtual const wxHtmlCell* Find(**int** *condition*, **const void*** *param*)

Returns pointer to itself if this cell matches condition (or if any of the cells following in the list matches), NULL otherwise. (In other words if you call top-level container's Find it will return pointer to the first cell that matches the condition)

It is recommended way how to obtain pointer to particular cell or to cell of some type (e.g. wxHtmlAnchorCell reacts on wxHTML_COND_ISANCHOR condition)

Parameters

condition

> Unique integer identifier of condition

param

> Optional parameters

Defined conditions

wxHTML_COND_ISANCHOR	Finds particular anchor. *param* is pointer to wxString with name of the anchor.
wxHTML_COND_USER	User-defined conditions start from this number.

wxHtmlCell::GetDescent

int GetDescent() **const**

Returns descent value of the cell (m_Descent member). See explanation:

PRESS F9 TO FORMAT PICTURE

wxHtmlCell::GetHeight

int GetHeight() **const**

Returns height of the cell (m_Height member).

wxHtmlCell::GetId

virtual wxString GetId() **const**

Returns unique cell identifier if there is any, empty string otherwise.

wxHtmlCell::GetLink

virtual wxHtmlLinkInfo* GetLink(**int** *x = 0*, **int** *y = 0*) **const**

Returns hypertext link if associated with this cell or NULL otherwise. See *wxHtmlLinkInfo* (p. 732). (Note: this makes sense only for visible tags).

Parameters

x,y

> Coordinates of position where the user pressed mouse button. These coordinates are used e.g. by COLORMAP. Values are relative to the upper left corner of THIS cell (i.e. from 0 to m_Width or m_Height)

wxHtmlCell::GetNext

wxHtmlCell* GetNext() **const**

Returns pointer to the next cell in list (see htmlcell.h if you're interested in details).

wxHtmlCell::GetParent

wxHtmlContainerCell* GetParent() **const**

Returns pointer to parent container.

wxHtmlCell::GetPosX

int GetPosX() **const**

Returns X position within parent (the value is relative to parent's upper left corner). The returned value is meaningful only if parent's *Layout* (p. 709) was called before!

wxHtmlCell::GetPosY

int GetPosY() **const**

Returns Y position within parent (the value is relative to parent's upper left corner). The returned value is meaningful only if parent's *Layout* (p. 709) was called before!

wxHtmlCell::GetWidth

int GetWidth() **const**

Returns width of the cell (m_Width member).

wxHtmlCell::Layout

virtual void Layout(**int** *w*)

This method performs two actions:

1. adjusts the cell's width according to the fact that maximal possible width is *w*. (this has sense when working with horizontal lines, tables etc.)

2. prepares layout (=fill-in m_PosX, m_PosY (and sometimes m_Height) members) based on actual width *w*

It must be called before displaying cells structure because m_PosX and m_PosY are undefined (or invalid) before calling Layout.

wxHtmlCell::OnMouseClick

virtual void OnMouseClick(**wxWindow*** *parent*, **int***x*, **int** *y*, **const wxMouseEvent&** *event*)

This function is simple event handler. Each time the user clicks mouse button over a cell within *wxHtmlWindow* (p. 749) this method of that cell is called. Default behavior is that it calls *wxHtmlWindow::LoadPage* (p. 753).

Note

If you need more "advanced" event handling you should use wxHtmlBinderCell instead.

Parameters

parent

> parent window (always wxHtmlWindow!)

x, y

> coordinates of mouse click (this is relative to cell's origin

left, middle, right

> boolean flags for mouse buttons. true if the left/middle/right button is pressed, false otherwise

wxHtmlCell::SetId

void SetId(**const wxString&** *id*)

Sets unique cell identifier. Default value is no identifier, i.e. empty string.

wxHtmlCell::SetLink

void SetLink(**const wxHtmlLinkInfo&** *link*)

Sets the hypertext link associated with this cell. (Default value is *wxHtmlLinkInfo* (p. 732)("", "") (no link))

wxHtmlCell::SetNext

void SetNext(**wxHtmlCell** **cell*)

Sets the next cell in the list. This shouldn't be called by user - it is to be used only by *wxHtmlContainerCell::InsertCell* (p. 712).

wxHtmlCell::SetParent

void SetParent(**wxHtmlContainerCell** **p*)

Sets parent container of this cell. This is called from*wxHtmlContainerCell::InsertCell* (p. 712).

wxHtmlCell::SetPos

void SetPos(**int** *x*, **int** *y*)

Sets the cell's position within parent container.

wxHtmlColourCell

This cell changes the colour of either the background or the foreground.

Derived from

wxHtmlCell (p. 705)

Include files

<wx/html/htmlcell.h>

wxHtmlColourCell::wxHtmlColourCell

wxHtmlColourCell(**wxColour** *clr*, **int** *flags = wxHTML_CLR_FOREGROUND*)

Constructor.

Parameters

clr

> The color

flags

> Can be one of following:

> **wxHTML_CLR_FOREGROUND**   change color of text

> **wxHTML_CLR_BACKGROUND**   change background color

wxHtmlContainerCell

The wxHtmlContainerCell class is an implementation of a cell that may contain more cells in it. It is heavily used in the wxHTML layout algorithm.

Derived from

wxHtmlCell (p. 705)

Include files

<wx/html/htmlcell.h>

See Also

Cells Overview (p. 1848)

wxHtmlContainerCell::wxHtmlContainerCell

 wxHtmlContainerCell(**wxHtmlContainerCell** **parent*)

Constructor. *parent* is pointer to parent container or NULL.

wxHtmlContainerCell::GetAlignHor

int GetAlignHor() **const**

Returns container's horizontal alignment.

wxHtmlContainerCell::GetAlignVer

int GetAlignVer() **const**

Returns container's vertical alignment.

wxHtmlContainerCell::GetBackgroundColour

wxColour GetBackgroundColour()

Returns the background colour of the container or `wxNullColour` if no background colour is set.

wxHtmlContainerCell::GetFirstCell

wxHtmlCell* GetFirstCell()

Returns pointer to the first cell in the list. You can then use wxHtmlCell's GetNext method to obtain pointer to the next cell in list.

Note: This shouldn't be used by the end user. If you need some way of finding particular cell in the list, try *Find* (p. 707) method instead.

wxHtmlContainerCell::GetIndent

int GetIndent(**int** *ind*) **const**

Returns the indentation. *ind* is one of the **wxHTML_INDENT_*** constants.

Note: You must call *GetIndentUnits* (p. 712) with same *ind* parameter in order to correctly interpret the returned integer value. It is NOT always in pixels!

wxHtmlContainerCell::GetIndentUnits

int GetIndentUnits(**int** *ind*) **const**

Returns the units of indentation for *ind* where *ind* is one of the **wxHTML_INDENT_*** constants.

wxHtmlContainerCell::InsertCell

void InsertCell(**wxHtmlCell** **cell*)

Inserts new cell into the container.

wxHtmlContainerCell::SetAlign

void SetAlign(**const wxHtmlTag&** *tag*)

Sets the container's alignment (both horizontal and vertical) according to the values stored in *tag*. (Tags ALIGN parameter is extracted.) In fact it is only a front-end to *SetAlignHor* (p. 713) and *SetAlignVer* (p. 713).

wxHtmlContainerCell::SetAlignHor

void SetAlignHor(**int** *al*)

Sets the container's *horizontal alignment*. During *Layout* (p. 709) each line is aligned according to *al* value.

Parameters

al

> new horizontal alignment. May be one of these values:

wxHTML_ALIGN_LEFT	lines are left-aligned (default)
wxHTML_ALIGN_JUSTIFY	lines are justified
wxHTML_ALIGN_CENTER	lines are centered
wxHTML_ALIGN_RIGHT	lines are right-aligned

wxHtmlContainerCell::SetAlignVer

void SetAlignVer(**int** *al*)

Sets the container's *vertical alignment*. This is per-line alignment!

Parameters

al

> new vertical alignment. May be one of these values:

wxHTML_ALIGN_BOTTOM	cells are over the line (default)
wxHTML_ALIGN_CENTER	cells are centered on line
wxHTML_ALIGN_TOP	cells are under the line

> PRESS F9 TO FORMAT PICTURE

wxHtmlContainerCell::SetBackgroundColour

void SetBackgroundColour(**const wxColour&** *clr*)

Sets the background colour for this container.

wxHtmlContainerCell::SetBorder

void SetBorder(**const wxColour&** *clr1*, **const wxColour&** *clr2*)

Sets the border (frame) colours. A border is a rectangle around the container.

Parameters

clr1

> Colour of top and left lines

clr2

> Colour of bottom and right lines

wxHtmlContainerCell::SetIndent

void SetIndent(**int** *i*, **int** *what*, **int** *units* = *wxHTML_UNITS_PIXELS*)

Sets the indentation (free space between borders of container and subcells).

Parameters

i

> Indentation value.

what

> Determines which of the four borders we're setting. It is OR combination of following constants:

wxHTML_INDENT_TOP	top border
wxHTML_INDENT_BOTTOM	bottom
wxHTML_INDENT_LEFT	left
wxHTML_INDENT_RIGHT	right
wxHTML_INDENT_HORIZONTAL	left and right
wxHTML_INDENT_VERTICAL	top and bottom
wxHTML_INDENT_ALL	all 4 borders

> PRESS F9 TO FORMAT PICTURE

units

> Units of *i*. This parameter affects interpretation of value.

wxHTML_UNITS_PIXELS	*i* is number of pixels

wxHTML_UNITS_PERCENT	*i* is interpreted as percents of width of parent container

wxHtmlContainerCell::SetMinHeight

void SetMinHeight(**int** *h*, **int** *align = wxHTML_ALIGN_TOP*)

Sets minimal height of the container.

When container's *Layout* (p. 709) is called, m_Height is set depending on layout of subcells to the height of area covered by layed-out subcells. Calling this method guarantees you that the height of container is never smaller than *h* - even if the subcells cover much smaller area.

Parameters

h

> The minimal height.

align

> If height of the container is lower than the minimum height, empty space must be inserted somewhere in order to ensure minimal height. This parameter is one of **wxHTML_ALIGN_TOP, wxHTML_ALIGN_BOTTOM, wxHTML_ALIGN_CENTER**. It refers to the *contents*, not to the empty place.

wxHtmlContainerCell::SetWidthFloat

void SetWidthFloat(**int** *w*, **int** *units*)

void SetWidthFloat(**const wxHtmlTag&** *tag*, **double** *pixel_scale = 1.0*)

Sets floating width adjustment.

The normal behaviour of container is that its width is the same as the width of parent container (and thus you can have only one sub-container per line). You can change this by setting FWA.

pixel_scale is number of real pixels that equals to 1 HTML pixel.

Parameters

w

> Width of the container. If the value is negative it means complement to full width of parent container (e.g.`SetWidthFloat(-50, wxHTML_UNITS_PIXELS)` sets the width of container to parent's width minus 50 pixels. This is useful when creating tables - you can call SetWidthFloat(50) and SetWidthFloat(-50))

units

Units of *w* This parameter affects the interpretation of value.

wxHTML_UNITS_PIXELS	*w* is number of pixels
wxHTML_UNITS_PERCENT	*w* is interpreted as percents of width of parent container

tag

In the second version of method, *w* and *units*info is extracted from tag's WIDTH parameter.

wxPython note: The second form of this method is named SetWidthFloatFromTag in wxPython.

wxHtmlDCRenderer

This class can render HTML document into a specified area of a DC. You can use it in your own printing code, although use of *wxHtmlEasyPrinting* (p. 718) or *wxHtmlPrintout* (p. 741) is strongly recommended.

Derived from

wxObject (p. 1012)

Include files

<wx/html/htmprint.h>

wxHtmlDCRenderer::wxHtmlDCRenderer

 wxHtmlDCRenderer()

Constructor.

wxHtmlDCRenderer::SetDC

void SetDC(**wxDC*** *dc*, **double** *pixel_scale = 1.0*)

Assign DC instance to the renderer.

pixel_scale can be used when rendering to high-resolution DCs (e.g. printer) to adjust size of pixel metrics. (Many dimensions in HTML are given in pixels -- e.g. image sizes. 300x300 image would be only one inch wide on typical printer. With pixel_scale = 3.0 it would be 3 inches.)

wxHtmlDCRenderer::SetFonts

void SetFonts(**wxString** *normal_face*, **wxString** *fixed_face*, **const int** **sizes = NULL*)

Sets fonts. See *wxHtmlWindow::SetFonts* (p. 756) for detailed description.

See also *SetSize* (p. 717).

wxHtmlDCRenderer::SetSize

void SetSize(**int** *width*, **int** *height*)

Set size of output rectangle, in pixels. Note that you **can't** change width of the rectangle between calls to *Render* (p. 717)! (You can freely change height.)

wxHtmlDCRenderer::SetHtmlText

void SetHtmlText(**const wxString&** *html*, **const wxString&** *basepath = wxEmptyString*, **bool** *isdir = true*)

Assign text to the renderer. *Render* (p. 717) then draws the text onto DC.

Parameters

html

> HTML text. This is *not* a filename.

basepath

> base directory (html string would be stored there if it was in file). It is used to determine path for loading images, for example.

isdir

> false if basepath is filename, true if it is directory name (see *wxFileSystem* (p. 540) for detailed explanation)

wxHtmlDCRenderer::Render

int Render(**int** *x*, **int** *y*, **int** *from = 0*, **int** *dont_render = false*)

Renders HTML text to the DC.

Parameters

x,y

> position of upper-left corner of printing rectangle (see *SetSize* (p. 717))

from

y-coordinate of the very first visible cell

dont_render

if true then this method only returns y coordinate of the next page and does not output anything

Returned value is y coordinate of first cell than didn't fit onto page. Use this value as *from* in next call to Render in order to print multipages document.

Caution!

The Following three methods **must** always be called before any call to Render (preferably in this order):

- *SetDC* (p. 716)

- *SetSize* (p. 717)

- *SetHtmlText* (p. 717)

Render() changes the DC's user scale and does NOT restore it.

wxHtmlDCRenderer::GetTotalHeight

int GetTotalHeight()

Returns the height of the HTML text. This is important if area height (see *SetSize* (p. 717)) is smaller that total height and thus the page cannot fit into it. In that case you're supposed to call *Render* (p. 717) as long as its return value is smaller than GetTotalHeight's.

wxHtmlEasyPrinting

This class provides very simple interface to printing architecture. It allows you to print HTML documents using only a few commands.

Note

Do not create this class on the stack only. You should create an instance on app startup and use this instance for all printing operations. The reason is that this class stores various settings in it.

Derived from

wxObject (p. 1012)

Include files

<wx/html/htmprint.h>

wxHtmlEasyPrinting::wxHtmlEasyPrinting

wxHtmlEasyPrinting(**const wxString&** *name* = *"Printing"*, **wxWindow*** *parentWindow* = *NULL*)

Constructor.

Parameters

name

> Name of the printing object. Used by preview frames and setup dialogs.

parentWindow

> pointer to the window that will own the preview frame and setup dialogs. May be NULL.

wxHtmlEasyPrinting::PreviewFile

bool PreviewFile(**const wxString&** *htmlfile*)

Preview HTML file.

Returns false in case of error -- call*wxPrinter::GetLastError* (p. 1063) to get detailed information about the kind of the error.

wxHtmlEasyPrinting::PreviewText

bool PreviewText(**const wxString&** *htmltext*, **const wxString&** *basepath* = *wxEmptyString*)

Preview HTML text (not file!).

Returns false in case of error -- call*wxPrinter::GetLastError* (p. 1063) to get detailed information about the kind of the error.

Parameters

htmltext

> HTML text.

basepath

> base directory (html string would be stored there if it was in file). It is used to determine path for loading images, for example.

wxHtmlEasyPrinting::PrintFile

bool PrintFile(**const wxString&** *htmlfile*)

Print HTML file.

Returns false in case of error -- call*wxPrinter::GetLastError* (p. 1063) to get detailed information about the kind of the error.

wxHtmlEasyPrinting::PrintText

bool PrintText(**const wxString&** *htmltext*, **const wxString&** *basepath* = *wxEmptyString*)

Print HTML text (not file!).

Returns false in case of error -- call*wxPrinter::GetLastError* (p. 1063) to get detailed information about the kind of the error.

Parameters

htmltext

> HTML text.

basepath

> base directory (html string would be stored there if it was in file). It is used to determine path for loading images, for example.

wxHtmlEasyPrinting::PageSetup

void PageSetup()

Display page setup dialog and allows the user to modify settings.

wxHtmlEasyPrinting::SetFonts

void SetFonts(**wxString** *normal_face*, **wxString** *fixed_face*, **const int** **sizes = NULL*)

Sets fonts. See *wxHtmlWindow::SetFonts* (p. 756) for detailed description.

wxHtmlEasyPrinting::SetHeader

void SetHeader(**const wxString&** *header*, **int** *pg = wxPAGE_ALL*)

Set page header.

Parameters

header

> HTML text to be used as header. You can use macros in it:

- @PAGENUM@ is replaced by page number

- @PAGESCNT@ is replaced by total number of pages

pg

one of wxPAGE_ODD, wxPAGE_EVEN and wxPAGE_ALL constants.

wxHtmlEasyPrinting::SetFooter

void SetFooter(**const wxString&** *footer*, **int** *pg = wxPAGE_ALL*)

Set page footer.

Parameters

footer

HTML text to be used as footer. You can use macros in it:

- @PAGENUM@ is replaced by page number

- @PAGESCNT@ is replaced by total number of pages

pg

one of wxPAGE_ODD, wxPAGE_EVEN and wxPAGE_ALL constants.

wxHtmlEasyPrinting::GetPrintData

wxPrintData* GetPrintData()

Returns pointer to *wxPrintData* (p. 1051) instance used by this class. You can set its parameters (via SetXXXX methods).

wxHtmlEasyPrinting::GetPageSetupData

wxPageSetupDialogData* GetPageSetupData()

Returns a pointer to *wxPageSetupDialogData* (p. 1019) instance used by this class. You can set its parameters (via SetXXXX methods).

wxHtmlFilter

This class is the parent class of input filters for *wxHtmlWindow* (p. 749). It allows you to read and display files of different file formats.

Derived from

wxObject (p. 1012)

<wx/html/htmlfilt.h>

See Also

Overview (p. 1848)

wxHtmlFilter::wxHtmlFilter

wxHtmlFilter()

Constructor.

wxHtmlFilter::CanRead

bool CanRead(**const wxFSFile&** *file*)

Returns true if this filter is capable of reading file *file*.

Example:

```
bool MyFilter::CanRead(const wxFSFile& file)
{
    return (file.GetMimeType() == "application/x-ugh");
}
```

wxHtmlFilter::ReadFile

wxString ReadFile(**const wxFSFile&** *file*)

Reads the file and returns string with HTML document.

Example:

```
wxString MyImgFilter::ReadFile(const wxFSFile& file)
{
    return "<html><body><img src=\"" +
            file.GetLocation() +
        "\"></body></html>";
}
```

wxHtmlHelpController

WARNING! Although this class has an API compatible with other wxWidgets help controllers as documented by *wxHelpController* (p. 696), it is recommended that you use

the enhanced capabilities of wxHtmlHelpController's API.

This help controller provides an easy way of displaying HTML help in your application (see *test* sample). The help system is based on **books** (see *AddBook* (p. 724)). A book is a logical section of documentation (for example "User's Guide" or "Programmer's Guide" or "C++ Reference" or "wxWidgets Reference"). The help controller can handle as many books as you want.

wxHTML uses Microsoft's HTML Help Workshop project files (.hhp, .hhk, .hhc) as its native format. The file format is described *here* (p. 1847). Have a look at docs/html/ directory where sample project files are stored.

You can use Tex2RTF to produce these files when generating HTML, if you set **htmlWorkshopFiles** to **true** in your tex2rtf.ini file.

Note

It is strongly recommended to use preprocessed **.hhp.cached** version of projects. It can be either created on-the-fly (see *SetTempDir* (p. 726)) or you can use **hhp2cached** utility from *utils/hhp2cached* to create it and distribute the cached version together with helpfiles. See *samples/html/help* sample for demonstration of its use.

See also

Information about wxBestHelpController (p. 696)

Derived from

wxHelpControllerBase

Include files

<wx/html/helpctrl.h>

wxHtmlHelpController::wxHtmlHelpController

 wxHtmlHelpController(**int** *style = wxHF_DEFAULT_STYLE*)

Constructor.

Parameters

style is combination of these flags:

wxHF_TOOLBAR	Help frame has toolbar.
wxHF_FLAT_TOOLBAR	Help frame has toolbar with flat buttons (aka coolbar).
wxHF_CONTENTS	Help frame has contents panel.
wxHF_INDEX	Help frame has index panel.

wxHF_SEARCH	Help frame has search panel.
wxHF_BOOKMARKS	Help frame has bookmarks controls.
wxHF_OPEN_FILES	Allow user to open arbitrary HTML document.
wxHF_PRINT	Toolbar contains "print" button.
wxHF_MERGE_BOOKS	Contents pane does not show book nodes. All books are merged together and appear as single book to the user.
wxHF_ICONS_BOOK	All nodes in contents pane have a book icon. This is how Microsoft's HTML help viewer behaves.
wxHF_ICONS_FOLDER	Book nodes in contents pane have a book icon, book's sections have a folder icon. This is the default.
wxHF_ICONS_BOOK_CHAPTER	Both book nodes and nodes of top-level sections of a book (i.e. chapters) have a book icon, all other sections (sections, subsections, ...) have a folder icon.
wxHF_DEFAULT_STYLE	`wxHF_TOOLBAR	wxHF_CONTENTS |` `wxHF_INDEX	wxHF_SEARCH |` `wxHF_BOOKMARKS	wxHF_PRINT`

wxHtmlHelpController::AddBook

bool AddBook(**const wxFileName&** *book_file*, **bool** *show_wait_msg*)

bool AddBook(**const wxString&** *book_url*, **bool** *show_wait_msg*)

Adds book (*.hhp file* (p. 1847) - HTML Help Workshop project file) into the list of loaded books. This must be called at least once before displaying any help.

book_file or *book_url* may be either .hhp file or ZIP archive that contains arbitrary number of .hhp files in top-level directory. This ZIP archive must have .zip or .htb extension (the latter stands for "HTML book"). In other words,`AddBook(wxFileName("help.zip"))`is possible and, in fact, recommended way.

Parameters

show_wait_msg

If true then a decoration-less window with progress message is displayed.

book_file

Help book filename. It is recommended to use this prototype instead of the one

taking URL, because it is less error-prone.

book_url

Help book URL (note that syntax of filename and URL is different on most platforms)

Note

Don't forget to install wxFileSystem ZIP handler with`wxFileSystem::AddHandler(new wxZipFSHandler);` before calling this method on a .zip or .htb file!

wxHtmlHelpController::CreateHelpFrame

virtual wxHtmlHelpFrame* CreateHelpFrame(wxHtmlHelpData * *data*)

This protected virtual method may be overridden so that the controller uses slightly different frame. See *samples/html/helpview* sample for an example.

wxHtmlHelpController::Display

void Display(const wxString& *x*)

Displays page *x*. This is THE important function - it is used to display the help in application.

You can specify the page in many ways:

- as direct filename of HTML document

- as chapter name (from contents) or as a book name

- as some word from index

- even as any word (will be searched)

Looking for the page runs in these steps:

1. try to locate file named x (if x is for example "doc/howto.htm")

2. try to open starting page of book named x

3. try to find x in contents (if x is for example "How To ...")

4. try to find x in index (if x is for example "How To ...")

5. switch to Search panel and start searching

void Display(const int *id*)

This alternative form is used to search help contents by numeric IDs.

wxPython note: The second form of this method is named DisplayId in wxPython.

wxHtmlHelpController::DisplayContents

void DisplayContents()

Displays help window and focuses contents panel.

wxHtmlHelpController::DisplayIndex

void DisplayIndex()

Displays help window and focuses index panel.

wxHtmlHelpController::KeywordSearch

bool KeywordSearch(**const wxString&** *keyword*, **wxHelpSearchMode** *mode* = *wxHELP_SEARCH_ALL*)

Displays help window, focuses search panel and starts searching. Returns true if the keyword was found. Optionally it searches through the index (mode = wxHELP_SEARCH_INDEX), default the content (mode = wxHELP_SEARCH_ALL).

Important: KeywordSearch searches only pages listed in .hhc file(s). You should list all pages in the contents file.

wxHtmlHelpController::ReadCustomization

void ReadCustomization(**wxConfigBase*** *cfg*, **wxString** *path* = *wxEmptyString*)

Reads the controller's setting (position of window, etc.)

wxHtmlHelpController::SetTempDir

void SetTempDir(**const wxString&** *path*)

Sets the path for storing temporary files - cached binary versions of index and contents files. These binary forms are much faster to read. Default value is empty string (empty string means that no cached data are stored). Note that these files are *not* deleted when program exits.

Once created these cached files will be used in all subsequent executions of your application. If cached files become older than corresponding .hhp file (e.g. if you regenerate documentation) it will be refreshed.

wxHtmlHelpController::SetTitleFormat

void SetTitleFormat(**const wxString&** *format*)

Sets format of title of the frame. Must contain exactly one "%s" (for title of displayed HTML page).

wxHtmlHelpController::UseConfig

void UseConfig(**wxConfigBase*** *config*, **const wxString&** *rootpath = wxEmptyString*)

Associates *config* object with the controller.

If there is associated config object, wxHtmlHelpController automatically reads and writes settings (including wxHtmlWindow's settings) when needed.

The only thing you must do is create wxConfig object and call UseConfig.

If you do not use *UseConfig*, wxHtmlHelpController will use default wxConfig object if available (for details see *wxConfigBase::Get* (p. 201) and *wxConfigBase::Set* (p. 205)).

wxHtmlHelpController::WriteCustomization

void WriteCustomization(**wxConfigBase*** *cfg*, **wxString** *path = wxEmptyString*)

Stores controllers setting (position of window etc.)

wxHtmlHelpData

This class is used by *wxHtmlHelpController* (p. 722) and *wxHtmlHelpFrame* (p. 729) to access HTML help items. It is internal class and should not be used directly - except for the case you're writing your own HTML help controller.

Derived from

wxObject (p. 1012)

Include files

<wx/html/helpdata.h>

wxHtmlHelpData::wxHtmlHelpData

wxHtmlHelpData()

Constructor.

wxHtmlHelpData::AddBook

bool AddBook(**const wxString&** *book_url*)

Adds new book. *book* is URL (not filename!) of HTML help project (hhp) or ZIP file that contains arbitrary number of .hhp projects (this zip file can have either .zip or .htb extension, htb stands for "html book"). Returns success.

wxHtmlHelpData::FindPageById

wxString FindPageById(**int** *id*)

Returns page's URL based on integer ID stored in project.

wxHtmlHelpData::FindPageByName

wxString FindPageByName(**const wxString&** *page*)

Returns page's URL based on its (file)name.

wxHtmlHelpData::GetBookRecArray

const wxHtmlBookRecArray& GetBookRecArray()

Returns array with help books info.

wxHtmlHelpData::GetContentsArray

const wxHtmlHelpDataItems& GetContentsArray()

Returns reference to array with contents entries.

wxHtmlHelpData::GetIndexArray

const wxHtmlHelpDataItems& GetIndexArray()

Returns reference to array with index entries.

wxHtmlHelpData::SetTempDir

void SetTempDir(**const wxString&** *path*)

Sets temporary directory where binary cached versions of MS HTML Workshop files will be stored. (This is turned off by default and you can enable this feature by setting non-empty temp dir.)

wxHtmlHelpFrame

This class is used by *wxHtmlHelpController* (p. 722) to display help. It is an internal class and should not be used directly - except for the case when you're writing your own HTML help controller.

Derived from

wxFrame (p. 579)

Include files

<wx/html/helpfrm.h>

wxHtmlHelpFrame::wxHtmlHelpFrame

 wxHtmlHelpFrame(**wxHtmlHelpData*** *data* = *NULL*)

 wxHtmlHelpFrame(**wxWindow*** *parent*, **int** *wxWindowID*, **const wxString&** *title* = *wxEmptyString*, **int** *style* = *wxHF_DEFAULT_STYLE*, **wxHtmlHelpData*** *data* = *NULL*)

Constructor.

style is combination of these flags:

wxHF_TOOLBAR	Help frame has toolbar.
wxHF_FLAT_TOOLBAR	Help frame has toolbar with flat buttons (aka coolbar).
wxHF_CONTENTS	Help frame has contents panel.
wxHF_INDEX	Help frame has index panel.
wxHF_SEARCH	Help frame has search panel.
wxHF_BOOKMARKS	Help frame has bookmarks controls.
wxHF_OPEN_FILES	Allow user to open arbitrary HTML document.
wxHF_PRINT	Toolbar contains "print" button.
wxHF_MERGE_BOOKS	Contents pane does not show book nodes. All books are merged together and appear as single book to the user.
wxHF_ICONS_BOOK	All nodes in contents pane have a book icon. This is how Microsoft's HTML help viewer behaves.
wxHF_ICONS_FOLDER	Book nodes in contents pane have a book icon,

	book's sections have a folder icon. This is the default.
wxHF_ICONS_BOOK_CHAPTER	Both book nodes and nodes of top-level sections of a book (i.e. chapters) have a book icon, all other sections (sections, subsections, ...) have a folder icon.
wxHF_DEFAULT_STYLE	`wxHF_TOOLBAR	wxHF_CONTENTS	wxHF_INDEX	wxHF_SEARCH	wxHF_BOOKMARKS	wxHF_PRINT`

wxHtmlHelpFrame::Create

bool Create(**wxWindow*** *parent*, **wxWindowID** *id*, **const wxString&** *title* = *wxEmptyString*, **int** *style* = *wxHF_DEFAULT_STYLE*)

Creates the frame. See *the constructor* (p. 729)for parameters description.

wxHtmlHelpFrame::CreateContents

void CreateContents()

Creates contents panel. (May take some time.)

Protected.

wxHtmlHelpFrame::CreateIndex

void CreateIndex()

Creates index panel. (May take some time.)

Protected.

wxHtmlHelpFrame::CreateSearch

void CreateSearch()

Creates search panel.

wxHtmlHelpFrame::Display

bool Display(**const wxString&** *x*)

bool Display(**const int** *id*)

Displays page x. If not found it will give the user the choice of searching books. Looking for the page runs in these steps:

1. try to locate file named x (if x is for example "doc/howto.htm")

2. try to open starting page of book x

3. try to find x in contents (if x is for example "How To ...")

4. try to find x in index (if x is for example "How To ...")

The second form takes numeric ID as the parameter. (uses extension to MS format, <param name="ID" value=id>)

wxPython note: The second form of this method is named DisplayId in wxPython.

wxHtmlHelpFrame::DisplayContents

bool DisplayContents()

Displays contents panel.

wxHtmlHelpFrame::DisplayIndex

bool DisplayIndex()

Displays index panel.

wxHtmlHelpFrame::GetData

wxHtmlHelpData* GetData()

Return wxHtmlHelpData object.

wxHtmlHelpFrame::KeywordSearch

bool KeywordSearch(**const wxString&** *keyword*, **wxHelpSearchMode** *mode =* *wxHELP_SEARCH_ALL*)

Search for given keyword. Optionally it searches through the index (mode = wxHELP_SEARCH_INDEX), default the content (mode = wxHELP_SEARCH_ALL).

wxHtmlHelpFrame::ReadCustomization

void ReadCustomization(**wxConfigBase*** *cfg*, **const wxString&** *path =* *wxEmptyString*)

Reads user's settings for this frame (see *wxHtmlHelpController::ReadCustomization* (p.

726))

wxHtmlHelpFrame::RefreshLists

void RefreshLists()

Refresh all panels. This is necessary if a new book was added.

Protected.

wxHtmlHelpFrame::SetTitleFormat

void SetTitleFormat(const wxString& *format*)

Sets the frame's title format. *format* must contain exactly one "%s" (it will be replaced by the page title).

wxHtmlHelpFrame::UseConfig

void UseConfig(wxConfigBase* *config*, const wxString& *rootpath* = *wxEmptyString*)

Add books to search choice panel.

wxHtmlHelpFrame::WriteCustomization

void WriteCustomization(wxConfigBase* *cfg*, const wxString& *path* = *wxEmptyString*)

Saves user's settings for this frame (see *wxHtmlHelpController::WriteCustomization* (p. 727)).

wxHtmlHelpFrame::AddToolbarButtons

virtual void AddToolbarButtons(wxToolBar **toolBar*, int *style*)

You may override this virtual method to add more buttons into help frame's toolbar. *toolBar* is a pointer to the toolbar and *style* is the style flag as passed to Create method.

wxToolBar::Realize is called immediately after returning from this function.

See *samples/html/helpview* for an example.

wxHtmlLinkInfo

This class stores all necessary information about hypertext links (as represented by `<A>` tag in HTML documents). In current implementation it stores URL and target frame

name. *Note that frames are not currently supported by wxHTML!*

Derived from

wxObject (p. 1012)

Include files

<wx/html/htmlcell.h>

wxHtmlLinkInfo::wxHtmlLinkInfo

wxHtmlLinkInfo()

Default ctor.

wxHtmlLinkInfo(**const wxString&** *href*, **const wxString&** *target = wxEmptyString*)

Construct hypertext link from HREF (aka URL) and TARGET (name of target frame).

wxHtmlLinkInfo::GetEvent

const wxMouseEvent * GetEvent()

Return pointer to event that generated OnLinkClicked event. Valid only within *wxHtmlWindow::OnLinkClicked* (p. 754), NULL otherwise.

wxHtmlLinkInfo::GetHtmlCell

const wxHtmlCell * GetHtmlCell()

Return pointer to the cell that was clicked. Valid only within *wxHtmlWindow::OnLinkClicked* (p. 754), NULL otherwise.

wxHtmlLinkInfo::GetHref

wxString GetHref()

Return *HREF* value of the <A> tag.

wxHtmlLinkInfo::GetTarget

wxString GetTarget()

Return *TARGET* value of the <A> tag (this value is used to specify in which frame should be the page pointed by *Href* (p. 733) opened).

wxHtmlListBox

wxHtmlListBox is an implementation of *wxVListBox* (p. 1477) which shows HTML content in the listbox rows. This is still an abstract base class and you will need to derive your own class from it (see htlbox sample for the example) but you will only need to override a single *OnGetItem()* (p. 735) function.

Derived from

wxVListBox (p. 1477)

Include files

<wx/htmllbox.h>

wxHtmlListBox::wxHtmlListBox

wxHtmlListBox(**wxWindow*** *parent*, **wxWindowID** *id = wxID_ANY*, **const wxPoint&** *pos = wxDefaultPosition*, **const wxSize&** *size = wxDefaultSize*, **size_t** *countItems = 0*, **long** *style = 0*, **const wxString&** *name = wxVListBoxNameStr*)

Normal constructor which calls *Create()* (p. 734)internally.

wxHtmlListBox()

Default constructor, you must call *Create()* (p. 734)later.

wxHtmlListBox::~wxHtmlListBox

~wxHtmlListBox()

Destructor cleans up whatever resources we use.

wxHtmlListBox::Create

bool Create(**wxWindow*** *parent*, **wxWindowID** *id = wxID_ANY*, **const wxPoint&** *pos = wxDefaultPosition*, **const wxSize&** *size = wxDefaultSize*, **long** *style = 0*, **const wxString&** *name = wxVListBoxNameStr*)

Creates the control and optionally sets the initial number of items in it (it may also be set or changed later with *SetItemCount()* (p. 1483)).

There are no special styles defined for wxHtmlListBox, in particular the wxListBox styles can not be used here.

Returns `true` on success or `false` if the control couldn't be created

wxHtmlListBox::GetFileSystem

wxFileSystem& GetFileSystem()

const wxFileSystem& GetFileSystem() **const**

Returns the *wxFileSystem* (p. 540) used by the HTML parser of this object. The file system object is used to resolve the paths in HTML fragments displayed in the control and you should use *wxFileSystem::ChangePathTo* (p. 540) if you use relative paths for the images or other resources embedded in your HTML.

wxHtmlListBox::GetSelectedTextBgColour

wxColour GetSelectedTextBgColour(**const wxColour&** *colBg*) **const**

This virtual function may be overridden to change the appearance of the background of the selected cells in the same way as *GetSelectedTextColour* (p. 735).

It should be rarely, if ever, used because *SetSelectionBackground* (p. 1484) allows to change the selection background for all cells at once and doing anything more fancy is probably going to look strangely.

See also

GetSelectedTextColour (p. 735)

wxHtmlListBox::GetSelectedTextColour

wxColour GetSelectedTextColour(**const wxColour&** *colFg*) **const**

This virtual function may be overridden to customize the appearance of the selected cells. It is used to determine how the colour *colFg* is going to look inside selection. By default all original colours are completely ignored and the standard, system-dependent, selection colour is used but the program may wish to override this to achieve some custom appearance.

See also

GetSelectedTextBgColour (p. 735),
SetSelectionBackground (p. 1484),
wxSystemSettings::GetColour (p. 1319)

wxHtmlListBox::OnGetItem

wxString OnGetItem(**size_t** *n*) **const**

This method must be implemented in the derived class and should return the body (i.e. without `<html>` nor `<body>` tags) of the HTML fragment for the given item.

wxHtmlListBox::OnGetItemMarkup

wxString OnGetItemMarkup(**size_t** *n*) **const**

This function may be overridden to decorate HTML returned by*OnGetItem()* (p. 735).

wxHtmlParser

Classes derived from this handle the **generic** parsing of HTML documents: it scans the document and divide it into blocks of tags (where one block consists of beginning and ending tag and of text between these two tags).

It is independent from wxHtmlWindow and can be used as stand-alone parser (Julian Smart's idea of speech-only HTML viewer or wget-like utility - see InetGet sample for example).

It uses system of tag handlers to parse the HTML document. Tag handlers are not statically shared by all instances but are created for each wxHtmlParser instance. The reason is that the handler may contain document-specific temporary data used during parsing (e.g. complicated structures like tables).

Typically the user calls only the *Parse* (p. 739) method.

Derived from

wxObject

Include files

<wx/html/htmlpars.h>

See also

Cells Overview (p. 1848),*Tag Handlers Overview* (p. 1849),*wxHtmlTag* (p. 743)

wxHtmlParser::wxHtmlParser

 wxHtmlParser()

Constructor.

wxHtmlParser::AddTag

void AddTag(**const wxHtmlTag&** *tag*)

This may (and may not) be overwritten in derived class.

This method is called each time new tag is about to be added. *tag* contains information

about the tag. (See *wxHtmlTag* (p. 743)for details.)

Default (wxHtmlParser) behaviour is this: First it finds a handler capable of handling this tag and then it calls handler's HandleTag method.

wxHtmlParser::AddTagHandler

virtual void AddTagHandler(**wxHtmlTagHandler** **handler*)

Adds handler to the internal list (& hash table) of handlers. This method should not be called directly by user but rather by derived class' constructor.

This adds the handler to this **instance** of wxHtmlParser, not to all objects of this class! (Static front-end to AddTagHandler is provided by wxHtmlWinParser).

All handlers are deleted on object deletion.

wxHtmlParser::AddText

virtual void AddWord(**const char*** *txt*)

Must be overwritten in derived class.

This method is called by *DoParsing* (p. 737)each time a part of text is parsed. *txt* is NOT only one word, it is substring of input. It is not formatted or preprocessed (so white spaces are unmodified).

wxHtmlParser::DoParsing

void DoParsing(**int** *begin_pos*, **int** *end_pos*)

void DoParsing()

Parses the m_Source from begin_pos to end_pos-1. (in noparams version it parses whole m_Source)

wxHtmlParser::DoneParser

virtual void DoneParser()

This must be called after DoParsing().

wxHtmlParser::GetFS

wxFileSystem* GetFS() **const**

Returns pointer to the file system. Because each tag handler has reference to it is parent parser it can easily request the file by calling

```
wxFSFile *f = m_Parser -> GetFS() -> OpenFile("image.jpg");
```

wxHtmlParser::GetProduct

virtual wxObject* GetProduct()

Returns product of parsing. Returned value is result of parsing of the document. The type of this result depends on internal representation in derived parser (but it must be derived from wxObject!).

See wxHtmlWinParser for details.

wxHtmlParser::GetSource

wxString* GetSource()

Returns pointer to the source being parsed.

wxHtmlParser::InitParser

virtual void InitParser(const wxString& *source*)

Setups the parser for parsing the *source* string. (Should be overridden in derived class)

wxHtmlParser::OpenURL

virtual wxFSFile* OpenURL(wxHtmlURLType *type*, const wxString& *url*)

Opens given URL and returns `wxFSFile` object that can be used to read data from it. This method may return NULL in one of two cases: either the URL doesn't point to any valid resource or the URL is blocked by overridden implementation of *OpenURL* in derived class.

Parameters

type

Indicates type of the resource. Is one of:

wxHTML_URL_PAGE	Opening a HTML page.
wxHTML_URL_IMAGE	Opening an image.
wxHTML_URL_OTHER	Opening a resource that doesn't fall into any other category.

url

URL being opened.

Notes

Always use this method in tag handlers instead of `GetFS()->OpenFile()` because it can block the URL and is thus more secure.

Default behaviour is to call *wxHtmlWindow::OnOpeningURL* (p. 754)of the associated wxHtmlWindow object (which may decide to block the URL or redirect it to another one),if there's any, and always open the URL if the parser is not used with wxHtmlWindow.

Returned `wxFSFile` object is not guaranteed to point to *url*, it might have been redirected!

wxHtmlParser::Parse

wxObject* Parse(**const wxString&** *source*)

Proceeds parsing of the document. This is end-user method. You can simply call it when you need to obtain parsed output (which is parser-specific)

The method does these things:

1. calls *InitParser(source)* (p. 738)

2. calls *DoParsing* (p. 737)

3. calls *GetProduct* (p. 738)

4. calls *DoneParser* (p. 737)

5. returns value returned by GetProduct

You shouldn't use InitParser, DoParsing, GetProduct or DoneParser directly.

wxHtmlParser::PushTagHandler

void PushTagHandler(**wxHtmlTagHandler*** *handler*, **wxString** *tags*)

Forces the handler to handle additional tags (not returned by *GetSupportedTags* (p. 747)). The handler should already be added to this parser.

Parameters

handler

the handler

tags

List of tags (in same format as GetSupportedTags's return value). The parser will

redirect these tags to *handler* (until call to *PopTagHandler* (p. 740)).

Example

Imagine you want to parse following pseudo-html structure:

```
<myitems>
    <param name="one" value="1">
    <param name="two" value="2">
</myitems>

<execute>
    <param program="text.exe">
</execute>
```

It is obvious that you cannot use only one tag handler for <param> tag. Instead you must use context-sensitive handlers for <param> inside <myitems> and <param> inside <execute>.

This is the preferred solution:

```
TAG_HANDLER_BEGIN(MYITEM, "MYITEMS")
    TAG_HANDLER_PROC(tag)
    {
        // ...something...

        m_Parser -> PushTagHandler(this, "PARAM");
        ParseInner(tag);
        m_Parser -> PopTagHandler();

        // ...something...
    }
TAG_HANDLER_END(MYITEM)
```

wxHtmlParser::PopTagHandler

void PopTagHandler()

Restores parser's state before last call to *PushTagHandler* (p. 739).

wxHtmlParser::SetFS

void SetFS(wxFileSystem *fs)

Sets the virtual file system that will be used to request additional files. (For example <IMG> tag handler requests wxFSFile with the image data.)

wxHtmlParser::StopParsing

void StopParsing()

Call this function to interrupt parsing from a tag handler. No more tags will be parsed afterward. This function may only be called from*wxHtmlParser::Parse* (p. 739) or any

function called by it (i.e. from tag handlers).

wxHtmlPrintout

This class serves as printout class for HTML documents.

Derived from

wxPrintout (p. 1065)

Include files

<wx/html/htmprint.h>

wxHtmlPrintout::wxHtmlPrintout

 wxHtmlPrintout(**const wxString&** *title* = *"Printout"*)

Constructor.

wxHtmlPrintout::AddFilter

static void AddFilter(**wxHtmlFilter*** *filter*)

Adds a filter to the static list of filters for wxHtmlPrintout. See *wxHtmlFilter* (p. 721) for further information.

wxHtmlPrintout::SetFonts

void SetFonts(**wxString** *normal_face*, **wxString** *fixed_face*, **const int** **sizes* = *NULL*)

Sets fonts. See *wxHtmlWindow::SetFonts* (p. 756) for detailed description.

wxHtmlPrintout::SetFooter

void SetFooter(**const wxString&** *footer*, **int** *pg* = *wxPAGE_ALL*)

Sets page footer.

Parameters

footer

> HTML text to be used as footer. You can use macros in it:

> > • @PAGENUM@ is replaced by page number

- @PAGESCNT@ is replaced by total number of pages

pg

> one of wxPAGE_ODD, wxPAGE_EVEN and wxPAGE_ALL constants.

wxHtmlPrintout::SetHeader

void SetHeader(**const wxString&** *header*, **int** *pg* = *wxPAGE_ALL*)

Sets page header.

Parameters

header

> HTML text to be used as header. You can use macros in it:

- @PAGENUM@ is replaced by page number
- @PAGESCNT@ is replaced by total number of pages

pg

> one of wxPAGE_ODD, wxPAGE_EVEN and wxPAGE_ALL constants.

wxHtmlPrintout::SetHtmlFile

void SetHtmlFile(**const wxString&** *htmlfile*)

Prepare the class for printing this HTML **file**. The file may be located on any virtual file system or it may be normal file.

wxHtmlPrintout::SetHtmlText

void SetHtmlText(**const wxString&** *html*, **const wxString&** *basepath* = *wxEmptyString*, **bool** *isdir* = *true*)

Prepare the class for printing this HTML text.

Parameters

html

> HTML text. (NOT file!)

basepath

> base directory (html string would be stored there if it was in file). It is used to determine path for loading images, for example.

isdir

> false if basepath is filename, true if it is directory name (see *wxFileSystem* (p. 540) for detailed explanation)

wxHtmlPrintout::SetMargins

void SetMargins(**float** *top = 25.2*, **float** *bottom = 25.2*, **float** *left = 25.2*, **float** *right = 25.2*, **float** *spaces = 5*)

Sets margins in millimeters. Defaults to 1 inch for margins and 0.5cm for space between text and header and/or footer

wxHtmlTag

This class represents a single HTML tag. It is used by *tag handlers* (p. 1849).

Derived from

wxObject

Include files

<wx/html/htmltag.h>

wxHtmlTag::wxHtmlTag

 wxHtmlTag(**wxHtmlTag** **parent*, **const wxString&** *source*, **int** *pos*, **int** *end_pos*, **wxHtmlTagsCache*** *cache*, **wxHtmlEntitiesParser** **entParser*)

Constructor. You will probably never have to construct a wxHtmlTag object yourself. Feel free to ignore the constructor parameters. Have a look at src/html/htmlpars.cpp if you're interested in creating it.

wxHtmlTag::GetAllParams

const wxString& GetAllParams() **const**

Returns a string containing all parameters.

Example : tag contains `<FONT SIZE=+2 COLOR="#000000">`. Call to tag.GetAllParams() would return `SIZE=+2 COLOR="#000000"`.

wxHtmlTag::GetBeginPos

int GetBeginPos() **const**

Returns beginning position of the text *between* this tag and paired ending tag. See explanation (returned position is marked with '|'):

```
bla bla bla <MYTAG> bla bla internal text</MYTAG> bla bla
                    |
```

wxHtmlTag::GetEndPos1

int GetEndPos1() **const**

Returns ending position of the text *between* this tag and paired ending tag. See explanation (returned position is marked with '|'):

```
bla bla bla <MYTAG> bla bla internal text</MYTAG> bla bla
                                          |
```

wxHtmlTag::GetEndPos2

int GetEndPos2() **const**

Returns ending position 2 of the text *between* this tag and paired ending tag. See explanation (returned position is marked with '|'):

```
bla bla bla <MYTAG> bla bla internal text</MYTAG> bla bla
                                                  |
```

wxHtmlTag::GetName

wxString GetName() **const**

Returns tag's name. The name is always in uppercase and it doesn't contain '<' or '/' characters. (So the name of `<FONT SIZE=+2>` tag is "FONT" and name of `</table>` is "TABLE")

wxHtmlTag::GetParam

wxString GetParam(**const wxString&** *par*, **bool** *with_commas = false*) **const**

Returns the value of the parameter. You should check whether the parameter exists or not (use *HasParam* (p. 746)) first.

Parameters

par

    The parameter's name.

with_commas

true if you want to get commas as well. See example.

Example

```
...
/* you have wxHtmlTag variable tag which is equal to
   HTML tag <FONT SIZE=+2 COLOR="#0000FF"> */
dummy = tag.GetParam("SIZE");
   // dummy == "+2"
dummy = tag.GetParam("COLOR");
   // dummy == "#0000FF"
dummy = tag.GetParam("COLOR", true);
   // dummy == "\"#0000FF\"" -- see the difference!!
```

wxHtmlTag::GetParamAsColour

bool GetParamAsColour(**const wxString&** *par*, **wxColour ****clr*) **const**

Interprets tag parameter *par* as colour specification and saves its value into wxColour variable pointed by *clr*.

Returns true on success and false if *par* is not colour specification or if the tag has no such parameter.

wxHtmlTag::GetParamAsInt

bool GetParamAsInt(**const wxString&** *par*, **int ****value*) **const**

Interprets tag parameter *par* as an integer and saves its value into int variable pointed by *value*.

Returns true on success and false if *par* is not an integer or if the tag has no such parameter.

wxHtmlTag::HasEnding

bool HasEnding() **const**

Returns true if this tag is paired with ending tag, false otherwise.

See the example of HTML document:

```
<html><body>
Hello<p>
How are you?
<p align=center>This is centered...</p>
Oops<br>Oooops!
</body></html>
```

In this example tags HTML and BODY have ending tags, first P and BR doesn't have

ending tag while the second P has. The third P tag (which is ending itself) of course doesn't have ending tag.

wxHtmlTag::HasParam

bool HasParam(**const wxString&** *par*) **const**

Returns true if the tag has a parameter of the given name. Example : `<FONT SIZE=+2 COLOR="#FF00FF">` has two parameters named "SIZE" and "COLOR".

Parameters

par

> the parameter you're looking for.

wxHtmlTag::ScanParam

wxString ScanParam(**const wxString&** *par*, **const wxChar** **format*, **void** **value*) **const**

This method scans the given parameter. Usage is exactly the same as sscanf's usage except that you don't pass a string but a parameter name as the first argument and you can only retrieve one value (i.e. you can use only one "%" element in *format*).

Parameters

par

> The name of the tag you want to query

format

> scanf()-like format string.

value

> pointer to a variable to store the value in

wxHtmlTagHandler

Derived from

wxObject (p. 1012)

Include files

<wx/html/htmlpars.h>

See Also

wxHtmlTagHandler::m_Parser

wxHtmlParser* m_Parser

This attribute is used to access parent parser. It is protected so that it can't be accessed by user but can be accessed from derived classes.

wxHtmlTagHandler::wxHtmlTagHandler

wxHtmlTagHandler()

Constructor.

wxHtmlTagHandler::GetSupportedTags

virtual wxString GetSupportedTags()

Returns list of supported tags. The list is in uppercase and tags are delimited by ','.
Example : `"I,B,FONT,P"`

wxHtmlTagHandler::HandleTag

virtual bool HandleTag(const wxHtmlTag& *tag*)

This is the core method of each handler. It is called each time one of supported tags is detected. *tag* contains all necessary info (see *wxHtmlTag* (p. 743) for details).

Return value

true if *ParseInner* (p. 747) was called, false otherwise.

Example

```
bool MyHandler::HandleTag(const wxHtmlTag& tag)
{
    ...
    // change state of parser (e.g. set bold face)
    ParseInner(tag);
    ...
    // restore original state of parser
}
```

You shouldn't call ParseInner if the tag is not paired with an ending one.

wxHtmlTagHandler::ParseInner

void ParseInner(**const wxHtmlTag&** *tag*)

This method calls parser's *DoParsing* (p. 737) method for the string between this tag and the paired ending tag:

```
...<A HREF="x.htm">Hello, world!</A>...
```

In this example, a call to ParseInner (with *tag* pointing to A tag) will parse 'Hello, world!'.

wxHtmlTagHandler::SetParser

virtual void SetParser(**wxHtmlParser** **parser*)

Assigns *parser* to this handler. Each **instance** of handler is guaranteed to be called only from the parser.

wxHtmlTagsModule

This class provides easy way of filling wxHtmlWinParser's table of tag handlers. It is used almost exclusively together with the set of*TAGS_MODULE_ * macros* (p. 1849)

Derived from

wxModule (p. 980)

Include files

<wx/html/winpars.h>

See Also

Tag Handlers (p. 1849),*wxHtmlTagHandler* (p. 746),*wxHtmlWinTagHandler* (p. 764),

wxHtmlTagsModule::FillHandlersTable

virtual void FillHandlersTable(**wxHtmlWinParser** **parser*)

You must override this method. In most common case its body consists only of lines of the following type:

```
parser -> AddTagHandler(new MyHandler);
```

I recommend using the **TAGS_MODULE_*** macros.

Paremeters

parser

Pointer to the parser that requested tables filling.

wxHtmlWidgetCell

wxHtmlWidgetCell is a class that provides a connection between HTML cells and widgets (an object derived from wxWindow). You can use it to display things like forms, input boxes etc. in an HTML window.

wxHtmlWidgetCell takes care of resizing and moving window.

Derived from

wxHtmlCell (p. 705)

Include files

<wx/html/htmlcell.h>

wxHtmlWidgetCell::wxHtmlWidgetCell

 wxHtmlWidgetCell(**wxWindow*** *wnd*, **int** *w = 0*)

Constructor.

Parameters

wnd

> Connected window. It is parent window **must** be the wxHtmlWindow object within which it is displayed!

w

> Floating width. If non-zero width of *wnd* window is adjusted so that it is always *w* percents of parent container's width. (For example w = 100 means that the window will always have same width as parent container)

wxHtmlWindow

wxHtmlWindow is probably the only class you will directly use unless you want to do something special (like adding new tag handlers or MIME filters).

The purpose of this class is to display HTML pages (either local file or downloaded via HTTP protocol) in a window. The width of the window is constant - given in the constructor - and virtual height is changed dynamically depending on page size. Once the window is created you can set its content by calling *SetPage(text)* (p. 757),*LoadPage(filename)* (p. 753) or*LoadFile* (p. 752).

Note

wxHtmlWindow uses the *wxImage* (p. 779) class for displaying images. Don't forget to

initialize all image formats you need before loading any page! (See
wxInitAllImageHandlers (p. 1590) and*wxImage::AddHandler* (p. 783).)

Derived from

wxScrolledWindow (p. 1150)

Include files

<wx/html/htmlwin.h>

Window styles

wxHW_SCROLLBAR_NEVER   Never display scrollbars, not even when the page is larger than the window.

wxHW_SCROLLBAR_AUTO   Display scrollbars only if page's size exceeds window's size.

wxHW_NO_SELECTION   Don't allow the user to select text.

wxHtmlWindow::wxHtmlWindow

wxHtmlWindow()

Default constructor.

wxHtmlWindow(**wxWindow** **parent*, **wxWindowID** *id = -1*, **const wxPoint&** *pos = wxDefaultPosition*, **const wxSize&** *size = wxDefaultSize*, **long** *style = wxHW_DEFAULT_STYLE*, **const wxString&** *name = "htmlWindow"*)

Constructor. The parameters are the same as for the *wxScrolledWindow* (p. 1150) constructor.

Parameters

style

> Window style. See *wxHtmlWindow* (p. 749).

wxHtmlWindow::AddFilter

static void AddFilter(**wxHtmlFilter** **filter*)

Adds *input filter* (p. 1848) to the static list of available filters. These filters are present by default:

-    `text/html` MIME type

-    `image/*` MIME types

-    Plain Text filter (this filter is used if no other filter matches)

wxHtmlWindow::AppendToPage

bool AppendToPage(**const wxString&** *source*)

Appends HTML fragment to currently displayed text and refreshes the window.

Parameters

source

HTML code fragment

Return value

false if an error occurred, true otherwise.

wxHtmlWindow::GetInternalRepresentation

wxHtmlContainerCell* GetInternalRepresentation() **const**

Returns pointer to the top-level container.

See also: *Cells Overview* (p. 1848), *Printing Overview* (p. 1846)

wxHtmlWindow::GetOpenedAnchor

wxString GetOpenedAnchor()

Returns anchor within currently opened page (see *GetOpenedPage* (p. 751)). If no page is opened or if the displayed page wasn't produced by call to LoadPage, empty string is returned.

wxHtmlWindow::GetOpenedPage

wxString GetOpenedPage()

Returns full location of the opened page. If no page is opened or if the displayed page wasn't produced by call to LoadPage, empty string is returned.

wxHtmlWindow::GetOpenedPageTitle

wxString GetOpenedPageTitle()

Returns title of the opened page or wxEmptyString if current page does not contain `<TITLE>` tag.

wxHtmlWindow::GetRelatedFrame

wxFrame* GetRelatedFrame() **const**

Returns the related frame.

wxHtmlWindow::HistoryBack

bool HistoryBack()

Moves back to the previous page. (each page displayed using *LoadPage* (p. 753) is stored in history list.)

wxHtmlWindow::HistoryCanBack

bool HistoryCanBack()

Returns true if it is possible to go back in the history (i.e. HistoryBack() won't fail).

wxHtmlWindow::HistoryCanForward

bool HistoryCanForward()

Returns true if it is possible to go forward in the history (i.e. HistoryBack() won't fail).

wxHtmlWindow::HistoryClear

void HistoryClear()

Clears history.

wxHtmlWindow::HistoryForward

bool HistoryForward()

Moves to next page in history.

wxHtmlWindow::LoadFile

virtual bool LoadFile(**const wxFileName&** *filename*)

Loads HTML page from file and displays it.

Return value

false if an error occurred, true otherwise

See also

LoadPage (p. 753)

wxHtmlWindow::LoadPage

virtual bool LoadPage(**const wxString&** *location*)

Unlike SetPage this function first loads HTML page from *location* and then displays it. See example:

```
htmlwin->LoadPage("help/myproject/index.htm");
```

Parameters

location

> The address of document. See *wxFileSystem* (p. 540) for details on address format and behaviour of "opener".

Return value

false if an error occurred, true otherwise

See also

LoadFile (p. 752)

wxHtmlWindow::OnCellClicked

virtual void OnCellClicked(**wxHtmlCell** **cell*, **wxCoord** *x*, **wxCoord** *y*, **const wxMouseEvent&** *event*)

This method is called when a mouse button is clicked inside wxHtmlWindow. The default behaviour is to call *OnLinkClicked* (p. 754) if the cell contains a hypertext link.

Parameters

cell

> The cell inside which the mouse was clicked, always a simple (i.e. non container) cell

x, y

> The logical coordinates of the click point

event

> The mouse event containing other information about the click

wxHtmlWindow::OnCellMouseHover

virtual void OnCellMouseHover(**wxHtmlCell** **cell*, **wxCoord** *x*, **wxCoord** *y*)

This method is called when a mouse moves over an HTML cell.

Parameters

cell

> The cell inside which the mouse is currently, always a simple (i.e. non container) cell

x, y

> The logical coordinates of the click point

wxHtmlWindow::OnLinkClicked

virtual void OnLinkClicked(**const wxHtmlLinkInfo&** *link*)

Called when user clicks on hypertext link. Default behaviour is to call *LoadPage* (p. 753) and do nothing else.

Also see *wxHtmlLinkInfo* (p. 732).

wxHtmlWindow::OnOpeningURL

virtual wxHtmlOpeningStatus OnOpeningURL(**wxHtmlURLType** *type*,**const wxString&** *url*, **wxString ****redirect*)

Called when an URL is being opened (either when the user clicks on a link or an image is loaded). The URL will be opened only if OnOpeningURL returns `wxHTML_OPEN`. This method is called by*wxHtmlParser::OpenURL* (p. 738). You can override OnOpeningURL to selectively block some URLs (e.g. for security reasons) or to redirect them elsewhere. Default behaviour is to always return `wxHTML_OPEN`.

Parameters

type

> Indicates type of the resource. Is one of**wxHTML_URL_PAGE** Opening a HTML page.

wxHTML_URL_IMAGE	Opening an image.
wxHTML_URL_OTHER	Opening a resource that doesn't fall into any other category.

url

> URL being opened.

redirect

> Pointer to wxString variable that must be filled with an URL if OnOpeningURL returns `wxHTML_REDIRECT`.

Return value **wxHTML_OPEN**   Open the URL.

wxHTML_BLOCK   Deny access to the URL, *wxHtmlParser::OpenURL* (p. 738) will return NULL.

wxHTML_REDIRECT   Don't open *url*, redirect to another URL. OnOpeningURL must fill **redirect* with the new URL. OnOpeningURL will be called again on returned URL.

wxHtmlWindow::OnSetTitle

virtual void OnSetTitle(**const wxString&** *title*)

Called on parsing `<TITLE>` tag.

wxHtmlWindow::ReadCustomization

virtual void ReadCustomization(**wxConfigBase** **cfg*, **wxString** *path* = *wxEmptyString*)

This reads custom settings from wxConfig. It uses the path 'path' if given, otherwise it saves info into currently selected path. The values are stored in sub-path `wxHtmlWindow`

Read values: all things set by SetFonts, SetBorders.

Parameters

cfg

> wxConfig from which you want to read the configuration.

path

> Optional path in config tree. If not given current path is used.

wxHtmlWindow::SelectAll

void SelectAll()

Selects all text in the window.

See also

SelectLine (p. 756),*SelectWord* (p. 756)

wxHtmlWindow::SelectionToText

wxString SelectionToText()

Returns current selection as plain text. Returns empty string if no text is currently selected.

wxHtmlWindow::SelectLine

void SelectLine(**const wxPoint&** *pos*)

Selects the line of text that *pos* points at. Note that *pos*is relative to the top of displayed page, not to window's origin, use*CalcUnscrolledPosition* (p. 1153)to convert physical coordinate.

See also

SelectAll (p. 755),*SelectWord* (p. 756)

wxHtmlWindow::SelectWord

void SelectWord(**const wxPoint&** *pos*)

Selects the word at position *pos*. Note that *pos*is relative to the top of displayed page, not to window's origin, use*CalcUnscrolledPosition* (p. 1153)to convert physical coordinate.

See also

SelectAll (p. 755),*SelectLine* (p. 756)

wxHtmlWindow::SetBorders

void SetBorders(**int** *b*)

This function sets the space between border of window and HTML contents. See image:

PRESS F9 TO FORMAT PICTURE

Parameters

b

indentation from borders in pixels

wxHtmlWindow::SetFonts

void SetFonts(**wxString** *normal_face*, **wxString** *fixed_face*, **const int** **sizes = NULL*)

This function sets font sizes and faces.

Parameters

normal_face

This is face name for normal (i.e. non-fixed) font. It can be either empty string (then the default face is chosen) or platform-specific face name. Examples are "helvetica" under Unix or "Times New Roman" under Windows.

fixed_face

The same thing for fixed face (<TT>..</TT>)

sizes

This is an array of 7 items of *int* type. The values represent size of font with HTML size from -2 to +4 (<FONT SIZE=-2> to <FONT SIZE=+4>). Default sizes are used if *sizes*is NULL.

Defaults

Default font sizes are defined by constants wxHTML_FONT_SIZE_1, wxHTML_FONT_SIZE_2, ..., wxHTML_FONT_SIZE_7. Note that they differ among platforms. Default face names are empty strings.

wxHtmlWindow::SetPage

bool SetPage(**const wxString&** *source*)

Sets HTML page and display it. This won't **load** the page!! It will display the *source*. See example:

```
htmlwin -> SetPage("<html><body>Hello, world!</body></html>");
```

If you want to load a document from some location use *LoadPage* (p. 753) instead.

Parameters

source

The HTML document source to be displayed.

Return value

false if an error occurred, true otherwise.

wxHtmlWindow::SetRelatedFrame

void SetRelatedFrame(**wxFrame*** *frame*, **const wxString&** *format*)

Sets the frame in which page title will be displayed. *format* is format of frame title, e.g.

"HtmlHelp : %s". It must contain exactly one %s. This%s is substituted with HTML page title.

wxHtmlWindow::SetRelatedStatusBar

void SetRelatedStatusBar(**int** *bar*)

After calling *SetRelatedFrame* (p. 757), this sets statusbar slot where messages will be displayed. (Default is -1 = no messages.)

Parameters

bar

> statusbar slot number (0..n)

wxHtmlWindow::ToText

wxString ToText()

Returns content of currently displayed page as plain text.

wxHtmlWindow::WriteCustomization

virtual void WriteCustomization(**wxConfigBase** **cfg*, **wxString** *path* = *wxEmptyString*)

Saves custom settings into wxConfig. It uses the path 'path' if given, otherwise it saves info into currently selected path. Regardless of whether the path is given or not, the function creates sub-path `wxHtmlWindow`.

Saved values: all things set by SetFonts, SetBorders.

Parameters

cfg

> wxConfig to which you want to save the configuration.

path

> Optional path in config tree. If not given, the current path is used.

wxHtmlWinParser

This class is derived from *wxHtmlParser* (p. 736) and its main goal is to parse HTML input so that it can be displayed in*wxHtmlWindow* (p. 749). It uses a special *wxHtmlWinTagHandler* (p. 764).

Notes

The product of parsing is a wxHtmlCell (resp. wxHtmlContainer) object.

Derived from

wxHtmlParser (p. 736)

Include files

<wx/html/winpars.h>

See Also

Handlers overview (p. 1849)

wxHtmlWinParser::wxHtmlWinParser

wxHtmlWinParser()

wxHtmlWinParser(**wxHtmlWindow** **wnd*)

Constructor. Don't use the default one, use constructor with*wnd* parameter (*wnd* is pointer to associated *wxHtmlWindow* (p. 749))

wxHtmlWinParser::AddModule

static void AddModule(**wxHtmlTagsModule** **module*)

Adds *module* (p. 1849) to the list of wxHtmlWinParser tag handler.

wxHtmlWinParser::CloseContainer

wxHtmlContainerCell* CloseContainer()

Closes the container, sets actual container to the parent one and returns pointer to it (see *Overview* (p. 1848)).

wxHtmlWinParser::CreateCurrentFont

virtual wxFont* CreateCurrentFont()

Creates font based on current setting (see *SetFontSize* (p. 764), *SetFontBold* (p. 763), *SetFontItalic* (p. 763), *SetFontFixed* (p. 763), *SetFontUnderlined* (p. 764)) and returns pointer to it. If the font was already created only a pointer is returned.

wxHtmlWinParser::GetActualColor

const wxColour& GetActualColor() const

Returns actual text colour.

wxHtmlWinParser::GetAlign

int GetAlign() const

Returns default horizontal alignment.

wxHtmlWinParser::GetCharHeight

int GetCharHeight() const

Returns (average) char height in standard font. It is used as DC-independent metrics.

Note: This function doesn't return the *actual* height. If you want to know the height of the current font, call `GetDC -> GetCharHeight()`.

wxHtmlWinParser::GetCharWidth

int GetCharWidth() const

Returns average char width in standard font. It is used as DC-independent metrics.

Note: This function doesn't return the *actual* width. If you want to know the height of the current font, call `GetDC -> GetCharWidth()`

wxHtmlWinParser::GetContainer

wxHtmlContainerCell* GetContainer() const

Returns pointer to the currently opened container (see *Overview* (p. 1848)). Common use:

```
m_WParser -> GetContainer() -> InsertCell(new ...);
```

wxHtmlWinParser::GetDC

wxDC* GetDC()

Returns pointer to the DC used during parsing.

wxHtmlWinParser::GetEncodingConverter

wxEncodingConverter * GetEncodingConverter() const

Returns *wxEncodingConverter* (p. 478) class used to do conversion between *input encoding* (p. 761) and *output encoding* (p. 762).

wxHtmlWinParser::GetFontBold

int GetFontBold() const

Returns true if actual font is bold, false otherwise.

wxHtmlWinParser::GetFontFace

wxString GetFontFace() const

Returns actual font face name.

wxHtmlWinParser::GetFontFixed

int GetFontFixed() const

Returns true if actual font is fixed face, false otherwise.

wxHtmlWinParser::GetFontItalic

int GetFontItalic() const

Returns true if actual font is italic, false otherwise.

wxHtmlWinParser::GetFontSize

int GetFontSize() const

Returns actual font size (HTML size varies from -2 to +4)

wxHtmlWinParser::GetFontUnderlined

int GetFontUnderlined() const

Returns true if actual font is underlined, false otherwise.

wxHtmlWinParser::GetInputEncoding

wxFontEncoding GetInputEncoding() const

Returns input encoding.

wxHtmlWinParser::GetLink

const wxHtmlLinkInfo& GetLink() const

Returns actual hypertext link. (This value has a non-empty *Href* (p. 733) string if the parser is between `<A>` and `</A>` tags, wxEmptyString otherwise.)

wxHtmlWinParser::GetLinkColor

const wxColour& GetLinkColor() const

Returns the colour of hypertext link text.

wxHtmlWinParser::GetOutputEncoding

wxFontEncoding GetOutputEncoding() const

Returns output encoding, i.e. closest match to document's input encoding that is supported by operating system.

wxHtmlWinParser::GetWindow

wxHtmlWindow* GetWindow()

Returns associated window (wxHtmlWindow). This may be NULL! (You should always test if it is non-NULL. For example `TITLE` handler sets window title only if some window is associated, otherwise it does nothing)

wxHtmlWinParser::OpenContainer

wxHtmlContainerCell* OpenContainer()

Opens new container and returns pointer to it (see *Overview* (p. 1848)).

wxHtmlWinParser::SetActualColor

void SetActualColor(const wxColour& *clr*)

Sets actual text colour. Note: this DOESN'T change the colour! You must create *wxHtmlColourCell* (p. 710) yourself.

wxHtmlWinParser::SetAlign

void SetAlign(**int** *a*)

Sets default horizontal alignment (see *wxHtmlContainerCell::SetAlignHor* (p. 713).) Alignment of newly opened container is set to this value.

wxHtmlWinParser::SetContainer

wxHtmlContainerCell* SetContainer(**wxHtmlContainerCell ****c*)

Allows you to directly set opened container. This is not recommended - you should use OpenContainer wherever possible.

wxHtmlWinParser::SetDC

virtual void SetDC(**wxDC** **dc*, **double** *pixel_scale = 1.0*)

Sets the DC. This must be called before *Parse* (p. 739)!*pixel_scale* can be used when rendering to high-resolution DCs (e.g. printer) to adjust size of pixel metrics. (Many dimensions in HTML are given in pixels -- e.g. image sizes. 300x300 image would be only one inch wide on typical printer. With pixel_scale = 3.0 it would be 3 inches.)

wxHtmlWinParser::SetFontBold

void SetFontBold(**int** *x*)

Sets bold flag of actualfont. *x* is either true of false.

wxHtmlWinParser::SetFontFace

void SetFontFace(**const wxString&** *face*)

Sets current font face to *face*. This affects either fixed size font or proportional, depending on context (whether the parser is inside <TT> tag or not).

wxHtmlWinParser::SetFontFixed

void SetFontFixed(**int** *x*)

Sets fixed face flag of actualfont. *x* is either true of false.

wxHtmlWinParser::SetFontItalic

void SetFontItalic(**int** *x*)

Sets italic flag of actualfont. *x* is either true of false.

wxHtmlWinParser::SetFontSize

void SetFontSize(**int** *s*)

Sets actual font size (HTML size varies from 1 to 7)

wxHtmlWinParser::SetFontUnderlined

void SetFontUnderlined(**int** *x*)

Sets underlined flag of actualfont. *x* is either true of false.

wxHtmlWinParser::SetFonts

void SetFonts(**wxString** *normal_face*, **wxString** *fixed_face*, **const int** **sizes = NULL*)

Sets fonts. See *wxHtmlWindow::SetFonts* (p. 756) for detailed description.

wxHtmlWinParser::SetInputEncoding

void SetInputEncoding(**wxFontEncoding** *enc*)

Sets input encoding. The parser uses this information to build conversion tables from document's encoding to some encoding supported by operating system.

wxHtmlWinParser::SetLink

void SetLink(**const wxHtmlLinkInfo&** *link*)

Sets actual hypertext link. Empty link is represented by *wxHtmlLinkInfo* (p. 732) with *Href* equal to wxEmptyString.

wxHtmlWinParser::SetLinkColor

void SetLinkColor(**const wxColour&** *clr*)

Sets colour of hypertext link.

wxHtmlWinTagHandler

This is basically wxHtmlTagHandler except that it is extended with protected member m_WParser pointing to the wxHtmlWinParser object (value of this member is identical to wxHtmlParser's m_Parser).

Derived from

wxHtmlTagHandler (p. 746)

Include files

<wx/html/winpars.h>

wxHtmlWinTagHandler::m_WParser

wxHtmlWinParser* m_WParser

Value of this attribute is identical to value of m_Parser. The only different is that m_WParser points to wxHtmlWinParser object while m_Parser points to wxHtmlParser object. (The same object, but overcast.)

wxHTTP

Derived from

wxProtocol (p. 1083)

Include files

<wx/protocol/http.h>

See also

wxSocketBase (p. 1203), *wxURL* (p. 1459)

wxHTTP::GetResponse

int GetResponse() **const**

Returns the HTTP response code returned by the server. Please refer to the RFC 2616 for the list of the responses.

wxHTTP::GetInputStream

wxInputStream * GetInputStream(**const wxString&** *path*)

Creates a new input stream on the specified path. You can use all except the seek functionality of wxStream. Seek isn't available on all streams. For example, http or ftp streams doesn't deal with it. Other functions like Tell and SeekI for this sort of stream. You will be notified when the EOF is reached by an error.

Note

You can know the size of the file you are getting using *wxStreamBase::GetSize()* (p.

1276). But there is a limitation: as HTTP servers aren't obliged to pass the size of the file, in some case, you will be returned 0xffffffff by GetSize(). In these cases, you should use the value returned by *wxInputStream::LastRead()* (p. 815): this value will be 0 when the stream is finished.

Return value

Returns the initialized stream. You will have to delete it yourself once you don't use it anymore. The destructor closes the network connection. The next time you will try to get a file the network connection will have to be reestablished: but you don't have to take care of this wxHTTP reestablishes it automatically.

See also

wxInputStream (p. 814)

wxHTTP::SetHeader

void SetHeader(**const wxString&** *header*, **const wxString&** *h_data*)

It sets data of a field to be sent during the next request to the HTTP server. The field name is specified by *header* and the content by *h_data*. This is a low level function and it assumes that you know what you are doing.

wxHTTP::GetHeader

wxString GetHeader(**const wxString&** *header*)

Returns the data attached with a field whose name is specified by *header*. If the field doesn't exist, it will return an empty string and not a NULL string.

Note

The header is not case-sensitive: I mean that "CONTENT-TYPE" and "content-type" represent the same header.

wxIcon

An icon is a small rectangular bitmap usually used for denoting a minimized application. It differs from a wxBitmap in always having a mask associated with it for transparent drawing. On some platforms, icons and bitmaps are implemented identically, since there is no real distinction between a wxBitmap with a mask and an icon; and there is no specific icon format on some platforms (X-based applications usually standardize on XPMs for small bitmaps and icons). However, some platforms (such as Windows) make the distinction, so a separate class is provided.

Derived from

wxBitmap (p. 79)

wxGDIObject (p. 607)
wxObject (p. 1012)

Include files

<wx/icon.h>

Predefined objects

Objects:

wxNullIcon

Remarks

It is usually desirable to associate a pertinent icon with a frame. Icons can also be used for other purposes, for example with *wxTreeCtrl* (p. 1424) and *wxListCtrl* (p. 852).

Icons have different formats on different platforms. Therefore, separate icons will usually be created for the different environments. Platform-specific methods for creating a **wxIcon** structure are catered for, and this is an occasion where conditional compilation will probably be required.

Note that a new icon must be created for every time the icon is to be used for a new window. In Windows, the icon will not be reloaded if it has already been used. An icon allocated to a frame will be deleted when the frame is deleted.

For more information please see *Bitmap and icon overview* (p. 1788).

See also

Bitmap and icon overview (p. 1788), *supported bitmap file formats* (p. 1789), *wxDC::DrawIcon* (p. 371), *wxCursor* (p. 226)

wxIcon::wxIcon

wxIcon()

Default constructor.

wxIcon(**const wxIcon&** *icon*)

Copy constructor.

wxIcon(**void*** *data*, **int** *type*, **int** *width*, **int** *height*, **int** *depth = -1*)

Creates an icon from the given data, which can be of arbitrary type.

wxIcon(**const char** *bits[]*, **int** *width*, **int** *height*
 int *depth = 1*)

Creates an icon from an array of bits.

wxIcon(**int** *width*, **int** *height*, **int** *depth = -1*)

Creates a new icon.

wxIcon(**char**** *bits*)

wxIcon(**const char**** *bits*)

Creates an icon from XPM data.

wxIcon(**const wxString&** *name*, **wxBitmapType** *type*, **int** *desiredWidth = -1*, **int** *desiredHeight = -1*)

Loads an icon from a file or resource.

wxIcon(**const wxIconLocation&** *loc*)

Loads an icon from the specified *location* (p. 775).

Parameters

bits

Specifies an array of pixel values.

width

Specifies the width of the icon.

height

Specifies the height of the icon.

desiredWidth

Specifies the desired width of the icon. This parameter only has an effect in Windows (32-bit) where icon resources can contain several icons of different sizes.

desiredWidth

Specifies the desired height of the icon. This parameter only has an effect in Windows (32-bit) where icon resources can contain several icons of different sizes.

depth

Specifies the depth of the icon. If this is omitted, the display depth of the screen is used.

name

This can refer to a resource name under MS Windows, or a filename under MS Windows and X. Its meaning is determined by the *flags* parameter.

loc

The object describing the location of the native icon, see *wxIconLocation* (p. 775).

type

May be one of the following:

wxBITMAP_TYPE_ICO   Load a Windows icon file.

wxBITMAP_TYPE_ICO_RESOURCE Load a Windows icon from the resource
database.

wxBITMAP_TYPE_GIF   Load a GIF bitmap file.

wxBITMAP_TYPE_XBM   Load an X bitmap file.

wxBITMAP_TYPE_XPM   Load an XPM bitmap file.

The validity of these flags depends on the platform and wxWidgets configuration. If all possible wxWidgets settings are used, the Windows platform supports ICO file, ICO resource, XPM data, and XPM file. Under wxGTK, the available formats are BMP file, XPM data, XPM file, and PNG file. Under wxMotif, the available formats are XBM data, XBM file, XPM data, XPM file.

Remarks

The first form constructs an icon object with no data; an assignment or another member function such as Create or LoadFile must be called subsequently.

The second and third forms provide copy constructors. Note that these do not copy the icon data, but instead a pointer to the data, keeping a reference count. They are therefore very efficient operations.

The fourth form constructs an icon from data whose type and value depends on the value of the *type* argument.

The fifth form constructs a (usually monochrome) icon from an array of pixel values, under both X and Windows.

The sixth form constructs a new icon.

The seventh form constructs an icon from pixmap (XPM) data, if wxWidgets has been configured to incorporate this feature.

To use this constructor, you must first include an XPM file. For example, assuming that the file `mybitmap.xpm` contains an XPM array of character pointers called mybitmap:

```
#include "mybitmap.xpm"

...

wxIcon *icon = new wxIcon(mybitmap);
```

A macro, wxICON, is available which creates an icon using an XPM on the appropriate platform, or an icon resource on Windows.

```
wxIcon icon(wxICON(mondrian));

// Equivalent to:
```

```
#if defined(__WXGTK__) || defined(__WXMOTIF__)
wxIcon icon(mondrian_xpm);
#endif

#if defined(__WXMSW__)
wxIcon icon("mondrian");
#endif
```

The eighth form constructs an icon from a file or resource. *name* can refer to a resource name under MS Windows, or a filename under MS Windows and X.

Under Windows, *type* defaults to wxBITMAP_TYPE_ICO_RESOURCE. Under X, *type* defaults to wxBITMAP_TYPE_XPM.

See also

wxIcon::CopyFromBitmap

void CopyFromBitmap(**const wxBitmap&** *bmp*)

Copies *bmp* bitmap to this icon. Under MS Windows the bitmap must have mask colour set.

wxIcon::LoadFile (p. 771)

wxPerl note: Constructors supported by wxPerl are:

- ::Icon->new(width, height, depth = -1)

- ::Icon->new(name, type, desiredWidth = -1, desiredHeight = -1)

- ::Icon->newFromBits(bits, width, height, depth = 1)

- ::Icon->newFromXPM(data)

wxIcon::~wxIcon

~wxIcon()

Destroys the wxIcon object and possibly the underlying icon data. Because reference counting is used, the icon may not actually be destroyed at this point - only when the reference count is zero will the data be deleted.

If the application omits to delete the icon explicitly, the icon will be destroyed automatically by wxWidgets when the application exits.

Do not delete an icon that is selected into a memory device context.

wxIcon::GetDepth

int GetDepth() const

Gets the colour depth of the icon. A value of 1 indicates a monochrome icon.

wxIcon::GetHeight

int GetHeight() const

Gets the height of the icon in pixels.

wxIcon::GetWidth

int GetWidth() const

Gets the width of the icon in pixels.

See also

wxIcon::GetHeight (p. 771)

wxIcon::LoadFile

bool LoadFile(const wxString& *name*, wxBitmapType *type*)

Loads an icon from a file or resource.

Parameters

name

> Either a filename or a Windows resource name. The meaning of *name* is
> determined by the *type* parameter.

type

> One of the following values:

> **wxBITMAP_TYPE_ICO** Load a Windows icon file.

> **wxBITMAP_TYPE_ICO_RESOURCE** Load a Windows icon from the resource
> database.

> **wxBITMAP_TYPE_GIF** Load a GIF bitmap file.

> **wxBITMAP_TYPE_XBM** Load an X bitmap file.

> **wxBITMAP_TYPE_XPM** Load an XPM bitmap file.

> The validity of these flags depends on the platform and wxWidgets configuration.

Return value

true if the operation succeeded, false otherwise.

See also

wxIcon::wxIcon (p. 767)

wxIcon::Ok

bool Ok() **const**

Returns true if icon data is present.

wxIcon::SetDepth

void SetDepth(**int** *depth*)

Sets the depth member (does not affect the icon data).

Parameters

depth

    Icon depth.

wxIcon::SetHeight

void SetHeight(**int** *height*)

Sets the height member (does not affect the icon data).

Parameters

height

    Icon height in pixels.

wxIcon::SetWidth

void SetWidth(**int** *width*)

Sets the width member (does not affect the icon data).

Parameters

width

    Icon width in pixels.

wxIcon::operator =

wxIcon& operator =(**const wxIcon&** *icon*)

Assignment operator. This operator does not copy any data, but instead passes a pointer to the data in *icon* and increments a reference counter. It is a fast operation.

Parameters

icon

> Icon to assign.

Return value

Returns 'this' object.

wxIcon::operator ==

bool operator ==(**const wxIcon&** *icon*)

Equality operator. This operator tests whether the internal data pointers are equal (a fast test).

Parameters

icon

> Icon to compare with 'this'

Return value

Returns true if the icons were effectively equal, false otherwise.

wxIcon::operator !=

bool operator !=(**const wxIcon&** *icon*)

Inequality operator. This operator tests whether the internal data pointers are unequal (a fast test).

Parameters

icon

> Icon to compare with 'this'

Return value

Returns true if the icons were unequal, false otherwise.

wxIconBundle

This class contains multiple copies of an icon in different sizes, see also *wxDialog::SetIcons* (p. 414) and*wxTopLevelWindow::SetIcons* (p. 1422).

Derived from

No base class

wxIconBundle::wxIconBundle

wxIconBundle()

Default constructor.

wxIconBundle(**const wxString&** *file*, **long** *type*)

Initializes the bundle with the icon(s) found in the file.

wxIconBundle(**const wxIcon&** *icon*)

Initializes the bundle with a single icon.

wxIconBundle(**const wxIconBundle&** *ic*)

Copy constructor.

wxIconBundle::~wxIconBundle

~wxIconBundle()

Destructor.

wxIconBundle::AddIcon

void AddIcon(**const wxString&** *file*, **long** *type*)

Adds all the icons contained in the file to the bundle; if the collection already contains icons with the same width and height, they are replaced by the new ones.

void AddIcon(**const wxIcon&** *icon*)

Adds the icon to the collection; if the collection already contains an icon with the same width and height, it is replaced by the new one.

wxIconBundle::GetIcon

const wxIcon& GetIcon(**const wxSize&** *size*) **const**

Returns the icon with the given size; if no such icon exists, returns the icon with size wxSYS_ICON_X/wxSYS_ICON_Y; if no such icon exists, returns the first icon in the bundle. If size = wxSize(-1, -1), returns the icon with size wxSYS_ICON_X/wxSYS_ICON_Y.

const wxIcon& GetIcon(**wxCoord** *size = -1*) **const**

Same as GetIcon(wxSize(size, size)).

wxIconBundle::operator=

const wxIconBundle& operator=(**const wxIconBundle&** *ic*)

Assignment operator.

wxIconLocation

wxIconLocation is a tiny class describing the location of an (external, i.e. not embedded into the application resources) icon. For most platforms it simply contains the file name but under some others (notably Windows) the same file may contain multiple icons and so this class also stores the index of the icon inside the file.

In any case, its details should be of no interest to the application code and most of them are not even documented here (on purpose) as it is only meant to be used as an opaque class: the application may get the object of this class from somewhere and the only reasonable thing to do with it later is to create a *wxIcon* (p. 766) from it.

Derived from

None.

Include files

<wx/iconloc.h>

See also

wxIcon (p. 766), *wxFileType::GetIcon* (p. 547)

wxIconLocation::IsOk

bool IsOk() **const**

Returns `true` if the object is valid, i.e. was properly initialized, and `false` otherwise.

wxIconizeEvent

An event being sent when the frame is iconized (minimized) or restored.

Currently only wxMSW and wxGTK generate such events.

Derived from

wxEvent (p. 483)
wxObject (p. 1012)

Include files

<wx/event.h>

Event table macros

To process an iconize event, use this event handler macro to direct input to a member function that takes a wxIconizeEvent argument.

EVT_ICONIZE(func)   Process a wxEVT_ICONIZE event.

See also
Event handling overview (p. 1751), *wxTopLevelWindow::Iconize* (p. 1420),
wxTopLevelWindow::IsIconized (p. 1420)

wxIconizeEvent::wxIconizeEvent

wxIconizeEvent(**int** *id = 0*, **bool** *iconized = true*)

Constructor.

wxIconizeEvent::Iconized

bool Iconized() **const**

Returns `true` if the frame has been iconized, `false` if it has been restored.

wxIdleEvent

This class is used for idle events, which are generated when the system is idle.

By default, idle events are sent to all windows. If this is causing a significant overhead in your application, you can call *wxIdleEvent::SetMode* (p. 778) with the value wxIDLE_PROCESS_SPECIFIED, and set the wxWS_EX_PROCESS_IDLE extra window style for every window which should receive idle events.

The function *wxWindow::OnInternalIdle* (p. 1520) is also provided for internal purposes,

and cannot be disabled. wxUpdateUIEvents are sent from OnInternalIdle.

Derived from

wxEvent (p. 483)
wxObject (p. 1012)

Include files

<wx/event.h>

Event table macros

To process an idle event, use this event handler macro to direct input to a member function that takes a wxIdleEvent argument.

EVT_IDLE(func)   Process a wxEVT_IDLE event.

Remarks

Idle events can be caught by the wxApp class, or by top-level window classes.

See also

Event handling overview (p. 1751), *wxUpdateUIEvent* (p. 1448),
wxWindow::OnInternalIdle (p. 1520)

wxIdleEvent::wxIdleEvent

 wxIdleEvent()

Constructor.

wxIdleEvent::CanSend

static bool CanSend(**wxWindow*** *window*)

Returns `true` if it is appropriate to send idle events to this window.

This function looks at the mode used (see *wxIdleEvent::SetMode* (p. 778)), and the wxWS_EX_PROCESS_IDLE style in *window* to determine whether idle events should be sent to this window now. By default this will always return `true` because the update mode is initially wxIDLE_PROCESS_ALL. You can change the mode to only send idle events to windows with the wxWS_EX_PROCESS_IDLE extra window style set.

See also

wxIdleEvent::SetMode (p. 778)

wxIdleEvent::GetMode

static wxIdleMode GetMode()

Static function returning a value specifying how wxWidgets will send idle events: to all windows, or only to those which specify that they will process the events.

See *wxIdleEvent::SetMode* (p. 778).

wxIdleEvent::RequestMore

void RequestMore(bool *needMore = true*)

Tells wxWidgets that more processing is required. This function can be called by an OnIdle handler for a window or window event handler to indicate that wxApp::OnIdle should forward the OnIdle event once more to the application windows. If no window calls this function during OnIdle, then the application will remain in a passive event loop (not calling OnIdle) until a new event is posted to the application by the windowing system.

See also

wxIdleEvent::MoreRequested (p. 778)

wxIdleEvent::MoreRequested

bool MoreRequested() const

Returns true if the OnIdle function processing this event requested more processing time.

See also

wxIdleEvent::RequestMore (p. 778)

wxIdleEvent::SetMode

static void SetMode(wxIdleMode *mode*)

Static function for specifying how wxWidgets will send idle events: to all windows, or only to those which specify that they will process the events.

mode can be one of the following values. The default is wxIDLE_PROCESS_ALL.

```
enum wxIdleMode
{
        // Send idle events to all windows
    wxIDLE_PROCESS_ALL,

        // Send idle events to windows that have
        // the wxWS_EX_PROCESS_IDLE flag specified
    wxIDLE_PROCESS_SPECIFIED
};
```

wxImage

This class encapsulates a platform-independent image. An image can be created from data, or using *wxBitmap::ConvertToImage* (p. 84). An image can be loaded from a file in a variety of formats, and is extensible to new formats via image format handlers. Functions are available to set and get image bits, so it can be used for basic image manipulation.

A wxImage cannot (currently) be drawn directly to a *wxDC* (p. 365). Instead, a platform-specific *wxBitmap* (p. 79) object must be created from it using the *wxBitmap::wxBitmap(wxImage,int depth)* (p. 80) constructor. This bitmap can then be drawn in a device context, using *wxDC::DrawBitmap* (p. 370).

One colour value of the image may be used as a mask colour which will lead to the automatic creation of a *wxMask* (p. 906) object associated to the bitmap object.

Alpha channel support

Starting from wxWidgets 2.5.0 wxImage supports alpha channel data, that is in addition to a byte for the red, green and blue colour components for each pixel it also stores a byte representing the pixel opacity. An alpha value of 0corresponds to a transparent pixel (null opacity) while a value of 255means that the pixel is 100% opaque.

Unlike RGB data, not all images have an alpha channel and before using *GetAlpha* (p. 787) you should check if this image contains an alpha channel with *HasAlpha* (p. 790). Note that currently only images loaded from PNG files with transparency information will have an alpha channel but alpha support will be added to the other formats as well (as well as support for saving images with alpha channel which also isn't implemented).

Available image handlers

The following image handlers are available. **wxBMPHandler** is always installed by default. To use other image formats, install the appropriate handler with *wxImage::AddHandler* (p. 783) or *wxInitAllImageHandlers* (p. 1590).

wxBMPHandler	For loading and saving, always installed.
wxPNGHandler	For loading (including alpha support) and saving.
wxJPEGHandler	For loading and saving.
wxGIFHandler	Only for loading, due to legal issues.
wxPCXHandler	For loading and saving (see below).
wxPNMHandler	For loading and saving (see below).
wxTIFFHandler	For loading and saving.
wxIFFHandler	For loading only.
wxXPMHandler	For loading and saving.

wxICOHandler	For loading and saving.
wxCURHandler	For loading and saving.
wxANIHandler	For loading only.

When saving in PCX format, **wxPCXHandler** will count the number of different colours in the image; if there are 256 or less colours, it will save as 8 bit, else it will save as 24 bit.

Loading PNMs only works for ASCII or raw RGB images. When saving in PNM format, **wxPNMHandler** will always save as raw RGB.

Derived from

wxObject (p. 1012)

Include files

<wx/image.h>

See also

wxBitmap (p. 79), *wxInitAllImageHandlers* (p. 1590)

wxImage::wxImage

wxImage()

Default constructor.

wxImage(**const wxImage&** *image*)

Copy constructor.

wxImage(**const wxBitmap&** *bitmap*)

(Deprecated form, use *wxBitmap::ConvertToImage* (p. 84)instead.) Constructs an image from a platform-dependent bitmap. This preserves mask information so that bitmaps and images can be converted back and forth without loss in that respect.

wxImage(**int** *width*, **int** *height*, **bool** *clear=true*)

Creates an image with the given width and height. If *clear* is true, the new image will be initialized to black. Otherwise, the image data will be uninitialized.

wxImage(**int** *width*, **int** *height*, **unsigned char*** *data*, **bool** *static_data* = `false`)

Creates an image from given data with the given width and height. If *static_data* is true, then wxImage will not delete the actual image data in its destructor, otherwise it will free it by calling*free()*.

wxImage(**const wxString&** *name*, **long** *type* = *wxBITMAP_TYPE_ANY*, **int** *index = -1*)

wxImage(**const wxString&** *name*, **const wxString&** *mimetype*, **int** *index = -1*)

Loads an image from a file.

wxImage(**wxInputStream&** *stream*, **long** *type = wxBITMAP_TYPE_ANY*, **int** *index = -1*)

wxImage(**wxInputStream&** *stream*, **const wxString&** *mimetype*, **int** *index = -1*)

Loads an image from an input stream.

wxImage(**const char**** *xpmData*)

Creates an image from XPM data.

Parameters

width

> Specifies the width of the image.

height

> Specifies the height of the image.

name

> Name of the file from which to load the image.

stream

> Opened input stream from which to load the image. Currently, the stream must support seeking.

type

> May be one of the following:

> | | |
> |---|---|
> | wxBITMAP_TYPE_BMP | Load a Windows bitmap file. |
> | wxBITMAP_TYPE_GIF | Load a GIF bitmap file. |
> | wxBITMAP_TYPE_JPEG | Load a JPEG bitmap file. |
> | wxBITMAP_TYPE_PNG | Load a PNG bitmap file. |
> | wxBITMAP_TYPE_PCX | Load a PCX bitmap file. |
> | wxBITMAP_TYPE_PNM | Load a PNM bitmap file. |
> | wxBITMAP_TYPE_TIF | Load a TIFF bitmap file. |
> | wxBITMAP_TYPE_XPM | Load a XPM bitmap file. |
> | wxBITMAP_TYPE_ICO | Load a Windows icon file (ICO). |

wxBITMAP_TYPE_CUR   Load a Windows cursor file (CUR).

wxBITMAP_TYPE_ANI   Load a Windows animated cursor file (ANI).

wxBITMAP_TYPE_ANY   Will try to autodetect the format.

mimetype

>   MIME type string (for example 'image/jpeg')

index

>   Index of the image to load in the case that the image file contains multiple images. This is only used by GIF, ICO and TIFF handlers. The default value (-1) means "choose the default image" and is interpreted as the first image (index=0) by the GIF and TIFF handler and as the largest and most colourful one by the ICO handler.

xpmData

>   A pointer to XPM image data.

Remarks

Depending on how wxWidgets has been configured, not all formats may be available.

Note: any handler other than BMP must be previously initialized with *wxImage::AddHandler* (p. 783) or *wxInitAllImageHandlers* (p. 1590).

```
Note: you can use GetOptionInt (p. 791) to get the  hotspot for loaded
cursor file:    int hotspot_x =
image.GetOptionInt(wxIMAGE_OPTION_CUR_HOTSPOT_X);
    int hotspot_y = image.GetOptionInt(wxIMAGE_OPTION_CUR_HOTSPOT_Y);
```

See also

wxImage::LoadFile (p. 793)

wxPython note: Constructors supported by wxPython are:

wxImage(name, flag)   Loads an image from a file

wxNullImage()   Create a null image (has no size or image data)

wxEmptyImage(width, height) Creates an empty image of the given size

wxImageFromMime(name, mimetype Creates an image from the given file of the given mimetype

wxImageFromBitmap(bitmap) Creates an image from a platform-dependent bitmap

wxPerl note: Constructors supported by wxPerl are:

•::Image->new(bitmap)

•::Image->new(icon)

•::Image->new(width, height)

•::Image->new(width, height, data)

•::Image->new(file, type, index)

•::Image->new(file, mimetype, index)

•::Image->new(stream, type, index)

•::Image->new(stream, mimetype, index)

wxImage::~wxImage

~wxImage()

Destructor.

wxImage::AddHandler

static void AddHandler(**wxImageHandler*** *handler*)

Adds a handler to the end of the static list of format handlers.

handler

A new image format handler object. There is usually only one instance of a given handler class in an application session.

See also

wxImageHandler (p. 802)

bool CanRead(**const wxString&** *filename*)

returns true if the current image handlers can read this file

wxPython note: In wxPython this static method is named `wxImage_AddHandler`.

wxImage::CleanUpHandlers

static void CleanUpHandlers()

Deletes all image handlers.

This function is called by wxWidgets on exit.

wxImage::ComputeHistogram

unsigned long ComputeHistogram(**wxImageHistogram&** *histogram*) **const**

Computes the histogram of the image. *histogram* is a reference to wxImageHistogram object. wxImageHistogram is a specialization of *wxHashMap* (p. 685) "template" and is defined as follows:

```
class WXDLLEXPORT wxImageHistogramEntry
{
public:
    wxImageHistogramEntry() : index(0), value(0) {}
    unsigned long index;
    unsigned long value;
};

WX_DECLARE_EXPORTED_HASH_MAP(unsigned long, wxImageHistogramEntry,
                             wxIntegerHash, wxIntegerEqual,
                             wxImageHistogram);
```

Return value

Returns number of colours in the histogram.

wxImage::ConvertAlphaToMask

bool ConvertAlphaToMask(**unsigned char** *threshold = 128*)

If the image has alpha channel, this method converts it to mask. All pixels with alpha value less than *threshold* are replaced with mask colour and the alpha channel is removed. Mask colour is chosen automatically using*FindFirstUnusedColour* (p. 785).

If the image image doesn't have alpha channel, ConvertAlphaToMask does nothing.

Return value

`false` if FindFirstUnusedColour returns `false`, `true` otherwise.

wxImage::ConvertToBitmap

wxBitmap ConvertToBitmap() **const**

Deprecated, use equivalent *wxBitmap constructor* (p. 80)(which takes wxImage and depth as its arguments) instead.

wxImage::ConvertToMono

wxImage ConvertToMono(**unsigned char** *r*, **unsigned char** *g*, **unsigned char** *b*) **const**

Returns monochromatic version of the image. The returned image has white colour where the original has *(r,g,b)* colour and black colour everywhere else.

wxImage::Copy

wxImage Copy() **const**

Returns an identical copy of the image.

wxImage::Create

bool Create(**int** *width*, **int** *height*, **bool** *clear=true*)

Creates a fresh image. If *clear* is true, the new image will be initialized to black. Otherwise, the image data will be uninitialized.

Parameters

width

> The width of the image in pixels.

height

> The height of the image in pixels.

Return value

true if the call succeeded, false otherwise.

wxImage::Destroy

void Destroy()

Destroys the image data.

wxImage::FindFirstUnusedColour

bool FindFirstUnusedColour(**unsigned char *** *r*, **unsigned char *** *g*, **unsigned char *** *b*, **unsigned char** *startR = 1*, **unsigned char** *startG = 0*, **unsigned char** *startB = 0*)

Parameters

r,g,b

> Pointers to variables to save the colour.

startR,startG,startB

> Initial values of the colour. Returned colour will have RGB values equal to or greater than these.

Finds the first colour that is never used in the image. The search begins at given initial colour and continues by increasing R, G and B components (in this order) by 1 until an

unused colour is found or the colour space exhausted.

Return value

Returns false if there is no unused colour left, true on success.

Notes

Note that this method involves computing the histogram, which is computationally intensive operation.

wxImage::FindHandler

static wxImageHandler* FindHandler(**const wxString&** *name*)

Finds the handler with the given name.

static wxImageHandler* FindHandler(**const wxString&** *extension*, **long** *imageType*)

Finds the handler associated with the given extension and type.

static wxImageHandler* FindHandler(**long** *imageType*)

Finds the handler associated with the given image type.

static wxImageHandler* FindHandlerMime(**const wxString&** *mimetype*)

Finds the handler associated with the given MIME type.

name

> The handler name.

extension

> The file extension, such as "bmp".

imageType

> The image type, such as wxBITMAP_TYPE_BMP.

mimetype

> MIME type.

Return value

A pointer to the handler if found, NULL otherwise.

See also

wxImageHandler (p. 802)

wxImage::GetImageExtWildcard

static wxString GetImageExtWildcard()

Iterates all registered wxImageHandler objects, and returns a string containing file extension masks suitable for passing to file open/save dialog boxes.

Return value

The format of the returned string is "(*.ext1;*.ext2)|*.ext1;*.ext2".

It is usually a good idea to prepend a description before passing the result to the dialog.

Example:

```
    wxFileDialog FileDlg( this, "Choose Image", ::wxGetWorkingDirectory(),
"", _("Image Files ") + wxImage::GetImageExtWildcard(), wxOPEN );
```

See also

wxImageHandler (p. 802)

wxImage::GetAlpha

unsigned char GetAlpha(**int** *x*, **int** *y*) **const**

Returns the alpha value for the given pixel. This function may only be called for the images with alpha channel, use *HasAlpha* (p. 790) to check for this.

The returned value is the *opacity* of the image, i.e. the value of 0corresponds to the transparent pixels while the value of 255 -- to the opaque ones.

unsigned char * GetAlpha() **const**

Returns pointer to the array storing the alpha values for this image. This pointer is NULL for the images without the alpha channel. If the image does have it, this pointer may be used to directly manipulate the alpha values which are stored as the *RGB* (p. 787) ones.

wxImage::GetBlue

unsigned char GetBlue(**int** *x*, **int** *y*) **const**

Returns the blue intensity at the given coordinate.

wxImage::GetData

unsigned char* GetData() **const**

Returns the image data as an array. This is most often used when doing direct image manipulation. The return value points to an array of characters in RGBRGBRGB... format in the top-to-bottom, left-to-right order, that is the first RGB triplet corresponds to

the pixel first pixel of the first row, the second one --- to the second pixel of the first row and so on until the end of the first row, with second row following after it and so on.

You should not delete the returned pointer nor pass it to*wxImage::SetData* (p. 799).

wxImage::GetGreen

unsigned char GetGreen(**int** *x*, **int** *y*) **const**

Returns the green intensity at the given coordinate.

wxImage::GetImageCount

static int GetImageCount(**const wxString&** *filename*, **long** *type =*
wxBITMAP_TYPE_ANY)

static int GetImageCount(**wxInputStream&** *stream*, **long** *type =*
wxBITMAP_TYPE_ANY)

If the image file contains more than one image and the image handler is capable of retrieving these individually, this function will return the number of available images.

name

> Name of the file to query.

stream

> Opened input stream with image data. Currently, the stream must support seeking.

type

> May be one of the following:

> | | |
> |---|---|
> | wxBITMAP_TYPE_BMP | Load a Windows bitmap file. |
> | wxBITMAP_TYPE_GIF | Load a GIF bitmap file. |
> | wxBITMAP_TYPE_JPEG | Load a JPEG bitmap file. |
> | wxBITMAP_TYPE_PNG | Load a PNG bitmap file. |
> | wxBITMAP_TYPE_PCX | Load a PCX bitmap file. |
> | wxBITMAP_TYPE_PNM | Load a PNM bitmap file. |
> | wxBITMAP_TYPE_TIF | Load a TIFF bitmap file. |
> | wxBITMAP_TYPE_XPM | Load a XPM bitmap file. |
> | wxBITMAP_TYPE_ICO | Load a Windows icon file (ICO). |
> | wxBITMAP_TYPE_CUR | Load a Windows cursor file (CUR). |

| wxBITMAP_TYPE_ANI | Load a Windows animated cursor file (ANI). |
| wxBITMAP_TYPE_ANY | Will try to autodetect the format. |

Return value

Number of available images. For most image handlers, this is 1 (exceptions are TIFF and ICO formats).

wxImage::GetHandlers

static wxList& GetHandlers()

Returns the static list of image format handlers.

See also

wxImageHandler (p. 802)

wxImage::GetHeight

int GetHeight() **const**

Gets the height of the image in pixels.

wxImage::GetMaskBlue

unsigned char GetMaskBlue() **const**

Gets the blue value of the mask colour.

wxImage::GetMaskGreen

unsigned char GetMaskGreen() **const**

Gets the green value of the mask colour.

wxImage::GetMaskRed

unsigned char GetMaskRed() **const**

Gets the red value of the mask colour.

wxImage::GetOrFindMaskColour

bool GetOrFindMaskColour(**unsigned char** **r*, **unsigned char** **g*, **unsigned char** **b*) **const**

Get the current mask colour or find a suitable unused colour that could be used as a mask colour. Returns `true` if the image currently has a mask.

wxImage::GetPalette

const wxPalette& GetPalette() const

Returns the palette associated with the image. Currently the palette is only used when converting to wxBitmap under Windows.

Eventually wxImage handlers will set the palette if one exists in the image file.

wxImage::GetRed

unsigned char GetRed(int *x*, int *y*) const

Returns the red intensity at the given coordinate.

wxImage::GetSubImage

wxImage GetSubImage(const wxRect& *rect*) const

Returns a sub image of the current one as long as the rect belongs entirely to the image.

wxImage::GetWidth

int GetWidth() const

Gets the width of the image in pixels.

See also

wxImage::GetHeight (p. 789)

wxImage::HasAlpha

bool HasAlpha() const

Returns true if this image has alpha channel, false otherwise.

See also

GetAlpha (p. 787), *SetAlpha* (p. 799)

wxImage::HasMask

bool HasMask() **const**

Returns true if there is a mask active, false otherwise.

wxImage::GetOption

wxString GetOption(**const wxString&** *name*) **const**

Gets a user-defined option. The function is case-insensitive to *name*.

For example, when saving as a JPEG file, the option **quality** is used, which is a number between 0 and 100 (0 is terrible, 100 is very good).

See also

wxImage::SetOption (p. 800), *wxImage::GetOptionInt* (p. 791), *wxImage::HasOption* (p. 791)

wxImage::GetOptionInt

int GetOptionInt(**const wxString&** *name*) **const**

Gets a user-defined option as an integer. The function is case-insensitive to *name*.

If the given option is not present, the function returns 0. Use *wxImage::HasOption* (p. 791) is 0 is a possibly valid value for the option.

Options for wxPNGHandlerwxIMAGE_OPTION_PNG_FORMAT   Format for saving a PNG file.

wxIMAGE_OPTION_PNG_BITDEPTH   Bit depth for every channel (R/G/B/A).

Supported values for wxIMAGE_OPTION_PNG_FORMAT:wxPNG_TYPE_COLOUR Stores RGB image.

wxPNG_TYPE_GREY   Stores grey image, converts from RGB.

wxPNG_TYPE_GREY_RED Stores grey image, uses red value as grey.

See also

wxImage::SetOption (p. 800), *wxImage::GetOption* (p. 791)

wxImage::HasOption

bool HasOption(**const wxString&** *name*) **const**

Returns true if the given option is present. The function is case-insensitive to *name*.

wxImage::InitAlpha

void InitAlpha()

Initializes the image alpha channel data. It is an error to call it if the image already has alpha data. If it doesn't, alpha data will be by default initialized to all pixels being fully opaque. But if the image has a a mask colour, all mask pixels will be completely transparent.

wxImage::InitStandardHandlers

static void InitStandardHandlers()

Internal use only. Adds standard image format handlers. It only install BMP for the time being, which is used by wxBitmap.

This function is called by wxWidgets on startup, and shouldn't be called by the user.

wxImage::InsertHandler

static void InsertHandler(**wxImageHandler*** *handler*)

Adds a handler at the start of the static list of format handlers.

handler

> A new image format handler object. There is usually only one instance of a given handler class in an application session.

wxImage::IsTransparent

bool IsTransparent(**int** *x*, **int** *y*, **unsigned char** *threshold = 128*) **const**

Returns `true` if the given pixel is transparent, i.e. either has the mask colour if this image has a mask or if this image has alpha channel and alpha value of this pixel is strictly less than *threshold*.

wxImage::LoadFile

bool LoadFile(**const wxString&** *name*, **long** *type* = *wxBITMAP_TYPE_ANY*, **int** *index* = *-1*)

bool LoadFile(**const wxString&** *name*, **const wxString&** *mimetype*, **int** *index* = *-1*)

Loads an image from a file. If no handler type is provided, the library will try to autodetect the format.

bool LoadFile(**wxInputStream&** *stream*, **long** *type*, **int** *index* = *-1*)

bool LoadFile(**wxInputStream&** *stream*, **const wxString&** *mimetype*, **int** *index* = *-1*)

Loads an image from an input stream.

Parameters

name

Name of the file from which to load the image.

stream

Opened input stream from which to load the image. Currently, the stream must support seeking.

type

One of the following values:

wxBITMAP_TYPE_BMP	Load a Windows image file.
wxBITMAP_TYPE_GIF	Load a GIF image file.
wxBITMAP_TYPE_JPEG	Load a JPEG image file.
wxBITMAP_TYPE_PCX	Load a PCX image file.
wxBITMAP_TYPE_PNG	Load a PNG image file.
wxBITMAP_TYPE_PNM	Load a PNM image file.
wxBITMAP_TYPE_TIF	Load a TIFF image file.
wxBITMAP_TYPE_XPM	Load a XPM image file.
wxBITMAP_TYPE_ICO	Load a Windows icon file (ICO).
wxBITMAP_TYPE_CUR	Load a Windows cursor file (CUR).
wxBITMAP_TYPE_ANI	Load a Windows animated cursor file (ANI).
wxBITMAP_TYPE_ANY	Will try to autodetect the format.

mimetype

>   MIME type string (for example 'image/jpeg')

index

>   Index of the image to load in the case that the image file contains multiple images. This is only used by GIF, ICO and TIFF handlers. The default value (-1) means "choose the default image" and is interpreted as the first image (index=0) by the GIF and TIFF handler and as the largest and most colourful one by the ICO handler.

Remarks

Depending on how wxWidgets has been configured, not all formats may be available.

```
Note: you can use GetOptionInt (p. 791) to get the  hotspot for loaded
cursor file:    int hotspot_x =
image.GetOptionInt(wxIMAGE_OPTION_CUR_HOTSPOT_X);
     int hotspot_y = image.GetOptionInt(wxIMAGE_OPTION_CUR_HOTSPOT_Y);
```

Return value

true if the operation succeeded, false otherwise. If the optional index parameter is out of range, false is returned and a call to wxLogError() takes place.

See also

wxImage::SaveFile (p. 796)

wxPython note: In place of a single overloaded method name, wxPython implements the following methods:

>   **LoadFile(filename, type)**   Loads an image of the given type from a file

>   **LoadMimeFile(filename, mimetype)**   Loads an image of the given mimetype from a file

wxPerl note: Methods supported by wxPerl are:

- >LoadFile(name, type)

- >LoadFile(name, mimetype)

wxImage::Ok

bool Ok() **const**

Returns true if image data is present.

wxImage::RemoveHandler

static bool RemoveHandler(**const wxString&** *name*)

Finds the handler with the given name, and removes it. The handler is not deleted.

name

> The handler name.

Return value

true if the handler was found and removed, false otherwise.

See also

wxImageHandler (p. 802)

wxImage::Mirror

wxImage Mirror(**bool** *horizontally = true*) **const**

Returns a mirrored copy of the image. The parameter *horizontally*indicates the orientation.

wxImage::Replace

void Replace(**unsigned char** *r1*, **unsigned char** *g1*, **unsigned char** *b1*,**unsigned char** *r2*, **unsigned char** *g2*, **unsigned char** *b2*)

Replaces the colour specified by *r1,g1,b1* by the colour *r2,g2,b2*.

wxImage::Rescale

wxImage & Rescale(**int** *width*, **int** *height*)

Changes the size of the image in-place by scaling it: after a call to this function, the image will have the given width and height.

Returns the (modified) image itself.

See also

Scale (p. 798)

wxImage::Resize

wxImage & Resize(**const wxSize&** *size*, **const wxPoint&** *pos*, **int** *red = -1*, **int** *green = -1*, **int** *blue = -1*)

Changes the size of the image in-place without scaling it by adding either a border with

the given colour or cropping as necessary. The image is pasted into a new image with the given *size* and background colour at the position *pos* relative to the upper left of the new image. If *red = green = blue = -1* then use either the current mask colour if set or find, use, and set a suitable mask colour for any newly exposed areas.

Returns the (modified) image itself.

See also

Size (p. 798)

wxImage::Rotate

wxImage Rotate(**double** *angle*, **const wxPoint&** *rotationCentre*, **bool** *interpolating =* *true*, **wxPoint*** *offsetAfterRotation = NULL*)

Rotates the image about the given point, by *angle* radians. Passing true to *interpolating* results in better image quality, but is slower. If the image has a mask, then the mask colour is used for the uncovered pixels in the rotated image background. Else, black (rgb 0, 0, 0) will be used.

Returns the rotated image, leaving this image intact.

wxImage::Rotate90

wxImage Rotate90(**bool** *clockwise = true*) **const**

Returns a copy of the image rotated 90 degrees in the direction indicated by *clockwise*.

wxImage::SaveFile

bool SaveFile(**const wxString&** *name*, **int** *type*) **const**

bool SaveFile(**const wxString&** *name*, **const wxString&** *mimetype*) **const**

Saves an image in the named file.

bool SaveFile(**const wxString&** *name*) **const**

Saves an image in the named file. File type is determined from the extension of the file name. Note that this function may fail if the extension is not recognized! You can use one of the forms above to save images to files with non-standard extensions.

bool SaveFile(**wxOutputStream&** *stream*, **int** *type*) **const**

bool SaveFile(**wxOutputStream&** *stream*, **const wxString&** *mimetype*) **const**

Saves an image in the given stream.

Parameters

name

> Name of the file to save the image to.

stream

> Opened output stream to save the image to.

type

> Currently these types can be used:

wxBITMAP_TYPE_BMP	Save a BMP image file.
wxBITMAP_TYPE_JPEG	Save a JPEG image file.
wxBITMAP_TYPE_PNG	Save a PNG image file.
wxBITMAP_TYPE_PCX	Save a PCX image file (tries to save as 8-bit if possible, falls back to 24-bit otherwise).
wxBITMAP_TYPE_PNM	Save a PNM image file (as raw RGB always).
wxBITMAP_TYPE_TIFF	Save a TIFF image file.
wxBITMAP_TYPE_XPM	Save a XPM image file.
wxBITMAP_TYPE_ICO	Save a Windows icon file (ICO) (the size may be up to 255 wide by 127 high. A single image is saved in 8 colors at the size supplied).
wxBITMAP_TYPE_CUR	Save a Windows cursor file (CUR).

mimetype

> MIME type.

Return value

true if the operation succeeded, false otherwise.

Remarks

Depending on how wxWidgets has been configured, not all formats may be available.

```
Note: you can use GetOptionInt (p. 791) to set the  hotspot before saving
an image into a cursor file (default hotspot is in  the centre of the
image):    image.SetOption(wxIMAGE_OPTION_CUR_HOTSPOT_X, hotspotX);
    image.SetOption(wxIMAGE_OPTION_CUR_HOTSPOT_Y, hotspotY);
```

See also

wxImage::LoadFile (p. 793)

wxPython note: In place of a single overloaded method name, wxPython implements

the following methods:

SaveFile(filename, type)   Saves the image using the given type to the named file

SaveMimeFile(filename, mimetype)   Saves the image using the given mimetype to the named file

wxPerl note: Methods supported by wxPerl are:

• >SaveFile(name, type)

• >SaveFile(name, mimetype)

wxImage::Scale

wxImage Scale(**int** *width*, **int** *height*) **const**

Returns a scaled version of the image. This is also useful for scaling bitmaps in general as the only other way to scale bitmaps is to blit a wxMemoryDC into another wxMemoryDC.

It may be mentioned that the GTK port uses this function internally to scale bitmaps when using mapping modes in wxDC.

Example:

```
// get the bitmap from somewhere
wxBitmap bmp = ...;

// rescale it to have size of 32*32
if ( bmp.GetWidth() != 32 || bmp.GetHeight() != 32 )
{
    wxImage image = bmp.ConvertToImage();
    bmp = wxBitmap(image.Scale(32, 32));

    // another possibility:
    image.Rescale(32, 32);
    bmp = image;
}
```

See also

Rescale (p. 795)

wxImage::Size

wxImage Size(**const wxSize&** *size*, **const wxPoint&** *pos*, **int** *red = -1*, **int** *green = -1*, **int** *blue = -1*) **const**

Returns a resized version of this image without scaling it by adding either a border with the given colour or cropping as necessary. The image is pasted into a new image with

the given *size* and background colour at the position *pos* relative to the upper left of the new image. If *red = green = blue = -1* then use either the current mask colour if set or find, use, and set a suitable mask colour for any newly exposed areas.

See also

Resize (p. 795)

wxImage::SetAlpha

void SetAlpha(**unsigned char ****alpha* = *NULL*,**bool** *static_data* = *false*)

This function is similar to *SetData* (p. 799) and has similar restrictions. The pointer passed to it may however be NULL in which case the function will allocate the alpha array internally -- this is useful to add alpha channel data to an image which doesn't have any. If the pointer is not NULL, it must have one byte for each image pixel and be allocated with malloc(). wxImage takes ownership of the pointer and will free it unless*static_data* parameter is set.to true -- in this case the caller should do it.

void SetAlpha(**int** *x*, **int** *y*, **unsigned char** *alpha*)

Sets the alpha value for the given pixel. This function should only be called if the image has alpha channel data, use *HasAlpha* (p. 790) to check for this.

wxImage::SetData

void SetData(**unsigned char****data*)

Sets the image data without performing checks. The data given must have the size (width*height*3) or results will be unexpected. Don't use this method if you aren't sure you know what you are doing.

The data must have been allocated with malloc(), **NOT** withoperator new.

After this call the pointer to the data is owned by the wxImage object, that will be responsible for deleting it. Do not pass to this function a pointer obtained through*wxImage::GetData* (p. 787).

wxImage::SetMask

void SetMask(**bool** *hasMask* = *true*)

Specifies whether there is a mask or not. The area of the mask is determined by the current mask colour.

wxImage::SetMaskColour

void SetMaskColour(**unsigned char** *red*, **unsigned char** *green*, **unsigned char** *blue*)

Sets the mask colour for this image (and tells the image to use the mask).

wxImage::SetMaskFromImage

bool SetMaskFromImage(**const wxImage&** *mask*, **unsigned char** *mr*, **unsigned char** *mg*, **unsigned char** *mb*)

Parameters

mask

> The mask image to extract mask shape from. Must have same dimensions as the image.

mr,mg,mb

> RGB value of pixels in *mask* that will be used to create the mask.

Sets image's mask so that the pixels that have RGB value of *mr,mg,mb* in *mask* will be masked in the image. This is done by first finding an unused colour in the image, setting this colour as the mask colour and then using this colour to draw all pixels in the image who corresponding pixel in *mask* has given RGB value.

Return value

Returns false if *mask* does not have same dimensions as the image or if there is no unused colour left. Returns true if the mask was successfully applied.

Notes

Note that this method involves computing the histogram, which is computationally intensive operation.

wxImage::SetOption

void SetOption(**const wxString&** *name*, **const wxString&** *value*)

void SetOption(**const wxString&** *name*, **int** *value*)

Sets a user-defined option. The function is case-insensitive to *name*.

For example, when saving as a JPEG file, the option **quality** is used, which is a number between 0 and 100 (0 is terrible, 100 is very good).

See also

wxImage::GetOption (p. 791), *wxImage::GetOptionInt* (p. 791), *wxImage::HasOption* (p. 791)

wxImage::SetPalette

void SetPalette(**const wxPalette&** *palette*)

Associates a palette with the image. The palette may be used when converting wxImage to wxBitmap (MSW only at present) or in file save operations (none as yet).

wxImage::SetRGB

void SetRGB(**int** *x*, **int** *y*, **unsigned char** *red*, **unsigned char** *green*, **unsigned char** *blue*)

Sets the pixel at the given coordinate. This routine performs bounds-checks for the coordinate so it can be considered a safe way to manipulate the data, but in some cases this might be too slow so that the data will have to be set directly. In that case you will have to get access to the image data using the *GetData* (p. 787) method.

wxImage::SetRGB

void SetRGB(**wxRect &** *rect*, **unsigned char** *red*, **unsigned char** *green*, **unsigned char** *blue*)

Sets the colour of the pixels within the given rectangle. This routine performs bounds-checks for the coordinate so it can be considered a safe way to manipulate the data.

wxImage::operator =

wxImage& operator =(**const wxImage&** *image*)

Assignment operator. This operator does not copy any data, but instead passes a pointer to the data in *image* and increments a reference counter. It is a fast operation.

Parameters

image

Image to assign.

Return value

Returns 'this' object.

wxImage::operator ==

bool operator ==(**const wxImage&** *image*) **const**

Equality operator. This operator tests whether the internal data pointers are equal (a fast test).

Parameters

image

> Image to compare with 'this'

Return value

Returns true if the images were effectively equal, false otherwise.

---

wxImage::operator !=

bool operator !=(**const wxImage&** *image*) **const**

Inequality operator. This operator tests whether the internal data pointers are unequal (a fast test).

Parameters

image

> Image to compare with 'this'

Return value

Returns true if the images were unequal, false otherwise.

---

wxImageHandler

This is the base class for implementing image file loading/saving, and image creation from data. It is used within wxImage and is not normally seen by the application.

If you wish to extend the capabilities of wxImage, derive a class from wxImageHandler and add the handler using *wxImage::AddHandler* (p. 783) in your application initialisation.

Note (Legal Issue)

This software is based in part on the work of the Independent JPEG Group.

(Applies when wxWidgets is linked with JPEG support. wxJPEGHandler uses libjpeg created by IJG.)

Derived from

wxObject (p. 1012)

Include files

<wx/image.h>

See also

wxImage (p. 779), *wxInitAllImageHandlers* (p. 1590)

---

wxImageHandler::wxImageHandler

wxImageHandler()

Default constructor. In your own default constructor, initialise the members m_name, m_extension and m_type.

wxImageHandler::~wxImageHandler

~wxImageHandler()

Destroys the wxImageHandler object.

wxImageHandler::GetName

wxString GetName() **const**

Gets the name of this handler.

wxImageHandler::GetExtension

wxString GetExtension() **const**

Gets the file extension associated with this handler.

wxImageHandler::GetImageCount

int GetImageCount(**wxInputStream&** *stream*)

If the image file contains more than one image and the image handler is capable of retrieving these individually, this function will return the number of available images.

stream

Opened input stream for reading image data. Currently, the stream must support seeking.

Return value

Number of available images. For most image handlers, this is 1 (exceptions are TIFF and ICO formats).

wxImageHandler::GetType

long GetType() **const**

Gets the image type associated with this handler.

wxImageHandler::GetMimeType

wxString GetMimeType() **const**

Gets the MIME type associated with this handler.

wxImageHandler::LoadFile

bool LoadFile(**wxImage*** *image*, **wxInputStream&** *stream*, **bool** *verbose=true*, **int** *index=0*)

Loads a image from a stream, putting the resulting data into *image*. If the image file contains more than one image and the image handler is capable of retrieving these individually, *index*indicates which image to read from the stream.

Parameters

image

> The image object which is to be affected by this operation.

stream

> Opened input stream for reading image data.

verbose

> If set to true, errors reported by the image handler will produce wxLogMessages.

index

> The index of the image in the file (starting from zero).

Return value

true if the operation succeeded, false otherwise.

See also

wxImage::LoadFile (p. 793), *wxImage::SaveFile* (p. 796), *wxImageHandler::SaveFile* (p. 804)

wxImageHandler::SaveFile

bool SaveFile(**wxImage*** *image*, **wxOutputStream&** *stream*)

Saves a image in the output stream.

Parameters

image

> The image object which is to be affected by this operation.

stream

> Opened output stream for writing the data.

Return value

true if the operation succeeded, false otherwise.

See also

wxImage::LoadFile (p. 793), *wxImage::SaveFile* (p. 796), *wxImageHandler::LoadFile* (p. 804)

wxImageHandler::SetName

void SetName(**const wxString&** *name*)

Sets the handler name.

Parameters

name

> Handler name.

wxImageHandler::SetExtension

void SetExtension(**const wxString&** *extension*)

Sets the handler extension.

Parameters

extension

> Handler extension.

wxImageHandler::SetMimeType

void SetMimeType(**const wxString&** *mimetype*)

Sets the handler MIME type.

Parameters

mimename

Handler MIME type.

wxImageHandler::SetType

void SetType(**long** *type*)

Sets the handler type.

Parameters

name

Handler type.

wxImageList

A wxImageList contains a list of images, which are stored in an unspecified form. Images can have masks for transparent drawing, and can be made from a variety of sources including bitmaps and icons.

wxImageList is used principally in conjunction with *wxTreeCtrl* (p. 1424) and *wxListCtrl* (p. 852) classes.

Derived from

wxObject (p. 1012)

Include files

<wx/imaglist.h>

See also

wxTreeCtrl (p. 1424), *wxListCtrl* (p. 852)

wxImageList::wxImageList

wxImageList()

Default constructor.

wxImageList(**int** *width*, **int** *height*, **const bool** *mask = true*, **int** *initialCount = 1*)

Constructor specifying the image size, whether image masks should be created, and the initial size of the list.

Parameters

width

Width of the images in the list.

height

Height of the images in the list.

mask

true if masks should be created for all images.

initialCount

The initial size of the list.

See also

wxImageList::Create (p. 808)

wxImageList::Add

int Add(**const wxBitmap&** *bitmap*, **const wxBitmap&** *mask = wxNullBitmap*)

Adds a new image using a bitmap and optional mask bitmap.

int Add(**const wxBitmap&** *bitmap*, **const wxColour&** *maskColour*)

Adds a new image using a bitmap and mask colour.

int Add(**const wxIcon&** *icon*)

Adds a new image using an icon.

Parameters

bitmap

Bitmap representing the opaque areas of the image.

mask

Monochrome mask bitmap, representing the transparent areas of the image.

maskColour

Colour indicating which parts of the image are transparent.

icon

Icon to use as the image.

Return value

The new zero-based image index.

Remarks

The original bitmap or icon is not affected by the **Add** operation, and can be deleted afterwards.

wxPython note: In place of a single overloaded method name, wxPython implements the following methods:

>>**Add(bitmap, mask=wxNullBitmap)**

>>**AddWithColourMask(bitmap, colour)**

>>**AddIcon(icon)**

wxImageList::Create

bool Create(**int** *width*, **int** *height*, **const bool** *mask* = *true*, **int** *initialCount* = *1*)

Initializes the list. See *wxImageList::wxImageList* (p. 806) for details.

wxImageList::Draw

bool Draw(**int** *index*, **wxDC&** *dc*, **int** *x*, **int** *y*, **int** *flags* = *wxIMAGELIST_DRAW_NORMAL*, **const bool** *solidBackground* = *false*)

Draws a specified image onto a device context.

Parameters

index

>Image index, starting from zero.

dc

>Device context to draw on.

x

>X position on the device context.

y

>Y position on the device context.

flags

>How to draw the image. A bitlist of a selection of the following:

>**wxIMAGELIST_DRAW_NORMAL**   Draw the image normally.

>**wxIMAGELIST_DRAW_TRANSPARENT**   Draw the image with transparency.

>**wxIMAGELIST_DRAW_SELECTED** Draw the image in selected state.

wxIMAGELIST_DRAW_FOCUSED   Draw the image in a focused state.

solidBackground

For optimisation - drawing can be faster if the function is told that the background is solid.

wxImageList::GetBitmap

wxBitmap GetBitmap(**int** *index*) **const**

Returns the bitmap corresponding to the given index.

wxImageList::GetIcon

wxIcon GetIcon(**int** *index*) **const**

Returns the icon corresponding to the given index.

wxImageList::GetImageCount

int GetImageCount() **const**

Returns the number of images in the list.

wxImageList::GetSize

bool GetSize(**int** *index*, **int&** *width*, **int &***height*) **const**

Retrieves the size of the images in the list. Currently, the *index* parameter is ignored as all images in the list have the same size.

Parameters

index

currently unused, should be 0

width

receives the width of the images in the list

height

receives the height of the images in the list

Return value

true if the function succeeded, false if it failed (for example, if the image list was not yet

initialized).

wxImageList::Remove

bool Remove(**int** *index*)

Removes the image at the given position.

wxImageList::RemoveAll

bool RemoveAll()

Removes all the images in the list.

wxImageList::Replace

bool Replace(**int** *index*, **const wxBitmap&** *bitmap*, **const wxBitmap&** *mask* = *wxNullBitmap*)

Replaces the existing image with the new image.

Windows only.

bool Replace(**int** *index*, **const wxIcon&** *icon*)

Replaces the existing image with the new image.

Parameters

bitmap

Bitmap representing the opaque areas of the image.

mask

Monochrome mask bitmap, representing the transparent areas of the image.

icon

Icon to use as the image.

Return value

true if the replacement was successful, false otherwise.

Remarks

The original bitmap or icon is not affected by the **Replace** operation, and can be deleted afterwards.

wxPython note: The second form is called `ReplaceIcon` in wxPython.

wxIndividualLayoutConstraint

Objects of this class are stored in the wxLayoutConstraint class as one of eight possible constraints that a window can be involved in.

Constraints are initially set to have the relationship wxUnconstrained, which means that their values should be calculated by looking at known constraints.

Derived from

wxObject (p. 1012)

Include files

<wx/layout.h>

See also

Overview and examples (p. 1768), *wxLayoutConstraints* (p. 837), *wxWindow::SetConstraints* (p. 1530).

Edges and relationships

The *wxEdge* enumerated type specifies the type of edge or dimension of a window.

wxLeft	The left edge.
wxTop	The top edge.
wxRight	The right edge.
wxBottom	The bottom edge.
wxCentreX	The x-coordinate of the centre of the window.
wxCentreY	The y-coordinate of the centre of the window.

The *wxRelationship* enumerated type specifies the relationship that this edge or dimension has with another specified edge or dimension. Normally, the user doesn't use these directly because functions such as *Below* and *RightOf* are a convenience for using the more general *Set* function.

wxUnconstrained	The edge or dimension is unconstrained (the default for edges.
wxAsIs	The edge or dimension is to be taken from the current window position or size (the default for dimensions.
wxAbove	The edge should be above another edge.
wxBelow	The edge should be below another edge.

wxLeftOf	The edge should be to the left of another edge.
wxRightOf	The edge should be to the right of another edge.
wxSameAs	The edge or dimension should be the same as another edge or dimension.
wxPercentOf	The edge or dimension should be a percentage of another edge or dimension.
wxAbsolute	The edge or dimension should be a given absolute value.

wxIndividualLayoutConstraint::wxIndividualLayoutConstraint

void wxIndividualLayoutConstraint()

Constructor. Not used by the end-user.

wxIndividualLayoutConstraint::Above

void Above(**wxWindow** **otherWin*, **int** *margin = 0*)

Constrains this edge to be above the given window, with an optional margin. Implicitly, this is relative to the top edge of the other window.

wxIndividualLayoutConstraint::Absolute

void Absolute(**int** *value*)

Constrains this edge or dimension to be the given absolute value.

wxIndividualLayoutConstraint::AsIs

void AsIs()

Sets this edge or constraint to be whatever the window's value is at the moment. If either of the width and height constraints are *as is*, the window will not be resized, but moved instead. This is important when considering panel items which are intended to have a default size, such as a button, which may take its size from the size of the button label.

wxIndividualLayoutConstraint::Below

void Below(**wxWindow** **otherWin*, **int** *margin = 0*)

Constrains this edge to be below the given window, with an optional margin. Implicitly, this is relative to the bottom edge of the other window.

wxIndividualLayoutConstraint::Unconstrained

void Unconstrained()

Sets this edge or dimension to be unconstrained, that is, dependent on other edges and dimensions from which this value can be deduced.

wxIndividualLayoutConstraint::LeftOf

void LeftOf(**wxWindow** **otherWin*, **int** *margin = 0*)

Constrains this edge to be to the left of the given window, with an optional margin. Implicitly, this is relative to the left edge of the other window.

wxIndividualLayoutConstraint::PercentOf

void PercentOf(**wxWindow** **otherWin*, **wxEdge** *edge*, **int** *per*)

Constrains this edge or dimension to be to a percentage of the given window, with an optional margin.

wxIndividualLayoutConstraint::RightOf

void RightOf(**wxWindow** **otherWin*, **int** *margin = 0*)

Constrains this edge to be to the right of the given window, with an optional margin. Implicitly, this is relative to the right edge of the other window.

wxIndividualLayoutConstraint::SameAs

void SameAs(**wxWindow** **otherWin*, **wxEdge** *edge*, **int** *margin = 0*)

Constrains this edge or dimension to be to the same as the edge of the given window, with an optional margin.

wxIndividualLayoutConstraint::Set

void Set(**wxRelationship** *rel*, **wxWindow** **otherWin*, **wxEdge** *otherEdge*, **int** *value = 0*, **int** *margin = 0*)

Sets the properties of the constraint. Normally called by one of the convenience functions such as Above, RightOf, SameAs.

wxInitDialogEvent

A wxInitDialogEvent is sent as a dialog or panel is being initialised. Handlers for this event can transfer data to the window. The default handler calls *wxWindow::TransferDataToWindow* (p. 1545).

Derived from

wxEvent (p. 483)
wxObject (p. 1012)

Include files

<wx/event.h>

Event table macros

To process an activate event, use these event handler macros to direct input to a member function that takes a wxInitDialogEvent argument.

EVT_INIT_DIALOG(func) Process a wxEVT_INIT_DIALOG event.

See also

Event handling overview (p. 1751)

wxInitDialogEvent::wxInitDialogEvent

wxInitDialogEvent(**int** *id = 0*)

Constructor.

wxInputStream

wxInputStream is an abstract base class which may not be used directly.

Derived from

wxStreamBase (p. 1275)

Include files

<wx/stream.h>

wxInputStream::wxInputStream

wxInputStream()

Creates a dummy input stream.

wxInputStream::~wxInputStream

~wxInputStream()

Destructor.

wxInputStream::CanRead

bool CanRead() **const**

Returns true if some data is available in the stream right now, so that calling *Read()* (p. 816) wouldn't block.

wxInputStream::GetC

char GetC()

Returns the first character in the input queue and removes it, blocking until it appears if necessary.

Note

If EOF, return value is undefined and LastRead() will return 0 and not 1.

wxInputStream::Eof

bool Eof() **const**

Returns true if the end of stream has been reached.

Note

For some streams Eof() will not return true until an attempt has been made to read past the end of the stream.*LastRead()* (p. 815)should be called after each read to check that a non-zero number of bytes have been read.

wxInputStream::LastRead

size_t LastRead() **const**

Returns the last number of bytes read.

wxInputStream::Peek

char Peek()

Returns the first character in the input queue without removing it.

Note

Blocks until something appears in the stream if necessary, if nothing ever does (i.e. EOF) LastRead() will return 0 (and the return value is undefined), otherwise LastRead() returns 1.

wxInputStream::Read

wxInputStream& Read(**void ****buffer*, **size_t** *size*)

Reads the specified amount of bytes and stores the data in *buffer*.

Warning

The buffer absolutely needs to have at least the specified size.

Return value

This function returns a reference on the current object, so the user can test any states of the stream right away.

wxInputStream& Read(**wxOutputStream&** *stream_out*)

Reads data from the input queue and stores it in the specified output stream. The data is read until an error is raised by one of the two streams.

Return value

This function returns a reference on the current object, so the user can test any states of the stream right away.

wxInputStream::SeekI

off_t SeekI(**off_t** *pos*, **wxSeekMode** *mode* = *wxFromStart*)

Changes the stream current position.

Parameters

pos

> Offset to seek to.

mode

> One of **wxFromStart**, **wxFromEnd**, **wxFromCurrent**.

Return value

The new stream position or wxInvalidOffset on error.

wxInputStream::TellI

off_t TellI() **const**

Returns the current stream position.

wxInputStream::Ungetch

size_t Ungetch(**const char*** *buffer*, **size_t** *size*)

This function is only useful in *read* mode. It is the manager of the "Write-Back" buffer. This buffer acts like a temporary buffer where data which has to be read during the next read IO call are put. This is useful when you get a big block of data which you didn't want to read: you can replace them at the top of the input queue by this way.

Be very careful about this call in connection with calling SeekI() on the same stream. Any call to SeekI() will invalidate any previous call to this method (otherwise you could SeekI() to one position, "unread" a few bytes there, SeekI() to another position and data would be either lost or corrupted).

Return value

Returns the amount of bytes saved in the Write-Back buffer.

bool Ungetch(**char** *c*)

This function acts like the previous one except that it takes only one character: it is sometimes shorter to use than the generic function.

wxIPaddress

wxIPaddress is an abstract base class for all internet protocol address objects. Currently, only *wxIPV4address* (p. 819) is implemented. An experimental implementation for IPV6, wxIPV6address, is being developed.

Derived from

wxSockAddress (p. 1202)

Include files

<wx/socket.h>

wxIPaddress::Hostname

virtual bool Hostname(**const wxString&** *hostname*)

Set the address to *hostname*, which can be a host name or an IP-style address in a

format dependent on implementation.

Return value

Returns true on success, false if something goes wrong (invalid hostname or invalid IP address).

virtual wxString Hostname()

Returns the hostname which matches the IP address.

wxIPaddress::IPAddress

virtual wxString IPAddress()

Returns a wxString containing the IP address.

wxIPaddress::Service

virtual bool Service(**const wxString&** *service*)

Set the port to that corresponding to the specified *service*.

Return value

Returns true on success, false if something goes wrong (invalid service).

virtual bool Service(**unsigned short** *service*)

Set the port to that corresponding to the specified *service*.

Return value

Returns true on success, false if something goes wrong (invalid service).

virtual unsigned short Service()

Returns the current service.

wxIPaddress::AnyAddress

virtual bool AnyAddress()

Internally, this is the same as setting the IP address to **INADDR_ANY**.

On IPV4 implementations, 0.0.0.0

On IPV6 implementations, ::

Return value

Returns true on success, false if something went wrong.

wxIPaddress::LocalHost

virtual bool LocalHost()

Set address to localhost.

On IPV4 implementations, 127.0.0.1

On IPV6 implementations, ::1

Return value

Returns true on success, false if something went wrong.

wxIPaddress::IsLocalHost

virtual bool IsLocalHost()

Determines if current address is set to localhost.

Return value

Returns true if address is localhost, false if internet address.

wxIPV4address

Derived from

wxIPaddress (p. 817)

Include files

<wx/socket.h>

wxIPV4address::Hostname

bool Hostname(**const wxString&** *hostname*)

Set the address to *hostname*, which can be a host name or an IP-style address in dot notation (a.b.c.d)

Return value

Returns true on success, false if something goes wrong (invalid hostname or invalid IP address).

wxString Hostname()

Returns the hostname which matches the IP address.

wxIPV4address::IPAddress

wxString IPAddress()

Returns a wxString containing the IP address in dot quad (127.0.0.1) format.

wxIPV4address::Service

bool Service(**const wxString&** *service*)

Set the port to that corresponding to the specified *service*.

Return value

Returns true on success, false if something goes wrong (invalid service).

bool Service(**unsigned short** *service*)

Set the port to that corresponding to the specified *service*.

Return value

Returns true on success, false if something goes wrong (invalid service).

unsigned short Service()

Returns the current service.

wxIPV4address::AnyAddress

bool AnyAddress()

Set address to any of the addresses of the current machine. Whenever possible, use this function instead of *wxIPV4address::LocalHost* (p. 820), as this correctly handles multi-homed hosts and avoids other small problems. Internally, this is the same as setting the IP address to **INADDR_ANY**.

Return value

Returns true on success, false if something went wrong.

wxIPV4address::LocalHost

bool LocalHost()

Set address to localhost (127.0.0.1). Whenever possible, use the *wxIPV4address::AnyAddress* (p. 820), function instead of this one, as this will correctly handle multi-homed hosts and avoid other small problems.

Return value

Returns true on success, false if something went wrong.

wxJoystick

wxJoystick allows an application to control one or more joysticks.

Derived from

wxObject (p. 1012)

Include files

<wx/joystick.h>

See also

wxJoystickEvent (p. 827)

wxJoystick::wxJoystick

 wxJoystick(**int** *joystick* = *wxJOYSTICK1*)

Constructor. *joystick* may be one of wxJOYSTICK1, wxJOYSTICK2, indicating the joystick controller of interest.

wxJoystick::~wxJoystick

 ~wxJoystick()

Destroys the wxJoystick object.

wxJoystick::GetButtonState

int GetButtonState() **const**

Returns the state of the joystick buttons. Every button is mapped to a single bit in the returned integer, with the first button being mapped to the least significant bit, and so on. A bitlist of wxJOY_BUTTONn identifiers, where n is 1, 2, 3 or 4 is available for historical reasons.

wxJoystick::GetManufacturerId

int GetManufacturerId() **const**

Returns the manufacturer id.

wxJoystick::GetMovementThreshold

int GetMovementThreshold() **const**

Returns the movement threshold, the number of steps outside which the joystick is deemed to have moved.

wxJoystick::GetNumberAxes

int GetNumberAxes() **const**

Returns the number of axes for this joystick.

wxJoystick::GetNumberButtons

int GetNumberButtons() **const**

Returns the number of buttons for this joystick.

wxJoystick::GetNumberJoysticks

int GetNumberJoysticks() **const**

Returns the number of joysticks currently attached to the computer.

wxJoystick::GetPollingMax

int GetPollingMax() **const**

Returns the maximum polling frequency.

wxJoystick::GetPollingMin

int GetPollingMin() **const**

Returns the minimum polling frequency.

wxJoystick::GetProductId

int GetProductId() **const**

Returns the product id for the joystick.

wxJoystick::GetProductName

wxString GetProductName() **const**

Returns the product name for the joystick.

wxJoystick::GetPosition

wxPoint GetPosition() **const**

Returns the x, y position of the joystick.

wxJoystick::GetPOVPosition

int GetPOVPosition() **const**

Returns the point-of-view position, expressed in continuous, one-hundredth of a degree units, but limited to return 0, 9000, 18000 or 27000. Returns -1 on error.

wxJoystick::GetPOVCTSPosition

int GetPOVCTSPosition() **const**

Returns the point-of-view position, expressed in continuous, one-hundredth of a degree units. Returns -1 on error.

wxJoystick::GetRudderMax

int GetRudderMax() **const**

Returns the maximum rudder position.

wxJoystick::GetRudderMin

int GetRudderMin() **const**

Returns the minimum rudder position.

wxJoystick::GetRudderPosition

int GetRudderPosition() **const**

Returns the rudder position.

wxJoystick::GetUMax

int GetUMax() **const**

Returns the maximum U position.

wxJoystick::GetUMin

int GetUMin() **const**

Returns the minimum U position.

wxJoystick::GetUPosition

int GetUPosition() **const**

Gets the position of the fifth axis of the joystick, if it exists.

wxJoystick::GetVMax

int GetVMax() **const**

Returns the maximum V position.

wxJoystick::GetVMin

int GetVMin() **const**

Returns the minimum V position.

wxJoystick::GetVPosition

int GetVPosition() **const**

Gets the position of the sixth axis of the joystick, if it exists.

wxJoystick::GetXMax

int GetXMax() **const**

Returns the maximum x position.

wxJoystick::GetXMin

int GetXMin() **const**

Returns the minimum x position.

wxJoystick::GetYMax

int GetYMax() **const**

Returns the maximum y position.

wxJoystick::GetYMin

int GetYMin() **const**

Returns the minimum y position.

wxJoystick::GetZMax

int GetZMax() **const**

Returns the maximum z position.

wxJoystick::GetZMin

int GetZMin() **const**

Returns the minimum z position.

wxJoystick::GetZPosition

int GetZPosition() **const**

Returns the z position of the joystick.

wxJoystick::HasPOV

bool HasPOV() **const**

Returns true if the joystick has a point of view control.

wxJoystick::HasPOV4Dir

bool HasPOV4Dir() **const**

Returns true if the joystick point-of-view supports discrete values (centered, forward, backward, left, and right).

wxJoystick::HasPOVCTS

bool HasPOVCTS() const

Returns true if the joystick point-of-view supports continuous degree bearings.

wxJoystick::HasRudder

bool HasRudder() const

Returns true if there is a rudder attached to the computer.

wxJoystick::HasU

bool HasU() const

Returns true if the joystick has a U axis.

wxJoystick::HasV

bool HasV() const

Returns true if the joystick has a V axis.

wxJoystick::HasZ

bool HasZ() const

Returns true if the joystick has a Z axis.

wxJoystick::IsOk

bool IsOk() const

Returns true if the joystick is functioning.

wxJoystick::ReleaseCapture

bool ReleaseCapture()

Releases the capture set by **SetCapture**.

Return value

true if the capture release succeeded.

See also

wxJoystick::SetCapture (p. 827), *wxJoystickEvent* (p. 827)

wxJoystick::SetCapture

bool SetCapture(**wxWindow*** *win*, **int** *pollingFreq = 0*)

Sets the capture to direct joystick events to *win*.

Parameters

win

> The window that will receive joystick events.

pollingFreq

> If zero, movement events are sent when above the threshold. If greater than zero, events are received every *pollingFreq* milliseconds.

Return value

true if the capture succeeded.

See also

wxJoystick::ReleaseCapture (p. 826), *wxJoystickEvent* (p. 827)

wxJoystick::SetMovementThreshold

void SetMovementThreshold(**int** *threshold*)

Sets the movement threshold, the number of steps outside which the joystick is deemed to have moved.

wxJoystickEvent

This event class contains information about mouse events, particularly events received by windows.

Derived from

wxEvent (p. 483)

Include files

<wx/event.h>

Event table macros

To process a mouse event, use these event handler macros to direct input to member functions that take a wxJoystickEvent argument.

EVT_JOY_BUTTON_DOWN(func)   Process a wxEVT_JOY_BUTTON_DOWN event.

EVT_JOY_BUTTON_UP(func)   Process a wxEVT_JOY_BUTTON_UP event.

EVT_JOY_MOVE(func)   Process a wxEVT_JOY_MOVE event.

EVT_JOY_ZMOVE(func)   Process a wxEVT_JOY_ZMOVE event.

EVT_JOYSTICK_EVENTS(func)   Processes all joystick events.

See also
wxJoystick (p. 821)

wxJoystickEvent::wxJoystickEvent

 wxJoystickEvent(**WXTYPE** *eventType = 0*, **int** *state = 0*, **int** *joystick = wxJOYSTICK1*, **int** *change = 0*)

Constructor.

wxJoystickEvent::ButtonDown

bool ButtonDown(**int** *button = wxJOY_BUTTON_ANY*) **const**

Returns true if the event was a down event from the specified button (or any button).

Parameters

button

Can be wxJOY_BUTTONn where n is 1, 2, 3 or 4; or wxJOY_BUTTON_ANY to indicate any button down event.

wxJoystickEvent::ButtonIsDown

bool ButtonIsDown(**int** *button = wxJOY_BUTTON_ANY*) **const**

Returns true if the specified button (or any button) was in a down state.

Parameters

button

Can be wxJOY_BUTTONn where n is 1, 2, 3 or 4; or wxJOY_BUTTON_ANY to indicate any button down event.

wxJoystickEvent::ButtonUp

bool ButtonUp(**int** *button = wxJOY_BUTTON_ANY*) **const**

Returns true if the event was an up event from the specified button (or any button).

Parameters

button

>Can be wxJOY_BUTTONn where n is 1, 2, 3 or 4; or wxJOY_BUTTON_ANY to indicate any button down event.

wxJoystickEvent::GetButtonChange

int GetButtonChange() const

Returns the identifier of the button changing state. This is a wxJOY_BUTTONn identifier, where n is one of 1, 2, 3, 4.

wxJoystickEvent::GetButtonState

int GetButtonState() const

Returns the down state of the buttons. This is a bitlist of wxJOY_BUTTONn identifiers, where n is one of 1, 2, 3, 4.

wxJoystickEvent::GetJoystick

int GetJoystick() const

Returns the identifier of the joystick generating the event - one of wxJOYSTICK1 and wxJOYSTICK2.

wxJoystickEvent::GetPosition

wxPoint GetPosition() const

Returns the x, y position of the joystick event.

wxJoystickEvent::GetZPosition

int GetZPosition() const

Returns the z position of the joystick event.

wxJoystickEvent::IsButton

bool IsButton() const

Returns true if this was a button up or down event (*not* 'is any button down?').

wxJoystickEvent::IsMove

bool IsMove() **const**

Returns true if this was an x, y move event.

wxJoystickEvent::IsZMove

bool IsZMove() **const**

Returns true if this was a z move event.

wxKeyEvent

This event class contains information about keypress (character) events.

Notice that there are three different kinds of keyboard events in wxWidgets: key down and up events and char events. The difference between the first two is clear - the first corresponds to a key press and the second to a key release - otherwise they are identical. Just note that if the key is maintained in a pressed state you will typically get a lot of (automatically generated) down events but only one up so it is wrong to assume that there is one up event corresponding to each down one.

Both key events provide untranslated key codes while the char event carries the translated one. The untranslated code for alphanumeric keys is always an upper case value. For the other keys it is one of `WXK_XXX` values from the *keycodes table* (p. 1669). The translated key is, in general, the character the user expects to appear as the result of the key combination when typing the text into a text entry zone, for example.

A few examples to clarify this (all assume that CAPS LOCK is unpressed and the standard US keyboard): when the `'A'` key is pressed, the key down event key code is equal to `ASCII A` == 65. But the char event key code is `ASCII a` == 97. On the other hand, if you press both SHIFT and `'A'` keys simultaneously , the key code in key down event will still be just `'A'` while the char event key code parameter will now be `'A'` as well.

Although in this simple case it is clear that the correct key code could be found in the key down event handler by checking the value returned by *ShiftDown()* (p. 834), in general you should use `EVT_CHAR` for this as for non alphanumeric keys the translation is keyboard-layout dependent and can only be done properly by the system itself.

Another kind of translation is done when the control key is pressed: for example, for CTRL-A key press the key down event still carries the same key code `'a'` as usual but the char event will have key code of 1, the ASCII value of this key combination.

You may discover how the other keys on your system behave interactively by running

the *text* (p. 1715) wxWidgets sample and pressing some keys in any of the text controls shown in it.

Note: If a key down (EVT_KEY_DOWN) event is caught and the event handler does not call event.Skip() then the coresponding char event (EVT_CHAR) will not happen. This is by design and enables the programs that handle both types of events to be a bit simpler.

Note for Windows programmers: The key and char events in wxWidgets are similar to but slightly different from Windows WM_KEYDOWN andWM_CHAR events. In particular, Alt-x combination will generate a char event in wxWidgets (unless it is used as an accelerator).

Tip: be sure to call event.Skip() for events that you don't process in key event function, otherwise menu shortcuts may cease to work under Windows.

Derived from

wxEvent (p. 483)

Include files

<wx/event.h>

Event table macros

To process a key event, use these event handler macros to direct input to member functions that take a wxKeyEvent argument.

EVT_KEY_DOWN(func)	Process a wxEVT_KEY_DOWN event (any key has been pressed).
EVT_KEY_UP(func)	Process a wxEVT_KEY_UP event (any key has been released).
EVT_CHAR(func)	Process a wxEVT_CHAR event.

wxKeyEvent::m_altDown

bool m_altDown

true if the Alt key is pressed down.

wxKeyEvent::m_controlDown

bool m_controlDown

true if control is pressed down.

wxKeyEvent::m_keyCode

long m_keyCode

Virtual keycode. See *Keycodes* (p. 1669) for a list of identifiers.

wxKeyEvent::m_metaDown

bool m_metaDown

true if the Meta key is pressed down.

wxKeyEvent::m_shiftDown

bool m_shiftDown

true if shift is pressed down.

wxKeyEvent::m_x

int m_x

X position of the event.

wxKeyEvent::m_y

int m_y

Y position of the event.

wxKeyEvent::wxKeyEvent

wxKeyEvent(**WXTYPE** *keyEventType*)

Constructor. Currently, the only valid event types are wxEVT_CHAR and wxEVT_CHAR_HOOK.

wxKeyEvent::AltDown

bool AltDown() **const**

Returns true if the Alt key was down at the time of the key event.

wxKeyEvent::CmdDown

bool CmdDown() **const**

"Cmd" is a pseudo key which is the same as Control for PC and Unix platforms but the special "Apple" (a.k.a as "Command") key under Macs: it makes often sense to use it instead of, say, ControlDown() because Cmd key is used for the same thing under Mac as Ctrl elsewhere (but Ctrl still exists, just not used for this purpose under Mac). So for non-Mac platforms this is the same as *ControlDown()* (p. 833) and under Mac this is the same as *MetaDown()* (p. 834).

wxKeyEvent::ControlDown

bool ControlDown() **const**

Returns true if the control key was down at the time of the key event.

wxKeyEvent::GetKeyCode

int GetKeyCode() **const**

Returns the virtual key code. ASCII events return normal ASCII values, while non-ASCII events return values such as **WXK_LEFT** for the left cursor key. See *Keycodes* (p. 1669) for a full list of the virtual key codes.

Note that in Unicode build, the returned value is meaningful only if the user entered a character that can be represented in current locale's default charset. You can obtain the corresponding Unicode character using*GetUnicodeKey* (p. 834).

wxKeyEvent::GetPosition

wxPoint GetPosition() **const**

void GetPosition(**long ****x*, **long ****y*) **const**

Obtains the position (in client coordinates) at which the key was pressed.

wxKeyEvent::GetRawKeyCode

wxUint32 GetRawKeyCode() **const**

Returns the raw key code for this event. This is a platform-dependent scan code which should only be used in advanced applications.

NB: Currently the raw key codes are not supported by all ports, use`#ifdef` `wxHAS_RAW_KEY_CODES` to determine if this feature is available.

wxKeyEvent::GetRawKeyFlags

wxUint32 GetRawKeyFlags() **const**

Returns the low level key flags for this event. The flags are platform-dependent and should only be used in advanced applications.

NB: Currently the raw key flags are not supported by all ports, use `#ifdef` `wxHAS_RAW_KEY_CODES` to determine if this feature is available.

wxKeyEvent::GetUnicodeKey

wxChar GetUnicodeKey() **const**

Returns the Unicode character corresponding to this key event.

This function is only available in Unicode build, i.e. when `wxUSE_UNICODE` is 1.

wxKeyEvent::GetX

long GetX() **const**

Returns the X position (in client coordinates) of the event.

wxKeyEvent::GetY

long GetY() **const**

Returns the Y (in client coordinates) position of the event.

wxKeyEvent::HasModifiers

bool HasModifiers() **const**

Returns true if either CTRL or ALT keys was down at the time of the key event. Note that this function does not take into account neither SHIFT nor META key states (the reason for ignoring the latter is that it is common for NUMLOCK key to be configured as META under X but the key presses even while NUMLOCK is on should be still processed normally).

wxKeyEvent::MetaDown

bool MetaDown() **const**

Returns true if the Meta key was down at the time of the key event.

wxKeyEvent::ShiftDown

bool ShiftDown() **const**

Returns true if the shift key was down at the time of the key event.

wxLayoutAlgorithm

wxLayoutAlgorithm implements layout of subwindows in MDI or SDI frames. It sends a wxCalculateLayoutEvent event to children of the frame, asking them for information about their size. For MDI parent frames, the algorithm allocates the remaining space to the MDI client window (which contains the MDI child frames). For SDI (normal) frames, a 'main' window is specified as taking up the remaining space.

Because the event system is used, this technique can be applied to any windows, which are not necessarily 'aware' of the layout classes (no virtual functions in wxWindow refer to wxLayoutAlgorithm or its events). However, you may wish to use *wxSashLayoutWindow* (p. 1129) for your subwindows since this class provides handlers for the required events, and accessors to specify the desired size of the window. The sash behaviour in the base class can be used, optionally, to make the windows user-resizable.

wxLayoutAlgorithm is typically used in IDE (integrated development environment) applications, where there are several resizable windows in addition to the MDI client window, or other primary editing window. Resizable windows might include toolbars, a project window, and a window for displaying error and warning messages.

When a window receives an OnCalculateLayout event, it should call SetRect in the given event object, to be the old supplied rectangle minus whatever space the window takes up. It should also set its own size accordingly. wxSashLayoutWindow::OnCalculateLayout generates an OnQueryLayoutInfo event which it sends to itself to determine the orientation, alignment and size of the window, which it gets from internal member variables set by the application.

The algorithm works by starting off with a rectangle equal to the whole frame client area. It iterates through the frame children, generating OnCalculateLayout events which subtract the window size and return the remaining rectangle for the next window to process. It is assumed (by wxSashLayoutWindow::OnCalculateLayout) that a window stretches the full dimension of the frame client, according to the orientation it specifies. For example, a horizontal window will stretch the full width of the remaining portion of the frame client area. In the other orientation, the window will be fixed to whatever size was specified by OnQueryLayoutInfo. An alignment setting will make the window 'stick' to the left, top, right or bottom of the remaining client area. This scheme implies that order of window creation is important. Say you wish to have an extra toolbar at the top of the frame, a project window to the left of the MDI client window, and an output window above the status bar. You should therefore create the windows in this order: toolbar, output window, project window. This ensures that the toolbar and output window take up space at the top and bottom, and then the remaining height in-between is used for the project window.

wxLayoutAlgorithm is quite independent of the way in which OnCalculateLayout chooses to interpret a window's size and alignment. Therefore you could implement a different window class with a new OnCalculateLayout event handler, that has a more

sophisticated way of laying out the windows. It might allow specification of whether stretching occurs in the specified orientation, for example, rather than always assuming stretching. (This could, and probably should, be added to the existing implementation).

Note: wxLayoutAlgorithm has nothing to do with wxLayoutConstraints. It is an alternative way of specifying layouts for which the normal constraint system is unsuitable.

Derived from

wxObject (p. 1012)

Include files

<wx/laywin.h>

Event handling

The algorithm object does not respond to events, but itself generates the following events in order to calculate window sizes.

EVT_QUERY_LAYOUT_INFO(func)	Process a wxEVT_QUERY_LAYOUT_INFO event, to get size, orientation and alignment from a window. See *wxQueryLayoutInfoEvent* (p. 1087).
EVT_CALCULATE_LAYOUT(func)	Process a wxEVT_CALCULATE_LAYOUT event, which asks the window to take a 'bite' out of a rectangle provided by the algorithm. See *wxCalculateLayoutEvent* (p. 120).

Data types

```
enum wxLayoutOrientation {
    wxLAYOUT_HORIZONTAL,
    wxLAYOUT_VERTICAL
};

enum wxLayoutAlignment {
    wxLAYOUT_NONE,
    wxLAYOUT_TOP,
    wxLAYOUT_LEFT,
    wxLAYOUT_RIGHT,
    wxLAYOUT_BOTTOM,
};
```

See also

wxSashEvent (p. 1127), *wxSashLayoutWindow* (p. 1129), *Event handling overview* (p. 1751)

wxCalculateLayoutEvent (p. 120), *wxQueryLayoutInfoEvent* (p. 1087), *wxSashLayoutWindow* (p. 1129), *wxSashWindow* (p. 1132)

wxLayoutAlgorithm::wxLayoutAlgorithm

wxLayoutAlgorithm()

Default constructor.

wxLayoutAlgorithm::~wxLayoutAlgorithm

~wxLayoutAlgorithm()

Destructor.

wxLayoutAlgorithm::LayoutFrame

bool LayoutFrame(**wxFrame*** *frame*, **wxWindow*** *mainWindow* = *NULL*) **const**

Lays out the children of a normal frame. *mainWindow* is set to occupy the remaining space.

This function simply calls *wxLayoutAlgorithm::LayoutWindow* (p. 837).

wxLayoutAlgorithm::LayoutMDIFrame

bool LayoutMDIFrame(**wxMDIParentFrame*** *frame*, **wxRect*** *rect* = *NULL*) **const**

Lays out the children of an MDI parent frame. If *rect* is non-NULL, the given rectangle will be used as a starting point instead of the frame's client area.

The MDI client window is set to occupy the remaining space.

wxLayoutAlgorithm::LayoutWindow

bool LayoutWindow(**wxWindow*** *parent*, **wxWindow*** *mainWindow* = *NULL*) **const**

Lays out the children of a normal frame or other window.

mainWindow is set to occupy the remaining space. If this is not specified, then the last window that responds to a calculate layout event in query mode will get the remaining space (that is, a non-query OnCalculateLayout event will not be sent to this window and the window will be set to the remaining size).

wxLayoutConstraints

Note: constraints are now deprecated and you should use *sizers* (p. 1771) instead.

Objects of this class can be associated with a window to define its layout constraints, with respect to siblings or its parent.

The class consists of the following eight constraints of class wxIndividualLayoutConstraint, some or all of which should be accessed directly to set the appropriate constraints.

- **left:** represents the left hand edge of the window

- **right:** represents the right hand edge of the window

- **top:** represents the top edge of the window

- **bottom:** represents the bottom edge of the window

- **width:** represents the width of the window

- **height:** represents the height of the window

- **centreX:** represents the horizontal centre point of the window

- **centreY:** represents the vertical centre point of the window

Most constraints are initially set to have the relationship wxUnconstrained, which means that their values should be calculated by looking at known constraints. The exceptions are *width* and *height*, which are set to wxAsIs to ensure that if the user does not specify a constraint, the existing width and height will be used, to be compatible with panel items which often have take a default size. If the constraint is wxAsIs, the dimension will not be changed.

```
wxPerl note: In wxPerl the constraints are accessed as  constraint =
Wx::LayoutConstraints->new();
  constraint->centreX->AsIs();
  constraint->centreY->Unconstrained();
```

Derived from

wxObject (p. 1012)

Include files

<wx/layout.h>

See also

Overview and examples (p. 1768), *wxIndividualLayoutConstraint* (p. 811), *wxWindow::SetConstraints* (p. 1530)

wxLayoutConstraints::wxLayoutConstraints

wxLayoutConstraints()

Constructor.

wxLayoutConstraints::bottom

wxIndividualLayoutConstraint bottom

Constraint for the bottom edge.

wxLayoutConstraints::centreX

wxIndividualLayoutConstraint centreX

Constraint for the horizontal centre point.

wxLayoutConstraints::centreY

wxIndividualLayoutConstraint centreY

Constraint for the vertical centre point.

wxLayoutConstraints::height

wxIndividualLayoutConstraint height

Constraint for the height.

wxLayoutConstraints::left

wxIndividualLayoutConstraint left

Constraint for the left-hand edge.

wxLayoutConstraints::right

wxIndividualLayoutConstraint right

Constraint for the right-hand edge.

wxLayoutConstraints::top

wxIndividualLayoutConstraint top

Constraint for the top edge.

wxLayoutConstraints::width

wxIndividualLayoutConstraint width

Constraint for the width.

wxList

wxList classes provide linked list functionality for wxWidgets, and for an application if it wishes. Depending on the form of constructor used, a list can be keyed on integer or string keys to provide a primitive look-up ability, but please note that this feature is **deprecated**. See *wxHashMap* (p. 685) for a faster method of storage when random access is required.

While wxList class in the previous versions of wxWidgets only could contain elements of type wxObject and had essentially untyped interface (thus allowing you to put apples in the list and read back oranges from it), the new wxList classes family may contain elements of any type and has much more strict type checking. Unfortunately, it also requires an additional line to be inserted in your program for each list class you use (which is the only solution short of using templates which is not done in wxWidgets because of portability issues).

The general idea is to have the base class wxListBase working with *void *data* but make all of its dangerous (because untyped) functions protected, so that they can only be used from derived classes which, in turn, expose a type safe interface. With this approach a new wxList-like class must be defined for each list type (i.e. list of ints, of wxStrings or of MyObjects). This is done with *WX_DECLARE_LIST* and *WX_DEFINE_LIST* macros like this (notice the similarity with WX_DECLARE_OBJARRAY and WX_IMPLEMENT_OBJARRAY macros):

Example

```
// this part might be in a header or source (.cpp) file
class MyListElement
{
    ... // whatever
};

// declare our list class: this macro declares and partly implements
MyList
// class (which derives from wxListBase)
WX_DECLARE_LIST(MyListElement, MyList);

...

// the only requirement for the rest is to be AFTER the full
declaration of
// MyListElement (for WX_DECLARE_LIST forward declaration is enough),
but
// usually it will be found in the source file and not in the header

#include <wx/listimpl.cpp>
WX_DEFINE_LIST(MyList);

// now MyList class may be used as a usual wxList, but all of its
methods
// will take/return the objects of the right (i.e. MyListElement)
type. You
// also have MyList::Node type which is the type-safe version of
wxNode.
```

```
      MyList list;
      MyListElement element;
      list.Append(element);       // ok
      list.Append(17);            // error: incorrect type

      // let's iterate over the list
      for ( MyList::Node *node = list.GetFirst(); node; node = node-
   >GetNext() )
      {
          MyListElement *current = node->GetData();

          ...process the current element...
      }
```

For compatibility with previous versions wxList and wxStringList classes are still defined, but their usage is deprecated and they will disappear in the future versions completely. The use of the latter is especially discouraged as it is not only unsafe but is also much less efficient than *wxArrayString* (p. 65) class.

In the documentation of the list classes below, the template notations are used even though these classes are not really templates at all -- but it helps to think about them as if they were. You should replace wxNode<T> with wxListName::Node and T itself with the list element type (i.e. the first parameter of WX_DECLARE_LIST).

Derived from

wxObject (p. 1012)

Include files

<wx/list.h>

Example

It is very common to iterate on a list as follows:

```
      ...
      wxWindow *win1 = new wxWindow(...);
      wxWindow *win2 = new wxWindow(...);

      wxList SomeList;
      SomeList.Append(win1);
      SomeList.Append(win2);

      ...

      wxNode *node = SomeList.GetFirst();
      while (node)
      {
        wxWindow *win = node->GetData();
        ...
        node = node->GetNext();
      }
```

To delete nodes in a list as the list is being traversed, replace

```
      ...
      node = node->GetNext();
      ...
```

with

```
        ...
        delete win;
        delete node;
        node = SomeList.GetFirst();
        ...
```

See *wxNode* (p. 998) for members that retrieve the data associated with a node, and members for getting to the next or previous node.

See also

wxNode (p. 998),*wxArray* (p. 53)

wxList::wxList

wxList()

wxList(**int** *n*, **T** **objects[]*)

wxList(**T** **object*, ...)

Note: keyed lists are deprecated and should not be used in new code.

wxList(**unsigned int** *key_type*)

Constructors. *key_type* is one of wxKEY_NONE, wxKEY_INTEGER, or wxKEY_STRING, and indicates what sort of keying is required (if any).

objects is an array of *n* objects with which to initialize the list.

The variable-length argument list constructor must be supplied with a terminating NULL.

wxList::~wxList

~wxList()

Destroys the list. Also destroys any remaining nodes, but does not destroy client data held in the nodes.

wxList::Append

wxNode<T> * Append(**T** **object*)

Note: keyed lists are deprecated and should not be used in new code.

wxNode<T> * Append(**long** *key*, **T** **object*)

wxNode<T> * Append(**const wxString&** *key*, **T ****object*)

Appends a new *wxNode* (p. 998) to the end of the list and puts a pointer to the *object* in the node. The last two forms store a key with the object for later retrieval using the key. The new node is returned in each case.

The key string is copied and stored by the list implementation.

wxList::Clear

void Clear()

Clears the list (but does not delete the client data stored with each node unless you called DeleteContents(`true`), in which case it deletes data).

wxList::DeleteContents

void DeleteContents(**bool** *destroy*)

If *destroy* is `true`, instructs the list to call *delete* on the client contents of a node whenever the node is destroyed. The default is `false`.

wxList::DeleteNode

bool DeleteNode(**wxNode<T> ****node*)

Deletes the given node from the list, returning `true` if successful.

wxList::DeleteObject

bool DeleteObject(**T ****object*)

Finds the given client *object* and deletes the appropriate node from the list, returning`true` if successful. The application must delete the actual object separately.

wxList::Erase

void Erase(**wxNode<T> ****node*)

Removes element at given position.

wxList::Find

wxNode<T> * Find(**T *** *object*)

Returns the node whose client date is *object* or NULL if none found.

Note: keyed lists are deprecated and should not be used in new code.

wxNode<T> * Find(**long** *key*)

wxNode<T> * Find(**const wxString&** *key*)

Returns the node whose stored key matches *key*. Use on a keyed list only.

wxList::GetCount

size_t GetCount() **const**

Returns the number of elements in the list.

wxList::GetFirst

wxNode<T> * GetFirst()

Returns the first node in the list (NULL if the list is empty).

wxList::GetLast

wxNode<T> * GetLast()

Returns the last node in the list (NULL if the list is empty).

wxList::IndexOf

int IndexOf(**T*** *obj*)

Returns the index of *obj* within the list or `wxNOT_FOUND` if *obj* is not found in the list.

wxList::Insert

wxNode<T> * Insert(**T** **object*)

Insert object at front of list.

wxNode<T> * Insert(**size_t** *position*, **T** **object*)

Insert object before *position*, i.e. the index of the new item in the list will be equal to *position*. *position* should be less than or equal to *GetCount* (p. 844); if it is equal to it, this is the same as calling *Append* (p. 842).

wxNode<T> * Insert(**wxNode<T>** **node*, **T** **object*)

Inserts the object before the given *node*.

wxList::IsEmpty

bool IsEmpty() **const**

Returns `true` if the list is empty, `false` otherwise.

wxList::Item

wxNode<T> * Item(**size_t** *index*) **const**

Returns the node at given position in the list.

wxList::Member

wxNode<T> * Member(**T ***object*)

NB: This function is deprecated, use *Find* (p. 843) instead.

Returns the node associated with *object* if it is in the list, NULL otherwise.

wxList::Nth

wxNode<T> * Nth(**int** *n*)

NB: This function is deprecated, use *Item* (p. 845) instead.

Returns the *nth* node in the list, indexing from zero (NULL if the list is empty or the nth node could not be found).

wxList::Number

int Number()

NB: This function is deprecated, use *GetCount* (p. 844) instead.

Returns the number of elements in the list.

wxList::Sort

void Sort(**wxSortCompareFunction** *compfunc*)

```
// Type of compare function for list sort operation (as in 'qsort')
typedef int (*wxSortCompareFunction)(const void *elem1, const void
*elem2);
```

Allows the sorting of arbitrary lists by giving a function to compare two list elements. We use the system **qsort** function for the actual sorting process.

If you use untyped wxList the sort function receives pointers to wxObject pointers (wxObject **), so be careful to dereference appropriately - but, of course, a better solution is to use list of appropriate type defined with WX_DECLARE_LIST.

Example:

```
int listcompare(const void *arg1, const void *arg2)
{
  return(compare(**(wxString **)arg1,    // use the wxString 'compare'
                 **(wxString **)arg2));  // function
}

void main()
{
  wxList list;

  list.Append(new wxString("DEF"));
  list.Append(new wxString("GHI"));
  list.Append(new wxString("ABC"));
  list.Sort(listcompare);
}
```

wxListbook

wxListbook is a class similar to *wxNotebook* (p. 999) but which uses a *wxListCtrl* (p. 852) to show the labels instead of the tabs.

There is no documentation for this class yet but its usage is identical to wxNotebook (except for the features clearly related to tabs only), so please refer to that class documentation for now. You can also use the *notebook sample* (p. 1713) to see wxListbook in action.

Derived from

wxControl (p. 215)
wxWindow (p. 1490)
wxEvtHandler (p. 486)
wxObject (p. 1012)

Include files

<wx/listbook.h>

Window styles

wxLB_DEFAULT	Choose the default location for the labels depending on the current platform (left everywhere except Mac where it is top).
wxLB_TOP	Place labels above the page area.
wxLB_LEFT	Place labels on the left side.
wxLB_RIGHT	Place labels on the right side.

| wxLB_BOTTOM | Place labels below the page area. |

See also

wxBookCtrl (p. 1797), *wxNotebook* (p. 999), *notebook sample* (p. 1713)

wxListBox

A listbox is used to select one or more of a list of strings. The strings are displayed in a scrolling box, with the selected string(s) marked in reverse video. A listbox can be single selection (if an item is selected, the previous selection is removed) or multiple selection (clicking an item toggles the item on or off independently of other selections).

List box elements are numbered from zero. Their number is limited in some platforms (e.g. ca. 2000 on GTK).

A listbox callback gets an event wxEVT_COMMAND_LISTBOX_SELECT for single clicks, and wxEVT_COMMAND_LISTBOX_DOUBLE_CLICKED for double clicks.

Derived from

wxControlWithItems (p. 215)
wxControl (p. 215)
wxWindow (p. 1490)
wxEvtHandler (p. 486)
wxObject (p. 1012)

Include files

<wx/listbox.h>

Window styles

wxLB_SINGLE	Single-selection list.
wxLB_MULTIPLE	Multiple-selection list: the user can toggle multiple items on and off.
wxLB_EXTENDED	Extended-selection list: the user can select multiple items using the SHIFT key and the mouse or special key combinations.
wxLB_HSCROLL	Create horizontal scrollbar if contents are too wide (Windows only).
wxLB_ALWAYS_SB	Always show a vertical scrollbar.
wxLB_NEEDED_SB	Only create a vertical scrollbar if needed.
wxLB_SORT	The listbox contents are sorted in alphabetical order.

Note that wxLB_SINGLE, wxLB_MULTIPLE and wxLB_EXTENDEDstyles are mutually exclusive and you can specify at most one of them (single selection is the default).

See also *window styles overview* (p. 1762).

Event handling

EVT_LISTBOX(id, func) Process a wxEVT_COMMAND_LISTBOX_SELECTED event, when an item on the list is selected.

EVT_LISTBOX_DCLICK(id, func) Process a wxEVT_COMMAND_LISTBOX_DOUBLECLIC KED event, when the listbox is double-clicked.

See also

wxChoice (p. 140), *wxComboBox* (p. 172), *wxListCtrl* (p. 852), *wxCommandEvent* (p. 180)

wxListBox::wxListBox

wxListBox()

Default constructor.

wxListBox(**wxWindow*** *parent*, **wxWindowID** *id*, **const wxPoint&** *pos* = *wxDefaultPosition*, **const wxSize&** *size* = *wxDefaultSize*, **int** *n*, **const wxString** *choices[]* = *NULL*, **long** *style = 0*, **const wxValidator&** *validator = wxDefaultValidator*, **const wxString&** *name = "listBox"*)

wxListBox(**wxWindow*** *parent*, **wxWindowID** *id*, **const wxPoint&** *pos*, **const wxSize&** *size*, **const wxArrayString&** *choices*, **long** *style = 0*, **const wxValidator&** *validator = wxDefaultValidator*, **const wxString&** *name = "listBox"*)

Constructor, creating and showing a list box.

Parameters

parent

Parent window. Must not be NULL.

id

Window identifier. A value of -1 indicates a default value.

pos

Window position.

size

Window size. If the default size (-1, -1) is specified then the window is sized appropriately.

n

> Number of strings with which to initialise the control.

choices

> An array of strings with which to initialise the control.

style

> Window style. See *wxListBox* (p. 847).

validator

> Window validator.

name

> Window name.

See also

wxListBox::Create (p. 849), *wxValidator* (p. 1461)

wxPython note: The wxListBox constructor in wxPython reduces the `n` and `choices` arguments are to a single argument, which is a list of strings.

wxPerl note: In wxPerl there is just an array reference in place of `n` and `choices`.

wxListBox::~wxListBox

void ~wxListBox()

Destructor, destroying the list box.

wxListBox::Create

bool Create(**wxWindow*** *parent*, **wxWindowID** *id*, **const wxPoint&** *pos* = *wxDefaultPosition*, **const wxSize&** *size* = *wxDefaultSize*, **int** *n*, **const wxString** *choices[] = NULL,* **long** *style = 0,* **const wxValidator&** *validator = wxDefaultValidator,* **const wxString&** *name = "listBox"*)

bool Create(**wxWindow*** *parent*, **wxWindowID** *id*, **const wxPoint&** *pos*, **const wxSize&** *size*, **const wxArrayString&** *choices*, **long** *style = 0,* **const wxValidator&** *validator = wxDefaultValidator*, **const wxString&** *name = "listBox"*)

Creates the listbox for two-step construction. See *wxListBox::wxListBox* (p. 848) for further details.

wxListBox::Deselect

void Deselect(**int** *n*)

Deselects an item in the list box.

Parameters

n

> The zero-based item to deselect.

Remarks

This applies to multiple selection listboxes only.

wxListBox::GetSelections

int GetSelections(**wxArrayInt&** *selections*) **const**

Fill an array of ints with the positions of the currently selected items.

Parameters

selections

> A reference to an wxArrayInt instance that is used to store the result of the query.

Return value

The number of selections.

Remarks

Use this with a multiple selection listbox.

See also

wxControlWithItems::GetSelection (p. 218), *wxControlWithItems::GetStringSelection* (p. 219), *wxControlWithItems::SetSelection* (p. 221)

wxPython note: The wxPython version of this method takes no parameters and returns a tuple of the selected items.

wxPerl note: In wxPerl this method takes no parameters and return the selected items as a list.

wxListBox::InsertItems

void InsertItems(**int** *nItems*, **const wxString** **items*, **int** *pos*)

void InsertItems(**const wxArrayString&** *nItems*, **int** *pos*)

Insert the given number of strings before the specified position.

Parameters

nItems

Number of items in the array *items*

items

Labels of items to be inserted

pos

Position before which to insert the items: for example, if *pos* is 0 the items will be inserted in the beginning of the listbox

wxPython note: The first two parameters are collapsed into a single parameter for wxPython, which is a list of strings.

wxPerl note: In wxPerl there is just an array reference in place of `nItems` and `items`.

wxListBox::IsSelected

bool IsSelected(**int** *n*) **const**

Determines whether an item is selected.

Parameters

n

The zero-based item index.

Return value

true if the given item is selected, false otherwise.

wxListBox::Set

void Set(**int** *n*, **const wxString*** *choices*, **void** ***clientData* = *NULL*)

void Set(**const wxArrayString&** *choices*, **void** ***clientData* = *NULL*)

Clears the list box and adds the given strings to it.

Parameters

n

The number of strings to set.

choices

An array of strings to set.

clientData

> Options array of client data pointers

Remarks

You may free the array from the calling program after this function has been called.

---

wxListBox::SetFirstItem

void SetFirstItem(**int** *n*)

void SetFirstItem(**const wxString&** *string*)

Set the specified item to be the first visible item. Windows only.

Parameters

n

> The zero-based item index.

string

> The string that should be visible.

wxListCtrl

A list control presents lists in a number of formats: list view, report view, icon view and small icon view. In any case, elements are numbered from zero. For all these modes, the items are stored in the control and must be added to it using *InsertItem* (p. 864) method.

A special case of report view quite different from the other modes of the list control is a virtual control in which the items data (including text, images and attributes) is managed by the main program and is requested by the control itself only when needed which allows to have controls with millions of items without consuming much memory. To use virtual list control you must use*SetItemCount* (p. 869) first and overload at least*OnGetItemText* (p. 865) (and optionally*OnGetItemImage* (p. 865) and*OnGetItemAttr* (p. 865)) to return the information about the items when the control requests it. Virtual list control can be used as a normal one except that no operations which can take time proportional to the number of items in the control happen -- this is required to allow having a practically infinite number of items. For example, in a multiple selection virtual list control, the selections won't be sent when many items are selected at once because this could mean iterating over all the items.

Using many of wxListCtrl features is shown in the*corresponding sample* (p. 1713).

To intercept events from a list control, use the event table macros described in *wxListEvent* (p. 871).

Derived from

wxControl (p. 215)
wxWindow (p. 1490)
wxEvtHandler (p. 486)
wxObject (p. 1012)

Include files

<wx/listctrl.h>

Window styles

wxLC_LIST	Multicolumn list view, with optional small icons. Columns are computed automatically, i.e. you don't set columns as in wxLC_REPORT. In other words, the list wraps, unlike a wxListBox.
wxLC_REPORT	Single or multicolumn report view, with optional header.
wxLC_VIRTUAL	The application provides items text on demand. May only be used with wxLC_REPORT.
wxLC_ICON	Large icon view, with optional labels.
wxLC_SMALL_ICON	Small icon view, with optional labels.
wxLC_ALIGN_TOP	Icons align to the top. Win32 default, Win32 only.
wxLC_ALIGN_LEFT	Icons align to the left.
wxLC_AUTOARRANGE	Icons arrange themselves. Win32 only.
wxLC_EDIT_LABELS	Labels are editable: the application will be notified when editing starts.
wxLC_NO_HEADER	No header in report mode.
wxLC_SINGLE_SEL	Single selection (default is multiple).
wxLC_SORT_ASCENDING	Sort in ascending order (must still supply a comparison callback in SortItems).
wxLC_SORT_DESCENDING	Sort in descending order (must still supply a comparison callback in SortItems).
wxLC_HRULES	Draws light horizontal rules between rows in report mode.
wxLC_VRULES	Draws light vertical rules between columns in report mode.

See also *window styles overview* (p. 1762).

Event handling

To process input from a list control, use these event handler macros to direct input to member functions that take a *wxListEvent* (p. 871) argument.

EVT_LIST_BEGIN_DRAG(id, func)   Begin dragging with the left mouse button.

EVT_LIST_BEGIN_RDRAG(id, func)   Begin dragging with the right mouse button.

EVT_LIST_BEGIN_LABEL_EDIT(id, func) Begin editing a label. This can be prevented by calling *Veto()* (p. 1012).

EVT_LIST_END_LABEL_EDIT(id, func)   Finish editing a label. This can be prevented by calling *Veto()* (p. 1012).

EVT_LIST_DELETE_ITEM(id, func)   Delete an item.

EVT_LIST_DELETE_ALL_ITEMS(id, func) Delete all items.

EVT_LIST_ITEM_SELECTED(id, func) The item has been selected.

EVT_LIST_ITEM_DESELECTED(id, func) The item has been deselected.

EVT_LIST_ITEM_ACTIVATED(id, func) The item has been activated (ENTER or double click).

EVT_LIST_ITEM_FOCUSED(id, func) The currently focused item has changed.

EVT_LIST_ITEM_MIDDLE_CLICK(id, func)   The middle mouse button has been clicked on an item.

EVT_LIST_ITEM_RIGHT_CLICK(id, func) The right mouse button has been clicked on an item.

EVT_LIST_KEY_DOWN(id, func)   A key has been pressed.

EVT_LIST_INSERT_ITEM(id, func)   An item has been inserted.

EVT_LIST_COL_CLICK(id, func)   A column (**m_col**) has been left-clicked.

EVT_LIST_COL_RIGHT_CLICK(id, func) A column (**m_col**) has been right-clicked.

EVT_LIST_COL_BEGIN_DRAG(id, func) The user started resizing a column - can be vetoed.

EVT_LIST_COL_DRAGGING(id, func) The divider between columns is being dragged.

EVT_LIST_COL_END_DRAG(id, func) A column has been resized by the user.

EVT_LIST_CACHE_HINT(id, func)   Prepare cache for a virtual list control

See also

wxListCtrl overview (p. 1796), *wxListBox* (p. 847), *wxTreeCtrl* (p. 1424), *wxImageList* (p. 806), *wxListEvent* (p. 871),*wxListItem* (p. 875)

wxListCtrl::wxListCtrl

wxListCtrl()

Default constructor.

wxListCtrl(**wxWindow*** *parent*, **wxWindowID** *id*, **const wxPoint&** *pos* = *wxDefaultPosition*, **const wxSize&** *size* = *wxDefaultSize*, **long** *style* = *wxLC_ICON*, **const wxValidator&** *validator* = *wxDefaultValidator*, **const wxString&** *name* = *wxListCtrlNameStr*)

Constructor, creating and showing a list control.

Parameters

parent

> Parent window. Must not be NULL.

id

> Window identifier. A value of -1 indicates a default value.

pos

> Window position.

size

> Window size. If the default size (-1, -1) is specified then the window is sized appropriately.

style

> Window style. See *wxListCtrl* (p. 852).

validator

> Window validator.

name

> Window name.

See also

wxListCtrl::Create (p. 856), *wxValidator* (p. 1461)

wxListCtrl::~wxListCtrl

void ~wxListCtrl()

Destructor, destroying the list control.

wxListCtrl::Arrange

bool Arrange(**int** *flag* = *wxLIST_ALIGN_DEFAULT*)

Arranges the items in icon or small icon view. This only has effect on Win32. *flag* is one of:

wxLIST_ALIGN_DEFAULT   Default alignment.

wxLIST_ALIGN_LEFT   Align to the left side of the control.

wxLIST_ALIGN_TOP   Align to the top side of the control.

wxLIST_ALIGN_SNAP_TO_GRID   Snap to grid.

wxListCtrl::AssignImageList

void AssignImageList(**wxImageList*** *imageList*, **int** *which*)

Sets the image list associated with the control and takes ownership of it (i.e. the control will, unlike when using SetImageList, delete the list when destroyed). *which* is one of wxIMAGE_LIST_NORMAL, wxIMAGE_LIST_SMALL, wxIMAGE_LIST_STATE (the last is unimplemented).

See also

wxListCtrl::SetImageList (p. 867)

wxListCtrl::ClearAll

void ClearAll()

Deletes all items and all columns.

wxListCtrl::Create

bool Create(**wxWindow*** *parent*, **wxWindowID** *id*, **const wxPoint&** *pos* = *wxDefaultPosition*, **const wxSize&** *size* = *wxDefaultSize*, **long** *style* = *wxLC_ICON*, **const wxValidator&** *validator* = *wxDefaultValidator*, **const wxString&** *name* = *wxListCtrlNameStr*)

Creates the list control. See *wxListCtrl::wxListCtrl* (p. 855) for further details.

wxListCtrl::DeleteAllItems

bool DeleteAllItems()

Deletes all the items in the list control.

NB: This function does *not* send the wxEVT_COMMAND_LIST_DELETE_ITEM event because deleting many items from the control would be too slow then (unlike *DeleteItem* (p. 857)).

wxListCtrl::DeleteColumn

bool DeleteColumn(**int** *col*)

Deletes a column.

wxListCtrl::DeleteItem

bool DeleteItem(**long** *item*)

Deletes the specified item. This function sends the wxEVT_COMMAND_LIST_DELETE_ITEM event for the item being deleted.

See also: *DeleteAllItems* (p. 856)

wxListCtrl::EditLabel

void EditLabel(**long** *item*)

Starts editing the label of the given item. This function generates a EVT_LIST_BEGIN_LABEL_EDIT event which can be vetoed so that no text control will appear for in-place editing.

If the user changed the label (i.e. s/he does not press ESC or leave the text control without changes, a EVT_LIST_END_LABEL_EDIT event will be sent which can be vetoed as well.

wxListCtrl::EnsureVisible

bool EnsureVisible(**long** *item*)

Ensures this item is visible.

wxListCtrl::FindItem

long FindItem(**long** *start*, **const wxString&** *str*, **const bool** *partial = false*)

Find an item whose label matches this string, starting from *start* or the beginning if *start* is -1.

long FindItem(**long** *start*, **long** *data*)

Find an item whose data matches this data, starting from *start* or the beginning if 'start' is

-1.

long FindItem(**long** *start*, **const wxPoint&** *pt*, **int** *direction*)

Find an item nearest this position in the specified direction, starting from*start* or the beginning if *start* is -1.

wxPython note: In place of a single overloaded method name, wxPython implements the following methods:

>   **FindItem(start, str, partial=false)**

>   **FindItemData(start, data)**

>   **FindItemAtPos(start, point, direction)**

wxPerl note: In wxPerl there are three methods instead of a single overloaded method:

>   **FindItem(start, str, partial = false)**

>   **FindItemData(start, data)**

>   **FindItemAtPos(start, point, direction)**

wxListCtrl::GetColumn

bool GetColumn(**int** *col*, **wxListItem&** *item*) **const**

Gets information about this column. See *wxListCtrl::SetItem* (p. 867) for more information.

wxPerl note: In wxPerl this method takes only the **col** parameter and returns a Wx::ListItem (or undef).

wxListCtrl::GetColumnCount

int GetColumnCount() **const**

Returns the number of columns.

wxListCtrl::GetColumnWidth

int GetColumnWidth(**int** *col*) **const**

Gets the column width (report view only).

wxListCtrl::GetCountPerPage

int GetCountPerPage() const

Gets the number of items that can fit vertically in the visible area of the list control (list or report view) or the total number of items in the list control (icon or small icon view).

wxListCtrl::GetEditControl

wxTextCtrl * GetEditControl() const

Returns the edit control being currently used to edit a label. Returns NULLif no label is being edited.

NB: It is currently only implemented for wxMSW.

wxListCtrl::GetImageList

wxImageList* GetImageList(int *which*) const

Returns the specified image list. *which* may be one of:

wxIMAGE_LIST_NORMAL   The normal (large icon) image list.

wxIMAGE_LIST_SMALL   The small icon image list.

wxIMAGE_LIST_STATE   The user-defined state image list (unimplemented).

wxListCtrl::GetItem

bool GetItem(wxListItem& *info*) const

Gets information about the item. See *wxListCtrl::SetItem* (p. 867) for more information.

You must call *info.SetId()* to the ID of item you're interested in before calling this method.

wxPython note: The wxPython version of this method takes an integer parameter for the item ID, an optional integer for the column number, and returns the wxListItem object.

wxPerl note: In wxPerl this method takes as parameter the **ID** of the item and (optionally) the column, and returns a Wx::ListItem object.

wxListCtrl::GetItemBackgroundColour

wxColour GetItemBackgroundColour(long *item*) const

Returns the colour for this item. If the item has no specific colour, returns an invalid colour (and not the default background control of the control itself).

See also

GetItemTextColour (p. 861)

wxListCtrl::GetItemCount

int GetItemCount() **const**

Returns the number of items in the list control.

wxListCtrl::GetItemData

long GetItemData(**long** *item*) **const**

Gets the application-defined data associated with this item.

wxListCtrl::GetItemPosition

bool GetItemPosition(**long** *item,* **wxPoint&** *pos*) **const**

Returns the position of the item, in icon or small icon view.

wxPython note: The wxPython version of this method accepts only the item ID and returns the wxPoint.

wxPerl note: In wxPerl this method takes only the **item** parameter and returns a Wx::Point (or undef).

wxListCtrl::GetItemRect

bool GetItemRect(**long** *item*, **wxRect&** *rect*, **int** *code = wxLIST_RECT_BOUNDS*) **const**

Returns the rectangle representing the item's size and position, in client coordinates.

code is one of wxLIST_RECT_BOUNDS, wxLIST_RECT_ICON, wxLIST_RECT_LABEL.

wxPython note: The wxPython version of this method accepts only the item ID and code and returns the wxRect.

wxPerl note: In wxPerl this method takes only the **item** parameter and returns a Wx::Rect (or undef).

wxListCtrl::GetItemSpacing

wxSize GetItemSpacing() **const**

Retrieves the spacing between icons in pixels: horizontal spacing is returned as x

component of the *wxSize* (p. 1173) object and the vertical spacing as its y component.

wxListCtrl::GetItemState

int GetItemState(**long** *item*, **long** *stateMask*) **const**

Gets the item state. For a list of state flags, see *wxListCtrl::SetItem* (p. 867).

The **stateMask** indicates which state flags are of interest.

wxListCtrl::GetItemText

wxString GetItemText(**long** *item*) **const**

Gets the item text for this item.

wxListCtrl::GetItemTextColour

wxColour GetItemTextColour(**long** *item*) **const**

Returns the colour for this item. If the item has no specific colour, returns an invalid colour (and not the default foreground control of the control itself as this wouldn't allow distinguishing between items having the same colour as the current control foreground and items with default colour which, hence, have always the same colour as the control).

wxListCtrl::GetNextItem

long GetNextItem(**long** *item*, **int** *geometry = wxLIST_NEXT_ALL*, **int** *state = wxLIST_STATE_DONTCARE*) **const**

Searches for an item with the given geometry or state, starting from*item* but excluding the *item* itself. If *item* is -1, the first item that matches the specified flags will be returned.

Returns the first item with given state following *item* or -1 if no such item found.

This function may be used to find all selected items in the control like this:

```
long item = -1;
for ( ;; )
{
    item = listctrl->GetNextItem(item,
                                 wxLIST_NEXT_ALL,
                                 wxLIST_STATE_SELECTED);
    if ( item == -1 )
        break;

    // this item is selected - do whatever is needed with it
    wxLogMessage("Item %ld is selected."), item);
}
```

geometry can be one of:

wxLIST_NEXT_ABOVE   Searches for an item above the specified item.

wxLIST_NEXT_ALL   Searches for subsequent item by index.

wxLIST_NEXT_BELOW   Searches for an item below the specified item.

wxLIST_NEXT_LEFT   Searches for an item to the left of the specified item.

wxLIST_NEXT_RIGHT   Searches for an item to the right of the specified item.

NB: this parameters is only supported by wxMSW currently and ignored on other platforms.

state can be a bitlist of the following:

wxLIST_STATE_DONTCARE   Don't care what the state is.

wxLIST_STATE_DROPHILITED   The item indicates it is a drop target.

wxLIST_STATE_FOCUSED The item has the focus.

wxLIST_STATE_SELECTEDThe item is selected.

wxLIST_STATE_CUT   The item is selected as part of a cut and paste operation.

wxListCtrl::GetSelectedItemCount

int GetSelectedItemCount() **const**

Returns the number of selected items in the list control.

wxListCtrl::GetTextColour

wxColour GetTextColour() **const**

Gets the text colour of the list control.

wxListCtrl::GetTopItem

long GetTopItem() **const**

Gets the index of the topmost visible item when in list or report view.

wxListCtrl::GetViewRect

wxRect GetViewRect() **const**

Returns the rectangle taken by all items in the control. In other words, if the controls

client size were equal to the size of this rectangle, no scrollbars would be needed and no free space would be left.

Note that this function only works in the icon and small icon views, not in list or report views (this is a limitation of the native Win32 control).

wxListCtrl::HitTest

long HitTest(**const wxPoint&** *point*, **int&** *flags*)

Determines which item (if any) is at the specified point, giving details in *flags*. Returns index of the item or `wxNOT_FOUND`if no item is at the specified point.*flags* will be a combination of the following flags:

wxLIST_HITTEST_ABOVE   Above the client area.

wxLIST_HITTEST_BELOW   Below the client area.

wxLIST_HITTEST_NOWHERE   In the client area but below the last item.

wxLIST_HITTEST_ONITEMICON   On the bitmap associated with an item.

wxLIST_HITTEST_ONITEMLABEL On the label (string) associated with an item.

wxLIST_HITTEST_ONITEMRIGHT   In the area to the right of an item.

wxLIST_HITTEST_ONITEMSTATEICON   On the state icon for a tree view item that is in a user-defined state.

wxLIST_HITTEST_TOLEFT To the right of the client area.

wxLIST_HITTEST_TORIGHT   To the left of the client area.

wxLIST_HITTEST_ONITEM Combination of wxLIST_HITTEST_ONITEMICON, wxLIST_HITTEST_ONITEMLABEL, wxLIST_HITTEST_ONITEMSTATEICON.

wxPython note: A tuple of values is returned in the wxPython version of this method. The first value is the item id and the second is the flags value mentioned above.

wxPerl note: In wxPerl this method only takes the **point** parameter and returns a 2-element list (item, flags).

wxListCtrl::InsertColumn

long InsertColumn(**long** *col*, **wxListItem&** *info*)

long InsertColumn(**long** *col*, **const wxString&** *heading*, **int** *format* = *wxLIST_FORMAT_LEFT*, **int** *width = -1*)

For report view mode (only), inserts a column. For more details, see *wxListCtrl::SetItem* (p. 867).

wxPython note: In place of a single overloaded method name, wxPython implements the following methods:

> **InsertColumn(col, heading, format=wxLIST_FORMAT_LEFT, width=-1)**
> Creates a column using a header string only.

> **InsertColumnItem(col, item)** Creates a column using a wxListItem.

wxListCtrl::InsertItem

long InsertItem(**wxListItem&** *info*)

Inserts an item, returning the index of the new item if successful, -1 otherwise.

long InsertItem(**long** *index*, **const wxString&** *label*)

Inserts a string item.

long InsertItem(**long** *index*, **int** *imageIndex*)

Inserts an image item.

long InsertItem(**long** *index*, **const wxString&** *label*, **int** *imageIndex*)

Insert an image/string item.

Parameters

info

> wxListItem object

index

> Index of the new item, supplied by the application

label

> String label

imageIndex

> index into the image list associated with this control and view style

wxPython note: In place of a single overloaded method name, wxPython implements the following methods:

> **InsertItem(item)**   Inserts an item using a wxListItem.

> **InsertStringItem(index, label)** Inserts a string item.

> **InsertImageItem(index, imageIndex)** Inserts an image item.

> **InsertImageStringItem(index, label, imageIndex)**   Insert an image/string item.

wxPerl note: In wxPerl there are four methods instead of a single overloaded method:

> **InsertItem(item)**   Inserts a Wx::ListItem
>
> **InsertStringItem(index, label)** Inserts a string item
>
> **InsertImageItem(index, imageIndex)** Inserts an image item
>
> **InsertImageStringItem(index, label, imageIndex)**   Inserts an item with a string and an image

wxListCtrl::OnGetItemAttr

virtual wxListItemAttr * OnGetItemAttr(long *item*)

This function may be overloaded in the derived class for a control with `wxLC_VIRTUAL` style. It should return the attribute for the for the specified `item` or `NULL` to use the default appearance parameters.

The base class version always returns `NULL`.

See also

OnGetItemImage (p. 865),
OnGetItemText (p. 865)

wxListCtrl::OnGetItemImage

virtual int OnGetItemImage(long *item*)

This function must be overloaded in the derived class for a control with `wxLC_VIRTUAL` style having an *image list* (p. 867)(if the control doesn't have an image list, it is not necessary to overload it). It should return the index of the items image in the controls image list or -1 for no image.

The base class version always returns -1.

See also

OnGetItemText (p. 865),
OnGetItemAttr (p. 865)

wxListCtrl::OnGetItemText

virtual wxString OnGetItemText(long *item*, long *column*) const

This function **must** be overloaded in the derived class for a control with `wxLC_VIRTUAL` style. It should return the string containing the text of the given *column* for the specified `item`.

See also

SetItemCount (p. 869),
OnGetItemImage (p. 865),
OnGetItemAttr (p. 865)

wxListCtrl::RefreshItem

void RefreshItem(long *item*)

Redraws the given *item*. This is only useful for the virtual list controls as without calling this function the displayed value of the item doesn't change even when the underlying data does change.

See also

RefreshItems (p. 866)

wxListCtrl::RefreshItems

void RefreshItems(long *itemFrom*, long *itemTo*)

Redraws the items between *itemFrom* and *itemTo*. The starting item must be less than or equal to the ending one.

Just as *RefreshItem* (p. 866) this is only useful for virtual list controls.

wxListCtrl::ScrollList

bool ScrollList(int *dx*, int *dy*)

Scrolls the list control. If in icon, small icon or report view mode, *dx* specifies the number of pixels to scroll. If in list view mode, *dx* specifies the number of columns to scroll. *dy* always specifies the number of pixels to scroll vertically.

NB: This method is currently only implemented in the Windows version.

wxListCtrl::SetBackgroundColour

void SetBackgroundColour(const wxColour& *col*)

Sets the background colour (GetBackgroundColour already implicit in wxWindow class).

wxListCtrl::SetColumn

bool SetColumn(**int** *col*, **wxListItem&** *item*)

Sets information about this column. See *wxListCtrl::SetItem* (p. 867) for more information.

wxListCtrl::SetColumnWidth

bool SetColumnWidth(**int** *col*, **int** *width*)

Sets the column width.

width can be a width in pixels or wxLIST_AUTOSIZE (-1) or wxLIST_AUTOSIZE_USEHEADER (-2). wxLIST_AUTOSIZE will resize the column to the length of its longest item. wxLIST_AUTOSIZE_USEHEADER will resize the column to the length of the header (Win32) or 80 pixels (other platforms).

In small or normal icon view, *col* must be -1, and the column width is set for all columns.

wxListCtrl::SetImageList

void SetImageList(**wxImageList*** *imageList*, **int** *which*)

Sets the image list associated with the control. *which* is one of wxIMAGE_LIST_NORMAL, wxIMAGE_LIST_SMALL, wxIMAGE_LIST_STATE (the last is unimplemented).

This method does not take ownership of the image list, you have to delete it yourself.

See also

wxListCtrl::AssignImageList (p. 856)

wxListCtrl::SetItem

bool SetItem(**wxListItem&** *info*)

long SetItem(**long** *index*, **int** *col*, **const** *wxString& label*, **int** *imageId = -1*)

Sets information about the item.

wxListItem is a class with the following members:

long m_mask	Indicates which fields are valid. See the list of valid mask flags below.
long m_itemId	The zero-based item position.
int m_col	Zero-based column, if in report mode.

long m_state	The state of the item. See the list of valid state flags below.
long m_stateMask	A mask indicating which state flags are valid. See the list of valid state flags below.
wxString m_text	The label/header text.
int m_image	The zero-based index into an image list.
long m_data	Application-defined data.
int m_format	For columns only: the format. Can be wxLIST_FORMAT_LEFT, wxLIST_FORMAT_RIGHT or wxLIST_FORMAT_CENTRE.
int m_width	For columns only: the column width.

The **m_mask** member contains a bitlist specifying which of the other fields are valid. The flags are:

wxLIST_MASK_STATE	The **m_state** field is valid.
wxLIST_MASK_TEXT	The **m_text** field is valid.
wxLIST_MASK_IMAGE	The **m_image** field is valid.
wxLIST_MASK_DATA	The **m_data** field is valid.
wxLIST_MASK_WIDTH	The **m_width** field is valid.
wxLIST_MASK_FORMAT	The **m_format** field is valid.

The **m_stateMask** and **m_state** members take flags from the following:

wxLIST_STATE_DONTCARE	Don't care what the state is. Win32 only.
wxLIST_STATE_DROPHILITED	The item is highlighted to receive a drop event. Win32 only.
wxLIST_STATE_FOCUSED	The item has the focus.
wxLIST_STATE_SELECTED	The item is selected.
wxLIST_STATE_CUT	The item is in the cut state. Win32 only.

The wxListItem object can also contain item-specific colour and font information: for this you need to call one of SetTextColour(), SetBackgroundColour() or SetFont() functions on it passing it the colour/font to use. If the colour/font is not specified, the default list control colour/font is used.

long SetItem(**long** *index*, **int** *col*, **const wxString&** *label*, **int** *imageId = -1*)

Sets a string field at a particular column.

wxPython note: In place of a single overloaded method name, wxPython implements the following methods:

| **SetItem(item)** | Sets information about the given wxListItem. |
| **SetStringItem(index, col, label, imageId)** | Sets a string or image at a given location. |

wxListCtrl::SetItemBackgroundColour

void SetItemBackgroundColour(**long** *item*, **const wxColour&** *col*)

Sets the background colour for this item. This function only works in report view.

The colour can be retrieved using*GetItemBackgroundColour* (p. 859).

wxListCtrl::SetItemCount

void SetItemCount(**long** *count*)

This method can only be used with virtual list controls. It is used to indicate to the control the number of items it contains. After calling it, the main program should be ready to handle calls to various item callbacks (such as*OnGetItemText* (p. 865)) for all items in the range from 0 to *count*.

wxListCtrl::SetItemData

bool SetItemData(**long** *item*, **long** *data*)

Associates application-defined data with this item.

wxListCtrl::SetItemImage

bool SetItemImage(**long** *item*, **int** *image*)

Sets the image associated with the item. The image is an index into the image list associated with the list control.

bool SetItemImage(**long** *item*, **int** *image*, **int** *selImage*)

Sets the unselected and selected images associated with the item. The images are indices into the image list associated with the list control. This form is deprecated: *selImage* is not used.

wxListCtrl::SetItemPosition

bool SetItemPosition(**long** *item*, **const wxPoint&** *pos*)

Sets the position of the item, in icon or small icon view. Windows only.

wxListCtrl::SetItemState

bool SetItemState(**long** *item*, **long** *state*, **long** *stateMask*)

Sets the item state. For a list of state flags, see *wxListCtrl::SetItem* (p. 867).

The **stateMask** indicates which state flags are valid.

wxListCtrl::SetItemText

void SetItemText(**long** *item*, **const wxString&** *text*)

Sets the item text for this item.

wxListCtrl::SetItemTextColour

void SetItemTextColour(**long** *item*, **const wxColour&** *col*)

Sets the colour for this item. This function only works in report view.

The colour can be retrieved using *GetItemTextColour* (p. 861).

wxListCtrl::SetSingleStyle

void SetSingleStyle(**long** *style*, **const bool** *add = true*)

Adds or removes a single window style.

wxListCtrl::SetTextColour

void SetTextColour(**const wxColour&** *col*)

Sets the text colour of the list control.

wxListCtrl::SetWindowStyleFlag

void SetWindowStyleFlag(**long** *style*)

Sets the whole window style, deleting all items.

wxListCtrl::SortItems

bool SortItems(**wxListCtrlCompare** *fnSortCallBack*, **long** *data*)

Call this function to sort the items in the list control. Sorting is done using the specified *fnSortCallBack* function. This function must have the following prototype:

```
int wxCALLBACK wxListCompareFunction(long item1, long item2, long
sortData)
```

It is called each time when the two items must be compared and should return 0 if the items are equal, negative value if the first item is less than the second one and positive value if the first one is greater than the second one (the same convention as used by `qsort(3)`).

Parameters

item1

> client data associated with the first item (**NOT** the index).

item2

> client data associated with the second item (**NOT** the index).

data

> the value passed to SortItems() itself.

Notice that the control may only be sorted on client data associated with the items, so you **must** use *SetItemData* (p. 869) if you want to be able to sort the items in the control.

Please see the *listctrl sample* (p. 1713) for an example of using this function.

wxPython note: wxPython uses the sortData parameter to pass the Python function to call, so it is not available for programmer use. Call SortItems with a reference to a callable object that expects two parameters.

wxPerl note: In wxPerl the comparison function must take just two parameters; however, you may use a closure to achieve an effect similar to the SortItems third parameter.

wxListEvent

A list event holds information about events associated with wxListCtrl objects.

Derived from

wxNotifyEvent (p. 1011)
wxCommandEvent (p. 180)
wxEvent (p. 483)
wxObject (p. 1012)

Include files

<wx/listctrl.h>

Event table macros

To process input from a list control, use these event handler macros to direct input to

member functions that take a wxListEvent argument.

EVT_LIST_BEGIN_DRAG(id, func)   Begin dragging with the left mouse button.

EVT_LIST_BEGIN_RDRAG(id, func)   Begin dragging with the right mouse button.

EVT_LIST_BEGIN_LABEL_EDIT(id, func) Begin editing a label. This can be prevented by calling *Veto()* (p. 1012).

EVT_LIST_END_LABEL_EDIT(id, func)   Finish editing a label. This can be prevented by calling *Veto()* (p. 1012).

EVT_LIST_DELETE_ITEM(id, func)   Delete an item.

EVT_LIST_DELETE_ALL_ITEMS(id, func) Delete all items.

EVT_LIST_ITEM_SELECTED(id, func) The item has been selected.

EVT_LIST_ITEM_DESELECTED(id, func) The item has been deselected.

EVT_LIST_ITEM_ACTIVATED(id, func) The item has been activated (ENTER or double click).

EVT_LIST_ITEM_FOCUSED(id, func) The currently focused item has changed.

EVT_LIST_ITEM_MIDDLE_CLICK(id, func)   The middle mouse button has been clicked on an item.

EVT_LIST_ITEM_RIGHT_CLICK(id, func) The right mouse button has been clicked on an item.

EVT_LIST_KEY_DOWN(id, func)   A key has been pressed.

EVT_LIST_INSERT_ITEM(id, func)   An item has been inserted.

EVT_LIST_COL_CLICK(id, func)   A column (**m_col**) has been left-clicked.

EVT_LIST_COL_RIGHT_CLICK(id, func) A column (**m_col**) (which can be -1 if the click occurred outside any column) has been right-clicked.

EVT_LIST_COL_BEGIN_DRAG(id, func) The user started resizing a column - can be vetoed.

EVT_LIST_COL_DRAGGING(id, func) The divider between columns is being dragged.

EVT_LIST_COL_END_DRAG(id, func) A column has been resized by the user.

EVT_LIST_CACHE_HINT(id, func)   Prepare cache for a virtual list control

See also
wxListCtrl (p. 852)

wxListEvent::wxListEvent

wxListEvent(**WXTYPE** *commandType = 0*, **int** *id = 0*)

Constructor.

wxListEvent::GetCacheFrom

long GetCacheFrom() **const**

For `EVT_LIST_CACHE_HINT` event only: return the first item which the list control advises us to cache.

wxListEvent::GetCacheTo

long GetCacheTo() **const**

For `EVT_LIST_CACHE_HINT` event only: return the last item (inclusive) which the list control advises us to cache.

wxListEvent::GetKeyCode

int GetKeyCode() **const**

Key code if the event is a keypress event.

wxListEvent::GetIndex

long GetIndex() **const**

The item index.

wxListEvent::GetColumn

int GetColumn() **const**

The column position: it is only used with `COL` events. For the column dragging events, it is the column to the left of the divider being dragged, for the column click events it may be -1 if the user clicked in the list control header outside any column.

wxListEvent::GetPoint

wxPoint GetPoint() **const**

The position of the mouse pointer if the event is a drag event.

wxListEvent::GetLabel

const wxString& GetLabel() **const**

The (new) item label for `EVT_LIST_END_LABEL_EDIT` event.

wxListEvent::GetText

const wxString& GetText() **const**

The text.

wxListEvent::GetImage

int GetImage() **const**

The image.

wxListEvent::GetData

long GetData() **const**

The data.

wxListEvent::GetMask

long GetMask() **const**

The mask.

wxListEvent::GetItem

const wxListItem& GetItem() **const**

An item object, used by some events. See also *wxListCtrl::SetItem* (p. 867).

wxListEvent::IsEditCancelled

bool IsEditCancelled() **const**

This method only makes sense for `EVT_LIST_END_LABEL_EDIT` message and returns `true` if it the label editing has been cancelled by the user (*GetLabel* (p. 874) returns an empty string in this case but it doesn't allow the application to distinguish between really cancelling the edit and the admittedly rare case when the user wants to rename it to an empty string).

wxListItem

This class stores information about a wxListCtrl item or column.

Derived from

wxObject (p. 1012)

Include files

<wx/listctrl.h>

wxListItem::wxListItem

wxListItem()

Constructor.

wxListItem::Clear

void Clear()

Resets the item state to the default.

wxListItem::GetAlign

wxListColumnFormat GetAlign() **const**

Returns the alignment for this item. Can be one of wxLIST_FORMAT_LEFT, wxLIST_FORMAT_RIGHT or wxLIST_FORMAT_CENTRE.

wxListItem::GetBackgroundColour

wxColour GetBackgroundColour() **const**

Returns the background colour for this item.

wxListItem::GetColumn

int GetColumn() **const**

Returns the zero-based column; meaningful only in report mode.

wxListItem::GetData

long GetData() const

Returns client data associated with the control.

wxListItem::GetFont

wxFont GetFont() const

Returns the font used to display the item.

wxListItem::GetId

long GetId() const

Returns the zero-based item position.

wxListItem::GetImage

int GetImage() const

Returns the zero-based index of the image associated with the item into the image list.

wxListItem::GetMask

long GetMask() const

Returns a bit mask indicating which fields of the structure are valid; can be any combination of the following values:

wxLIST_MASK_STATE	**GetState** is valid.
wxLIST_MASK_TEXT	**GetText** is valid.
wxLIST_MASK_IMAGE	**GetImage** is valid.
wxLIST_MASK_DATA	**GetData** is valid.
wxLIST_MASK_WIDTH	**GetWidth** is valid.
wxLIST_MASK_FORMAT	**GetFormat** is valid.

wxListItem::GetState

long GetState() const

Returns a bit field representing the state of the item. Can be any combination of:

wxLIST_STATE_DONTCARE	Don't care what the state is. Win32 only.

wxLIST_STATE_DROPHILITED   The item is highlighted to receive a drop event. Win32 only.

wxLIST_STATE_FOCUSED The item has the focus.

wxLIST_STATE_SELECTEDThe item is selected.

wxLIST_STATE_CUT   The item is in the cut state. Win32 only.

wxListItem::GetText

const wxString& GetText() **const**

Returns the label/header text.

wxListItem::GetTextColour

wxColour GetTextColour() **const**

Returns the text colour.

wxListItem::GetWidth

int GetWidth() **const**

Meaningful only for column headers in report mode. Returns the column width.

wxListItem::SetAlign

void SetAlign(**wxListColumnFormat** *align*)

Sets the alignment for the item. See also*wxListItem::GetAlign* (p. 875)

wxListItem::SetBackgroundColour

void SetBackgroundColour(**const wxColour&** *colBack*)

Sets the background colour for the item.

wxListItem::SetColumn

void SetColumn(**int** *col*)

Sets the zero-based column. Meaningful only in report mode.

wxListItem::SetData

void SetData(**long** *data*)

void SetData(**void*** *data*)

Sets client data for the item.

wxListItem::SetFont

void SetFont(**const wxFont&** *font*)

Sets the font for the item.

wxListItem::SetId

void SetId(**long** *id*)

Sets the zero-based item position.

wxListItem::SetImage

void SetImage(**int** *image*)

Sets the zero-based index of the image associated with the item into the image list.

wxListItem::SetMask

void SetMask(**long** *mask*)

Sets the mask of valid fields. See *wxListItem::GetMask* (p. 876).

wxListItem::SetState

void SetState(**long** *state*)

Sets the item state flags (note that the valid state flags are influenced by the value of the state mask, see*wxListItem::SetStateMask* (p. 878)). See *wxListItem::GetState* (p. 876) for valid flag values.

wxListItem::SetStateMask

void SetStateMask(**long** *stateMask*)

Sets the bitmask that is used to determine which of the state flags are to be set. See also *wxListItem::SetState* (p. 878).

wxListItem::SetText

void SetText(**const wxString&** *text*)

Sets the text label for the item.

wxListItem::SetTextColour

void SetTextColour(**const wxColour&** *colText*)

Sets the text colour for the item.

wxListItem::SetWidth

void SetWidth(**int** *width*)

Meaningful only for column headers in report mode. Sets the column width.

wxListView

This class currently simply presents a simpler to use interface for the *wxListCtrl* (p. 852) -- it can be thought of as a *façade*for that complicated class. Using it is preferable to using *wxListCtrl* (p. 852) directly whenever possible because in the future some ports might implement wxListView but not the full set of wxListCtrl features.

Other than different interface, this class is identical to wxListCtrl. In particular, it uses the same events, same windows styles and so on.

Derived from

wxListCtrl (p. 852)
wxControl (p. 215)
wxWindow (p. 1490)
wxEvtHandler (p. 486)
wxObject (p. 1012)

Include files

<wx/listctrl.h>

wxListView::ClearColumnImage

void ClearColumnImage(**int** *col*)

Resets the column image -- after calling this function, no image will be shown.

Parameters

col

the column to clear image for

See also

SetColumnImage (p. 881)

wxListView::Focus

void Focus(**long** *index*)

Sets focus to the item with the given *index*.

wxListView::GetFirstSelected

long GetFirstSelected() **const**

Returns the first selected item in a (presumably) multiple selection control. Tigether with *GetNextSelected* (p. 880) it can be used to iterate over all selected items in the control.

Return value

The fisrt selected item, if any, -1 otherwise.

wxListView::GetFocusedItem

long GetFocusedItem() **const**

Returns the currently focused item or -1 if none.

See also

IsSelected (p. 881),
Focus (p. 880)

wxListView::GetNextSelected

long GetNextSelected(**long** *item*) **const**

Used together with *GetFirstSelected* (p. 880) to iterate over all selected items in the control.

Return value

Returns the next selected item or -1 if there are no more of them.

wxListView::IsSelected

bool IsSelected(**long** *index*)

Returns `true` if the item with the given *index* is selected, `false` otherwise.

See also

GetFirstSelected (p. 880),
GetNextSelected (p. 880)

wxListView::Select

void Select(long n, **bool** *on = true*)

Selects or unselects the given item.

Parameters

n

the item to select or unselect

on

if `true` (default), selects the item, otherwise unselects it

See also

SetItemState (p. 870)

wxListView::SetColumnImage

void SetColumnImage(**int** *col*, **int** *image*)

Sets the column image for the specified column. To use the column images, the control must have a valid image list with at least one image.

Parameters

col

the column to set image for

image

the index of the column image in the controls image list

See also

ClearColumnImage (p. 879),
SetImageList (p. 867)

wxLocale

wxLocale class encapsulates all language-dependent settings and is a generalization of the C locale concept.

In wxWidgets this class manages message catalogs which contain the translations of the strings used to the current language.

wxPerl note: In wxPerl you can't use the '_' function name, so the `Wx::Locale` module can export the `gettext` and `gettext_noop` under any given name.

```
# this imports gettext ( equivalent to Wx::GetTranslation
# and gettext_noop ( a noop )
# into your module
use Wx::Locale qw(:default);

# ....

# use the functions
print gettext( ``Panic!'' );

button = Wx::Button->new( window, -1, gettext( ``Label'' ) );
```

If you need to translate a lot of strings, then adding gettext( ) around each one is a long task (that is why _( ) was introduced), so just choose a shorter name for gettext:

```
#
use Wx::Locale 'gettext' => 't',
               'gettext_noop' => 'gettext_noop';

# ...

# use the functions
print t( ``Panic!!'' );

# ...
```

Derived from

No base class

See also

Internationalization overview (p. 1736),
Internat sample (p. 1712)

Include files

<wx/intl.h>

Supported languages

See *list of recognized language constants* (p. 1671). These constants may be used to

specify the language in *Init* (p. 888) and are returned by *GetSystemLanguage* (p. 887):

wxLocale::wxLocale

wxLocale()

This is the default constructor and it does nothing to initialize the object: *Init()* (p. 888) must be used to do that.

wxLocale(**int** *language*, **int** *flags* =   *wxLOCALE_LOAD_DEFAULT |* *wxLOCALE_CONV_ENCODING*)

See *Init()* (p. 888) for parameters description.

wxLocale(**const char** **szName*, **const char** **szShort = NULL*, **const char** **szLocale =* *NULL*, **bool** *bLoadDefault = true*, **bool** *bConvertEncoding = false*)

See *Init()* (p. 888) for parameters description.

The call of this function has several global side effects which you should understand: first of all, the application locale is changed - note that this will affect many of standard C library functions such as printf() or strftime(). Second, this wxLocale object becomes the new current global locale for the application and so all subsequent calls to wxGetTranslation() will try to translate the messages using the message catalogs for this locale.

wxLocale::~wxLocale

~wxLocale()

The destructor, like the constructor, also has global side effects: the previously set locale is restored and so the changes described in *Init* (p. 888) documentation are rolled back.

wxLocale::AddCatalog

bool AddCatalog(**const char** **szDomain*)

bool AddCatalog(**const char** **szDomain*, **wxLanguage***msgIdLanguage*, **const char** **msgIdCharset*)

Add a catalog for use with the current locale: it is searched for in standard places (current directory first, then the system one), but you may also prepend additional directories to the search path with *AddCatalogLookupPathPrefix()* (p. 884).

All loaded catalogs will be used for message lookup by*GetString()* (p. 886) for the current locale.

Returns true if catalog was successfully loaded, false otherwise (which might mean that the catalog is not found or that it isn't in the correct format).

The second form of this method takes two additional arguments,*msgIdLanguage* and *msgIdCharset*.

msgIdLanguage specifies the language of "msgid" strings in source code (i.e. arguments to *GetString* (p. 886),*wxGetTranslation* (p. 1610) and the *_()* (p. 1613) macro). It is used if AddCatalog cannot find any catalog for current language: if the language is same as source code language, then strings from source code are used instead.

msgIdCharset lets you specify the charset used for msgids in sources in case they use 8-bit characters (e.g. German or French strings). This argument has no effect in Unicode build, because literals in sources are Unicode strings; you have to use compiler-specific method of setting the right charset when compiling with Unicode.

By default (i.e. when you use the first form), msgid strings are assumed to be in English and written only using 7-bit ASCII characters.

If you have to deal with non-English strings or 8-bit characters in the source code, see the instructions in *Writing non-English applications* (p. 1737).

wxLocale::AddCatalogLookupPathPrefix

void AddCatalogLookupPathPrefix(**const wxString&** *prefix*)

Add a prefix to the catalog lookup path: the message catalog files will be looked up under prefix/<lang>/LC_MESSAGES, prefix/<lang> and prefix (in this order).

This only applies to subsequent invocations of AddCatalog().

wxLocale::AddLanguage

static void AddLanguage(**const wxLanguageInfo&** *info*)

Adds custom, user-defined language to the database of known languages. This database is used in conjunction with the first form of *Init* (p. 888).

wxLanguageInfo is defined as follows:

```
struct WXDLLEXPORT wxLanguageInfo
{
    int Language;                    // wxLanguage id
    wxString CanonicalName;          // Canonical name, e.g. fr_FR
#ifdef __WIN32__
    wxUint32 WinLang, WinSublang;    // Win32 language identifiers
                                     // (LANG_xxxx, SUBLANG_xxxx)
#endif
    wxString Description;            // human-readable name of the language
};
```

Language should be greater than wxLANGUAGE_USER_DEFINED.

wxPerl note: In wxPerl Wx::LanguageInfo has only one method:

Wx::LanguageInfo->new(language, canonicalName, WinLang, WinSubLang,

Description)

wxLocale::FindLanguageInfo

static wxLanguageInfo * FindLanguageInfo(**const wxString&** *locale*)

This function may be used to find the language description structure for the given locale, specified either as a two letter ISO language code (for example, "pt"), a language code followed by the country code ("pt_BR") or a full, human readable, language description ("Portuguese-Brazil").

Returns the information for the given language or NULL if this language is unknown. Note that even if the returned pointer is valid, the caller should*not* delete it.

See also

GetLanguageInfo (p. 885)

wxLocale::GetCanonicalName

wxString GetCanonicalName() **const**

Returns the canonical form of current locale name. Canonical form is the one that is used on UNIX systems: it is a two- or five-letter string in xx or xx_YY format, where xx is ISO 639 code of language and YY is ISO 3166 code of the country. Examples are "en", "en_GB", "en_US" or "fr_FR".

This form is internally used when looking up message catalogs.

Compare *GetSysName* (p. 887).

wxLocale::GetLanguage

int GetLanguage() **const**

Returns *wxLanguage* (p. 882) constant of current language. Note that you can call this function only if you used the form of*Init* (p. 888) that takes wxLanguage argument.

wxLocale::GetLanguageInfo

static wxLanguageInfo * GetLanguageInfo(**int** *lang*) **const**

Returns a pointer to wxLanguageInfo structure containing information about the given language or NULL if this language is unknown. Note that even if the returned pointer is valid, the caller should *not* delete it.

See *AddLanguage* (p. 884) for the wxLanguageInfo description.

As with *Init* (p. 888), wxLANGUAGE_DEFAULT has the special meaning if passed as an

argument to this function and in this case the result of *GetSystemLanguage()* (p. 887) is used.

wxLocale::GetLanguageName

static wxString GetLanguageName(int *lang*) const

Returns English name of the given language or empty string if this language is unknown.

See *GetLanguageInfo* (p. 885) for a remark about special meaning of `wxLANGUAGE_DEFAULT`.

wxLocale::GetLocale

const char* GetLocale() const

Returns the locale name as passed to the constructor or *Init()* (p. 888). This is full, human-readable name, e.g. "English" or "French".

wxLocale::GetName

const wxString& GetName() const

Returns the current short name for the locale (as given to the constructor or the Init() function).

wxLocale::GetString

const char* GetString(const char *szOrigString*, const char *szDomain = NULL*) const

const char* GetString(const char *szOrigString*, const char *szOrigString2*, size_t *n*, const char *szDomain = NULL*) const

Retrieves the translation for a string in all loaded domains unless the szDomain parameter is specified (and then only this catalog/domain is searched).

Returns original string if translation is not available (in this case an error message is generated the first time a string is not found; use *wxLogNull* (p. 1743) to suppress it).

The second form is used when retrieving translation of string that has different singular and plural form in English or different plural forms in some other language. It takes two extra arguments: *szOrigString*parameter must contain the singular form of the string to be converted. It is also used as the key for the search in the catalog. The *szOrigString2* parameter is the plural form (in English). The parameter *n* is used to determine the plural form. If no message catalog is found *szOrigString* is returned if 'n == 1', otherwise *szOrigString2*. See GNU gettext manual
(`http://www.gnu.org/manual/gettext/html_chapter/gettext_10.html#S`

EC150) for additional information on plural forms handling.

This method is called by the *wxGetTranslation* (p. 1610)function and *_()* (p. 1613) macro.

Remarks

Domains are searched in the last to first order, i.e. catalogs added later override those added before.

wxLocale::GetHeaderValue

wxString GetHeaderValue(**const char** **szHeader*, **const char** **szDomain = NULL*) **const**

Returns the header value for header *szHeader*. The search for *szHeader* is case sensitive. If an *szDomain*is passed, this domain is searched. Else all domains will be searched until a header has been found. The return value is the value of the header if found. Else this will be empty.

wxLocale::GetSysName

wxString GetSysName() **const**

Returns current platform-specific locale name as passed to setlocale().

Compare *GetCanonicalName* (p. 885).

wxLocale::GetSystemEncoding

static wxFontEncoding GetSystemEncoding() **const**

Tries to detect the user's default font encoding. Returns *wxFontEncoding* (p. 558) value or **wxFONTENCODING_SYSTEM** if it couldn't be determined.

wxLocale::GetSystemEncodingName

static wxString GetSystemEncodingName() **const**

Tries to detect the name of the user's default font encoding. This string isn't particularly useful for the application as its form is platform-dependent and so you should probably use *GetSystemEncoding* (p. 887) instead.

Returns a user-readable string value or an empty string if it couldn't be determined.

wxLocale::GetSystemLanguage

static int GetSystemLanguage() **const**

Tries to detect the user's default language setting. Returns *wxLanguage* (p. 882) value or **wxLANGUAGE_UNKNOWN** if the language-guessing algorithm failed.

wxLocale::Init

bool Init(**int** *language = wxLANGUAGE_DEFAULT*, **int** *flags = wxLOCALE_LOAD_DEFAULT | wxLOCALE_CONV_ENCODING*)

bool Init(**const char** **szName*, **const char** **szShort = NULL*, **const char** **szLocale = NULL*, **bool** *bLoadDefault = true*, **bool** *bConvertEncoding = false*)

The second form is deprecated, use the first one unless you know what you are doing.

Parameters

language

> *wxLanguage* (p. 882) identifier of the locale. wxLANGUAGE_DEFAULT has special meaning -- wxLocale will use system's default language (see *GetSystemLanguage* (p. 887)).

flags

> Combination of the following:

> **wxLOCALE_LOAD_DEFAULT**   Load the message catalog for the given locale containing the translations of standard wxWidgets messages automatically.

> **wxLOCALE_CONV_ENCODING**   Automatically convert message catalogs to platform's default encoding. Note that it will do only basic conversion between well-known pair like iso8859-1 and windows-1252 or iso8859-2 and windows-1250. See *Writing non-English applications* (p. 1737) for detailed description of this behaviour. Note that this flag is meaningless in Unicode build.

szName

> The name of the locale. Only used in diagnostic messages.

szShort

> The standard 2 letter locale abbreviation; it is used as the directory prefix when looking for the message catalog files.

szLocale

> The parameter for the call to setlocale(). Note that it is platform-specific.

bLoadDefault

> May be set to false to prevent loading of the message catalog for the given locale containing the translations of standard wxWidgets messages. This parameter would be rarely used in normal circumstances.

bConvertEncoding

> May be set to true to do automatic conversion of message catalogs to platform's native encoding. Note that it will do only basic conversion between well-known pair like iso8859-1 and windows-1252 or iso8859-2 and windows-1250. See *Writing non-English applications* (p. 1737) for detailed description of this behaviour.

The call of this function has several global side effects which you should understand: first of all, the application locale is changed - note that this will affect many of standard C library functions such as printf() or strftime(). Second, this wxLocale object becomes the new current global locale for the application and so all subsequent calls to *wxGetTranslation()* (p. 1610) will try to translate the messages using the message catalogs for this locale.

Returns true on success or false if the given locale couldn't be set.

wxLocale::IsLoaded

bool IsLoaded(**const char*** *domain*) **const**

Check if the given catalog is loaded, and returns true if it is.

According to GNU gettext tradition, each catalog normally corresponds to 'domain' which is more or less the application name.

See also: *AddCatalog* (p. 883)

wxLocale::IsOk

bool IsOk() **const**

Returns true if the locale could be set successfully.

wxLog

wxLog class defines the interface for the *log targets* used by wxWidgets logging functions as explained in the *wxLog overview* (p. 1743). The only situations when you need to directly use this class is when you want to derive your own log target because the existing ones don't satisfy your needs. Another case is if you wish to customize the behaviour of the standard logging classes (all of which respect the wxLog settings): for example, set which trace messages are logged and which are not or change (or even remove completely) the timestamp on the messages.

Otherwise, it is completely hidden behind the *wxLogXXX()* functions and you may not even know about its existence.

See *log overview* (p. 1743) for the descriptions of wxWidgets logging facilities.

Derived from

No base class

Include files

<wx/log.h>

Static functions

The functions in this section work with and manipulate the active log target. The *OnLog()* (p. 893) is called by the *wxLogXXX()* functions and invokes the *DoLog()* (p. 894) of the active log target if any. Get/Set methods are used to install/query the current active target and, finally, *DontCreateOnDemand()* (p. 894) disables the automatic creation of a standard log target if none actually exists. It is only useful when the application is terminating and shouldn't be used in other situations because it may easily lead to a loss of messages.

OnLog (p. 893)
GetActiveTarget (p. 893)
SetActiveTarget (p. 893)
DontCreateOnDemand (p. 894)
Suspend (p. 893)
Resume (p. 893)

Logging functions

There are two functions which must be implemented by any derived class to actually process the log messages: *DoLog* (p. 894) and *DoLogString* (p. 894). The second function receives a string which just has to be output in some way and the easiest way to write a new log target is to override just this function in the derived class. If more control over the output format is needed, then the first function must be overridden which allows to construct custom messages depending on the log level or even do completely different things depending on the message severity (for example, throw away all messages except warnings and errors, show warnings on the screen and forward the error messages to the user's (or programmer's) cell phone - maybe depending on whether the timestamp tells us if it is day or night in the current time zone).

There also functions to support message buffering. Why are they needed? Some of wxLog implementations, most notably the standard wxLogGui class, buffer the messages (for example, to avoid showing the user a zillion of modal message boxes one after another -- which would be really annoying). *Flush()* (p. 894) shows them all and clears the buffer contents. This function doesn't do anything if the buffer is already

empty.

Flush (p. 894)
FlushActive (p. 894)

Customization

The functions below allow some limited customization of wxLog behaviour without writing a new log target class (which, aside of being a matter of several minutes, allows you to do anything you want).

The verbose messages are the trace messages which are not disabled in the release mode and are generated by *wxLogVerbose* (p. 1650). They are not normally shown to the user because they present little interest, but may be activated, for example, in order to help the user find some program problem.

As for the (real) trace messages, their handling depends on the settings of the (application global) *trace mask*. There are two ways to specify it: either by using *SetTraceMask* (p. 895) and *GetTraceMask* (p. 896) and using *wxLogTrace* (p. 1651) which takes an integer mask or by using *AddTraceMask* (p. 892) for string trace masks.

The difference between bit-wise and string trace masks is that a message using integer trace mask will only be logged if all bits of the mask are set in the current mask while a message using string mask will be logged simply if the mask had been added before to the list of allowed ones.

For example,

```
// wxTraceOleCalls is one of standard bit masks
wxLogTrace(wxTraceRefCount | wxTraceOleCalls, "Active object ref count:
%d", nRef);
```

will do something only if the current trace mask contains both `wxTraceRefCount` and `wxTraceOle`, but

```
// wxTRACE_OleCalls is one of standard string masks
wxLogTrace(wxTRACE_OleCalls, "IFoo::Bar() called");
```

will log the message if it was preceded by

```
wxLog::AddTraceMask(wxTRACE_OleCalls);
```

Using string masks is simpler and allows to easily add custom ones, so this is the preferred way of working with trace messages. The integer trace mask is kept for compatibility and for additional (but very rarely needed) flexibility only.

The standard trace masks are given in *wxLogTrace* (p. 1651) documentation.

Finally, the *wxLog::DoLog()* function automatically prepends a time stamp to all the messages. The format of the time stamp may be changed: it can be any string with % specifications fully described in the documentation of the standard *strftime()* function. For example, the default format is "[%d/%b/%y %H:%M:%S] " which gives something like "[17/Sep/98 22:10:16] " (without quotes) for the current date. Setting an empty string as

the time format disables timestamping of the messages completely.

NB: Timestamping is disabled for Visual C++ users in debug builds by default because otherwise it would be impossible to directly go to the line from which the log message was generated by simply clicking in the debugger window on the corresponding error message. If you wish to enable it, please use*SetTimestamp* (p. 895) explicitly.

AddTraceMask (p. 892)
RemoveTraceMask (p. 896)
ClearTraceMasks (p. 892)
GetTraceMasks (p. 892)
IsAllowedTraceMask (p. 896)
SetVerbose (p. 895)
GetVerbose (p. 895)
SetTimestamp (p. 895)
GetTimestamp (p. 895)
SetTraceMask (p. 895)
GetTraceMask (p. 896)

wxLog::AddTraceMask

static void AddTraceMask(**const wxString&** *mask*)

Add the *mask* to the list of allowed masks for*wxLogTrace* (p. 1651).

See also

RemoveTraceMask (p. 896)*GetTraceMasks* (p. 892)

wxLog::ClearTraceMasks

static void ClearTraceMasks()

Removes all trace masks previously set with*AddTraceMask* (p. 892).

See also

RemoveTraceMask (p. 896)

wxLog::GetTraceMasks

static const wxArrayString & GetTraceMasks()

Returns the currently allowed list of string trace masks.

See also

AddTraceMask (p. 892).

wxLog::OnLog

static void OnLog(**wxLogLevel** *level*, **const char *** *message*)

Forwards the message at specified level to the *DoLog()* function of the active log target if there is any, does nothing otherwise.

wxLog::GetActiveTarget

static wxLog * GetActiveTarget()

Returns the pointer to the active log target (may be NULL).

wxLog::SetActiveTarget

static wxLog * SetActiveTarget(**wxLog *** *logtarget*)

Sets the specified log target as the active one. Returns the pointer to the previous active log target (may be NULL). To supress logging use a new instance of wxLogNull not NULL. If the active log target is set to NULL a new default log target will be created when logging occurs.

wxLog::Suspend

static void Suspend()

Suspends the logging until *Resume* (p. 893) is called. Note that the latter must be called the same number of times as the former to undo it, i.e. if you call Suspend() twice you must call Resume() twice as well.

Note that suspending the logging means that the log sink won't be be flushed periodically, it doesn't have any effect if the current log target does the logging immediately without waiting for *Flush* (p. 894) to be called (the standard GUI log target only shows the log dialog when it is flushed, so Suspend() works as expected with it).

See also

Resume (p. 893),
wxLogNull (p. 1743)

wxLog::Resume

static void Resume()

Resumes logging previously suspended by a call to*Suspend* (p. 893). All messages logged in the meanwhile will be flushed soon.

wxLog::DoLog

virtual void DoLog(**wxLogLevel** *level*, **const wxChar** **msg*, **time_t** *timestamp*)

Called to process the message of the specified severity. *msg* is the text of the message as specified in the call of *wxLogXXX()* function which generated it and *timestamp* is the moment when the message was generated.

The base class version prepends the timestamp to the message, adds a prefix corresponding to the log level and then calls*DoLogString* (p. 894) with the resulting string.

wxLog::DoLogString

virtual void DoLogString(**const wxChar** **msg*, **time_t** *timestamp*)

Called to log the specified string. The timestamp is already included into the string but still passed to this function.

A simple implementation may just send the string to `stdout` or, better,`stderr`.

wxLog::DontCreateOnDemand

static void DontCreateOnDemand()

Instructs wxLog to not create new log targets on the fly if there is none currently. (Almost) for internal use only: it is supposed to be called by the application shutdown code.

Note that this function also calls*ClearTraceMasks* (p. 892).

wxLog::Flush

virtual void Flush()

Shows all the messages currently in buffer and clears it. If the buffer is already empty, nothing happens.

wxLog::FlushActive

static void FlushActive()

Flushes the current log target if any, does nothing if there is none.

See also

Flush (p. 894)

wxLog::SetVerbose

static void SetVerbose(**bool** *verbose = true*)

Activates or deactivates verbose mode in which the verbose messages are logged as the normal ones instead of being silently dropped.

wxLog::GetVerbose

static bool GetVerbose()

Returns whether the verbose mode is currently active.

wxLog::SetLogLevel

static void SetLogLevel(**wxLogLevel** *logLevel*)

Specifies that log messages with level > logLevel should be ignored and not sent to the active log target.

wxLog::GetLogLevel

static wxLogLevel GetLogLevel()

Returns the current log level limit.

wxLog::SetTimestamp

void SetTimestamp(**const char *** *format*)

Sets the timestamp format prepended by the default log targets to all messages. The string may contain any normal characters as well as %prefixed format specificators, see *strftime()* manual for details. Passing a NULL value (not empty string) to this function disables message timestamping.

wxLog::GetTimestamp

const char * GetTimestamp() **const**

Returns the current timestamp format string.

wxLog::SetTraceMask

static void SetTraceMask(**wxTraceMask** *mask*)

Sets the trace mask, see *Customization* (p. 891)section for details.

wxLog::GetTraceMask

Returns the current trace mask, see *Customization* (p. 891) section for details.

wxLog::IsAllowedTraceMask

static bool IsAllowedTraceMask(**const wxChar ***mask*)

Returns true if the *mask* is one of allowed masks for*wxLogTrace* (p. 1651).

See also: *AddTraceMask* (p. 892),*RemoveTraceMask* (p. 896)

wxLog::RemoveTraceMask

static void RemoveTraceMask(**const wxString&** *mask*)

Remove the *mask* from the list of allowed masks for*wxLogTrace* (p. 1651).

See also: *AddTraceMask* (p. 892)

wxLogChain

This simple class allows to chain log sinks, that is to install a new sink but keep passing log messages to the old one instead of replacing it completely as*SetActiveTarget* (p. 893) does.

It is especially useful when you want to divert the logs somewhere (for example to a file or a log window) but also keep showing the error messages using the standard dialogs as *wxLogGui* (p. 1743) does by default.

Example of usage:

```
wxLogChain *logChain = new wxLogChain(new wxLogStderr);

// all the log messages are sent to stderr and also processed as usually
...

// don't delete logChain directly as this would leave a dangling
// pointer as active log target, use SetActiveTarget() instead
delete wxLog::SetActiveTarget(...something else or NULL...);
```

Derived from

wxLog (p. 889)

Include files

<wx/log.h>

wxLogChain::wxLogChain

wxLogChain(**wxLog ****logger*)

Sets the specified `logger` (which may be `NULL`) as the default log target but the log messages are also passed to the previous log target if any.

wxLogChain::~wxLogChain

~wxLogChain()

Destroys the previous log target.

wxLogChain::GetOldLog

wxLog * GetOldLog() **const**

Returns the pointer to the previously active log target (which may be `NULL`).

wxLogChain::IsPassingMessages

bool IsPassingMessages() **const**

Returns `true` if the messages are passed to the previously active log target (default) or `false` if *PassMessages* (p. 897)had been called.

wxLogChain::PassMessages

void PassMessages(**bool** *passMessages*)

By default, the log messages are passed to the previously active log target. Calling this function with `false` parameter disables this behaviour (presumably temporarily, as you shouldn't use wxLogChain at all otherwise) and it can be reenabled by calling it again with *passMessages* set to `true`.

wxLogChain::SetLog

void SetLog(**wxLog ****logger*)

Sets another log target to use (may be `NULL`). The log target specified in the *constructor* (p. 897) or in a previous call to this function is deleted.

This doesn't change the old log target value (the one the messages are forwarded to) which still remains the same as was active when wxLogChain object was created.

wxLogGui

This is the default log target for the GUI wxWidgets applications. It is passed to *wxLog::SetActiveTarget* (p. 893) at the program startup and is deleted by wxWidgets during the program shut down.

Derived from

wxLog (p. 889)

Include files

<wx/log.h>

wxLogGui::wxLogGui

wxLogGui()

Default constructor.

wxLogNull

This class allows to temporarily suspend logging. All calls to the log functions during the life time of an object of this class are just ignored.

In particular, it can be used to suppress the log messages given by wxWidgets itself but it should be noted that it is rarely the best way to cope with this problem as **all** log messages are suppressed, even if they indicate a completely different error than the one the programmer wanted to suppress.

For instance, the example of the overview:

```
wxFile file;

// wxFile.Open() normally complains if file can't be opened, we don't
want it
 {
   wxLogNull logNo;
   if ( !file.Open("bar") )
     ... process error ourselves ...
 } // ~wxLogNull called, old log sink restored

wxLogMessage("..."); // ok
```

would be better written as:

```
wxFile file;

// don't try to open file if it doesn't exist, we are prepared to deal
with
 // this ourselves - but all other errors are not expected
```

```
if ( wxFile::Exists("bar") )
{
    // gives an error message if the file couldn't be opened
    file.Open("bar");
}
else
{
    ...
}
```

Derived from

wxLog (p. 889)

Include files

<wx/log.h>

wxLogNull::wxLogNull

wxLogNull()

Suspends logging.

wxLogNull::~wxLogNull

Resumes logging.

wxLogPassThrough

A special version of *wxLogChain* (p. 896) which uses itself as the new log target. Maybe more clearly, it means that this is a log target which forwards the log messages to the previously installed one in addition to processing them itself.

Unlike *wxLogChain* (p. 896) which is usually used directly as is, this class must be derived from to implement *DoLog* (p. 894)and/or *DoLogString* (p. 894) methods.

Derived from

wxLogChain (p. 896)

Include files

<wx/log.h>

wxLogPassThrough::wxLogPassThrough

Default ctor installs this object as the current active log target.

wxLogStderr

This class can be used to redirect the log messages to a C file stream (not to be confused with C++ streams). It is the default log target for the non-GUI wxWidgets applications which send all the output to stderr.

Derived from

wxLog (p. 889)

Include files

<wx/log.h>

See also

wxLogStream (p. 900)

wxLogStderr::wxLogStderr

 wxLogStderr(**FILE** **fp = NULL*)

Constructs a log target which sends all the log messages to the given FILE. If it is NULL, the messages are sent to stderr.

wxLogStream

This class can be used to redirect the log messages to a C++ stream.

Please note that this class is only available if wxWidgets was compiled with the standard iostream library support (wxUSE_STD_IOSTREAM must be on).

Derived from

wxLog (p. 889)

Include files

<wx/log.h>

See also

wxLogStderr (p. 900),
wxStreamToTextRedirector (p. 1283)

wxLogStream::wxLogStream

wxLogStream(**std::ostream** **ostr = NULL*)

Constructs a log target which sends all the log messages to the given output stream. If it is NULL, the messages are sent to cerr.

wxLogTextCtrl

Using these target all the log messages can be redirected to a text control. The text control must have been created with wxTE_MULTILINE style by the caller previously.

Derived from

wxLog (p. 889)

Include files

<wx/log.h>

See also

wxLogTextCtrl (p. 901),
wxStreamToTextRedirector (p. 1283)

wxLogTextCtrl::wxLogTextCtrl

wxLogTextCtrl(**wxTextCtrl** **textctrl*)

Constructs a log target which sends all the log messages to the given text control. The *textctrl* parameter cannot be NULL.

wxLogWindow

This class represents a background log window: to be precise, it collects all log messages in the log frame which it manages but also passes them on to the log target which was active at the moment of its creation. This allows, for example, to show all the log messages in a frame but still continue to process them normally by showing the standard log dialog.

Derived from

wxLogPassThrough (p. 899)

Include files

<wx/log.h>

See also

wxLogTextCtrl (p. 901)

wxLogWindow::wxLogWindow

 wxLogWindow(**wxFrame** **parent*, **const wxChar** **title*, **bool** *show* = `true`, **bool** *passToOld* = `true`)

Creates the log frame window and starts collecting the messages in it.

Parameters

parent

The parent window for the log frame, may be `NULL`

title

The title for the log frame

show

`true` to show the frame initially (default), otherwise*wxLogWindow::Show* (p. 902) must be called later.

passToOld

`true` to process the log messages normally in addition to logging them in the log frame (default), `false` to only log them in the log frame.

wxLogWindow::Show

void Show(**bool** *show* = `true`)

Shows or hides the frame.

wxLogWindow::GetFrame

wxFrame * GetFrame() **const**

Returns the associated log frame window. This may be used to position or resize it but use *wxLogWindow::Show* (p. 902) to show or hide it.

wxLogWindow::OnFrameCreate

virtual void OnFrameCreate(**wxFrame** **frame*)

Called immediately after the log frame creation allowing for any extra initializations.

wxLogWindow::OnFrameClose

virtual bool OnFrameClose(**wxFrame** **frame*)

Called if the user closes the window interactively, will not be called if it is destroyed for another reason (such as when program exits).

Return `true` from here to allow the frame to close, `false` to prevent this from happening.

See also

wxLogWindow::OnFrameDelete (p. 903)

wxLogWindow::OnFrameDelete

virtual void OnFrameDelete(**wxFrame** **frame*)

Called right before the log frame is going to be deleted: will always be called unlike *OnFrameClose()* (p. 903).

wxLongLong

This class represents a signed 64 bit long number. It is implemented using the native 64 bit type where available (machines with 64 bit longs or compilers which have (an analog of) *long long* type) and uses the emulation code in the other cases which ensures that it is the most efficient solution for working with 64 bit integers independently of the architecture.

wxLongLong defines all usual arithmetic operations such as addition, subtraction, bitwise shifts and logical operations as well as multiplication and division (not yet for the machines without native *long long*). It also has operators for implicit construction from and conversion to the native *long long* type if it exists and *long*.

You would usually use this type in exactly the same manner as any other (built-in) arithmetic type. Note that wxLongLong is a signed type, if you want unsigned values use wxULongLong which has exactly the same API as wxLongLong except when explicitly mentioned otherwise.

If a native (i.e. supported directly by the compiler) 64 bit integer type was found to exist, *wxLongLong_t* macro will be defined to correspond to it. Also, in this case only, two additional macros will be defined: *wxLongLongFmtSpec* (p. 1632) for printing 64 bit integers using the standard `printf()` function (but see also *ToString()* (p. 905) for a more portable solution) and *wxLL* (p. 1632) for defining 64 bit integer compile-time constants.

Derived from

No base class

Include files

<wx/longlong.h>

wxLongLong::wxLongLong

wxLongLong()

Default constructor initializes the object to 0.

wxLongLong::wxLongLong

wxLongLong(**wxLongLong_t** *ll*)

Constructor from native long long (only for compilers supporting it).

wxLongLong::wxLongLong

wxLongLong(**long** *hi*, **unsigned long** *lo*)

Constructor from 2 longs: the high and low part are combined into one wxLongLong.

wxLongLong::operator=

wxLongLong& operator operator=(**wxLongLong_t** *ll*)

Assignment operator from native long long (only for compilers supporting it).

wxLongLong::Abs

wxLongLong Abs() **const**

wxLongLong& Abs()

Returns an absolute value of wxLongLong - either making a copy (const version) or modifying it in place (the second one). Not in wxULongLong.

wxLongLong::Assign

wxLongLong& Assign(**double** *d*)

This allows to convert a double value to wxLongLong type. Such conversion is not

always possible in which case the result will be silently truncated in a platform-dependent way. Not in wxULongLong.

wxLongLong::GetHi

long GetHi() **const**

Returns the high 32 bits of 64 bit integer.

wxLongLong::GetLo

unsigned long GetLo() **const**

Returns the low 32 bits of 64 bit integer.

wxLongLong::GetValue

wxLongLong_t GetValue() **const**

Convert to native long long (only for compilers supporting it)

wxLongLong::ToLong

long ToLong() **const**

Truncate wxLongLong to long. If the conversion loses data (i.e. the wxLongLong value is outside the range of built-in long type), an assert will be triggered in debug mode.

wxLongLong::ToString

wxString ToString() **const**

Returns the string representation of a wxLongLong.

wxLongLong::operator+

wxLongLong operator+(**const wxLongLong&** *ll*) **const**

Adds 2 wxLongLongs together and returns the result.

wxLongLong::operator+=

wxLongLong& operator+(**const wxLongLong&** *ll*)

Add another wxLongLong to this one.

wxLongLong::operator++

wxLongLong& operator++()

wxLongLong& operator++(**int**)

Pre/post increment operator.

wxLongLong::operator-

wxLongLong operator-() **const**

Returns the value of this wxLongLong with opposite sign. Not in wxULongLong.

wxLongLong::operator-

wxLongLong operator-(**const wxLongLong&** *ll*) **const**

Subtracts 2 wxLongLongs and returns the result.

wxLongLong::operator-=

wxLongLong& operator-(**const wxLongLong&** *ll*)

Subtracts another wxLongLong from this one.

wxLongLong::operator--

wxLongLong& operator--()

wxLongLong& operator--(**int**)

Pre/post decrement operator.

wxMask

This class encapsulates a monochrome mask bitmap, where the masked area is black and the unmasked area is white. When associated with a bitmap and drawn in a device context, the unmasked area of the bitmap will be drawn, and the masked area will not be drawn.

Derived from

wxObject (p. 1012)

Include files

<wx/bitmap.h>

Remarks

A mask may be associated with a *wxBitmap* (p. 79). It is used in *wxDC::Blit* (p. 367) when the source device context is a *wxMemoryDC* (p. 935) with wxBitmap selected into it that contains a mask.

See also

wxBitmap (p. 79), *wxDC::Blit* (p. 367), *wxMemoryDC* (p. 935)

wxMask::wxMask

wxMask()

Default constructor.

wxMask(**const** *wxBitmap* **(p. 79)&** *bitmap*)

Constructs a mask from a monochrome bitmap.

wxPython note: This is the default constructor for wxMask in wxPython.

wxMask(**const** *wxBitmap* **(p. 79)&** *bitmap*, **const** *wxColour* **(p. 164)&** *colour*)

Constructs a mask from a bitmap and a colour that indicates the background.

wxPython note: wxPython has an alternate wxMask constructor matching this form called `wxMaskColour`.

wxMask(**const wxBitmap&** *bitmap*, **int** *index*)

Constructs a mask from a bitmap and a palette index that indicates the background. Not yet implemented for GTK.

Parameters

bitmap

A valid bitmap.

colour

A colour specifying the transparency RGB values.

index

Index into a palette, specifying the transparency colour.

wxMask::~wxMask

~wxMask()

Destroys the wxMask object and the underlying bitmap data.

wxMask::Create

bool Create(**const wxBitmap&** *bitmap*)

Constructs a mask from a monochrome bitmap.

bool Create(**const wxBitmap&** *bitmap*, **const wxColour&** *colour*)

Constructs a mask from a bitmap and a colour that indicates the background.

bool Create(**const wxBitmap&** *bitmap*, **int** *index*)

Constructs a mask from a bitmap and a palette index that indicates the background. Not yet implemented for GTK.

Parameters

bitmap

A valid bitmap.

colour

A colour specifying the transparency RGB values.

index

Index into a palette, specifying the transparency colour.

wxMaximizeEvent

An event being sent when the frame is maximized or restored.

Derived from

wxEvent (p. 483)
wxObject (p. 1012)

Include files

<wx/event.h>

Event table macros

To process a maximize event, use this event handler macro to direct input to a member function that takes a wxMaximizeEvent argument.

EVT_MAXIMIZE(func)   Process a wxEVT_MAXIMIZE event.

Event handling overview (p. 1751), *wxTopLevelWindow::Maximize* (p. 1421),
wxTopLevelWindow::IsMaximized (p. 1421)

wxMaximizeEvent::wxMaximizeEvent

wxMaximizeEvent(**int** *id = 0*)

Constructor.

wxMBConv

This class is the base class of a hierarchy of classes capable of converting text strings
between multibyte (SBCS or DBCS) encodings and Unicode. It is itself a wrapper around
the standard libc mbstowcs() and wcstombs() routines, and has one predefined instance,
wxConvLibc.

Derived from

No base class

Include files

<wx/strconv.h>

See also

wxCSConv (p. 225), *wxEncodingConverter* (p. 478), *wxMBConv classes overview* (p.
1733)

wxMBConv::wxMBConv

wxMBConv()

Constructor.

wxMBConv::MB2WC

virtual size_t MB2WC(**wchar_t ****outputBuf*, **const char ****psz*, **size_t** *outputSize*) **const**

Converts from a string *psz* in multibyte encoding to Unicode putting the output into the
buffer *outputBuf* of the maximum size *outputSize* (in wide characters, not bytes). If
outputBuf is NULL, only the length of the string which would result from the conversion is
calculated and returned. Note that this is the length and not size, i.e. the returned value

does **not** include the trailing NUL. But when the function is called with a non-NULL *outputBuf*, the *outputSize* parameter should be the size of the buffer and so it **should** take into account the trailing NUL.

Parameters

outputBuf

> the output buffer, may be NULL if the caller is only interested in the length of the resulting string

psz

> the NUL-terminated input string, cannot be NULL

outputSize

> the size of the output buffer (in wide characters, **including** the NUL) , ignored if *outputBuf* is NULL

Return value

The length of the converted string (in wide characters, **excluding** the NUL)

wxMBConv::WC2MB

virtual size_t WC2MB(**char*** *buf*, **const wchar_t*** *psz*, **size_t** *n*) **const**

Converts from Unicode to multibyte encoding. The semantics of this function (including the return value meaning) is the same as for *MB2WC* (p. 909).

Notice that when the function is called with a non-NULL buffer, the *n* parameter should be the size of the buffer and so it **should** take into account the trailing NUL, which might take two or four bytes for some encodings (UTF-16 and UTF-32).

wxMBConv::cMB2WC

const wxWCharBuffer cMB2WC(**const char*** *psz*) **const**

Converts from multibyte encoding to Unicode by calling MB2WC, allocating a temporary wxWCharBuffer to hold the result.

wxMBConv::cWC2MB

const wxCharBuffer cWC2MB(**const wchar_t*** *psz*) **const**

Converts from Unicode to multibyte encoding by calling WC2MB, allocating a temporary wxCharBuffer to hold the result.

wxMBConv::cMB2WX

const char* cMB2WX(**const char*** *psz*) **const**

const wxWCharBuffer cMB2WX(**const char*** *psz*) **const**

Converts from multibyte encoding to the current wxChar type (which depends on whether wxUSE_UNICODE is set to 1). If wxChar is char, it returns the parameter unaltered. If wxChar is wchar_t, it returns the result in a wxWCharBuffer. The macro wxMB2WXbuf is defined as the correct return type (without const).

wxMBConv::cWX2MB

const char* cWX2MB(**const wxChar*** *psz*) **const**

const wxCharBuffer cWX2MB(**const wxChar*** *psz*) **const**

Converts from the current wxChar type to multibyte encoding. If wxChar is char, it returns the parameter unaltered. If wxChar is wchar_t, it returns the result in a wxCharBuffer. The macro wxWX2MBbuf is defined as the correct return type (without const).

wxMBConv::cWC2WX

const wchar_t* cWC2WX(**const wchar_t*** *psz*) **const**

const wxCharBuffer cWC2WX(**const wchar_t*** *psz*) **const**

Converts from Unicode to the current wxChar type. If wxChar is wchar_t, it returns the parameter unaltered. If wxChar is char, it returns the result in a wxCharBuffer. The macro wxWC2WXbuf is defined as the correct return type (without const).

wxMBConv::cWX2WC

const wchar_t* cWX2WC(**const wxChar*** *psz*) **const**

const wxWCharBuffer cWX2WC(**const wxChar*** *psz*) **const**

Converts from the current wxChar type to Unicode. If wxChar is wchar_t, it returns the parameter unaltered. If wxChar is char, it returns the result in a wxWCharBuffer. The macro wxWX2WCbuf is defined as the correct return type (without const).

wxMBConvFile

This class used to define the class instance **wxConvFileName**, but nowadays **wxConvFileName** is either of type wxConvLibc (on most platforms) or wxConvUTF8 (on MacOS X). **wxConvFileName** converts filenames between filesystem multibyte encoding and Unicode. **wxConvFileName** can also be set to a something else at run-

time which is used e.g. by wxGTK to use a class which checks the environment variable **G_FILESYSTEM_ENCODING** indicating that filenames should not be interpreted as UTF8 and also for converting invalid UTF8 characters (e.g. if there is a filename in iso8859_1) to strings with octal values.

Since some platforms (such as Win32) use Unicode in the filenames, and others (such as Unix) use multibyte encodings, this class should only be used directly if wxMBFILES is defined to 1. A convenience macro, wxFNCONV, is defined to wxConvFileName->cWX2MB in this case. You could use it like this:

```
wxChar *name = wxT("rawfile.doc");
FILE *fil = fopen(wxFNCONV(name), "r");
```

(although it would be better to use wxFopen(name, wxT("r")) in this case.)

Derived from

wxMBConv (p. 909)

Include files

<wx/strconv.h>

See also

wxMBConv classes overview (p. 1733)

wxMBConvFile::MB2WC

size_t MB2WC(**wchar_t*** *buf*, **const char*** *psz*, **size_t** *n*) **const**

Converts from multibyte filename encoding to Unicode. Returns the size of the destination buffer.

wxMBConvFile::WC2MB

size_t WC2MB(**char*** *buf*, **const wchar_t*** *psz*, **size_t** *n*) **const**

Converts from Unicode to multibyte filename encoding. Returns the size of the destination buffer.

wxMBConvUTF7

This class converts between the UTF-7 encoding and Unicode. It has one predefined instance, **wxConvUTF7**.

WARNING: this class is not implemented yet.

wxMBConv (p. 909)

Include files

<wx/strconv.h>

See also

wxMBConvUTF8 (p. 913), *wxMBConv classes overview* (p. 1733)

wxMBConvUTF7::MB2WC

size_t MB2WC(**wchar_t*** *buf*, **const char*** *psz*, **size_t** *n*) **const**

Converts from UTF-7 encoding to Unicode. Returns the size of the destination buffer.

wxMBConvUTF7::WC2MB

size_t WC2MB(**char*** *buf*, **const wchar_t*** *psz*, **size_t** *n*) **const**

Converts from Unicode to UTF-7 encoding. Returns the size of the destination buffer.

wxMBConvUTF8

This class converts between the UTF-8 encoding and Unicode. It has one predefined instance, **wxConvUTF8**.

Derived from

wxMBConv (p. 909)

Include files

<wx/strconv.h>

See also

wxMBConvUTF7 (p. 912), *wxMBConv classes overview* (p. 1733)

Remarks

UTF-8 is a compatibility encoding used to encode Unicode text into anything that was originally written for 8-bit strings, including (but not limited to) filenames, transfer protocols, and database fields. Notable properties include:

- Variable-length encoding able to encode up to 31 bits per character

- ASCII characters (character values under 128) are encoded as plain ASCII (1 byte per character)

- Null bytes do not occur in the encoding, except when there's an actual Unicode null character

- Preserves sort ordering for plain 8-bit comparison routines like strcmp()

- High bit patterns disambiguates character boundaries, and makes it easy to detect whether a string is encoded with UTF-8 or not

All of these properties make UTF-8 a very favorable solution in any situation where full Unicode character support is desired while remaining compatible with code written with only 8-bit extended-ASCII characters in mind.

wxMBConvUTF8::MB2WC

size_t MB2WC(**wchar_t*** *buf*, **const char*** *psz*, **size_t** *n*) **const**

Converts from UTF-8 encoding to Unicode. Returns the size of the destination buffer.

wxMBConvUTF8::WC2MB

size_t WC2MB(**char*** *buf*, **const wchar_t*** *psz*, **size_t** *n*) **const**

Converts from Unicode to UTF-8 encoding. Returns the size of the destination buffer.

wxMBConvUTF16

This class is used to convert between multibyte encodings and UTF-16 Unicode encoding (also known as UCS-2). Unlike *UTF-8* (p. 913) encoding, UTF-16 uses words and not bytes and hence depends on the byte ordering: big or little endian. Hence this class is provided in two versions: wxMBConvUTF16LE and wxMBConvUTF16BE and wxMBConvUTF16 itself is just a typedef for one of them (native for the given platform, e.g. LE under Windows and BE under Mac).

Derived from

wxMBConv (p. 909)

Include files

<wx/strconv.h>

See also

wxMBConvUTF8 (p. 913), *wxMBConvUTF32* (p. 915), *wxMBConv classes overview* (p. 1733)

wxMBConvUTF16::MB2WC

size_t MB2WC(**wchar_t*** *buf*, **const char*** *psz*, **size_t** *n*) **const**

Converts from UTF-16 encoding to Unicode. Returns the size of the destination buffer.

wxMBConvUTF16::WC2MB

size_t WC2MB(**char*** *buf*, **const wchar_t*** *psz*, **size_t** *n*) **const**

Converts from Unicode to UTF-16 encoding. Returns the size of the destination buffer.

wxMBConvUTF32

This class is used to convert between multibyte encodings and UTF-32 Unicode encoding (also known as UCS-4). Unlike *UTF-8* (p. 913) encoding, UTF-32 uses (double) words and not bytes and hence depends on the byte ordering: big or little endian. Hence this class is provided in two versions: wxMBConvUTF32LE and wxMBConvUTF32BE and wxMBConvUTF32 itself is just a typedef for one of them (native for the given platform, e.g. LE under Windows and BE under Mac).

Derived from

wxMBConv (p. 909)

Include files

<wx/strconv.h>

See also

wxMBConvUTF8 (p. 913), *wxMBConvUTF16* (p. 914), *wxMBConv classes overview* (p. 1733)

wxMBConvUTF32::MB2WC

size_t MB2WC(**wchar_t*** *buf*, **const char*** *psz*, **size_t** *n*) **const**

Converts from UTF-32 encoding to Unicode. Returns the size of the destination buffer.

wxMBConvUTF32::WC2MB

size_t WC2MB(**char*** *buf*, **const wchar_t*** *psz*, **size_t** *n*) **const**

Converts from Unicode to UTF-32 encoding. Returns the size of the destination buffer.

wxMDIChildFrame

An MDI child frame is a frame that can only exist on a *wxMDIClientWindow* (p. 919), which is itself a child of *wxMDIParentFrame* (p. 920).

Derived from

wxFrame (p. 579)
wxWindow (p. 1490)
wxEvtHandler (p. 486)
wxObject (p. 1012)

Include files

<wx/mdi.h>

Window styles

wxCAPTION	Puts a caption on the frame.
wxDEFAULT_FRAME_STYLE	Defined as **wxMINIMIZE_BOX | wxMAXIMIZE_BOX | wxTHICK_FRAME | wxSYSTEM_MENU | wxCAPTION**.
wxICONIZE	Display the frame iconized (minimized) (Windows only).
wxMAXIMIZE	Displays the frame maximized (Windows only).
wxMAXIMIZE_BOX	Displays a maximize box on the frame (Windows and Motif only).
wxMINIMIZE	Identical to **wxICONIZE**.
wxMINIMIZE_BOX	Displays a minimize box on the frame (Windows and Motif only).
wxRESIZE_BORDER	Displays a resizeable border around the window (Motif only; for Windows, it is implicit in wxTHICK_FRAME).
wxSTAY_ON_TOP	Stay on top of other windows (Windows only).
wxSYSTEM_MENU	Displays a system menu (Windows and Motif only).
wxTHICK_FRAME	Displays a thick frame around the window (Windows and Motif only).

See also *window styles overview* (p. 1762).

Remarks

Although internally an MDI child frame is a child of the MDI client window, in wxWidgets

you create it as a child of *wxMDIParentFrame* (p. 920). You can usually forget that the client window exists.

MDI child frames are clipped to the area of the MDI client window, and may be iconized on the client window.

You can associate a menubar with a child frame as usual, although an MDI child doesn't display its menubar under its own title bar. The MDI parent frame's menubar will be changed to reflect the currently active child frame. If there are currently no children, the parent frame's own menubar will be displayed.

See also

wxMDIClientWindow (p. 919), *wxMDIParentFrame* (p. 920), *wxFrame* (p. 579)

wxMDIChildFrame::wxMDIChildFrame

wxMDIChildFrame()

Default constructor.

wxMDIChildFrame(**wxMDIParentFrame*** *parent*, **wxWindowID** *id*, **const wxString&** *title*, **const wxPoint&** *pos* = *wxDefaultPosition*, **const wxSize&** *size* = *wxDefaultSize*, **long** *style* = *wxDEFAULT_FRAME_STYLE*, **const wxString&** *name* = *"frame"*)

Constructor, creating the window.

Parameters

parent

> The window parent. This should not be NULL.

id

> The window identifier. It may take a value of -1 to indicate a default value.

title

> The caption to be displayed on the frame's title bar.

pos

> The window position. A value of (-1, -1) indicates a default position, chosen by either the windowing system or wxWidgets, depending on platform.

size

> The window size. A value of (-1, -1) indicates a default size, chosen by either the windowing system or wxWidgets, depending on platform.

style

The window style. See *wxMDIChildFrame* (p. 916).

name

The name of the window. This parameter is used to associate a name with the item, allowing the application user to set Motif resource values for individual windows.

Remarks

None.

See also

wxMDIChildFrame::Create (p. 918)

wxMDIChildFrame::~wxMDIChildFrame

~wxMDIChildFrame()

Destructor. Destroys all child windows and menu bar if present.

wxMDIChildFrame::Activate

void Activate()

Activates this MDI child frame.

See also

wxMDIChildFrame::Maximize (p. 918), *wxMDIChildFrame::Restore* (p. 919)

wxMDIChildFrame::Create

bool Create(**wxWindow*** *parent*, **wxWindowID** *id*, **const wxString&** *title*, **const wxPoint&** *pos* = *wxDefaultPosition*, **const wxSize&** *size* = *wxDefaultSize*, **long** *style* = *wxDEFAULT_FRAME_STYLE*, **const wxString&** *name* = *"frame"*)

Used in two-step frame construction. See *wxMDIChildFrame::wxMDIChildFrame* (p. 917) for further details.

wxMDIChildFrame::Maximize

void Maximize(**bool***maximize*)

Maximizes this MDI child frame.

See also

wxMDIChildFrame::Activate (p. 918), *wxMDIChildFrame::Restore* (p. 919)

wxMDIChildFrame::Restore

void Restore()

Restores this MDI child frame (unmaximizes).

See also

wxMDIChildFrame::Activate (p. 918), *wxMDIChildFrame::Maximize* (p. 918)

wxMDIClientWindow

An MDI client window is a child of *wxMDIParentFrame* (p. 920), and manages zero or more *wxMDIChildFrame* (p. 916) objects.

Derived from

wxWindow (p. 1490)
wxEvtHandler (p. 486)
wxObject (p. 1012)

Include files

<wx/mdi.h>

Remarks

The client window is the area where MDI child windows exist. It doesn't have to cover the whole parent frame; other windows such as toolbars and a help window might coexist with it. There can be scrollbars on a client window, which are controlled by the parent window style.

The **wxMDIClientWindow** class is usually adequate without further derivation, and it is created automatically when the MDI parent frame is created. If the application needs to derive a new class, the function *wxMDIParentFrame::OnCreateClient* (p. 926) must be overridden in order to give an opportunity to use a different class of client window.

Under Windows 95, the client window will automatically have a sunken border style when the active child is not maximized, and no border style when a child is maximized.

See also

wxMDIChildFrame (p. 916), *wxMDIParentFrame* (p. 920), *wxFrame* (p. 579)

wxMDIClientWindow::wxMDIClientWindow

wxMDIClientWindow()

Default constructor.

wxMDIClientWindow(**wxMDIParentFrame*** *parent*, **long** *style = 0*)

Constructor, creating the window.

Parameters

parent

>   The window parent.

style

>   The window style. Currently unused.

Remarks

The second style of constructor is called within *wxMDIParentFrame::OnCreateClient* (p. 926).

See also

wxMDIParentFrame::wxMDIParentFrame (p. 922), *wxMDIParentFrame::OnCreateClient* (p. 926)

wxMDIClientWindow::~wxMDIClientWindow

~wxMDIClientWindow()

Destructor.

wxMDIClientWindow::CreateClient

bool CreateClient(**wxMDIParentFrame*** *parent*, **long** *style = 0*)

Used in two-step frame construction. See *wxMDIClientWindow::wxMDIClientWindow* (p. 919) for further details.

wxMDIParentFrame

An MDI (Multiple Document Interface) parent frame is a window which can contain MDI child frames in its own 'desktop'. It is a convenient way to avoid window clutter, and is used in many popular Windows applications, such as Microsoft Word(TM).

Derived from

wxFrame (p. 579)
wxWindow (p. 1490)

wxEvtHandler (p. 486)
wxObject (p. 1012)

Include files

<wx/mdi.h>

Remarks

There may be multiple MDI parent frames in a single application, but this probably only makes sense within programming development environments.

Child frames may be of class *wxMDIChildFrame* (p. 916) (contained within the parent frame) or *wxFrame* (p. 579) (shown as a top-level frame).

An MDI parent frame always has a *wxMDIClientWindow* (p. 919) associated with it, which is the parent for MDI client frames. This client window may be resized to accommodate non-MDI windows, as seen in Microsoft Visual C++ (TM) and Microsoft Publisher (TM), where a documentation window is placed to one side of the workspace.

MDI remains popular despite dire warnings from Microsoft itself that MDI is an obsolete user interface style.

The implementation is native in Windows, and simulated under Motif. Under Motif, the child window frames will often have a different appearance from other frames because the window decorations are simulated.

Window styles

wxCAPTION	Puts a caption on the frame.
wxDEFAULT_FRAME_STYLE	Defined as **wxMINIMIZE_BOX | wxMAXIMIZE_BOX | wxTHICK_FRAME | wxSYSTEM_MENU | wxCAPTION**.
wxHSCROLL	Displays a horizontal scrollbar in the *client window*, allowing the user to view child frames that are off the current view.
wxICONIZE	Display the frame iconized (minimized) (Windows only).
wxMAXIMIZE	Displays the frame maximized (Windows only).
wxMAXIMIZE_BOX	Displays a maximize box on the frame (Windows and Motif only).
wxMINIMIZE	Identical to **wxICONIZE**.
wxMINIMIZE_BOX	Displays a minimize box on the frame (Windows and Motif only).
wxRESIZE_BORDER	Displays a resizeable border around the window (Motif only; for Windows, it is implicit in wxTHICK_FRAME).
wxSTAY_ON_TOP	Stay on top of other windows (Windows only).

wxSYSTEM_MENU   Displays a system menu (Windows and Motif only).

wxTHICK_FRAME   Displays a thick frame around the window (Windows and
                           Motif only).

wxVSCROLL   Displays a vertical scrollbar in the *client window*, allowing
                           the user to view child frames that are off the current view.

wxFRAME_NO_WINDOW_MENU   Under Windows, removes the Window menu that is
                           normally added automatically.

See also *window styles overview* (p. 1762).

See also

wxMDIChildFrame (p. 916), *wxMDIClientWindow* (p. 919), *wxFrame* (p. 579), *wxDialog*
(p. 406)

wxMDIParentFrame::wxMDIParentFrame

 wxMDIParentFrame()

Default constructor.

 wxMDIParentFrame(**wxWindow*** *parent*, **wxWindowID** *id*, **const wxString&** *title*,
const wxPoint& *pos = wxDefaultPosition*, **const wxSize&** *size = wxDefaultSize*, **long**
style = wxDEFAULT_FRAME_STYLE | wxVSCROLL | wxHSCROLL, **const wxString&**
name = "frame")

Constructor, creating the window.

Parameters

parent

> The window parent. This should be NULL.

id

> The window identifier. It may take a value of -1 to indicate a default value.

title

> The caption to be displayed on the frame's title bar.

pos

> The window position. A value of (-1, -1) indicates a default position, chosen by
> either the windowing system or wxWidgets, depending on platform.

size

The window size. A value of (-1, -1) indicates a default size, chosen by either the windowing system or wxWidgets, depending on platform.

style

The window style. See *wxMDIParentFrame* (p. 920).

name

The name of the window. This parameter is used to associate a name with the item, allowing the application user to set Motif resource values for individual windows.

Remarks

During the construction of the frame, the client window will be created. To use a different class from *wxMDIClientWindow* (p. 919), override *wxMDIParentFrame::OnCreateClient* (p. 926).

Under Windows 95, the client window will automatically have a sunken border style when the active child is not maximized, and no border style when a child is maximized.

See also

wxMDIParentFrame::Create (p. 924), *wxMDIParentFrame::OnCreateClient* (p. 926)

wxMDIParentFrame::~wxMDIParentFrame

~wxMDIParentFrame()

Destructor. Destroys all child windows and menu bar if present.

wxMDIParentFrame::ActivateNext

void ActivateNext()

Activates the MDI child following the currently active one.

See also

wxMDIParentFrame::ActivatePrevious (p. 923)

wxMDIParentFrame::ActivatePrevious

void ActivatePrevious()

Activates the MDI child preceding the currently active one.

See also

wxMDIParentFrame::ActivateNext (p. 923)

wxMDIParentFrame::ArrangeIcons

void ArrangeIcons()

Arranges any iconized (minimized) MDI child windows.

See also

wxMDIParentFrame::Cascade (p. 924), *wxMDIParentFrame::Tile* (p. 927)

wxMDIParentFrame::Cascade

void Cascade()

Arranges the MDI child windows in a cascade.

See also

wxMDIParentFrame::Tile (p. 927), *wxMDIParentFrame::ArrangeIcons* (p. 924)

wxMDIParentFrame::Create

bool Create(**wxWindow*** *parent*, **wxWindowID** *id*, **const wxString&** *title*, **const wxPoint&** *pos* = *wxDefaultPosition*, **const wxSize&** *size* = *wxDefaultSize*, **long** *style* = *wxDEFAULT_FRAME_STYLE | wxVSCROLL | wxHSCROLL*, **const wxString&** *name* = *"frame"*)

Used in two-step frame construction. See *wxMDIParentFrame::wxMDIParentFrame* (p. 922) for further details.

wxMDIParentFrame::GetClientSize

virtual void GetClientSize(**int*** *width*, **int*** *height*) **const**

This gets the size of the frame 'client area' in pixels.

Parameters

width

      Receives the client width in pixels.

height

      Receives the client height in pixels.

Remarks

The client area is the area which may be drawn on by the programmer, excluding title bar, border, status bar, and toolbar if present.

If you wish to manage your own toolbar (or perhaps you have more than one), provide an **OnSize** event handler. Call **GetClientSize** to find how much space there is for your windows and don't forget to set the size and position of the MDI client window as well as your toolbar and other windows (but not the status bar).

If you have set a toolbar with *wxMDIParentFrame::SetToolbar* (p. 926), the client size returned will have subtracted the toolbar height. However, the available positions for the client window and other windows of the frame do not start at zero - you must add the toolbar height.

The position and size of the status bar and toolbar (if known to the frame) are always managed by **wxMDIParentFrame**, regardless of what behaviour is defined in your **OnSize** event handler. However, the client window position and size are always set in **OnSize**, so if you override this event handler, make sure you deal with the client window.

You do not have to manage the size and position of MDI child windows, since they are managed automatically by the client window.

See also

wxMDIParentFrame::GetToolBar (p. 925), *wxMDIParentFrame::SetToolBar* (p. 926), *wxMDIClientWindow* (p. 919)

wxPython note: The wxPython version of this method takes no arguments and returns a tuple containing width and height.

wxMDIParentFrame::GetActiveChild

wxMDIChildFrame* GetActiveChild() **const**

Returns a pointer to the active MDI child, if there is one.

wxMDIParentFrame::GetClientWindow

wxMDIClientWindow* GetClientWindow() **const**

Returns a pointer to the client window.

See also

wxMDIParentFrame::OnCreateClient (p. 926)

wxMDIParentFrame::GetToolBar

virtual wxWindow* GetToolBar() **const**

Returns the window being used as the toolbar for this frame.

See also

wxMDIParentFrame::SetToolBar (p. 926)

wxMDIParentFrame::GetWindowMenu

wxMenu* GetWindowMenu() const

Returns the current Window menu (added by wxWidgets to the menubar). This function is available under Windows only.

wxMDIParentFrame::OnCreateClient

virtual wxMDIClientWindow* OnCreateClient()

Override this to return a different kind of client window. If you override this function, you must create your parent frame in two stages, or your function will never be called, due to the way C++ treats virtual functions called from constructors. For example:

```
frame = new MyParentFrame;
frame->Create(parent, myParentFrameId, wxT("My Parent Frame"));
```

Remarks

You might wish to derive from *wxMDIClientWindow* (p. 919) in order to implement different erase behaviour, for example, such as painting a bitmap on the background.

Note that it is probably impossible to have a client window that scrolls as well as painting a bitmap or pattern, since in **OnScroll**, the scrollbar positions always return zero. (Solutions to: julian.smart@btopenworld.com).

See also

wxMDIParentFrame::GetClientWindow (p. 925), *wxMDIClientWindow* (p. 919)

wxMDIParentFrame::SetToolBar

virtual void SetToolBar(wxWindow* *toolbar*)

Sets the window to be used as a toolbar for this MDI parent window. It saves the application having to manage the positioning of the toolbar MDI client window.

Parameters

toolbar

Toolbar to manage.

Remarks

When the frame is resized, the toolbar is resized to be the width of the frame client area, and the toolbar height is kept the same.

The parent of the toolbar must be this frame.

If you wish to manage your own toolbar (or perhaps you have more than one), don't call this function, and instead manage your subwindows and the MDI client window by providing an **OnSize** event handler. Call *wxMDIParentFrame::GetClientSize* (p. 924) to find how much space there is for your windows.

Note that SDI (normal) frames and MDI child windows must always have their toolbars managed by the application.

See also

wxMDIParentFrame::GetToolBar (p. 925)*, wxMDIParentFrame::GetClientSize* (p. 924)

wxMDIParentFrame::SetWindowMenu

void SetWindowMenu(**wxMenu*** *menu*)

Call this to change the current Window menu. Ownership of the menu object passes to the frame when you call this function.

This call is available under Windows only.

To remove the window completely, use the wxFRAME_NO_WINDOW_MENU window style.

wxMDIParentFrame::Tile

void Tile(**wxOrientation** *orient = wxHORIZONTAL*)

Tiles the MDI child windows either horizontally or vertically depending on whether *orient* is wxHORIZONTAL or wxVERTICAL.

Currently only implemented for MSW, does nothing under the other platforms.

See also

wxMDIParentFrame::Cascade (p. 924)*, wxMDIParentFrame::ArrangeIcons* (p. 924)

wxMediaCtrl

wxMediaCtrl is a class that allows a way to convieniently display types of media, such as videos, audio files, natively through native codecs.

wxMediaCtrl uses native backends to render media, for example on Windows there is a ActiveMovie/DirectShow backend, and on Macintosh there is a QuickTime backend.

Derived from

wxControl (p. 215)

Include files

<wx/mediactrl.h>

Rendering media

Depending upon the backend, wxMediaCtrl can render and display pretty much any kind of media that the native system can - such as an image, mpeg video, or mp3 (without license restrictions - sense it relies on native system calls that may not technically have mp3 decoding available, for example, it falls outside the realm of licensing restrictions).

For general operation, all you need to do is call *wxMediaCtrl::Load* (p. 931) to load the file you want to render, then call *wxMediaCtrl::Play* (p. 932) to show the video/audio of the media.

More complex operations are generally more heavily dependant on the capabilities of the backend. For example, QuickTime cannot set the playback rate of certain streaming media - while DirectShow is slightly more flexible in that regard.

Operation

When wxMediaCtrl plays a file, it plays until the stop position is reached (currently the end of the file/stream). Right before it hits the end of the stream, it fires off a EVT_MEDIA_STOP event to its parent window, at which point the event handler can choose to veto the event, preventing the stream from actually stopping.

```
Example://connect to the media event
this->Connect(wxMY_ID, wxEVT_MEDIA_STOP, (wxObjectEventFunction)
(wxEventFunction)(wxMediaEventFunction) &MyFrame::OnMediaStop);

//...
void MyFrame::OnMediaStop(const wxMediaEvent& evt)
{
    if(bUserWantsToSeek)
    {
        m_mediactrl->SetPosition(
            m_mediactrl->GetDuration() << 1
                                );
        evt.Veto();
    }
}
```

When wxMediaCtrl stops, either by the EVT_MEDIA_STOP not being vetoed, or by manually calling *wxMediaCtrl::Stop* (p. 932), where it actually stops is not at the beginning, rather, but at the beginning of the stream. That is, when it stops and play is called, playback is gauranteed to start at the beginning of the media. This is because some streams are not seekable, and when stop is called on them they return to the beginning, thus wxMediaCtrl tries to keep consistant for all types of media.

Choosing a backend

Generally, you should almost certainly leave this part up to wxMediaCtrl - but if you need a certain backend for a particular reason, such as QuickTime for playing .mov files, all you need to do to choose a specific backend is to pass the name of the backend class to *wxMediaCtrl::Create* (p. 930).

The following are valid backend identifiers -**wxMEDIABACKEND_DIRECTSHOW**
Use ActiveMovie/DirectShow. Requires wxUSE_DIRECTSHOW to be enabled, requires linkage with the static library strmiids.lib, and is available on Windows Only.

wxMEDIABACKEND_QUICKTIME	Use QuickTime. Windows and Mac Only. WARNING: On Mac Systems lower than OSX 10.2 this defaults to emulating window positioning and suffers from several bugs, including not working correctly embedded in a wxNotebook.
wxMEDIABACKEND_MCI	Use Media Command Interface. Windows Only.
wxMEDIABACKEND_GSTREAMER	Use GStreamer. Unix Only.

wxMediaCtrl::wxMediaCtrl

wxMediaCtrl()

Default constructor - you `must` call Create before calling any other methods of wxMediaCtrl.

wxMediaCtrl(**wxWindow*** *parent*, **const wxString&** *fileName = wxT("")*, **wxWindowID** *id*, **const wxPoint&** *pos = wxDefaultPosition*, **const wxSize&** *size = wxDefaultSize*, **long** *style = 0*, **const wxString&** *szBackend = wxT("")*, **const wxValidator&***validator = wxDefaultValidator*, **const wxString&** *name = wxPanelNameStr*)

Constructor that calls *Create* (p. 930). You may prefer to call *Create* (p. 930) directly to check to see if wxMediaCtrl is available on the system.

parent

parent of this control. Must not be NULL.

id

id to use for events

fileName

If not empty, the path of a file to open.

pos

> Position to put control at.

size

> Size to put the control at and to stretch movie to.

style

> Optional styles.

szBackend

> Name of backend you want to use, leave blank to make wxMediaCtrl figure it out.

validator

> validator to use.

name

> Window name.

wxMediaCtrl::Create

bool Create(**wxWindow*** *parent*, **const wxString&** *fileName = wxT("")*, **wxWindowID** *id*, **const wxPoint&** *pos = wxDefaultPosition*, **const wxSize&** *size = wxDefaultSize*, **long** *style = 0*, **const wxString&** *szBackend = wxT("")*, **const wxValidator&***validator = wxDefaultValidator*, **const wxString&** *name = wxPanelNameStr*)

Creates this control. Returns `false` if it can't load the movie located at `fileName` or it cannot load one of its native backends.

If you specify a file to open via `fileName` and you don't specify a backend to use, wxMediaCtrl tries each of its backends until one that can render the path referred to by `fileName` can be found.

parent

> parent of this control. Must not be NULL.

id

> id to use for events

fileName

> If not empty, the path of a file to open.

pos

> Position to put control at.

size

    Size to put the control at and to stretch movie to.

style

    Optional styles.

szBackend

    Name of backend you want to use, leave blank to make wxMediaCtrl figure it out.

validator

    validator to use.

name

    Window name.

wxMediaCtrl::Length

wxFileOffset GetDuration()

Obtains the length - the total amount of time the movie has in milliseconds.

wxMediaCtrl::Tell

wxFileOffset GetPosition()

Obtains the current position in time within the movie in milliseconds.

wxMediaCtrl::GetState

wxMediaCtrlState GetState()

Obtains the state the playback of the movie is in -

wxMEDIASTATE_STOPPED	The movie has stopped.
wxMEDIASTATE_PAUSED	The movie is paused.
wxMEDIASTATE_PLAYING	The movie is currently playing.

wxMediaCtrl::Load

bool Load(const wxString& *fileName*)

Loads the file that `fileName` refers to. Returns false if loading fails.

wxMediaCtrl::Pause

bool Pause()

Pauses playback of the movie.

wxMediaCtrl::Play

bool Play()

Resumes playback of the movie.

wxMediaCtrl::Seek

wxFileOffset SetPosition(**wxFileOffset** *where*, **wxSeekMode** *mode*)

Seeks to a position within the movie.

wxMediaCtrl::Stop

bool Stop()

Stops the media.

See *Operation* (p. 928) for an overview of how stopping works.

wxMediaEvent

Event *wxMediaCtrl* (p. 927) uses.

Derived from

wxNotifyEvent (p. 1011)

Include files

<wx/mediactrl.h>

Event table macros

EVT_MEDIA_STOP(func)	Triggerred right before the media stops. You can Veto this event to prevent it from stopping, causing it to continue playing - even if it has reached that end of the media.
EVT_MEDIA_FINISHED(func)	Sent when a media has finished playing in a *wxMediaCtrl* (p. 927). Note that if you connect to this event and don't skip it, it will override the looping behaviour of the media control.

wxMemoryBuffer

A **wxMemoryBuffer** is a useful data structure for storing arbitrary sized blocks of memory. wxMemoryBuffer guarantees deletion of the memory block when the object is destroyed.

Derived from

None

Include files

<wx/buffer.h>

wxMemoryBuffer::wxMemoryBuffer

wxMemoryBuffer(**const wxMemoryBuffer&** *src*)

Copy constructor, refcounting is used for performance , but wxMemoryBuffer is not a copy-on-write structure so changes made to one buffer effect all copies made from it.

wxMemoryBuffer(**size_t** *size*)

Create a new buffer.

size

size of new buffer.

wxMemoryBuffer::GetData

void* GetData()

Return a pointer to the data in the buffer.

wxMemoryBuffer::GetBufSize

size_t GetBufSize()

Returns the size of the buffer.

wxMemoryBuffer::GetDataLen

size_t GetDataLen()

Returns the length of the valid data in the buffer.

wxMemoryBuffer::SetBufSize

void SetBufSize(**size_t** *size*)

Ensures the buffer has at least *size* bytes available.

wxMemoryBuffer::SetDataLen

void SetDataLen(**size_t** *size*)

Sets the length of the data stored in the buffer. Mainly useful for truncating existing data.

size

New length of the valid data in the buffer. This is distinct from the allocated size

wxMemoryBuffer::GetWriteBuf

void * GetWriteBuf(**size_t** *sizeNeeded*)

Ensure the buffer is big enough and return a pointer to the buffer which can be used to directly write into the buffer up to *sizeNeeded* bytes.

wxMemoryBuffer::UngetWriteBuf

void UngetWriteBuf(**size_t** *sizeUsed*)

Update the buffer after completing a direct write, which you must have used GetWriteBuf() to initialise.

sizeUsed

The amount of data written in to buffer by the direct write

wxMemoryBuffer::GetAppendBuf

void * GetAppendBuf(**size_t** *sizeNeeded*)

Ensure that the buffer is big enough and return a pointer to the start of the empty space in the buffer. This pointer can be used to directly write data into the buffer, this new data will be appended to the existing data.

sizeNeeded

Amount of extra space required in the buffer for the append operation

wxMemoryBuffer::UngetAppendBuf

void UngetAppendBuf(**size_t** *sizeUsed*)

Update the length after completing a direct append, which you must have used GetAppendBuf() to initialise.

sizeUsed

> This is the amount of new data that has been appended.

wxMemoryBuffer::AppendByte

void AppendByte(**char** *data*)

Append a single byte to the buffer.

data

> New byte to append to the buffer.

wxMemoryBuffer::AppendData

void AppendData(**void*** *data*, **size_t** *len*)

Single call to append a data block to the buffer.

data

> Pointer to block to append to the buffer.

len

> Length of data to append.

wxMemoryDC

A memory device context provides a means to draw graphics onto a bitmap. When drawing in to a mono-bitmap, using `wxWHITE`, `wxWHITE_PEN` and`wxWHITE_BRUSH`will draw the background colour (i.e. 0) whereas all other colours will draw the foreground colour (i.e. 1).

Derived from

wxDC (p. 365)
wxObject (p. 1012)

Include files

<wx/dcmemory.h>

Remarks

A bitmap must be selected into the new memory DC before it may be used for anything. Typical usage is as follows:

```
// Create a memory DC
wxMemoryDC temp_dc;
temp_dc.SelectObject(test_bitmap);

// We can now draw into the memory DC...
// Copy from this DC to another DC.
old_dc.Blit(250, 50, BITMAP_WIDTH, BITMAP_HEIGHT, temp_dc, 0, 0);
```

Note that the memory DC *must* be deleted (or the bitmap selected out of it) before a bitmap can be reselected into another memory DC.

See also

wxBitmap (p. 79), *wxDC* (p. 365)

wxMemoryDC::wxMemoryDC

wxMemoryDC()

Constructs a new memory device context.

Use the *Ok* member to test whether the constructor was successful in creating a usable device context. Don't forget to select a bitmap into the DC before drawing on it.

wxMemoryDC::SelectObject

void SelectObject(const wxBitmap& *bitmap*)

Selects the given bitmap into the device context, to use as the memory bitmap. Selecting the bitmap into a memory DC allows you to draw into the DC (and therefore the bitmap) and also to use **Blit** to copy the bitmap to a window. For this purpose, you may find *wxDC::DrawIcon* (p. 371) easier to use instead.

If the argument is wxNullBitmap (or some other uninitialised wxBitmap) the current bitmap is selected out of the device context, and the original bitmap restored, allowing the current bitmap to be destroyed safely.

wxMemoryFSHandler

This *wxFileSystem* (p. 540) handler can store arbitrary data in memory stream and make them accessible via URL. It is particularly suitable for storing bitmaps from resources or included XPM files so that they can be used with wxHTML.

Filenames are prefixed with "memory:", e.g. "memory:myfile.html".

Example:

```
#ifndef __WXMSW__
#include "logo.xpm"
#endif

void MyFrame::OnAbout(wxCommandEvent&)
{
    wxBusyCursor bcur;
    wxFileSystem::AddHandler(new wxMemoryFSHandler);
    wxMemoryFSHandler::AddFile("logo.pcx", wxBITMAP(logo),
wxBITMAP_TYPE_PCX);
    wxMemoryFSHandler::AddFile("about.htm",
                                "<html><body>About: "
                                "<img
src=\"memory:logo.pcx\"></body></html>");

    wxDialog dlg(this, -1, wxString(_("About")));
    wxBoxSizer *topsizer;
    wxHtmlWindow *html;
    topsizer = new wxBoxSizer(wxVERTICAL);
    html = new wxHtmlWindow(&dlg, -1, wxDefaultPosition,
                            wxSize(380, 160), wxHW_SCROLLBAR_NEVER);
    html->SetBorders(0);
    html->LoadPage("memory:about.htm");
    html->SetSize(html->GetInternalRepresentation()->GetWidth(),
                  html->GetInternalRepresentation()->GetHeight());
    topsizer->Add(html, 1, wxALL, 10);
    topsizer->Add(new wxStaticLine(&dlg, -1), 0, wxEXPAND | wxLEFT |
    wxRIGHT, 10);
    topsizer->Add(new wxButton(&dlg, wxID_OK, "Ok"),
                  0, wxALL | wxALIGN_RIGHT, 15);
    dlg.SetAutoLayout(true);
    dlg.SetSizer(topsizer);
    topsizer->Fit(&dlg);
    dlg.Centre();
    dlg.ShowModal();

    wxMemoryFSHandler::RemoveFile("logo.pcx");
    wxMemoryFSHandler::RemoveFile("about.htm");
}
```

Derived from

wxFileSystemHandler (p. 542)

Include files

<wx/fs_mem.h>

wxMemoryFSHandler::wxMemoryFSHandler

wxMemoryFSHandler()

Constructor.

wxMemoryFSHandler::AddFile

static void AddFile(**const wxString&** *filename*, **wxImage&** *image*, **long** *type*)

static void AddFile(**const wxString&** *filename*, **const wxBitmap&** *bitmap*, **long** *type*)

static void AddFile(**const wxString&** *filename*, **const wxString&** *textdata*)

static void AddFile(**const wxString&** *filename*, **const void*** *binarydata*, **size_t** *size*)

Add file to list of files stored in memory. Stored data (bitmap, text or raw data) will be copied into private memory stream and available under name "memory:" + *filename*.

The *type* argument is one of `wxBITMAP_TYPE_XXX` constants. Note that you must use a *type* value (aka image format) that wxWidgets can save (e.g. JPG, PNG, see *wxImage documentation* (p. 779))!

wxMemoryFSHandler::RemoveFile

static void RemoveFile(**const wxString&** *filename*)

Remove file from memory FS and free occupied memory.

wxMemoryInputStream

Derived from

wxInputStream (p. 814)

Include files

<wx/mstream.h>

See also

wxStreamBuffer (p. 1277), *wxMemoryOutputStream* (p. 939)

wxMemoryInputStream::wxMemoryInputStream

wxMemoryInputStream(**const char *** *data*, **size_t** *len*)

Initializes a new read-only memory stream which will use the specified buffer*data* of length *len*. The stream does not take ownership of the buffer, i.e. that it will not delete in its destructor.

wxMemoryInputStream(**const wxMemoryOutputStream&** *stream*)

Creates a new read-only memory stream, initilalizing it with the data from the given output stream *stream*.

wxMemoryInputStream::~wxMemoryInputStream

~wxMemoryInputStream()

Destructor.

wxMemoryInputStream::GetInputStreamBuffer

wxStreamBuffer * GetInputStreamBuffer() **const**

Returns the pointer to the stream object used as an internal buffer for that stream.

wxMemoryOutputStream

Derived from

wxOutputStream (p. 1016)

Include files

<wx/mstream.h>

See also

wxStreamBuffer (p. 1277)

wxMemoryOutputStream::wxMemoryOutputStream

wxMemoryOutputStream(**char *** *data = NULL*, **size_t** *length = 0*)

If *data* is NULL, then it will initialize a new empty buffer which will grow if required.

Warning

If the buffer is created, it will be destroyed at the destruction of the stream.

wxMemoryOutputStream::~wxMemoryOutputStream

~wxMemoryOutputStream()

Destructor.

wxMemoryOutputStream::CopyTo

size_t CopyTo(**char ****buffer*, **size_t** *len*) **const**

CopyTo allowed you to transfer data from the internal buffer of wxMemoryOutputStream to an external buffer. *len* specifies the size of the buffer.

Returned value

CopyTo returns the number of bytes copied to the buffer. Generally it is either len or the size of the stream buffer.

wxMemoryOutputStream::GetOutputStreamBuffer

wxStreamBuffer * GetOutputStreamBuffer() **const**

Returns the pointer to the stream object used as an internal buffer for that stream.

wxMenu

A menu is a popup (or pull down) list of items, one of which may be selected before the menu goes away (clicking elsewhere dismisses the menu). Menus may be used to construct either menu bars or popup menus.

A menu item has an integer ID associated with it which can be used to identify the selection, or to change the menu item in some way. A menu item with a special identifier -1 is a separator item and doesn't have an associated command but just makes a separator line appear in the menu.

NB: Please note that *wxID_ABOUT* and *wxID_EXIT* are predefined by wxWidgets and have a special meaning since entries using these IDs will be taken out of the normal menus under MacOS X and will be inserted into the system menu (following the appropriate MacOS X interface guideline). On PalmOS *wxID_EXIT* is disabled according to Palm OS Companion guidelines.

Menu items may be either normal items, check items or radio items. Normal items don't have any special properties while the check items have a boolean flag associated to them and they show a checkmark in the menu when the flag is set. wxWidgets automatically toggles the flag value when the item is clicked and its value may be retrieved using either *IsChecked* (p. 950) method of wxMenu or wxMenuBar itself or by using *wxEvent::IsChecked* (p. 184) when you get the menu notification for the item in question.

The radio items are similar to the check items except that all the other items in the same radio group are unchecked when a radio item is checked. The radio group is formed by a contiguous range of radio items, i.e. it starts at the first item of this kind and ends with the first item of a different kind (or the end of the menu). Notice that because the radio groups are defined in terms of the item positions inserting or removing the items in the menu containing the radio items risks to not work correctly. Finally note that radio items are not supported under Motif.

Allocation strategy

All menus except the popup ones must be created on the heap. All menus attached to a

menubar or to another menu will be deleted by their parent when it is deleted. As the frame menubar is deleted by the frame itself, it means that normally all menus used are deleted automatically.

Derived from

wxEvtHandler (p. 486)
wxObject (p. 1012)

Include files

<wx/menu.h>

Event handling

If the menu is part of a menubar, then *wxMenuBar* (p. 953) event processing is used.

With a popup menu, there is a variety of ways to handle a menu selection event (wxEVT_COMMAND_MENU_SELECTED).

1. Derive a new class from wxMenu and define event table entries using the EVT_MENU macro.

2. Set a new event handler for wxMenu, using an object whose class has EVT_MENU entries.

3. Provide EVT_MENU handlers in the window which pops up the menu, or in an ancestor of this window.

4. Define a callback of type wxFunction, which you pass to the wxMenu constructor. The callback takes a reference to the menu, and a reference to a*wxCommandEvent* (p. 180). This method is deprecated and should not be used in the new code, it is provided for backwards compatibility only.

See also

wxMenuBar (p. 953), *wxWindow::PopupMenu* (p. 1521), *Event handling overview* (p. 1751)

wxMenu::wxMenu

 wxMenu(**const wxString&** *title = ""*, **long** *style = 0*)

Constructs a wxMenu object.

Parameters

title

> A title for the popup menu: the empty string denotes no title.

style

If set to `wxMENU_TEAROFF`, the menu will be detachable (wxGTK only).

 wxMenu(**long** *style*)

Constructs a wxMenu object.

Parameters

style

If set to `wxMENU_TEAROFF`, the menu will be detachable (wxGTK only).

wxMenu::~wxMenu

 ~wxMenu()

Destructor, destroying the menu.

Note: under Motif, a popup menu must have a valid parent (the window it was last popped up on) when being destroyed. Therefore, make sure you delete or re-use the popup menu *before* destroying the parent window. Re-use in this context means popping up the menu on a different window from last time, which causes an implicit destruction and recreation of internal data structures.

wxMenu::Append

wxMenuItem* Append(**int** *id*, **const wxString&** *item*, **const wxString&** *helpString* = *""*, **wxItemKind** *kind* = *wxITEM_NORMAL*)

Adds a string item to the end of the menu.

wxMenuItem* Append(**int** *id*, **const wxString&** *item*, **wxMenu** **subMenu*, **const wxString&** *helpString* = *""*)

Adds a pull-right submenu to the end of the menu. Append the submenu to the parent menu *after* you have added your menu items, or accelerators may not be registered properly.

wxMenuItem* Append(**wxMenuItem*** *menuItem*)

Adds a menu item object. This is the most generic variant of Append() method because it may be used for both items (including separators) and submenus and because you can also specify various extra properties of a menu item this way, such as bitmaps and fonts.

Parameters

id

The menu command identifier.

item

The string to appear on the menu item.

menu

Pull-right submenu.

kind

May be `wxITEM_SEPARATOR`, `wxITEM_NORMAL`, `wxITEM_CHECK` or `wxITEM_RADIO`

helpString

An optional help string associated with the item. By default, the handler for the wxEVT_MENU_HIGHLIGHT event displays this string in the status line.

menuItem

A menuitem object. It will be owned by the wxMenu object after this function is called, so do not delete it yourself.

Remarks

This command can be used after the menu has been shown, as well as on initial creation of a menu or menubar.

The *item* string for the normal menu items (not submenus or separators) may include the accelerator which can be used to activate the menu item from keyboard. The accelerator string follows the item label and is separated from it by a `TAB` character (`'\t'`). Its general syntax is any combination of `"CTRL"`, `"ALT"` and `"SHIFT"` strings (case doesn't matter) separated by either `'-'` or `'+'` characters and followed by the accelerator itself. The accelerator may be any alphanumeric character, any function key (from `F1` to `F12`) or one of the special characters listed in the table below (again, case doesn't matter):

`DEL` or `DELETE`	Delete key
`INS` or `INSERT`	Insert key
`ENTER` or `RETURN`	Enter key
`PGUP`	PageUp key
`PGDN`	PageDown key
`LEFT`	Left cursor arrow key
`RIGHT`	Right cursor arrow key
`UP`	Up cursor arrow key
`DOWN`	Down cursor arrow key
`HOME`	Home key

END	End key
SPACE	Space
TAB	Tab key
ESC or ESCAPE	Escape key (Windows only)

See also

wxMenu::AppendSeparator (p. 945), *wxMenu::AppendCheckItem* (p. 944), *wxMenu::AppendRadioItem* (p. 944), *wxMenu::Insert* (p. 949), *wxMenu::SetLabel* (p. 952), *wxMenu::GetHelpString* (p. 947), *wxMenu::SetHelpString* (p. 952), *wxMenuItem* (p. 964)

wxPython note: In place of a single overloaded method name, wxPython implements the following methods:

> **Append(id, string, helpStr="", checkable=false)**
>
> **AppendMenu(id, string, aMenu, helpStr="")**
>
> **AppendItem(aMenuItem)**

wxMenu::AppendCheckItem

wxMenuItem* AppendCheckItem(**int** *id*, **const wxString&** *item*, **const wxString&** *helpString = ""*)

Adds a checkable item to the end of the menu.

See also

wxMenu::Append (p. 942), *wxMenu::InsertCheckItem* (p. 949)

wxMenu::AppendRadioItem

wxMenuItem* AppendRadioItem(**int** *id*, **const wxString&** *item*, **const wxString&** *helpString = ""*)

Adds a radio item to the end of the menu. All consequent radio items form a group and when an item in the group is checked, all the others are automatically unchecked.

NB: Currently only implemented under Windows and GTK, use `#if wxHAS_RADIO_MENU_ITEMS` to test for availability of this feature.

See also

wxMenu::Append (p. 942), *wxMenu::InsertRadioItem* (p. 949)

wxMenu::AppendSeparator

wxMenuItem* AppendSeparator()

Adds a separator to the end of the menu.

See also

wxMenu::Append (p. 942), *wxMenu::InsertSeparator* (p. 949)

wxMenu::Break

void Break()

Inserts a break in a menu, causing the next appended item to appear in a new column.

wxMenu::Check

void Check(**int** *id*, **const bool** *check*)

Checks or unchecks the menu item.

Parameters

id

> The menu item identifier.

check

> If true, the item will be checked, otherwise it will be unchecked.

See also

wxMenu::IsChecked (p. 950)

wxMenu::Delete

void Delete(**int** *id*)

void Delete(**wxMenuItem** *item)

Deletes the menu item from the menu. If the item is a submenu, it will**not** be deleted. Use *Destroy* (p. 946) if you want to delete a submenu.

Parameters

id

> Id of the menu item to be deleted.

item

>   Menu item to be deleted.

See also

wxMenu::FindItem (p. 946), *wxMenu::Destroy* (p. 946), *wxMenu::Remove* (p. 951)

wxMenu::Destroy

void Destroy(**int** *id*)

void Destroy(**wxMenuItem ****item*)

Deletes the menu item from the menu. If the item is a submenu, it will be deleted. Use *Remove* (p. 951) if you want to keep the submenu (for example, to reuse it later).

Parameters

id

>   Id of the menu item to be deleted.

item

>   Menu item to be deleted.

See also

wxMenu::FindItem (p. 946), *wxMenu::Deletes* (p. 945), *wxMenu::Remove* (p. 951)

wxMenu::Enable

void Enable(**int** *id*, **const bool** *enable*)

Enables or disables (greys out) a menu item.

Parameters

id

>   The menu item identifier.

enable

>   true to enable the menu item, false to disable it.

See also

wxMenu::IsEnabled (p. 950)

wxMenu::FindItem

int FindItem(**const wxString&** *itemString*) **const**

Finds the menu item id for a menu item string.

wxMenuItem * FindItem(**int** *id*, **wxMenu *****menu = NULL*) **const**

Finds the menu item object associated with the given menu item identifier and, optionally, the (sub)menu it belongs to.

wxPerl note: In wxPerl this method takes just the `id` parameter; in scalar context it returns the associated `Wx::MenuItem`, in list context it returns a 2-element list (`item, submenu )`

Parameters

itemString

Menu item string to find.

id

Menu item identifier.

menu

If the pointer is not NULL, it will be filled with the item's parent menu (if the item was found)

Return value

First form: menu item identifier, or `wxNOT_FOUND` if none is found.

Second form: returns the menu item object, or NULL if it is not found.

Remarks

Any special menu codes are stripped out of source and target strings before matching.

wxPython note: The name of this method in wxPython is `FindItemById` and it does not support the second parameter.

wxMenu::FindItemByPosition

wxMenuItem* FindItemByPosition(**size_t** *position*) **const**

Returns the wxMenuItem given a position in the menu.

wxMenu::GetHelpString

wxString GetHelpString(**int** *id*) **const**

Returns the help string associated with a menu item.

Parameters

id

The menu item identifier.

Return value

The help string, or the empty string if there is no help string or the item was not found.

See also

wxMenu::SetHelpString (p. 952), *wxMenu::Append* (p. 942)

wxMenu::GetLabel

wxString GetLabel(**int** *id*) **const**

Returns a menu item label.

Parameters

id

The menu item identifier.

Return value

The item label, or the empty string if the item was not found.

See also

wxMenu::SetLabel (p. 952)

wxMenu::GetMenuItemCount

size_t GetMenuItemCount() **const**

Returns the number of items in the menu.

wxMenu::GetMenuItems

wxMenuItemList& GetMenuItems() **const**

Returns the list of items in the menu. wxMenuItemList is a pseudo-template list class containing wxMenuItem pointers.

wxMenu::GetTitle

wxString GetTitle() **const**

Returns the title of the menu.

Remarks

This is relevant only to popup menus, use *wxMenuBar::GetLabelTop* (p. 958) for the menus in the menubar.

See also

wxMenu::SetTitle (p. 953)

wxMenu::Insert

wxMenuItem* Insert(**size_t** *pos*, **wxMenuItem ****item*)

wxMenuItem* Insert(**size_t** *pos*, **int** *id*, **const wxString&** *item*, **const wxString&** *helpString = ""*, **wxItemKind** *kind = wxITEM_NORMAL*)

Inserts the given *item* before the position *pos*. Inserting the item at position *GetMenuItemCount* (p. 948) is the same as appending it.

See also

wxMenu::Append (p. 942), *wxMenu::Prepend* (p. 950)

wxMenu::InsertCheckItem

wxMenuItem* InsertCheckItem(**size_t** *pos*, **int** *id*, **const wxString&** *item*, **const wxString&** *helpString = ""*)

Inserts a checkable item at the given position.

See also

wxMenu::Insert (p. 949), *wxMenu::AppendCheckItem* (p. 944)

wxMenu::InsertRadioItem

wxMenuItem* InsertRadioItem(**size_t** *pos*, **int** *id*, **const wxString&** *item*, **const wxString&** *helpString = ""*)

Inserts a radio item at the given position.

See also

wxMenu::Insert (p. 949), *wxMenu::AppendRadioItem* (p. 944)

wxMenu::InsertSeparator

wxMenuItem* InsertSeparator(**size_t** *pos*)

Inserts a separator at the given position.

See also

wxMenu::Insert (p. 949), *wxMenu::AppendSeparator* (p. 945)

wxMenu::IsChecked

bool IsChecked(**int** *id*) **const**

Determines whether a menu item is checked.

Parameters

id

> The menu item identifier.

Return value

true if the menu item is checked, false otherwise.

See also

wxMenu::Check (p. 945)

wxMenu::IsEnabled

bool IsEnabled(**int** *id*) **const**

Determines whether a menu item is enabled.

Parameters

id

> The menu item identifier.

Return value

true if the menu item is enabled, false otherwise.

See also

wxMenu::Enable (p. 946)

wxMenu::Prepend

wxMenuItem* Prepend(**wxMenuItem** *item)

wxMenuItem* Prepend(**int** *id*, **const wxString&** *item*, **const wxString&** *helpString =
""*, **wxItemKind** *kind = wxITEM_NORMAL*)

Inserts the given *item* at position 0, i.e. before all the other existing items.

See also

wxMenu::Append (p. 942), *wxMenu::Insert* (p. 949)

wxMenu::PrependCheckItem

wxMenuItem* PrependCheckItem(**int** *id*, **const wxString&** *item*, **const wxString&**
helpString = "")

Inserts a checkable item at position 0.

See also

wxMenu::Prepend (p. 950), *wxMenu::AppendCheckItem* (p. 944)

wxMenu::PrependRadioItem

wxMenuItem* PrependRadioItem(**int** *id*, **const wxString&** *item*, **const wxString&**
helpString = "")

Inserts a radio item at position 0.

See also

wxMenu::Prepend (p. 950), *wxMenu::AppendRadioItem* (p. 944)

wxMenu::PrependSeparator

wxMenuItem* PrependSeparator()

Inserts a separator at position 0.

See also

wxMenu::Prepend (p. 950), *wxMenu::AppendSeparator* (p. 945)

wxMenu::Remove

wxMenuItem * Remove(**int** *id*)

wxMenuItem * Remove(**wxMenuItem ****item*)

Removes the menu item from the menu but doesn't delete the associated C++ object.
This allows to reuse the same item later by adding it back to the menu (especially useful

with submenus).

Parameters

id

> The identifier of the menu item to remove.

item

> The menu item to remove.

Return value

The item which was detached from the menu.

wxMenu::SetHelpString

void SetHelpString(**int** *id*, **const wxString&** *helpString*)

Sets an item's help string.

Parameters

id

> The menu item identifier.

helpString

> The help string to set.

See also

wxMenu::GetHelpString (p. 947)

wxMenu::SetLabel

void SetLabel(**int** *id*, **const wxString&** *label*)

Sets the label of a menu item.

Parameters

id

> The menu item identifier.

label

> The menu item label to set.

See also

wxMenu::Append (p. 942), *wxMenu::GetLabel* (p. 948)

wxMenu::SetTitle

void SetTitle(**const wxString&** *title*)

Sets the title of the menu.

Parameters

title

> The title to set.

Remarks

This is relevant only to popup menus, use *wxMenuBar::SetLabelTop* (p. 962) for the menus in the menubar.

See also

wxMenu::GetTitle (p. 948)

wxMenu::UpdateUI

void UpdateUI(**wxEvtHandler*** *source = NULL*) **const**

Sends events to *source* (or owning window if NULL) to update the menu UI. This is called just before the menu is popped up with *wxWindow::PopupMenu* (p. 1521), but the application may call it at other times if required.

See also

wxUpdateUIEvent (p. 1448)

wxMenuBar

A menu bar is a series of menus accessible from the top of a frame.

Derived from

wxWindow (p. 1490)
wxEvtHandler (p. 486)
wxObject (p. 1012)

Include files

<wx/menu.h>

Event handling

To respond to a menu selection, provide a handler for EVT_MENU, in the frame that contains the menu bar. If you have a toolbar which uses the same identifiers as your EVT_MENU entries, events from the toolbar will also be processed by your EVT_MENU event handlers.

Tip: under Windows, if you discover that menu shortcuts (for example, Alt-F to show the file menu) are not working, check any EVT_CHAR events you are handling in child windows. If you are not calling `event.Skip()` for events that you don't process in these event handlers, menu shortcuts may cease to work.

See also

wxMenu (p. 940), *Event handling overview* (p. 1751)

wxMenuBar::wxMenuBar

wxMenuBar(**long** *style = 0*)

Default constructor.

wxMenuBar(**size_t** *n*, **wxMenu*** *menus[]*, **const wxString** *titles[]*, **long** *style = 0*)

Construct a menu bar from arrays of menus and titles.

Parameters

n

> The number of menus.

menus

> An array of menus. Do not use this array again - it now belongs to the menu bar.

titles

> An array of title strings. Deallocate this array after creating the menu bar.

style

> If `wxMB_DOCKABLE` the menu bar can be detached (wxGTK only).

wxPython note: Only the default constructor is supported in wxPython. Use *wxMenuBar::Append* (p. 955) instead.

wxPerl note: wxPerl only supports the first constructor: use *wxMenuBar::Append* (p. 955) instead.

wxMenuBar::~wxMenuBar

void ~wxMenuBar()

Destructor, destroying the menu bar and removing it from the parent frame (if any).

wxMenuBar::Append

bool Append(**wxMenu ****menu*, **const wxString&** *title*)

Adds the item to the end of the menu bar.

Parameters

menu

> The menu to add. Do not deallocate this menu after calling **Append**.

title

> The title of the menu.

Return value

true on success, false if an error occurred.

See also

wxMenuBar::Insert (p. 959)

wxMenuBar::Check

void Check(**int** *id*, **const bool** *check*)

Checks or unchecks a menu item.

Parameters

id

> The menu item identifier.

check

> If true, checks the menu item, otherwise the item is unchecked.

Remarks

Only use this when the menu bar has been associated with a frame; otherwise, use the wxMenu equivalent call.

wxMenuBar::Enable

void Enable(**int** *id*, **const bool** *enable*)

Enables or disables (greys out) a menu item.

Parameters

id

> The menu item identifier.

enable

> true to enable the item, false to disable it.

Remarks

Only use this when the menu bar has been associated with a frame; otherwise, use the wxMenu equivalent call.

wxMenuBar::EnableTop

void EnableTop(**int** *pos*, **const bool** *enable*)

Enables or disables a whole menu.

Parameters

pos

> The position of the menu, starting from zero.

enable

> true to enable the menu, false to disable it.

Remarks

Only use this when the menu bar has been associated with a frame.

wxMenuBar::FindMenu

int FindMenu(**const wxString&** *title*) **const**

Returns the index of the menu with the given *title* or `wxNOT_FOUND` if no such menu exists in this menubar. The *title* parameter may specify either the menu title (with accelerator characters, i.e. `"&File"`) or just the menu label (`"File"`) indifferently.

wxMenuBar::FindMenuItem

int FindMenuItem(**const wxString&** *menuString*, **const wxString&** *itemString*) **const**

Finds the menu item id for a menu name/menu item string pair.

Parameters

menuString

> Menu title to find.

itemString

> Item to find.

Return value

The menu item identifier, or `wxNOT_FOUND` if none was found.

Remarks

Any special menu codes are stripped out of source and target strings before matching.

wxMenuBar::FindItem

wxMenuItem * FindItem(**int** *id*, **wxMenu** ***menu = NULL*) **const**

Finds the menu item object associated with the given menu item identifier.

Parameters

id

> Menu item identifier.

menu

> If not NULL, menu will get set to the associated menu.

Return value

The found menu item object, or NULL if one was not found.

wxMenuBar::GetHelpString

wxString GetHelpString(**int** *id*) **const**

Gets the help string associated with the menu item identifier.

Parameters

id

> The menu item identifier.

Return value

The help string, or the empty string if there was no help string or the menu item was not

found.

See also

wxMenuBar::SetHelpString (p. 961)

wxMenuBar::GetLabel

wxString GetLabel(**int** *id*) **const**

Gets the label associated with a menu item.

Parameters

id

> The menu item identifier.

Return value

The menu item label, or the empty string if the item was not found.

Remarks

Use only after the menubar has been associated with a frame.

wxMenuBar::GetLabelTop

wxString GetLabelTop(**int** *pos*) **const**

Returns the label of a top-level menu. Note that the returned string does not include the accelerator characters which could have been specified in the menu title string during its construction.

Parameters

pos

> Position of the menu on the menu bar, starting from zero.

Return value

The menu label, or the empty string if the menu was not found.

Remarks

Use only after the menubar has been associated with a frame.

See also

wxMenuBar::SetLabelTop (p. 962)

wxMenuBar::GetMenu

wxMenu* GetMenu(**int** *menuIndex*) **const**

Returns the menu at *menuIndex* (zero-based).

wxMenuBar::GetMenuCount

int GetMenuCount() **const**

Returns the number of menus in this menubar.

wxMenuBar::Insert

bool Insert(**size_t** *pos*, **wxMenu ****menu*, **const wxString&** *title*)

Inserts the menu at the given position into the menu bar. Inserting menu at position 0 will insert it in the very beginning of it, inserting at position *GetMenuCount()* (p. 959) is the same as calling *Append()* (p. 955).

Parameters

pos

> The position of the new menu in the menu bar

menu

> The menu to add. wxMenuBar owns the menu and will free it.

title

> The title of the menu.

Return value

true on success, false if an error occurred.

See also

wxMenuBar::Append (p. 955)

wxMenuBar::IsChecked

bool IsChecked(**int** *id*) **const**

Determines whether an item is checked.

Parameters

id

The menu item identifier.

Return value

true if the item was found and is checked, false otherwise.

wxMenuBar::IsEnabled

bool IsEnabled(**int** *id*) **const**

Determines whether an item is enabled.

Parameters

id

The menu item identifier.

Return value

true if the item was found and is enabled, false otherwise.

wxMenuBar::Refresh

void Refresh()

Redraw the menu bar

wxMenuBar::Remove

wxMenu * Remove(**size_t** *pos*)

Removes the menu from the menu bar and returns the menu object - the caller is responsible for deleting it. This function may be used together with *wxMenuBar::Insert* (p. 959) to change the menubar dynamically.

See also

wxMenuBar::Replace (p. 960)

wxMenuBar::Replace

wxMenu * Replace(**size_t** *pos*, **wxMenu ***menu*, **const wxString&** *title*)

Replaces the menu at the given position with another one.

Parameters

pos

The position of the new menu in the menu bar

menu

The menu to add.

title

The title of the menu.

Return value

The menu which was previously at position *pos*. The caller is responsible for deleting it.

See also

wxMenuBar::Insert (p. 959), *wxMenuBar::Remove* (p. 960)

wxMenuBar::SetHelpString

void SetHelpString(**int** *id*, **const wxString&** *helpString*)

Sets the help string associated with a menu item.

Parameters

id

Menu item identifier.

helpString

Help string to associate with the menu item.

See also

wxMenuBar::GetHelpString (p. 957)

wxMenuBar::SetLabel

void SetLabel(**int** *id*, **const wxString&** *label*)

Sets the label of a menu item.

Parameters

id

Menu item identifier.

label

Menu item label.

Remarks

Use only after the menubar has been associated with a frame.

See also

wxMenuBar::GetLabel (p. 958)

---

wxMenuBar::SetLabelTop

void SetLabelTop(**int** *pos*, **const wxString&** *label*)

Sets the label of a top-level menu.

Parameters

pos

> The position of a menu on the menu bar, starting from zero.

label

> The menu label.

Remarks

Use only after the menubar has been associated with a frame.

See also

wxMenuBar::GetLabelTop (p. 958)

---

wxMenuEvent

This class is used for a variety of menu-related events. Note that these do not include menu command events, which are handled using *wxCommandEvent* (p. 180) objects.

The default handler for wxEVT_MENU_HIGHLIGHT displays help text in the first field of the status bar.

Derived from

wxEvent (p. 483)
wxObject (p. 1012)

Include files

<wx/event.h>

Event table macros

To process a menu event, use these event handler macros to direct input to member

functions that take a wxMenuEvent argument. They can only be processed by a menubar's frame.

EVT_MENU_OPEN(func)	A menu is about to be opened. On Windows, this is only sent once for each navigation of the menubar (up until all menus have closed).
EVT_MENU_CLOSE(func)	A menu has been just closed.
EVT_MENU_HIGHLIGHT(id, func)	The menu item with the specified id has been highlighted: used to show help prompts in the status bar by *wxFrame* (p. 579)
EVT_MENU_HIGHLIGHT_ALL(func)	A menu item has been highlighted, i.e. the currently selected menu item has changed.

See also

Command events (p. 180),
Event handling overview (p. 1751)

wxMenuEvent::wxMenuEvent

 wxMenuEvent(**WXTYPE** *id = 0*, **int** *id = 0*, **wxMenu*** *menu = NULL*)

Constructor.

wxMenuEvent::GetMenu

wxMenu * GetMenu() **const**

Returns the menu which is being opened or closed. This method should only be used with the `OPEN` and `CLOSE` events.

wxMenuEvent::GetMenuId

int GetMenuId() **const**

Returns the menu identifier associated with the event. This method should be only used with the `HIGHLIGHT` events.

wxMenuEvent::IsPopup

bool IsPopup() **const**

Returns `true` if the menu which is being opened or closed is a popup menu, `false` if it is a normal one.

This method should only be used with the OPEN and CLOSE events.

wxMenuItem

A menu item represents an item in a menu. Note that you usually don't have to deal with it directly as *wxMenu* (p. 940) methods usually construct an object of this class for you.

Also please note that the methods related to fonts and bitmaps are currently only implemented for Windows and GTK+.

Derived from

wxOwnerDrawn (Windows only)
wxObject (p. 1012)

Include files

<wx/menuitem.h>

See also

wxMenuBar (p. 953), *wxMenu* (p. 940)

wxMenuItem::wxMenuItem

 wxMenuItem(**wxMenu*** *parentMenu = NULL*, **int** *id = wxID_SEPARATOR*, **const wxString&** *text = ""*, **const wxString&** *helpString = ""*, **wxItemKind** *kind = wxITEM_NORMAL*, **wxMenu*** *subMenu = NULL*)

Constructs a wxMenuItem object.

Parameters

parentMenu

Menu that the menu item belongs to.

id

Identifier for this menu item, or wxID_SEPARATOR to indicate a separator.

text

Text for the menu item, as shown on the menu. An accelerator key can be specified using the ampersand '&' character. In order to embed an ampersand character in the menu item text, the ampersand must be doubled.

helpString

Optional help string that will be shown on the status bar.

kind

> May be `wxITEM_SEPARATOR`, `wxITEM_NORMAL`, `wxITEM_CHECK` or `wxITEM_RADIO`

subMenu

> If non-NULL, indicates that the menu item is a submenu.

wxMenuItem::~wxMenuItem

 ~wxMenuItem()

Destructor.

wxMenuItem::Check

void Check(**bool** *check = true*)

Checks or unchecks the menu item.

Note that this only works when the item is already appended to a menu.

wxMenuItem::Enable

void Enable(**bool** *enable = true*)

Enables or disables the menu item.

wxMenuItem::GetBackgroundColour

wxColour& GetBackgroundColour() **const**

Returns the background colour associated with the menu item (Windows only).

wxMenuItem::GetBitmap

wxBitmap& GetBitmap(**bool** *checked = true*) **const**

Returns the checked or unchecked bitmap (Windows only).

wxMenuItem::GetFont

wxFont& GetFont() **const**

Returns the font associated with the menu item (Windows only).

wxMenuItem::GetHelp

wxString GetHelp() const

Returns the help string associated with the menu item.

wxMenuItem::GetId

int GetId() const

Returns the menu item identifier.

wxMenuItem::GetKind

wxItemKind GetKind() const

Returns the item kind, one of `wxITEM_SEPARATOR`, `wxITEM_NORMAL`, `wxITEM_CHECK` or `wxITEM_RADIO`.

wxMenuItem::GetLabel

wxString GetLabel() const

Returns the text associated with the menu item without any accelerator characters it might contain.

See also

GetText (p. 967), *GetLabelFromText* (p. 966)

wxMenuItem::GetLabelFromText

static wxString GetLabelFromText(const wxString& *text*)

Strips all accelerator characters and mnemonics from the given *text*. For example,

```
wxMenuItem::GetLabelFromText("&Hello\tCtrl-H");
```

will return just `"Hello"`.

See also

GetText (p. 967), *GetLabel* (p. 966)

wxMenuItem::GetMarginWidth

int GetMarginWidth() const

Gets the width of the menu item checkmark bitmap (Windows only).

wxMenuItem::GetMenu

wxMenu* GetMenu() **const**

Returns the menu this menu item is in, or NULL if this menu item is not attached.

wxMenuItem::GetName

wxString GetName() **const**

Returns the text associated with the menu item.

NB: this function is deprecated, please use *GetText* (p. 967) or *GetLabel* (p. 966) instead.

wxMenuItem::GetText

wxString GetText() **const**

Returns the text associated with the menu item, such as it was passed to the wxMenuItem constructor, i.e. with any accelerator characters it may contain.

See also

GetLabel (p. 966), *GetLabelFromText* (p. 966)

wxMenuItem::GetSubMenu

wxMenu* GetSubMenu() **const**

Returns the submenu associated with the menu item, or NULL if there isn't one.

wxMenuItem::GetTextColour

wxColour& GetTextColour() **const**

Returns the text colour associated with the menu item (Windows only).

wxMenuItem::IsCheckable

bool IsCheckable() **const**

Returns true if the item is checkable.

wxMenuItem::IsChecked

bool IsChecked() **const**

Returns true if the item is checked.

wxMenuItem::IsEnabled

bool IsEnabled() **const**

Returns true if the item is enabled.

wxMenuItem::IsSeparator

bool IsSeparator() **const**

Returns true if the item is a separator.

wxMenuItem::IsSubMenu

bool IsSubMenu() **const**

Returns true if the item is a submenu.

wxMenuItem::SetBackgroundColour

void SetBackgroundColour(**const wxColour&** *colour*) **const**

Sets the background colour associated with the menu item (Windows only).

wxMenuItem::SetBitmap

void SetBitmap(**const wxBitmap&** *bmp*)

Sets the bitmap for the menu item (Windows and GTK+ only). It is equivalent to *SetBitmaps* (p. 968)(bmp, wxNullBitmap).

wxMenuItem::SetBitmaps

void SetBitmaps(**const wxBitmap&** *checked*, **const wxBitmap&** *unchecked* = *wxNullBitmap*)

Sets the checked/unchecked bitmaps for the menu item (Windows only). The first bitmap is also used as the single bitmap for uncheckable menu items.

wxMenuItem::SetFont

void SetFont(**const wxFont&** *font*)

Sets the font associated with the menu item (Windows only).

wxMenuItem::SetHelp

void SetHelp(**const wxString&** *helpString*)

Sets the help string.

wxMenuItem::SetMarginWidth

void SetMarginWidth(**int** *width*) **const**

Sets the width of the menu item checkmark bitmap (Windows only).

wxMenuItem::SetMenu

void SetMenu(**const wxMenu***menu*)

Sets the parent menu which will contain this menu item.

wxMenuItem::SetSubMenu

void SetSubMenu(**const wxMenu***menu*)

Sets the submenu of this menu item.

wxMenuItem::SetText

void SetText(**const wxString&** *text*)

Sets the text associated with the menu item.

wxMenuItem::SetTextColour

void SetTextColour(**const wxColour&** *colour*)

Sets the text colour associated with the menu item (Windows only).

wxMessageDialog

This class represents a dialog that shows a single or multi-line message, with a choice of OK, Yes, No and Cancel buttons.

Derived from

wxDialog (p. 406)
wxWindow (p. 1490)
wxEvtHandler (p. 486)
wxObject (p. 1012)

Include files

<wx/msgdlg.h>

See also

wxMessageDialog overview (p. 1801)

wxMessageDialog::wxMessageDialog

 wxMessageDialog(**wxWindow*** *parent*, **const wxString&** *message*, **const wxString&** *caption = "Message box"*, **long** *style = wxOK | wxCANCEL*, **const wxPoint&** *pos = wxDefaultPosition*)

Constructor. Use *wxMessageDialog::ShowModal* (p. 971) to show the dialog.

Parameters

parent

Parent window.

message

Message to show on the dialog.

caption

The dialog caption.

style

A dialog style (bitlist) containing flags chosen from the following:

wxOK	Show an OK button.
wxCANCEL	Show a Cancel button.
wxYES_NO	Show Yes and No buttons.
wxYES_DEFAULT	Used with **wxYES_NO**, makes **Yes** button the default - which is the default behaviour.

wxNO_DEFAULT	Used with **wxYES_NO**, makes **No** button the default.
wxICON_EXCLAMATION	Shows an exclamation mark icon.
wxICON_HAND	Shows an error icon.
wxICON_ERROR	Shows an error icon - the same as wxICON_HAND.
wxICON_QUESTION	Shows a question mark icon.
wxICON_INFORMATION	Shows an information (i) icon.
wxSTAY_ON_TOP	The message box stays on top of all other window, even those of the other applications (Windows only).

pos

Dialog position. Not Windows.

wxMessageDialog::~wxMessageDialog

~wxMessageDialog()

Destructor.

wxMessageDialog::ShowModal

int ShowModal()

Shows the dialog, returning one of wxID_OK, wxID_CANCEL, wxID_YES, wxID_NO.

wxMetafile

A **wxMetafile** represents the MS Windows metafile object, so metafile operations have no effect in X. In wxWidgets, only sufficient functionality has been provided for copying a graphic to the clipboard; this may be extended in a future version. Presently, the only way of creating a metafile is to use a wxMetafileDC.

Derived from

wxObject (p. 1012)

Include files

<wx/metafile.h>

See also

wxMetafileDC (p. 973)

wxMetafile::wxMetafile

wxMetafile(**const wxString&** *filename = ""*)

Constructor. If a filename is given, the Windows disk metafile is read in. Check whether this was performed successfully by using the *wxMetafile::Ok* (p. 972) member.

wxMetafile::~wxMetafile

~wxMetafile()

Destructor.

wxMetafile::Ok

bool Ok()

Returns true if the metafile is valid.

wxMetafile::Play

bool Play(**wxDC ****dc*)

Plays the metafile into the given device context, returning true if successful.

wxMetafile::SetClipboard

bool SetClipboard(**int** *width = 0*, **int** *height = 0*)

Passes the metafile data to the clipboard. The metafile can no longer be used for anything, but the wxMetafile object must still be destroyed by the application.

Below is a example of metafile, metafile device context and clipboard use from the `hello.cpp` example. Note the way the metafile dimensions are passed to the clipboard, making use of the device context's ability to keep track of the maximum extent of drawing commands.

```
wxMetafileDC dc;
if (dc.Ok())
{
  Draw(dc, false);
  wxMetafile *mf = dc.Close();
  if (mf)
  {
    bool success = mf->SetClipboard((int)(dc.MaxX() + 10),
(int)(dc.MaxY() + 10));
    delete mf;
```

```
      }
    }
```

wxMetafileDC

This is a type of device context that allows a metafile object to be created (Windows only), and has most of the characteristics of a normal **wxDC**. The *wxMetafileDC::Close* (p. 974) member must be called after drawing into the device context, to return a metafile. The only purpose for this at present is to allow the metafile to be copied to the clipboard (see *wxMetafile* (p. 971)).

Adding metafile capability to an application should be easy if you already write to a wxDC; simply pass the wxMetafileDC to your drawing function instead. You may wish to conditionally compile this code so it is not compiled under X (although no harm will result if you leave it in).

Note that a metafile saved to disk is in standard Windows metafile format, and cannot be imported into most applications. To make it importable, call the function *::wxMakeMetafilePlaceable* (p. 1625) after closing your disk-based metafile device context.

Derived from

wxDC (p. 365)
wxObject (p. 1012)

Include files

<wx/metafile.h>

See also

wxMetafile (p. 971), *wxDC* (p. 365)

wxMetafileDC::wxMetafileDC

wxMetafileDC(**const wxString&** *filename* = *""*)

Constructor. If no filename is passed, the metafile is created in memory.

wxMetafileDC::~wxMetafileDC

~wxMetafileDC()

Destructor.

wxMetafileDC::Close

wxMetafile * Close()

This must be called after the device context is finished with. A metafile is returned, and ownership of it passes to the calling application (so it should be destroyed explicitly).

wxMimeTypesManager

This class allows the application to retrieve the information about all known MIME types from a system-specific location and the filename extensions to the MIME types and vice versa. After initialization the functions*wxMimeTypesManager::GetFileTypeFromMimeType* (p. 976) and *wxMimeTypesManager::GetFileTypeFromExtension* (p. 976) may be called: they will return a *wxFileType* (p. 545) object which may be further queried for file description, icon and other attributes.

Windows: MIME type information is stored in the registry and no additional initialization is needed.

Unix: MIME type information is stored in the files mailcap and mime.types (system-wide) and .mailcap and .mime.types in the current user's home directory: all of these files are searched for and loaded if found by default. However, additional functions *wxMimeTypesManager::ReadMailcap* (p. 976) and *wxMimeTypesManager::ReadMimeTypes* (p. 977) are provided to load additional files.

If GNOME or KDE desktop environment is installed, then wxMimeTypesManager gathers MIME information from respective files (e.g. .kdelnk files under KDE).

NB: Currently, wxMimeTypesManager is limited to reading MIME type information but it will support modifying it as well in future versions.

Global objects

Global instance of wxMimeTypesManager is always available. It is defined as follows:

```
wxMimeTypesManager *wxTheMimeTypesManager;
```

It is recommended to use this instance instead of creating your own because gathering MIME information may take quite a long on Unix systems.

Derived from

No base class.

Include files

<wx/mimetype.h>

See also

wxFileType (p. 545)

Helper functions

All of these functions are static (i.e. don't need a wxMimeTypesManager object to call them) and provide some useful operations for string representations of MIME types. Their usage is recommended instead of directly working with MIME types using wxString functions.

IsOfType (p. 976)

Constructor and destructor

NB: You won't normally need to use more than one wxMimeTypesManager object in a program.

wxMimeTypesManager (p. 975)
~wxMimeTypesManager (p. 976)

Query database

These functions are the heart of this class: they allow to find a *file type* (p. 545) object from either file extension or MIME type. If the function is successful, it returns a pointer to the wxFileType object which **must** be deleted by the caller, otherwise NULL will be returned.

GetFileTypeFromMimeType (p. 976)
GetFileTypeFromExtension (p. 976)

Initialization functions

Unix: These functions may be used to load additional files (except for the default ones which are loaded automatically) containing MIME information in either mailcap(5) or mime.types(5) format.

ReadMailcap (p. 976)
ReadMimeTypes (p. 977)
AddFallbacks (p. 976)

wxMimeTypesManager::wxMimeTypesManager

wxMimeTypesManager()

Constructor puts the object in the "working" state, no additional initialization are needed - but *ReadXXX* (p. 975) may be used to load additional mailcap/mime.types files.

wxMimeTypesManager::~wxMimeTypesManager

~wxMimeTypesManager()

Destructor is not virtual, so this class should not be derived from.

wxMimeTypesManager::AddFallbacks

void AddFallbacks(**const wxFileTypeInfo** **fallbacks*)

This function may be used to provide hard-wired fallbacks for the MIME types and extensions that might not be present in the system MIME database.

Please see the typetest sample for an example of using it.

wxMimeTypesManager::GetFileTypeFromExtension

wxFileType* GetFileTypeFromExtension(**const wxString&** *extension*)

Gather information about the files with given extension and return the corresponding *wxFileType* (p. 545) object or NULL if the extension is unknown.

wxMimeTypesManager::GetFileTypeFromMimeType

wxFileType* GetFileTypeFromMimeType(**const wxString&** *mimeType*)

Gather information about the files with given MIME type and return the corresponding *wxFileType* (p. 545) object or NULL if the MIME type is unknown.

wxMimeTypesManager::IsOfType

bool IsOfType(**const wxString&** *mimeType*, **const wxString&** *wildcard*)

This function returns true if either the given *mimeType* is exactly the same as *wildcard* or if it has the same category and the subtype of *wildcard* is '*'. Note that the '*' wildcard is not allowed in *mimeType* itself.

The comparison don by this function is case insensitive so it is not necessary to convert the strings to the same case before calling it.

wxMimeTypesManager::ReadMailcap

bool ReadMailcap(**const wxString&** *filename*, **bool** *fallback* = *false*)

Load additional file containing information about MIME types and associated information in mailcap format. See metamail(1) and mailcap(5) for more information.

fallback parameter may be used to load additional mailcap files without overriding the settings found in the standard files: normally, entries from files loaded with ReadMailcap will override the entries from files loaded previously (and the standard ones are loaded in the very beginning), but this will not happen if this parameter is set to true (default is false).

The return value is true if there were no errors in the file or false otherwise.

wxMimeTypesManager::ReadMimeTypes

bool ReadMimeTypes(**const wxString&** *filename*)

Load additional file containing information about MIME types and associated information in mime.types file format. See metamail(1) and mailcap(5) for more information.

The return value is true if there were no errors in the file or false otherwise.

wxMiniFrame

A miniframe is a frame with a small title bar. It is suitable for floating toolbars that must not take up too much screen area.

Derived from

wxFrame (p. 579)
wxWindow (p. 1490)
wxEvtHandler (p. 486)
wxObject (p. 1012)

Include files

<wx/minifram.h>

Window styles

wxICONIZE	Display the frame iconized (minimized) (Windows only).
wxCAPTION	Puts a caption on the frame.
wxDEFAULT_FRAME_STYLE	Defined as **wxMINIMIZE_BOX | wxMAXIMIZE_BOX | wxTHICK_FRAME | wxSYSTEM_MENU | wxCAPTION | wxCLOSE_BOX**.
wxMINIMIZE	Identical to **wxICONIZE**.
wxMINIMIZE_BOX	Displays a minimize box on the frame (Windows and Motif only).
wxMAXIMIZE	Displays the frame maximized (Windows only).
wxMAXIMIZE_BOX	Displays a maximize box on the frame (Windows and Motif

only).

wxCLOSE_BOX	Displays a close box on the frame.
wxSTAY_ON_TOP	Stay on top of other windows (Windows only).
wxSYSTEM_MENU	Displays a system menu (Windows and Motif only).
wxTHICK_FRAME	Displays a thick frame around the window (Windows and Motif only).
wxTINY_CAPTION_HORIZ	This style is obsolete and not used any longer.
wxTINY_CAPTION_VERT	This style is obsolete and not used any longer.
wxRESIZE_BORDER	Displays a resizeable border around the window (Motif only; for Windows, it is implicit in **wxTHICK_FRAME**).

See also *window styles overview* (p. 1762). Note that all the window styles above are ignored under GTK and the mini frame cannot be resized by the user.

Remarks

This class has miniframe functionality under Windows and GTK, i.e. the presence of mini frame will not be noted in the task bar and focus behaviour is different. On other platforms, it behaves like a normal frame.

See also

wxMDIParentFrame (p. 920), *wxMDIChildFrame* (p. 916), *wxFrame* (p. 579), *wxDialog* (p. 406)

wxMiniFrame::wxMiniFrame

wxMiniFrame()

Default constructor.

wxMiniFrame(**wxWindow*** *parent*, **wxWindowID** *id*, **const wxString&** *title*, **const wxPoint&** *pos* = *wxDefaultPosition*, **const wxSize&** *size* = *wxDefaultSize*, **long** *style* = *wxDEFAULT_FRAME_STYLE*, **const wxString&** *name* = *"frame"*)

Constructor, creating the window.

Parameters

parent

> The window parent. This may be NULL. If it is non-NULL, the frame will always be displayed on top of the parent window on Windows.

id

The window identifier. It may take a value of -1 to indicate a default value.

title

The caption to be displayed on the frame's title bar.

pos

The window position. A value of (-1, -1) indicates a default position, chosen by either the windowing system or wxWidgets, depending on platform.

size

The window size. A value of (-1, -1) indicates a default size, chosen by either the windowing system or wxWidgets, depending on platform.

style

The window style. See *wxMiniFrame* (p. 977).

name

The name of the window. This parameter is used to associate a name with the item, allowing the application user to set Motif resource values for individual windows.

Remarks

The frame behaves like a normal frame on non-Windows platforms.

See also

wxMiniFrame::Create (p. 979)

wxMiniFrame::~wxMiniFrame

void ~wxMiniFrame()

Destructor. Destroys all child windows and menu bar if present.

wxMiniFrame::Create

bool Create(**wxWindow*** *parent*, **wxWindowID** *id*, **const wxString&** *title*, **const wxPoint&** *pos* = *wxDefaultPosition*, **const wxSize&** *size* = *wxDefaultSize*, **long** *style* = *wxDEFAULT_FRAME_STYLE*, **const wxString&** *name = "frame"*)

Used in two-step frame construction. See *wxMiniFrame::wxMiniFrame* (p. 978) for further details.

wxMirrorDC

wxMirrorDC is a simple wrapper class which is always associated with a real *wxDC* (p. 365) object and either forwards all of its operations to it without changes (no mirroring takes place) or exchanges *x* and *y* coordinates which makes it possible to reuse the same code to draw a figure and its mirror -- i.e. reflection related to the diagonal line x == y.

wxMirrorDC has been added in wxWidgets version 2.5.0.

Derived from

wxDC (p. 365)

Include files

<wx/dcmirror.h>

wxMirrorDC::wxMirrorDC

 wxMirrorDC(**wxDC&** *dc*, **bool** *mirror*)

Creates a (maybe) mirrored DC associated with the real *dc*. Everything drawn on wxMirrorDC will appear (and maybe mirrored) on *dc*.

mirror specifies if we do mirror (if it is `true`) or not (if it is `false`).

wxModule

The module system is a very simple mechanism to allow applications (and parts of wxWidgets itself) to define initialization and cleanup functions that are automatically called on wxWidgets startup and exit.

To define a new kind of module, derive a class from wxModule, override the OnInit and OnExit functions, and add the DECLARE_DYNAMIC_CLASS and IMPLEMENT_DYNAMIC_CLASS to header and implementation files (which can be the same file). On initialization, wxWidgets will find all classes derived from wxModule, create an instance of each, and call each OnInit function. On exit, wxWidgets will call the OnExit function for each module instance.

Note that your module class does not have to be in a header file.

For example:

```
// A module to allow DDE initialization/cleanup
// without calling these functions from app.cpp or from
// the user's application.

class wxDDEModule: public wxModule
{
DECLARE_DYNAMIC_CLASS(wxDDEModule)
public:
```

```
        wxDDEModule() {}
        bool OnInit() { wxDDEInitialize(); return true; };
        void OnExit() { wxDDECleanUp(); };
    };

    IMPLEMENT_DYNAMIC_CLASS(wxDDEModule, wxModule)
```

Derived from

wxObject (p. 1012)

Include files

<wx/module.h>

wxModule::wxModule

wxModule()

Constructs a wxModule object.

wxModule::~wxModule

~wxModule()

Destructor.

wxModule::OnExit

virtual void OnExit()

Provide this function with appropriate cleanup for your module.

wxModule::OnInit

virtual bool OnInit()

Provide this function with appropriate initialization for your module. If the function returns false, wxWidgets will exit immediately.

wxMouseCaptureChangedEvent

An mouse capture changed event is sent to a window that loses its mouse capture. This is called even if wxWindow::ReleaseCapture was called by the application code. Handling this event allows an application to cater for unexpected capture releases which might otherwise confuse mouse handling code.

This event is implemented under Windows only.

Derived from

wxEvent (p. 483)
wxObject (p. 1012)

Include files

<wx/event.h>

Event table macros

To process this event, use the following event handler macro to direct input to a member function that takes a wxMouseCaptureChangedEvent argument.

EVT_MOUSE_CAPTURE_CHANGED(func)   Process a
                                                wxEVT_MOUSE_CAPTURE_CHANGED
                                                event.

See also

Event handling overview (p. 1751), *wxWindow::CaptureMouse* (p. 1494),
wxWindow::ReleaseMouse (p. 1524), *wxWindow::GetCapture* (p. 1505)

wxMouseCaptureChangedEvent::wxMouseCaptureChangedEvent

 wxMouseCaptureChangedEvent(**wxWindowID** *windowId = 0*, **wxWindow***
gainedCapture = NULL)

Constructor.

wxActivateEvent::GetCapturedWindow

wxWindow* GetCapturedWindow() **const**

Returns the window that gained the capture, or NULL if it was a non-wxWidgets window.

wxMouseEvent

This event class contains information about the events generated by the mouse: they include mouse buttons press and release events and mouse move events.

All mouse events involving the buttons use wxMOUSE_BTN_LEFT for the left mouse button, wxMOUSE_BTN_MIDDLE for the middle one and wxMOUSE_BTN_RIGHT for the right one. Note that not all mice have a middle button so a portable application should avoid relying on the events from it.

NB: Note that under Windows CE mouse enter and leave events are not natively supported by the system but are generated by wxWidgets itself. This has several drawbacks: the LEAVE_WINDOW event might be received some time after the mouse left the window and the state variables for it may have changed during this time.

NB: Note the difference between methods like*LeftDown* (p. 990) and*LeftIsDown* (p. 990): the former returns `true`when the event corresponds to the left mouse button click while the latter returns `true` if the left mouse button is currently being pressed. For example, when the user is dragging the mouse you can use*LeftIsDown* (p. 990) to test whether the left mouse button is (still) depressed. Also, by convention, if*LeftDown* (p. 990) returns `true`,*LeftIsDown* (p. 990) will also return `true` in wxWidgets whatever the underlying GUI behaviour is (which is platform-dependent). The same applies, of course, to other mouse buttons as well.

Derived from

wxEvent (p. 483)

Include files

<wx/event.h>

Event table macros

To process a mouse event, use these event handler macros to direct input to member functions that take a wxMouseEvent argument.

EVT_LEFT_DOWN(func)	Process a wxEVT_LEFT_DOWN event. The handler of this event should normally call *event.Skip()* (p. 486) to allow the default processing to take place as otherwise the window under mouse wouldn't get the focus.
EVT_LEFT_UP(func)	Process a wxEVT_LEFT_UP event.
EVT_LEFT_DCLICK(func)	Process a wxEVT_LEFT_DCLICK event.
EVT_MIDDLE_DOWN(func)	Process a wxEVT_MIDDLE_DOWN event.
EVT_MIDDLE_UP(func)	Process a wxEVT_MIDDLE_UP event.
EVT_MIDDLE_DCLICK(func)	Process a wxEVT_MIDDLE_DCLICK event.
EVT_RIGHT_DOWN(func)	Process a wxEVT_RIGHT_DOWN event.
EVT_RIGHT_UP(func)	Process a wxEVT_RIGHT_UP event.
EVT_RIGHT_DCLICK(func)	Process a wxEVT_RIGHT_DCLICK event.
EVT_MOTION(func)	Process a wxEVT_MOTION event.
EVT_ENTER_WINDOW(func)	Process a wxEVT_ENTER_WINDOW event.
EVT_LEAVE_WINDOW(func)	Process a wxEVT_LEAVE_WINDOW event.

EVT_MOUSEWHEEL(func)	Process a wxEVT_MOUSEWHEEL event.
EVT_MOUSE_EVENTS(func)	Process all mouse events.

wxMouseEvent::m_altDown

bool m_altDown

true if the Alt key is pressed down.

wxMouseEvent::m_controlDown

bool m_controlDown

true if control key is pressed down.

wxMouseEvent::m_leftDown

bool m_leftDown

true if the left mouse button is currently pressed down.

wxMouseEvent::m_middleDown

bool m_middleDown

true if the middle mouse button is currently pressed down.

wxMouseEvent::m_rightDown

bool m_rightDown

true if the right mouse button is currently pressed down.

wxMouseEvent::m_metaDown

bool m_metaDown

true if the Meta key is pressed down.

wxMouseEvent::m_shiftDown

bool m_shiftDown

true if shift is pressed down.

wxMouseEvent::m_x

long m_x

X-coordinate of the event.

wxMouseEvent::m_y

long m_y

Y-coordinate of the event.

wxMouseEvent::m_wheelRotation

int m_wheelRotation

The distance the mouse wheel is rotated.

wxMouseEvent::m_wheelDelta

int m_wheelDelta

The wheel delta, normally 120.

wxMouseEvent::m_linesPerAction

int m_linesPerAction

The configured number of lines (or whatever) to be scrolled per wheel action.

wxMouseEvent::wxMouseEvent

wxMouseEvent(**WXTYPE** *mouseEventType = 0*)

Constructor. Valid event types are:

- **wxEVT_ENTER_WINDOW**
- **wxEVT_LEAVE_WINDOW**
- **wxEVT_LEFT_DOWN**
- **wxEVT_LEFT_UP**

- • **wxEVT_LEFT_DCLICK**

- • **wxEVT_MIDDLE_DOWN**

- • **wxEVT_MIDDLE_UP**

- • **wxEVT_MIDDLE_DCLICK**

- • **wxEVT_RIGHT_DOWN**

- • **wxEVT_RIGHT_UP**

- • **wxEVT_RIGHT_DCLICK**

- • **wxEVT_MOTION**

- • **wxEVT_MOUSEWHEEL**

wxMouseEvent::AltDown

bool AltDown()

Returns true if the Alt key was down at the time of the event.

wxMouseEvent::Button

bool Button(**int** *button*)

Returns true if the identified mouse button is changing state. Valid values of *button* are:

wxMOUSE_BTN_LEFT	check if left button was pressed
wxMOUSE_BTN_MIDDLE	check if middle button was pressed
wxMOUSE_BTN_RIGHT	check if right button was pressed
wxMOUSE_BTN_ANY	check if any button was pressed

wxMouseEvent::ButtonDClick

bool ButtonDClick(**int** *but = wxMOUSE_BTN_ANY*)

If the argument is omitted, this returns true if the event was a mouse double click event. Otherwise the argument specifies which double click event was generated (see *Button* (p. 986) for the possible values).

wxMouseEvent::ButtonDown

bool ButtonDown(**int** *but = -1*)

If the argument is omitted, this returns true if the event was a mouse button down event. Otherwise the argument specifies which button-down event was generated (see *Button* (p. 986) for the possible values).

wxMouseEvent::ButtonUp

bool ButtonUp(**int** *but = -1*)

If the argument is omitted, this returns true if the event was a mouse button up event. Otherwise the argument specifies which button-up event was generated (see *Button* (p. 986) for the possible values).

wxMouseEvent::CmdDown

bool CmdDown() **const**

Same as *MetaDown* (p. 990) under Mac, same as *ControlDown* (p. 987) elsewhere.

See also

wxKeyEvent::CmdDown (p. 832)

wxMouseEvent::ControlDown

bool ControlDown()

Returns true if the control key was down at the time of the event.

wxMouseEvent::Dragging

bool Dragging()

Returns true if this was a dragging event (motion while a button is depressed).

See also

Moving (p. 991)

wxMouseEvent::Entering

bool Entering()

Returns true if the mouse was entering the window.

See also *wxMouseEvent::Leaving* (p. 989).

wxMouseEvent::GetButton

int GetButton() **const**

Returns the mouse button which generated this event or `wxMOUSE_BTN_NONE` if no button is involved (for mouse move, enter or leave event, for example). Otherwise `wxMOUSE_BTN_LEFT` is returned for the left button down, up and double click events, `wxMOUSE_BTN_MIDDLE` and `wxMOUSE_BTN_RIGHT` for the same events for the middle and the right buttons respectively.

wxMouseEvent::GetPosition

wxPoint GetPosition() **const**

void GetPosition(**wxCoord*** *x*, **wxCoord*** *y*) **const**

void GetPosition(**long*** *x*, **long*** *y*) **const**

Sets *x and *y to the position at which the event occurred.

Returns the physical mouse position in pixels.

Note that if the mouse event has been artificially generated from a special keyboard combination (e.g. under Windows when the "menu" key is pressed), the returned position is `wxDefaultPosition`.

wxMouseEvent::GetLogicalPosition

wxPoint GetLogicalPosition(**const wxDC&** *dc*) **const**

Returns the logical mouse position in pixels (i.e. translated according to the translation set for the DC, which usually indicates that the window has been scrolled).

wxMouseEvent::GetLinesPerAction

int GetLinesPerAction() **const**

Returns the configured number of lines (or whatever) to be scrolled per wheel action. Defaults to three.

wxMouseEvent::GetWheelRotation

int GetWheelRotation() **const**

Get wheel rotation, positive or negative indicates direction of rotation. Current devices all send an event when rotation is equal to +/-WheelDelta, but this allows for finer resolution devices to be created in the future. Because of this you shouldn't assume that one event is equal to 1 line or whatever, but you should be able to either do partial line

scrolling or wait until +/-WheelDelta rotation values have been accumulated before scrolling.

wxMouseEvent::GetWheelDelta

int GetWheelDelta() **const**

Get wheel delta, normally 120. This is the threshold for action to be taken, and one such action (for example, scrolling one increment) should occur for each delta.

wxMouseEvent::GetX

long GetX() **const**

Returns X coordinate of the physical mouse event position.

wxMouseEvent::GetY

long GetY()

Returns Y coordinate of the physical mouse event position.

wxMouseEvent::IsButton

bool IsButton() **const**

Returns true if the event was a mouse button event (not necessarily a button down event - that may be tested using *ButtonDown*).

wxMouseEvent::IsPageScroll

bool IsPageScroll() **const**

Returns true if the system has been setup to do page scrolling with the mouse wheel instead of line scrolling.

wxMouseEvent::Leaving

bool Leaving() **const**

Returns true if the mouse was leaving the window.

See also *wxMouseEvent::Entering* (p. 987).

wxMouseEvent::LeftDClick

bool LeftDClick() **const**

Returns true if the event was a left double click.

wxMouseEvent::LeftDown

bool LeftDown() **const**

Returns true if the left mouse button changed to down.

wxMouseEvent::LeftIsDown

bool LeftIsDown() **const**

Returns true if the left mouse button is currently down, independent of the current event type.

Please notice that it is **not** the same as*LeftDown* (p. 990) which returns true if the left mouse button was just pressed. Rather, it describes the state of the mouse button before the event happened.

This event is usually used in the mouse event handlers which process "move mouse" messages to determine whether the user is (still) dragging the mouse.

wxMouseEvent::LeftUp

bool LeftUp() **const**

Returns true if the left mouse button changed to up.

wxMouseEvent::MetaDown

bool MetaDown() **const**

Returns true if the Meta key was down at the time of the event.

wxMouseEvent::MiddleDClick

bool MiddleDClick() **const**

Returns true if the event was a middle double click.

wxMouseEvent::MiddleDown

bool MiddleDown() **const**

Returns true if the middle mouse button changed to down.

wxMouseEvent::MiddleIsDown

bool MiddleIsDown() const

Returns true if the middle mouse button is currently down, independent of the current event type.

wxMouseEvent::MiddleUp

bool MiddleUp() const

Returns true if the middle mouse button changed to up.

wxMouseEvent::Moving

bool Moving() const

Returns true if this was a motion event and no mouse buttons were pressed. If any mouse button is held pressed, then this method returns `false` and *Dragging* (p. 987) returns `true`.

wxMouseEvent::RightDClick

bool RightDClick() const

Returns true if the event was a right double click.

wxMouseEvent::RightDown

bool RightDown() const

Returns true if the right mouse button changed to down.

wxMouseEvent::RightIsDown

bool RightIsDown() const

Returns true if the right mouse button is currently down, independent of the current event type.

wxMouseEvent::RightUp

bool RightUp() **const**

Returns true if the right mouse button changed to up.

wxMouseEvent::ShiftDown

bool ShiftDown() **const**

Returns true if the shift key was down at the time of the event.

wxMoveEvent

A move event holds information about move change events.

Derived from

wxEvent (p. 483)
wxObject (p. 1012)

Include files

<wx/event.h>

Event table macros

To process a move event, use this event handler macro to direct input to a member function that takes a wxMoveEvent argument.

EVT_MOVE(func)	Process a wxEVT_MOVE event, which is generated when a window is moved.

See also

wxPoint (p. 1045), *Event handling overview* (p. 1751)

wxMoveEvent::wxMoveEvent

wxMoveEvent(**const wxPoint&** *pt*, **int** *id = 0*)

Constructor.

wxMoveEvent::GetPosition

wxPoint GetPosition() **const**

Returns the position of the window generating the move change event.

wxMultiChoiceDialog

This class represents a dialog that shows a list of strings, and allows the user to select one or more.

Derived from

wxDialog (p. 406)
wxWindow (p. 1490)
wxEvtHandler (p. 486)
wxObject (p. 1012)

Include files

<wx/choicdlg.h>

See also

wxMultiChoiceDialog overview (p. 1801),*wxSingleChoiceDialog* (p. 1169)

wxMultiChoiceDialog::wxMultiChoiceDialog

 wxMultiChoiceDialog(**wxWindow*** *parent*, **const wxString&** *message*, **const wxString&** *caption*, **int** *n*, **const wxString*** *choices*, **long** *style* = *wxCHOICEDLG_STYLE*, **const wxPoint&** *pos* = *wxDefaultPosition*)

 wxMultiChoiceDialog(**wxWindow*** *parent*, **const wxString&** *message*, **const wxString&** *caption*, **const wxArrayString&** *choices*, **long** *style* = *wxCHOICEDLG_STYLE*, **const wxPoint&** *pos* = *wxDefaultPosition*)

Constructor taking an array of wxString choices.

Parameters

parent

      Parent window.

message

      Message to show on the dialog.

caption

      The dialog caption.

n

      The number of choices.

choices

An array of strings, or a string list, containing the choices.

style

A dialog style (bitlist) containing flags chosen from standard dialog styles and the following:

wxOK	Show an OK button.
wxCANCEL	Show a Cancel button.
wxCENTRE	Centre the message. Not Windows.

The default value is equivalent to **wxDEFAULT_DIALOG_STYLE | wxRESIZE_BORDER | wxOK | wxCANCEL | wxCENTRE**.

pos

Dialog position. Not Windows.

Remarks

Use *wxMultiChoiceDialog::ShowModal* (p. 994) to show the dialog.

wxPython note: For Python the two parameters n and choices are collapsed into a multi parameter choices which is expected to be a Python list of strings.

wxPerl note: In wxPerl there is just an array reference in place of n.

wxMultiChoiceDialog::GetSelections

wxArrayInt GetSelection() **const**

Returns array with indexes of selected items.

wxMultiChoiceDialog::SetSelections

void SetSelections(**const wxArrayInt&** *selections*) **const**

Sets selected items from the array of selected items' indexes.

wxMultiChoiceDialog::ShowModal

int ShowModal()

Shows the dialog, returning either wxID_OK or wxID_CANCEL.

wxMutex

A mutex object is a synchronization object whose state is set to signaled when it is not

owned by any thread, and nonsignaled when it is owned. Its name comes from its usefulness in coordinating mutually-exclusive access to a shared resource as only one thread at a time can own a mutex object.

Mutexes may be recursive in the sense that a thread can lock a mutex which it had already locked before (instead of dead locking the entire process in this situation by starting to wait on a mutex which will never be released while the thread is waiting) but using them is not recommended and they are **not**recursive by default. The reason for this is that recursive mutexes are not supported by all Unix flavours and, worse, they cannot be used with *wxCondition* (p. 189).

For example, when several thread use the data stored in the linked list, modifications to the list should be only allowed to one thread at a time because during a new node addition the list integrity is temporarily broken (this is also called *program invariant*).

Example

```
     // this variable has an "s_" prefix because it is static: seeing an
"s_" in
     // a multithreaded program is in general a good sign that you should
use a
     // mutex (or a critical section)
     static wxMutex *s_mutexProtectingTheGlobalData;

     // we store some numbers in this global array which is presumably used
by
     // several threads simultaneously
     wxArrayInt s_data;

     void MyThread::AddNewNode(int num)
     {
         // ensure that no other thread accesses the list
         s_mutexProtectingTheGlobalList->Lock();

         s_data.Add(num);

         s_mutexProtectingTheGlobalList->Unlock();
     }

     // return true the given number is greater than all array elements
     bool MyThread::IsGreater(int num)
     {
         // before using the list we must acquire the mutex
         wxMutexLocker lock(s_mutexProtectingTheGlobalData);

         size_t count = s_data.Count();
         for ( size_t n = 0; n < count; n++ )
         {
             if ( s_data[n] > num )
                 return false;
         }

         return true;
     }
```

Notice how wxMutexLocker was used in the second function to ensure that the mutex is unlocked in any case: whether the function returns true or false (because the destructor

of the local object *lock* is always called). Using this class instead of directly using wxMutex is, in general safer and is even more so if your program uses C++ exceptions.

Constants

```
enum wxMutexType
{
    // normal mutex: try to always use this one
    wxMUTEX_DEFAULT,

    // recursive mutex: don't use these ones with wxCondition
    wxMUTEX_RECURSIVE
};
```

Derived from

None.

Include files

<wx/thread.h>

See also

wxThread (p. 1377), *wxCondition* (p. 189), *wxMutexLocker* (p. 997), *wxCriticalSection* (p. 223)

wxMutex::wxMutex

wxMutex(**wxMutexType** *type* = *wxMUTEX_DEFAULT*)

Default constructor.

wxMutex::~wxMutex

~wxMutex()

Destroys the wxMutex object.

wxMutex::Lock

wxMutexError Lock()

Locks the mutex object.

Return value

One of:

wxMUTEX_NO_ERROR   There was no error.

wxMUTEX_DEAD_LOCK	A deadlock situation was detected.
wxMUTEX_BUSY	The mutex is already locked by another thread.

wxMutex::TryLock

wxMutexError TryLock()

Tries to lock the mutex object. If it can't, returns immediately with an error.

Return value

One of:

wxMUTEX_NO_ERROR	There was no error.
wxMUTEX_DEAD_LOCK	A deadlock situation was detected.
wxMUTEX_BUSY	The mutex is already locked by another thread.

wxMutex::Unlock

wxMutexError Unlock()

Unlocks the mutex object.

Return value

One of:

wxMUTEX_NO_ERROR	There was no error.
wxMUTEX_DEAD_LOCK	A deadlock situation was detected.
wxMUTEX_BUSY	The mutex is already locked by another thread.
wxMUTEX_UNLOCKED	The calling thread tries to unlock an unlocked mutex.

wxMutexLocker

This is a small helper class to be used with *wxMutex* (p. 994) objects. A wxMutexLocker acquires a mutex lock in the constructor and releases (or unlocks) the mutex in the destructor making it much more difficult to forget to release a mutex (which, in general, will promptly lead to the serious problems). See *wxMutex* (p. 994) for an example of wxMutexLocker usage.

Derived from

None.

Include files

<wx/thread.h>

See also

wxMutex (p. 994), *wxCriticalSectionLocker* (p. 224)

wxMutexLocker::wxMutexLocker

 wxMutexLocker(**wxMutex&** *mutex*)

Constructs a wxMutexLocker object associated with mutex and locks it. Call *IsLocked* (p. 998) to check if the mutex was successfully locked.

wxMutexLocker::~wxMutexLocker

 ~wxMutexLocker()

Destructor releases the mutex if it was successfully acquired in the ctor.

wxMutexLocker::IsOk

bool IsOk() **const**

Returns true if mutex was acquired in the constructor, false otherwise.

wxNode

wxNodeBase is the node structure used in linked lists (see *wxList* (p. 840)) and derived classes. You should never use wxNodeBase class directly, however, because it works with untyped (`void *`) data and this is unsafe. Use wxNodeBase-derived classes which are automatically defined by WX_DECLARE_LIST and WX_DEFINE_LIST macros instead as described in*wxList* (p. 840) documentation (see example there). Also note that although there is a class called wxNode, it is defined for backwards compatibility only and usage of this class is strongly deprecated.

In the documentation below, the type `T` should be thought of as a "template" parameter: this is the type of data stored in the linked list or, in other words, the first argument of WX_DECLARE_LIST macro. Also, wxNode is written as wxNode<T> even though it isn't really a template class -- but it helps to think of it as if it were.

Derived from

None.

Include files

<wx/list.h>

See also

wxList (p. 840), *wxHashTable* (p. 693)

wxNode<T>::GetData

T * GetData() **const**

Retrieves the client data pointer associated with the node.

wxNode<T>::GetNext

wxNode<T> * GetNext() **const**

Retrieves the next node or NULL if this node is the last one.

wxNode<T>::GetPrevious

wxNode<T> * GetPrevious()

Retrieves the previous node or NULL if this node is the first one in the list.

wxNode<T>::SetData

void SetData(**T ****data*)

Sets the data associated with the node (usually the pointer will have been set when the node was created).

wxNode<T>::IndexOf

int IndexOf()

Returns the zero-based index of this node within the list. The return value will be `wxNOT_FOUND` if the node has not been added to a list yet.

wxNotebook

This class represents a notebook control, which manages multiple windows with associated tabs.

To use the class, create a wxNotebook object and call *AddPage* (p. 1002) or *InsertPage* (p. 1006), passing a window to be used as the page. Do not explicitly delete the window for a page that is currently managed by wxNotebook.

wxNotebookPage is a typedef for wxWindow.

Derived from

wxControl (p. 215)
wxWindow (p. 1490)
wxEvtHandler (p. 486)
wxObject (p. 1012)

Include files

<wx/notebook.h>

Window styles

wxNB_TOP	Place tabs on the top side.
wxNB_LEFT	Place tabs on the left side.
wxNB_RIGHT	Place tabs on the right side.
wxNB_BOTTOM	Place tabs under instead of above the notebook pages.
wxNB_FIXEDWIDTH	(Windows only) All tabs will have same width.
wxNB_MULTILINE	(Windows only) There can be several rows of tabs.
wxNB_NOPAGETHEME	(Windows only) Display a solid colour on notebook pages, and not a gradient, which can reduce performance.
wxNB_FLAT	(Windows CE only) Show tabs in a flat style.

The styles `wxNB_LEFT`, `RIGHT` and `BOTTOM` are not supported under Microsoft Windows XP when using visual themes.

See also *window styles overview* (p. 1762).

Event handling

To process input from a notebook control, use the following event handler macros to direct input to member functions that take a *wxNotebookEvent* (p. 1008) argument.

EVT_NOTEBOOK_PAGE_CHANGED(id, func)   The page selection was changed. Processes a wxEVT_COMMAND_NOTEBOOK_PAGE_CHANGED event.

EVT_NOTEBOOK_PAGE_CHANGING(id, func)   The page selection is about to be changed. Processes a wxEVT_COMMAND_NOTEBOOK_PAGE_CHANGING event. This event can be *vetoed* (p.

1012).

Page backgrounds

On Windows XP, the default theme paints a gradient on the notebook's pages. If you wish to suppress this theme, for aesthetic or performance reasons, there are three ways of doing it. You can use wxNB_NOPAGETHEME to disable themed drawing for a particular notebook, you can call `wxSystemOptions::SetOption` to disable it for the whole application, or you can disable it for individual pages by using `SetBackgroundColour`.

To disable themed pages globally:

```
wxSystemOptions::SetOption(wxT("msw.notebook.themed-background"), 0);
```

Set the value to 1 to enable it again.

To give a single page a solid background that more or less fits in with the overall theme, use:

```
wxColour col = notebook->GetThemeBackgroundColour();
if (col.Ok())
{
    page->SetBackgroundColour(col);
}
```

On platforms other than Windows, or if the application is not using Windows themes, `GetThemeBackgroundColour` will return an uninitialised colour object, and the above code will therefore work on all platforms.

See also

wxBookCtrl (p. 1797), *wxNotebookEvent* (p. 1008), *wxImageList* (p. 806), *notebook sample* (p. 1713)

wxNotebook::wxNotebook

wxNotebook()

Default constructor.

wxNotebook(**wxWindow*** *parent*, **wxWindowID** *id*, **const wxPoint&** *pos = wxDefaultPosition*, **const wxSize&** *size = wxDefaultSize*, **long** *style = 0*, **const wxString&** *name = wxNotebookNameStr*)

Constructs a notebook control.

Note that sometimes you can reduce flicker by passing the wxCLIP_CHILDREN window style.

Parameters

parent

> The parent window. Must be non-NULL.

id

> The window identifier.

pos

> The window position.

size

> The window size.

style

> The window style. See *wxNotebook* (p. 999).

name

> The name of the control (used only under Motif).

wxNotebook::~wxNotebook

 ~wxNotebook()

Destroys the wxNotebook object.

wxNotebook::AddPage

bool AddPage(**wxNotebookPage*** *page*, **const wxString&** *text*, **bool** *select = false*, **int** *imageId = -1*)

Adds a new page.

The call to this function may generate the page changing events.

Parameters

page

> Specifies the new page.

text

> Specifies the text for the new page.

select

> Specifies whether the page should be selected.

imageId

> Specifies the optional image index for the new page.

Return value

true if successful, false otherwise.

Remarks

Do not delete the page, it will be deleted by the notebook.

See also

wxNotebook::InsertPage (p. 1006)

wxNotebook::AdvanceSelection

void AdvanceSelection(**bool** *forward = true*)

Cycles through the tabs.

The call to this function generates the page changing events.

wxNotebook::AssignImageList

void AssignImageList(**wxImageList*** *imageList*)

Sets the image list for the page control and takes ownership of the list.

See also

wxImageList (p. 806),*SetImageList* (p. 1007)

wxNotebook::Create

bool Create(**wxWindow*** *parent*, **wxWindowID** *id*, **const wxPoint&** *pos* = *wxDefaultPosition*, **const wxSize&** *size*, **long** *style = 0*, **const wxString&** *name* = *wxNotebookNameStr*)

Creates a notebook control. See *wxNotebook::wxNotebook* (p. 1001) for a description of the parameters.

wxNotebook::DeleteAllPages

bool DeleteAllPages()

Deletes all pages.

wxNotebook::DeletePage

bool DeletePage(**size_t** *page*)

Deletes the specified page, and the associated window.

The call to this function generates the page changing events.

wxNotebook::GetCurrentPage

wxWindow * GetCurrentPage() **const**

Returns the currently selected notebook age or NULL.

wxNotebook::GetImageList

wxImageList* GetImageList() **const**

Returns the associated image list.

See also

wxImageList (p. 806), *wxNotebook::SetImageList* (p. 1007)

wxNotebook::GetPage

wxNotebookPage* GetPage(**size_t** *page*)

Returns the window at the given page position.

wxNotebook::GetPageCount

size_t GetPageCount() **const**

Returns the number of pages in the notebook control.

wxNotebook::GetPageImage

int GetPageImage(**size_t** *nPage*) **const**

Returns the image index for the given page.

wxNotebook::GetPageText

wxString GetPageText(**size_t** *nPage*) **const**

Returns the string for the given page.

wxNotebook::GetRowCount

int GetRowCount() **const**

Returns the number of rows in the notebook control.

wxNotebook::GetSelection

int GetSelection() **const**

Returns the currently selected page, or -1 if none was selected.

Note that this method may return either the previously or newly selected page when called from the EVT_NOTEBOOK_PAGE_CHANGED handler depending on the platform and so *wxNotebookEvent::GetSelection* (p. 1009) should be used instead in this case.

wxNotebook::GetThemeBackgroundColour

wxColour GetThemeBackgroundColour() **const**

If running under Windows and themes are enabled for the application, this function returns a suitable colour for painting the background of a notebook page, and can be passed to SetBackgroundColour. Otherwise, an uninitialised colour will be returned.

wxNotebook::HitTest

int HitTest(**const wxPoint&** *pt*, **long** **flags = NULL*)

Returns the index of the tab at the specified position or wxNOT_FOUNDif none. If *flags* parameter is non NULL, the position of the point inside the tab is returned as well.

NB: This method is currently only implemented under wxMSW and wxUniv.

Parameters

pt

Specifies the point for the hit test.

flags

Return value for detailed information. One of the following values:**wxNB_HITTEST_NOWHERE**
There was no tab under this point.

wxNB_HITTEST_ONICON   The point was over an icon (currently wxMSW only).

| **wxNB_HITTEST_ONLABEL** | The point was over a label (currently wxMSW only). |
| **wxNB_HITTEST_ONITEM** | The point was over an item, but not on the label or icon. |

Return value

Returns the zero-based tab index or `wxNOT_FOUND` if there is no tab is at the specified position.

wxNotebook::InsertPage

bool InsertPage(**size_t** *index*, **wxNotebookPage*** *page*, **const wxString&** *text*, **bool** *select = false*, **int** *imageId = -1*)

Inserts a new page at the specified position.

Parameters

index

> Specifies the position for the new page.

page

> Specifies the new page.

text

> Specifies the text for the new page.

select

> Specifies whether the page should be selected.

imageId

> Specifies the optional image index for the new page.

Return value

true if successful, false otherwise.

Remarks

Do not delete the page, it will be deleted by the notebook.

See also

wxNotebook::AddPage (p. 1002)

wxNotebook::OnSelChange

void OnSelChange(**wxNotebookEvent&** *event*)

An event handler function, called when the page selection is changed.

See also

wxNotebookEvent (p. 1008)

wxNotebook::RemovePage

bool RemovePage(**size_t** *page*)

Deletes the specified page, without deleting the associated window.

wxNotebook::SetImageList

void SetImageList(**wxImageList*** *imageList*)

Sets the image list for the page control. It does not take ownership of the image list, you must delete it yourself.

See also

wxImageList (p. 806),*AssignImageList* (p. 1003)

wxNotebook::SetPadding

void SetPadding(**const wxSize&** *padding*)

Sets the amount of space around each page's icon and label, in pixels.

NB: The vertical padding cannot be changed in wxGTK.

wxNotebook::SetPageSize

void SetPageSize(**const wxSize&** *size*)

Sets the width and height of the pages.

NB: This method is currently not implemented for wxGTK.

wxNotebook::SetPageImage

bool SetPageImage(**size_t** *page*, **int** *image*)

Sets the image index for the given page. *image* is an index into the image list which was

set with *wxNotebook::SetImageList* (p. 1007).

wxNotebook::SetPageText

bool SetPageText(**size_t** *page*, **const wxString&** *text*)

Sets the text for the given page.

wxNotebook::SetSelection

int SetSelection(**size_t** *page*)

Sets the selection for the given page, returning the previous selection.

The call to this function generates the page changing events.

See also

wxNotebook::GetSelection (p. 1005)

wxNotebookEvent

This class represents the events generated by a notebook control: currently, there are two of them. The PAGE_CHANGING event is sent before the current page is changed. It allows the program to examine the current page (which can be retrieved with *GetOldSelection()* (p. 1009)) and to veto the page change by calling *Veto()* (p. 1012) if, for example, the current values in the controls of the old page are invalid.

The second event - PAGE_CHANGED - is sent after the page has been changed and the program cannot veto it any more, it just informs it about the page change.

To summarize, if the program is interested in validating the page values before allowing the user to change it, it should process the PAGE_CHANGING event, otherwise PAGE_CHANGED is probably enough. In any case, it is probably unnecessary to process both events at once.

Derived from

wxNotifyEvent (p. 1011)
wxCommandEvent (p. 180)
wxEvent (p. 483)
wxObject (p. 1012)

Include files

<wx/notebook.h>

Event handling

To process input from a notebook control, use the following event handler macros to

direct input to member functions that take a *wxNotebookEvent* (p. 1008) argument.

EVT_NOTEBOOK_PAGE_CHANGED(id, func)   The page selection was changed. Processes a wxEVT_COMMAND_NOTEBOOK_PAGE_CHANGED event.

EVT_NOTEBOOK_PAGE_CHANGING(id, func)   The page selection is about to be changed. Processes a wxEVT_COMMAND_NOTEBOOK_PAGE_CHANGING event. This event can be *vetoed* (p. 1012).

See also

wxNotebook (p. 999)

wxNotebookEvent::wxNotebookEvent

 wxNotebookEvent(**wxEventType** *eventType = wxEVT_NULL*, **int** *id = 0*, **int** *sel = -1*, **int** *oldSel = -1*)

Constructor (used internally by wxWidgets only).

wxNotebookEvent::GetOldSelection

int GetOldSelection() **const**

Returns the page that was selected before the change, -1 if none was selected.

wxNotebookEvent::GetSelection

int GetSelection() **const**

Returns the currently selected page, or -1 if none was selected.

NB: under Windows, GetSelection() will return the same value as *GetOldSelection()* (p. 1009) when called from EVT_NOTEBOOK_PAGE_CHANGING handler and not the page which is going to be selected. Also note that the values of selection and old selection returned for an event generated in response to a call to *wxNotebook::SetSelection* (p. 1008) shouldn't be trusted as they are currently inconsistent under different platforms (but in this case you presumably don't need them anyhow as you already have the corresponding information).

wxNotebookEvent::SetOldSelection

void SetOldSelection(**int** *page*)

Sets the id of the page selected before the change.

wxNotebookEvent::SetSelection

void SetSelection(**int** *page*)

Sets the selection member variable.

See also

wxNotebookEvent::GetSelection (p. 1009)

wxNotebookSizer

This class is deprecated and should not be used in new code! It is no longer needed, *wxNotebook* **(p. 999) control can be inserted into any sizer class and its minimal size will be determined correctly. See** *wxSizer overview* **(p. 1771) for more information.**

wxNotebookSizer is a specialized sizer to make sizers work in connection with using notebooks. This sizer is different from any other sizer as you must not add any children to it - instead, it queries the notebook class itself. The only thing this sizer does is to determine the size of the biggest page of the notebook and report an adjusted minimal size to a more toplevel sizer.

Derived from

wxSizer (p. 1176)
wxObject (p. 1012)

Include files

<wx/sizer.h>

See also

wxSizer (p. 1176), *wxNotebook* (p. 999), *Sizer overview* (p. 1771)

wxNotebookSizer::wxNotebookSizer

 wxNotebookSizer(**wxNotebook*** *notebook*)

Constructor. It takes an associated notebook as its only parameter.

wxNotebookSizer::GetNotebook

wxNotebook* GetNotebook()

Returns the notebook associated with the sizer.

wxNotifyEvent

This class is not used by the event handlers by itself, but is a base class for other event classes (such as *wxNotebookEvent* (p. 1008)).

It (or an object of a derived class) is sent when the controls state is being changed and allows the program to *Veto()* (p. 1012) this change if it wants to prevent it from happening.

Derived from

wxCommandEvent (p. 180)
wxEvent (p. 483)
wxEvtHandler (p. 486)
wxObject (p. 1012)

Include files

<wx/event.h>

Event table macros

None

See also

wxNotebookEvent (p. 1008)

wxNotifyEvent::wxNotifyEvent

wxNotifyEvent(**wxEventType** *eventType = wxEVT_NULL*, **int** *id = 0*)

Constructor (used internally by wxWidgets only).

wxNotifyEvent::Allow

void Allow()

This is the opposite of *Veto()* (p. 1012): it explicitly allows the event to be processed. For most events it is not necessary to call this method as the events are allowed anyhow but some are forbidden by default (this will be mentioned in the corresponding event description).

wxNotifyEvent::IsAllowed

bool IsAllowed() **const**

Returns true if the change is allowed (*Veto()* (p. 1012) hasn't been called) or false otherwise (if it was).

wxNotifyEvent::Veto

void Veto()

Prevents the change announced by this event from happening.

It is in general a good idea to notify the user about the reasons for vetoing the change because otherwise the applications behaviour (which just refuses to do what the user wants) might be quite surprising.

wxObject

This is the root class of all wxWidgets classes. It declares a virtual destructor which ensures that destructors get called for all derived class objects where necessary.

wxObject is the hub of a dynamic object creation scheme, enabling a program to create instances of a class only knowing its string class name, and to query the class hierarchy.

The class contains optional debugging versions of **new** and **delete**, which can help trace memory allocation and deallocation problems.

wxObject can be used to implement reference counted objects, such as wxPen, wxBitmap and others.

See also

wxClassInfo (p. 144), *Debugging overview* (p. 1746), *wxObjectRefData* (p. 1015)

wxObject::wxObject

 wxObject()

 wxObject(**const wxObject&** *other*)

Default and copy constructors.

wxObject::~wxObject

 wxObject()

Destructor. Performs dereferencing, for those objects that use reference counting.

wxObject::m_refData

wxObjectRefData* m_refData

Pointer to an object which is the object's reference-counted data.

See also

wxObject::Ref (p. 1014), *wxObject::UnRef* (p. 1015), *wxObject::SetRefData* (p. 1015), *wxObject::GetRefData* (p. 1013), *wxObjectRefData* (p. 1015)

wxObject::Dump

void Dump(**ostream&** *stream*)

A virtual function that may be redefined by derived classes to allow dumping of memory states.

This function is only defined in debug build and doesn't exist at all if `__WXDEBUG__` is not defined.

Parameters

stream

      Stream on which to output dump information.

Remarks

Currently wxWidgets does not define Dump for derived classes, but programmers may wish to use it for their own applications. Be sure to call the Dump member of the class's base class to allow all information to be dumped.

The implementation of this function in wxObject just writes the class name of the object.

wxObject::GetClassInfo

wxClassInfo * GetClassInfo()

This virtual function is redefined for every class that requires run-time type information, when using DECLARE_CLASS macros.

wxObject::GetRefData

wxObjectRefData* GetRefData() **const**

Returns the **m_refData** pointer.

wxObject::Ref (p. 1014), *wxObject::UnRef* (p. 1015), *wxObject::m_refData* (p. 1013), *wxObject::SetRefData* (p. 1015), *wxObjectRefData* (p. 1015)

wxObject::IsKindOf

bool IsKindOf(**wxClassInfo ****info*)

Determines whether this class is a subclass of (or the same class as) the given class.

Parameters

info

> A pointer to a class information object, which may be obtained by using the CLASSINFO macro.

Return value

true if the class represented by *info* is the same class as this one or is derived from it.

Example

```
bool tmp = obj->IsKindOf(CLASSINFO(wxFrame));
```

wxObject::Ref

void Ref(**const wxObject&** *clone*)

Makes this object refer to the data in *clone*.

Parameters

clone

> The object to 'clone'.

Remarks

First this function calls *wxObject::UnRef* (p. 1015) on itself to decrement (and perhaps free) the data it is currently referring to.

It then sets its own m_refData to point to that of *clone*, and increments the reference count inside the data.

See also

wxObject::UnRef (p. 1015), *wxObject::m_refData* (p. 1013), *wxObject::SetRefData* (p. 1015), *wxObject::GetRefData* (p. 1013), *wxObjectRefData* (p. 1015)

wxObject::SetRefData

void SetRefData(**wxObjectRefData*** *data*)

Sets the **m_refData** pointer.

See also

wxObject::Ref (p. 1014), *wxObject::UnRef* (p. 1015), *wxObject::m_refData* (p. 1013), *wxObject::GetRefData* (p. 1013), *wxObjectRefData* (p. 1015)

wxObject::UnRef

void UnRef()

Decrements the reference count in the associated data, and if it is zero, deletes the data. The **m_refData** member is set to NULL.

See also

wxObject::Ref (p. 1014), *wxObject::m_refData* (p. 1013), *wxObject::SetRefData* (p. 1015), *wxObject::GetRefData* (p. 1013), *wxObjectRefData* (p. 1015)

wxObject::operator new

void * new(**size_t** *size*, **const wxString&** *filename = NULL*, **int** *lineNum = 0*)

The *new* operator is defined for debugging versions of the library only, when the identifier __WXDEBUG__ is defined. It takes over memory allocation, allowing wxDebugContext operations.

wxObject::operator delete

void delete(**void** *buf*)

The *delete* operator is defined for debugging versions of the library only, when the identifier __WXDEBUG__ is defined. It takes over memory deallocation, allowing wxDebugContext operations.

wxObjectRefData

This class is used to store reference-counted data. Derive classes from this to store your own data. When retrieving information from a **wxObject**'s reference data, you will need to cast to your own derived class.

Friends

wxObject (p. 1012)

wxObject (p. 1012)

wxObjectRefData::m_count

int m_count

Reference count. When this goes to zero during a *wxObject::UnRef* (p. 1015), an object can delete the **wxObjectRefData** object.

wxObjectRefData::wxObjectRefData

wxObjectRefData()

Default constructor. Initialises the **m_count** member to 1.

wxObjectRefData::~wxObjectRefData

wxObjectRefData()

Destructor.

wxOutputStream

wxOutputStream is an abstract base class which may not be used directly.

Derived from

wxStreamBase (p. 1275)

Include files

<wx/stream.h>

wxOutputStream::wxOutputStream

wxOutputStream()

Creates a dummy wxOutputStream object.

wxOutputStream::~wxOutputStream

~wxOutputStream()

Destructor.

wxOutputStream::Close

bool Close()

Closes the stream, returning `false` if an error occurs. The stream is closed implicitly in the destructor if Close() is not called explicitly.

If this stream wraps another stream or some other resource such as a file, then the underlying resource is closed too if it is owned by this stream, or left open otherwise.

wxOutputStream::LastWrite

size_t LastWrite() **const**

Returns the number of bytes written during the last *Write()* (p. 1018). It may return 0 even if there is no error on the stream if it is only temporarily impossible to write to it.

wxOutputStream::PutC

void PutC(**char** *c*)

Puts the specified character in the output queue and increments the stream position.

wxOutputStream::SeekO

off_t SeekO(**off_t** *pos*, **wxSeekMode** *mode* = *wxFromStart*)

Changes the stream current position.

Parameters

pos

> Offset to seek to.

mode

> One of **wxFromStart**, **wxFromEnd**, **wxFromCurrent**.

Return value

The new stream position or wxInvalidOffset on error.

wxOutputStream::TellO

off_t TellO() **const**

Returns the current stream position.

wxOutputStream::Write

wxOutputStream& Write(**const void ****buffer*, **size_t** *size*)

Writes up to the specified amount of bytes using the data of *buffer*. Note that not all data can always be written so you must check the number of bytes really written to the stream using *LastWrite()* (p. 1017) when this function returns. In some cases (for example a write end of a pipe which is currently full) it is even possible that there is no errors and zero bytes have been written.

This function returns a reference on the current object, so the user can test any states of the stream right away.

wxOutputStream& Write(**wxInputStream&** *stream_in*)

Reads data from the specified input stream and stores them in the current stream. The data is read until an error is raised by one of the two streams.

wxPageSetupDialog

This class represents the page setup common dialog. The page setup dialog is standard from Windows 95 on, replacing the print setup dialog (which is retained in Windows and wxWidgets for backward compatibility). On Windows 95 and NT 4.0 and above, the page setup dialog is native to the windowing system, otherwise it is emulated.

The page setup dialog contains controls for paper size (A4, A5 etc.), orientation (landscape or portrait), and controls for setting left, top, right and bottom margin sizes in millimetres.

When the dialog has been closed, you need to query the *wxPageSetupDialogData* (p. 1019) object associated with the dialog.

Note that the OK and Cancel buttons do not destroy the dialog; this must be done by the application.

Derived from

wxDialog (p. 406)
wxWindow (p. 1490)
wxEvtHandler (p. 486)
wxObject (p. 1012)

Include files

<wx/printdlg.h>

See also

wxPrintDialog (p. 1057), *wxPageSetupDialogData* (p. 1019)

wxPageSetupDialog::wxPageSetupDialog

wxPageSetupDialog(**wxWindow*** *parent*, **wxPageSetupDialogData*** *data* = *NULL*)

Constructor. Pass a parent window, and optionally a pointer to a block of page setup data, which will be copied to the print dialog's internal data.

wxPageSetupDialog::~wxPageSetupDialog

~wxPageSetupDialog()

Destructor.

wxPageSetupDialog::GetPageSetupData

wxPageSetupDialogData& GetPageSetupData()

Returns the *page setup data* (p. 1019) associated with the dialog.

wxPageSetupDialog::ShowModal

int ShowModal()

Shows the dialog, returning wxID_OK if the user pressed OK, and wxID_CANCEL otherwise.

wxPageSetupDialogData

This class holds a variety of information related to *wxPageSetupDialog* (p. 1018).

It contains a *wxPrintData* (p. 1051) member which is used to hold basic printer configuration data (as opposed to the user-interface configuration settings stored by wxPageSetupDialogData).

Derived from

wxObject (p. 1012)

Include files

<wx/cmndata.h>

See also

wxPageSetupDialog (p. 1018)

wxPageSetupDialogData::wxPageSetupDialogData

wxPageSetupDialogData()

Default constructor.

wxPageSetupDialogData(**wxPageSetupDialogData&** *data*)

Copy constructor.

wxPageSetupDialogData(**wxPrintData&** *printData*)

Construct an object from a print data object.

wxPageSetupDialogData::~wxPageSetupDialogData

~wxPageSetupDialogData()

Destructor.

wxPageSetupDialogData::EnableHelp

void EnableHelp(**bool** *flag*)

Enables or disables the 'Help' button (Windows only).

wxPageSetupDialogData::EnableMargins

void EnableMargins(**bool** *flag*)

Enables or disables the margin controls (Windows only).

wxPageSetupDialogData::EnableOrientation

void EnableOrientation(**bool** *flag*)

Enables or disables the orientation control (Windows only).

wxPageSetupDialogData::EnablePaper

void EnablePaper(**bool** *flag*)

Enables or disables the paper size control (Windows only).

wxPageSetupDialogData::EnablePrinter

void EnablePrinter(**bool** *flag*)

Enables or disables the **Printer** button, which invokes a printer setup dialog.

wxPageSetupDialogData::GetDefaultMinMargins

bool GetDefaultMinMargins() **const**

Returns true if the page setup dialog will take its minimum margin values from the currently selected printer properties. Windows only.

wxPageSetupDialogData::GetEnableMargins

bool GetEnableMargins() **const**

Returns true if the margin controls are enabled (Windows only).

wxPageSetupDialogData::GetEnableOrientation

bool GetEnableOrientation() **const**

Returns true if the orientation control is enabled (Windows only).

wxPageSetupDialogData::GetEnablePaper

bool GetEnablePaper() **const**

Returns true if the paper size control is enabled (Windows only).

wxPageSetupDialogData::GetEnablePrinter

bool GetEnablePrinter() **const**

Returns true if the printer setup button is enabled.

wxPageSetupDialogData::GetEnableHelp

bool GetEnableHelp() **const**

Returns true if the printer setup button is enabled.

wxPageSetupDialogData::GetDefaultInfo

bool GetDefaultInfo() **const**

Returns true if the dialog will simply return default printer information (such as orientation) instead of showing a dialog. Windows only.

wxPageSetupDialogData::GetMarginTopLeft

wxPoint GetMarginTopLeft() **const**

Returns the left (x) and top (y) margins in millimetres.

wxPageSetupDialogData::GetMarginBottomRight

wxPoint GetMarginBottomRight() **const**

Returns the right (x) and bottom (y) margins in millimetres.

wxPageSetupDialogData::GetMinMarginTopLeft

wxPoint GetMinMarginTopLeft() **const**

Returns the left (x) and top (y) minimum margins the user can enter (Windows only). Units are in millimetres

wxPageSetupDialogData::GetMinMarginBottomRight

wxPoint GetMinMarginBottomRight() **const**

Returns the right (x) and bottom (y) minimum margins the user can enter (Windows only). Units are in millimetres

wxPageSetupDialogData::GetPaperId

wxPaperSize GetPaperId() **const**

Returns the paper id (stored in the internal wxPrintData object).

For further information, see *wxPrintData::SetPaperId* (p. 1055).

wxPageSetupDialogData::GetPaperSize

wxSize GetPaperSize() **const**

Returns the paper size in millimetres.

wxPageSetupDialogData::GetPrintData

wxPrintData& GetPrintData()

Returns a reference to the *print data* (p. 1051) associated with this object.

wxPageSetupDialogData::Ok

bool Ok() **const**

Returns true if the print data associated with the dialog data is valid. This can return false on Windows if the current printer is not set, for example. On all other platforms, it returns true.

wxPageSetupDialogData::SetDefaultInfo

void SetDefaultInfo(**bool** *flag*)

Pass true if the dialog will simply return default printer information (such as orientation) instead of showing a dialog. Windows only.

wxPageSetupDialogData::SetDefaultMinMargins

void SetDefaultMinMargins(**bool** *flag*)

Pass true if the page setup dialog will take its minimum margin values from the currently selected printer properties. Windows only. Units are in millimetres

wxPageSetupDialogData::SetMarginTopLeft

void SetMarginTopLeft(**const wxPoint&** *pt*)

Sets the left (x) and top (y) margins in millimetres.

wxPageSetupDialogData::SetMarginBottomRight

void SetMarginBottomRight(**const wxPoint&** *pt*)

Sets the right (x) and bottom (y) margins in millimetres.

wxPageSetupDialogData::SetMinMarginTopLeft

void SetMinMarginTopLeft(**const wxPoint&** *pt*)

Sets the left (x) and top (y) minimum margins the user can enter (Windows only). Units are in millimetres.

wxPageSetupDialogData::SetMinMarginBottomRight

void SetMinMarginBottomRight(**const wxPoint&** *pt*)

Sets the right (x) and bottom (y) minimum margins the user can enter (Windows only). Units are in millimetres.

wxPageSetupDialogData::SetPaperId

void SetPaperId(**wxPaperSize&** *id*)

Sets the paper size id. For further information, see *wxPrintData::SetPaperId* (p. 1055).

Calling this function overrides the explicit paper dimensions passed in *wxPageSetupDialogData::SetPaperSize* (p. 1024).

wxPageSetupDialogData::SetPaperSize

void SetPaperSize(**const wxSize&** *size*)

Sets the paper size in millimetres. If a corresponding paper id is found, it will be set in the internal wxPrintData object, otherwise the paper size overrides the paper id.

wxPageSetupDialogData::SetPrintData

void SetPrintData(**const wxPrintData&** *printData*)

Sets the *print data* (p. 1051) associated with this object.

wxPageSetupDialogData::operator =

void operator =(**const wxPrintData&** *data*)

Assigns print data to this object.

void operator =(**const wxPageSetupDialogData&** *data*)

Assigns page setup data to this object.

wxPaintDC

A wxPaintDC must be constructed if an application wishes to paint on the client area of a window from within an **OnPaint** event. This should normally be constructed as a temporary stack object; don't store a wxPaintDC object. If you have an OnPaint handler, you *must* create a wxPaintDC object within it even if you don't actually use it.

Using wxPaintDC within OnPaint is important because it automatically sets the clipping

area to the damaged area of the window. Attempts to draw outside this area do not appear.

To draw on a window from outside **OnPaint**, construct a *wxClientDC* (p. 147) object.

To draw on the whole window including decorations, construct a *wxWindowDC* (p. 1547) object (Windows only).

Derived from

wxWindowDC (p. 1547)
wxDC (p. 365)

Include files

<wx/dcclient.h>

See also

wxDC (p. 365), *wxMemoryDC* (p. 935), *wxPaintDC* (p. 1024), *wxWindowDC* (p. 1547), *wxScreenDC* (p. 1142)

---

wxPaintDC::wxPaintDC

wxPaintDC(**wxWindow*** *window*)

Constructor. Pass a pointer to the window on which you wish to paint.

wxPaintEvent

A paint event is sent when a window's contents needs to be repainted.

Derived from

wxEvent (p. 483)
wxObject (p. 1012)

Include files

<wx/event.h>

Event table macros

To process a paint event, use this event handler macro to direct input to a member function that takes a wxPaintEvent argument.

EVT_PAINT(func)   Process a wxEVT_PAINT event.

See also
Event handling overview (p. 1751)

Remarks

Note that In a paint event handler, the application must *always* create a *wxPaintDC* (p. 1024) object, even if you do not use it. Otherwise, under MS Windows, refreshing for this and other windows will go wrong.

For example:

```
void MyWindow::OnPaint(wxPaintEvent& event)
{
    wxPaintDC dc(this);

    DrawMyDocument(dc);
}
```

You can optimize painting by retrieving the rectangles that have been damaged and only repainting these. The rectangles are in terms of the client area, and are unscrolled, so you will need to do some calculations using the current view position to obtain logical, scrolled units.

Here is an example of using the *wxRegionIterator* (p. 1115) class:

```
// Called when window needs to be repainted.
void MyWindow::OnPaint(wxPaintEvent& event)
{
  wxPaintDC dc(this);

  // Find Out where the window is scrolled to
  int vbX,vbY;                      // Top left corner of client
  GetViewStart(&vbX,&vbY);

  int vX,vY,vW,vH;                  // Dimensions of client area in pixels
  wxRegionIterator upd(GetUpdateRegion()); // get the update rect list

  while (upd)
  {
    vX = upd.GetX();
    vY = upd.GetY();
    vW = upd.GetW();
    vH = upd.GetH();

    // Alternatively we can do this:
    // wxRect rect(upd.GetRect());

    // Repaint this rectangle
    ...some code...

    upd ++ ;
  }
}
```

wxPaintEvent::wxPaintEvent

wxPaintEvent(**int** *id = 0*)

Constructor.

wxPalette

A palette is a table that maps pixel values to RGB colours. It allows the colours of a low-depth bitmap, for example, to be mapped to the available colours in a display.

Derived from

wxGDIObject (p. 607)
wxObject (p. 1012)

Include files

<wx/palette.h>

Predefined objects

Objects:

wxNullPalette

See also

wxDC::SetPalette (p. 384), *wxBitmap* (p. 79)

wxPalette::wxPalette

wxPalette()

Default constructor.

wxPalette(**const wxPalette&** *palette*)

Copy constructor. This uses reference counting so is a cheap operation.

wxPalette(**int** *n*, **const unsigned char*** *red*,
const unsigned char* *green*, **const unsigned char*** *blue*)

Creates a palette from arrays of size *n*, one for each red, blue or green component.

Parameters

palette

> A pointer or reference to the palette to copy.

n

> The number of indices in the palette.

red

> An array of red values.

green

> An array of green values.

blue

> An array of blue values.

See also

wxPalette::Create (p. 1028)

wxPerl note: In wxPerl the third constructor form takes as parameters 3 array references (they must be of the same length).

wxPalette::~wxPalette

~wxPalette()

Destructor.

wxPalette::Create

bool Create(**int** *n*, **const unsigned char*** *red*, **const unsigned char*** *green*, **const unsigned char*** *blue*)

Creates a palette from arrays of size *n*, one for each red, blue or green component.

Parameters

n

> The number of indices in the palette.

red

> An array of red values.

green

> An array of green values.

blue

> An array of blue values.

Return value

true if the creation was successful, false otherwise.

See also

wxPalette::wxPalette (p. 1027)

wxPalette::GetColoursCount

int GetColoursCount() **const**

Returns number of entries in palette.

wxPalette::GetPixel

int GetPixel(**const unsigned char** *red*, **const unsigned char** *green*, **const unsigned char** *blue*) **const**

Returns a pixel value (index into the palette) for the given RGB values.

Parameters

red

> Red value.

green

> Green value.

blue

> Blue value.

Return value

The nearest palette index.

See also

wxPalette::GetRGB (p. 1029)

wxPalette::GetRGB

bool GetRGB(**int** *pixel*, **const unsigned char*** *red*, **const unsigned char*** *green*, **const unsigned char*** *blue*) **const**

Returns RGB values for a given palette index.

Parameters

pixel

> The palette index.

red

> Receives the red value.

green

Receives the green value.

blue

Receives the blue value.

Return value

true if the operation was successful.

See also

wxPalette::GetPixel (p. 1029)

wxPerl note: In wxPerl this method takes only the `pixel` parameter and returns a 3-element list (or the empty list upon failure).

wxPalette::Ok

bool Ok() **const**

Returns true if palette data is present.

wxPalette::operator =

wxPalette& operator =(**const wxPalette&** *palette*)

Assignment operator, using reference counting. Returns a reference to 'this'.

wxPalette::operator ==

bool operator ==(**const wxPalette&** *palette*)

Equality operator. Two palettes are equal if they contain pointers to the same underlying palette data. It does not compare each attribute, so two independently-created palettes using the same parameters will fail the test.

wxPalette::operator !=

bool operator !=(**const wxPalette&** *palette*)

Inequality operator. Two palettes are not equal if they contain pointers to different underlying palette data. It does not compare each attribute.

wxPanel

A panel is a window on which controls are placed. It is usually placed within a frame. It

contains minimal extra functionality over and above its parent class wxWindow; its main purpose is to be similar in appearance and functionality to a dialog, but with the flexibility of having any window as a parent.

Note: if not all characters are being intercepted by your OnKeyDown or OnChar handler, it may be because you are using the wxTAB_TRAVERSAL style, which grabs some keypresses for use by child controls.

Derived from

wxWindow (p. 1490)
wxEvtHandler (p. 486)
wxObject (p. 1012)

Include files

<wx/panel.h>

Window styles

There are no specific styles for this window.

See also *window styles overview* (p. 1762).

Remarks

By default, a panel has the same colouring as a dialog.

See also

wxDialog (p. 406)

wxPanel::wxPanel

wxPanel()

Default constructor.

wxPanel(**wxWindow*** *parent*, **wxWindowID** *id = -1*, **const wxPoint&** *pos = wxDefaultPosition*, **const wxSize&** *size = wxDefaultSize*, **long** *style = wxTAB_TRAVERSAL*, **const wxString&** *name = "panel"*)

Constructor.

Parameters

parent

> The parent window.

id

An identifier for the panel. A value of -1 is taken to mean a default.

pos

The panel position. A value of (-1, -1) indicates a default position, chosen by either the windowing system or wxWidgets, depending on platform.

size

The panel size. A value of (-1, -1) indicates a default size, chosen by either the windowing system or wxWidgets, depending on platform.

style

The window style. See *wxPanel* (p. 1030).

name

Used to associate a name with the window, allowing the application user to set Motif resource values for individual dialog boxes.

See also

wxPanel::Create (p. 1032)

wxPanel::~wxPanel

 ~wxPanel()

Destructor. Deletes any child windows before deleting the physical window.

wxPanel::Create

bool Create(**wxWindow*** *parent*, **wxWindowID** *id*, **const wxPoint&** *pos* = *wxDefaultPosition*, **const wxSize&** *size* = *wxDefaultSize*, **long** *style* = *wxTAB_TRAVERSAL*, **const wxString&** *name* = *"panel"*)

Used for two-step panel construction. See *wxPanel::wxPanel* (p. 1031) for details.

wxPanel::GetDefaultItem

wxWindow* GetDefaultItem() **const**

Returns a pointer to the button which is the default for this window, or NULL. The default button is the one activated by pressing the Enter key.

wxPanel::InitDialog

void InitDialog()

Sends a *wxInitDialogEvent* (p. 813), which in turn transfers data to the dialog via validators.

See also

wxInitDialogEvent (p. 813)

wxPanel::OnSysColourChanged

void OnSysColourChanged(**wxSysColourChangedEvent&** *event*)

The default handler for wxEVT_SYS_COLOUR_CHANGED.

Parameters

event

> The colour change event.

Remarks

Changes the panel's colour to conform to the current settings (Windows only). Add an event table entry for your panel class if you wish the behaviour to be different (such as keeping a user-defined background colour). If you do override this function, call wxEvent::Skip to propagate the notification to child windows and controls.

See also

wxSysColourChangedEvent (p. 1314)

wxPanel::SetDefaultItem

void SetDefaultItem(**wxButton** **btn*)

Changes the default button for the panel.

See also

GetDefaultItem (p. 1032)

wxPanel::SetFocus

virtual void SetFocus()

Overrides *wxWindow::SetFocus* (p. 1533). This method uses the (undocumented) mix-in class wxControlContainer which manages the focus and TAB logic for controls which usually have child controls. In practice, if you call this method and the control has at least one child window, the focus will be given to the child window.

See also

wxFocusEvent (p. 557)*wxWindow::SetFocus* (p. 1533)

---
wxPanel::SetFocusIgnoringChildren

virtual void SetFocusIgnoringChildren()

In contrast to *wxPanel::SetFocus* (p. 1033) (see above) this will set the focus to the panel even of there are child windows in the panel. This is only rarely needed.

See also

wxFocusEvent (p. 557)*wxPanel::SetFocus* (p. 1033)

---
wxPasswordEntryDialog

This class represents a dialog that requests a one-line password string from the user. It is implemented as a generic wxWidgets dialog.

Derived from

wxTextEntryDialog (p. 1361)
wxDialog (p. 406)
wxWindow (p. 1490)
wxEvtHandler (p. 486)
wxObject (p. 1012)

Include files

<wx/textdlg.h>

See also

wxPassowrdEntryDialog overview (p. 1801)

---
wxPasswordEntryDialog::wxPasswordEntryDialog

 wxPasswordEntryDialog(**wxWindow*** *parent*, **const wxString&** *message*, **const wxString&** *caption* = *"Enter password"*, **const wxString&** *defaultValue* = *""*, **long** *style* = *wxOK | wxCANCEL | wxCENTRE*, **const wxPoint&** *pos* = *wxDefaultPosition*)

Constructor. Use *wxTextEntryDialog::ShowModal* (p. 1363) to show the dialog.

Parameters

parent

> Parent window.

message

> Message to show on the dialog.

defaultValue

> The default value, which may be the empty string.

style

> A dialog style, specifying the buttons (wxOK, wxCANCEL) and an optional wxCENTRE style. You do not need to specify the **wxTE_PASSWORD** style, it is always applied.

pos

> Dialog position.

wxPathList

The path list is a convenient way of storing a number of directories, and when presented with a filename without a directory, searching for an existing file in those directories. Storing the filename only in an application's files and using a locally-defined list of directories makes the application and its files more portable.

Use the *wxFileNameFromPath* global function to extract the filename from the path.

Derived from

wxList (p. 840)
wxObject (p. 1012)

Include files

<wx/filefn.h>

See also

wxList (p. 840)

wxPathList::wxPathList

 wxPathList()

Constructor.

wxPathList::AddEnvList

void AddEnvList(**const wxString&** *env_variable*)

Finds the value of the given environment variable, and adds all paths to the path list. Useful for finding files in the PATH variable, for example.

wxPathList::Add

void Add(**const wxString&** *path*)

Adds the given directory to the path list, but does not check if the path was already on the list (use *wxPathList::Member()* (p. 1036) for this).

wxPathList::EnsureFileAccessible

void EnsureFileAccessible(**const wxString&** *filename*)

Given a full filename (with path), ensures that files in the same path can be accessed using the pathlist. It does this by stripping the filename and adding the path to the list if not already there.

wxPathList::FindAbsoluteValidPath

wxString FindAbsoluteValidPath(**const wxString&** *file*)

Searches for a full path for an existing file by appending *file* to successive members of the path list. If the file wasn't found, an empty string is returned.

wxPathList::FindValidPath

wxString FindValidPath(**const wxString&** *file*)

Searches for a full path for an existing file by appending *file* to successive members of the path list. If the file wasn't found, an empty string is returned. This path may be relative to the current working directory.

wxPathList::Member

bool Member(**const wxString&** *file*)

true if the path is in the path list (ignoring case).

wxPen

A pen is a drawing tool for drawing outlines. It is used for drawing lines and painting the outline of rectangles, ellipses, etc. It has a colour, a width and a style.

Derived from

wxGDIObject (p. 607)
wxObject (p. 1012)

Include files

<wx/pen.h>

Predefined objects

Objects:

wxNullPen

Pointers:

wxRED_PEN
wxCYAN_PEN
wxGREEN_PEN
wxBLACK_PEN
wxWHITE_PEN
wxTRANSPARENT_PEN
wxBLACK_DASHED_PEN
wxGREY_PEN
wxMEDIUM_GREY_PEN
wxLIGHT_GREY_PEN

Remarks

On a monochrome display, wxWidgets shows all non-white pens as black.

Do not initialize objects on the stack before the program commences, since other required structures may not have been set up yet. Instead, define global pointers to objects and create them in *OnInit* or when required.

An application may wish to dynamically create pens with different characteristics, and there is the consequent danger that a large number of duplicate pens will be created. Therefore an application may wish to get a pointer to a pen by using the global list of pens **wxThePenList**, and calling the member function **FindOrCreatePen**. See the entry for *wxPenList* (p. 1043).

wxPen uses a reference counting system, so assignments between brushes are very cheap. You can therefore use actual wxPen objects instead of pointers without efficiency problems. Once one wxPen object changes its data it will create its own pen data internally so that other pens, which previously shared the data using the reference counting, are not affected.

See also

wxPenList (p. 1043), *wxDC* (p. 365), *wxDC::SetPen* (p. 384)

wxPen::wxPen

wxPen()

Default constructor. The pen will be uninitialised, and *wxPen::Ok* (p. 1041) will return false.

wxPen(**const wxColour&** *colour*, **int** *width = 1*, **int** *style = `wxSOLID`*)

Constructs a pen from a colour object, pen width and style.

wxPen(**const wxString&** *colourName*, **int** *width*, **int** *style*)

Constructs a pen from a colour name, pen width and style.

wxPen(**const wxBitmap&** *stipple*, **int** *width*)

Constructs a stippled pen from a stipple bitmap and a width.

wxPen(**const wxPen&** *pen*)

Copy constructor. This uses reference counting so is a cheap operation.

Parameters

colour

> A colour object.

colourName

> A colour name.

width

> Pen width. Under Windows, the pen width cannot be greater than 1 if the style is wxDOT, wxLONG_DASH, wxSHORT_DASH, wxDOT_DASH, or wxUSER_DASH.

stipple

> A stipple bitmap.

pen

> A pointer or reference to a pen to copy.

style

> The style may be one of the following:

wxSOLID	Solid style.
wxTRANSPARENT	No pen is used.
wxDOT	Dotted style.
wxLONG_DASH	Long dashed style.

wxSHORT_DASH	Short dashed style.
wxDOT_DASH	Dot and dash style.
wxSTIPPLE	Use the stipple bitmap.
wxUSER_DASH	Use the user dashes: see *wxPen::SetDashes* (p. 1042).
wxBDIAGONAL_HATCH	Backward diagonal hatch.
wxCROSSDIAG_HATCH	Cross-diagonal hatch.
wxFDIAGONAL_HATCH	Forward diagonal hatch.
wxCROSS_HATCH	Cross hatch.
wxHORIZONTAL_HATCH	Horizontal hatch.
wxVERTICAL_HATCH	Vertical hatch.

Remarks

Different versions of Windows and different versions of other platforms support *very* different subsets of the styles above - there is no similarity even between Windows95 and Windows98 - so handle with care.

If the named colour form is used, an appropriate **wxColour** structure is found in the colour database.

See also

wxPen::SetStyle (p. 1042), *wxPen::SetColour* (p. 1041), *wxPen::SetWidth* (p. 1043), *wxPen::SetStipple* (p. 1042)

wxPerl note: Constructors supported by wxPerl are:

- ::Pen->new(colour, width, style)

- ::Pen->new(colourName, width, style)

- ::Pen->new(stipple, width)

wxPen::~wxPen

~wxPen()

Destructor.

Remarks

The destructor may not delete the underlying pen object of the native windowing system, since wxBrush uses a reference counting system for efficiency.

Although all remaining pens are deleted when the application exits, the application should try to clean up all pens itself. This is because wxWidgets cannot know if a pointer to the pen object is stored in an application data structure, and there is a risk of double deletion.

wxPen::GetCap

int GetCap() **const**

Returns the pen cap style, which may be one of **wxCAP_ROUND**, **wxCAP_PROJECTING** and **wxCAP_BUTT**. The default is **wxCAP_ROUND**.

See also

wxPen::SetCap (p. 1041)

wxPen::GetColour

wxColour& GetColour() **const**

Returns a reference to the pen colour.

See also

wxPen::SetColour (p. 1041)

wxPen::GetDashes

int GetDashes(**wxDash**** *dashes*) **const**

Gets an array of dashes (defined as char in X, DWORD under Windows).*dashes* is a pointer to the internal array. Do not deallocate or store this pointer. The function returns the number of dashes associated with this pen.

See also

wxPen::SetDashes (p. 1042)

wxPen::GetJoin

int GetJoin() **const**

Returns the pen join style, which may be one of **wxJOIN_BEVEL**, **wxJOIN_ROUND** and **wxJOIN_MITER**. The default is **wxJOIN_ROUND**.

See also

wxPen::SetJoin (p. 1042)

wxPen::GetStipple

wxBitmap* GetStipple() **const**

Gets a pointer to the stipple bitmap.

See also

wxPen::SetStipple (p. 1042)

wxPen::GetStyle

int GetStyle() **const**

Returns the pen style.

See also

wxPen::wxPen (p. 1037), *wxPen::SetStyle* (p. 1042)

wxPen::GetWidth

int GetWidth() **const**

Returns the pen width.

See also

wxPen::SetWidth (p. 1043)

wxPen::Ok

bool Ok() **const**

Returns true if the pen is initialised.

wxPen::SetCap

void SetCap(**int** *capStyle*)

Sets the pen cap style, which may be one of **wxCAP_ROUND**, **wxCAP_PROJECTING** and **wxCAP_BUTT**. The default is **wxCAP_ROUND**.

See also

wxPen::GetCap (p. 1040)

wxPen::SetColour

void SetColour(**wxColour&** *colour*)

void SetColour(**const wxString&** *colourName*)

void SetColour(**int** *red*, **int** *green*, **int** *blue*)

The pen's colour is changed to the given colour.

See also

wxPen::GetColour (p. 1040)

wxPen::SetDashes

void SetDashes(**int** *n*, **wxDash*** *dashes*)

Associates an array of pointers to dashes (defined as char in X, DWORD under Windows) with the pen. The array is not deallocated by wxPen, but neither must it be deallocated by the calling application until the pen is deleted or this function is called with a NULL array.

See also

wxPen::GetDashes (p. 1040)

wxPen::SetJoin

void SetJoin(**int** *join_style*)

Sets the pen join style, which may be one of **wxJOIN_BEVEL**, **wxJOIN_ROUND** and **wxJOIN_MITER**. The default is **wxJOIN_ROUND**.

See also

wxPen::GetJoin (p. 1040)

wxPen::SetStipple

void SetStipple(**wxBitmap*** *stipple*)

Sets the bitmap for stippling.

See also

wxPen::GetStipple (p. 1041)

wxPen::SetStyle

void SetStyle(**int** *style*)

Set the pen style.

See also

wxPen::wxPen (p. 1037)

wxPen::SetWidth

void SetWidth(**int** *width*)

Sets the pen width.

See also

wxPen::GetWidth (p. 1041)

wxPen::operator =

wxPen& operator =(**const wxPen&** *pen*)

Assignment operator, using reference counting. Returns a reference to 'this'.

wxPen::operator ==

bool operator ==(**const wxPen&** *pen*)

Equality operator. Two pens are equal if they contain pointers to the same underlying pen data. It does not compare each attribute, so two independently-created pens using the same parameters will fail the test.

wxPen::operator !=

bool operator !=(**const wxPen&** *pen*)

Inequality operator. Two pens are not equal if they contain pointers to different underlying pen data. It does not compare each attribute.

wxPenList

There is only one instance of this class: **wxThePenList**. Use this object to search for a previously created pen of the desired type and create it if not already found. In some windowing systems, the pen may be a scarce resource, so it can pay to reuse old resources if possible. When an application finishes, all pens will be deleted and their resources freed, eliminating the possibility of 'memory leaks'. However, it is best not to rely on this automatic cleanup because it can lead to double deletion in some circumstances.

There are two mechanisms in recent versions of wxWidgets which make the pen list less useful than it once was. Under Windows, scarce resources are cleaned up internally if they are not being used. Also, a referencing counting mechanism applied to all GDI objects means that some sharing of underlying resources is possible. You don't have to keep track of pointers, working out when it is safe delete a pen, because the referencing counting does it for you. For example, you can set a pen in a device context, and then immediately delete the pen you passed, because the pen is 'copied'.

So you may find it easier to ignore the pen list, and instead create and copy pens as you see fit. If your Windows resource meter suggests your application is using too many resources, you can resort to using GDI lists to share objects explicitly.

The only compelling use for the pen list is for wxWidgets to keep track of pens in order to clean them up on exit. It is also kept for backward compatibility with earlier versions of wxWidgets.

See also

wxPen (p. 1036)

wxPenList::wxPenList

void wxPenList()

Constructor. The application should not construct its own pen list: use the object pointer **wxThePenList**.

wxPenList::AddPen

void AddPen(**wxPen*** *pen*)

Used internally by wxWidgets to add a pen to the list.

wxPenList::FindOrCreatePen

wxPen* FindOrCreatePen(**const wxColour&** *colour*, **int** *width*, **int** *style*)

Finds a pen with the specified attributes and returns it, else creates a new pen, adds it to the pen list, and returns it.

wxPen* FindOrCreatePen(**const wxString&** *colourName*, **int** *width*, **int** *style*)

Finds a pen with the specified attributes and returns it, else creates a new pen, adds it to the pen list, and returns it.

Parameters

colour

Colour object.

colourName

Colour name, which should be in the *colour database* (p. 169).

width

Width of pen.

style

Pen style. See *wxPen::wxPen* (p. 1037) for a list of styles.

wxPenList::RemovePen

void RemovePen(**wxPen*** *pen*)

Used by wxWidgets to remove a pen from the list.

wxPoint

A **wxPoint** is a useful data structure for graphics operations. It simply contains integer *x* and *y* members.

See also *wxRealPoint* (p. 1098) for a floating point version.

Derived from

None

Include files

<wx/gdicmn.h>

See also

wxRealPoint (p. 1098)

wxPoint::wxPoint

wxPoint()

wxPoint(**int** *x*, **int** *y*)

Create a point.

wxPoint::x

int x

x member.

wxPoint::y

int y

y member.

wxPostScriptDC

This defines the wxWidgets Encapsulated PostScript device context, which can write PostScript files on any platform. See *wxDC* (p. 365) for descriptions of the member functions.

Derived from

wxDC (p. 365)
wxObject (p. 1012)

Include files

<wx/dcps.h>

wxPostScriptDC::wxPostScriptDC

 wxPostScriptDC(**const wxPrintData&** *printData*)

Constructs a PostScript printer device context from a *wxPrintData* (p. 1051) object.

 wxPostScriptDC(**const wxString&** *output*, **bool** *interactive = true*,
 wxWindow *parent)

Constructor. *output* is an optional file for printing to, and if *interactive* is true a dialog box will be displayed for adjusting various parameters. *parent* is the parent of the printer dialog box.

Use the *Ok* member to test whether the constructor was successful in creating a usable device context.

See *Printer settings* (p. 1625) for functions to set and get PostScript printing settings.

This constructor and the global printer settings are now deprecated; use the wxPrintData constructor instead.

wxPostScriptDC::SetResolution

static void SetResolution(**int** *ppi*)

Set resolution (in pixels per inch) that will be used in PostScript output. Default is 720ppi.

wxPostScriptDC::GetResolution

static int GetResolution()

Return resolution used in PostScript output. See *SetResolution* (p. 1047).

wxPreviewCanvas

A preview canvas is the default canvas used by the print preview system to display the preview.

Derived from

wxScrolledWindow (p. 1150)
wxWindow (p. 1490)
wxevthandler (p. 486)
wxObject (p. 1012)

Include files

<wx/print.h>

See also

wxPreviewFrame (p. 1049), *wxPreviewControlBar* (p. 1048), *wxPrintPreview* (p. 1069)

wxPreviewCanvas::wxPreviewCanvas

 wxPreviewCanvas(**wxPrintPreview*** *preview*, **wxWindow*** *parent*, **const wxPoint&** *pos = wxDefaultPosition*, **const wxSize&** *size = wxDefaultSize*, **long** *style = 0*, **const wxString&** *name = "canvas"*)

Constructor.

wxPreviewCanvas::~wxPreviewCanvas

 ~wxPreviewCanvas()

Destructor.

wxPreviewCanvas::OnPaint

void OnPaint(**wxPaintEvent&** *event*)

Calls *wxPrintPreview::PaintPage* (p. 1071) to refresh the canvas.

wxPreviewControlBar

This is the default implementation of the preview control bar, a panel with buttons and a zoom control. You can derive a new class from this and override some or all member functions to change the behaviour and appearance; or you can leave it as it is.

Derived from

wxPanel (p. 1030)
wxWindow (p. 1490)
wxEvtHandler (p. 486)
wxObject (p. 1012)

Include files

<wx/print.h>

See also

wxPreviewFrame (p. 1049), *wxPreviewCanvas* (p. 1047), *wxPrintPreview* (p. 1069)

wxPreviewControlBar::wxPreviewControlbar

 wxPreviewControlBar(**wxPrintPreview*** *preview*, **long** *buttons*, **wxWindow*** *parent*, **const wxPoint&** *pos = wxDefaultPosition*, **const wxSize&** *size = wxDefaultSize*, **long** *style = 0*, **const wxString&** *name = "panel"*)

Constructor.

The buttons parameter may be a combination of the following, using the bitwise 'or' operator.

wxPREVIEW_PRINT	Create a print button.
wxPREVIEW_NEXT	Create a next page button.
wxPREVIEW_PREVIOUS	Create a previous page button.
wxPREVIEW_ZOOM	Create a zoom control.
wxPREVIEW_DEFAULT	Equivalent to a combination of wxPREVIEW_PREVIOUS, wxPREVIEW_NEXT and wxPREVIEW_ZOOM.

wxPreviewControlBar::~wxPreviewControlBar

~wxPreviewControlBar()

Destructor.

wxPreviewControlBar::CreateButtons

void CreateButtons()

Creates buttons, according to value of the button style flags.

wxPreviewControlBar::GetPrintPreview

wxPrintPreview * GetPrintPreview()

Gets the print preview object associated with the control bar.

wxPreviewControlBar::GetZoomControl

int GetZoomControl()

Gets the current zoom setting in percent.

wxPreviewControlBar::SetZoomControl

void SetZoomControl(**int** *percent*)

Sets the zoom control.

wxPreviewFrame

This class provides the default method of managing the print preview interface. Member functions may be overridden to replace functionality, or the class may be used without derivation.

Derived from

wxFrame (p. 579)
wxWindow (p. 1490)
wxEvtHandler (p. 486)
wxObject (p. 1012)

Include files

<wx/print.h>

See also

wxPreviewCanvas (p. 1047), *wxPreviewControlBar* (p. 1048), *wxPrintPreview* (p. 1069)

wxPreviewFrame::wxPreviewFrame

 wxPreviewFrame(**wxPrintPreview*** *preview*, **wxWindow*** *parent*, **const wxString&** *title*, **const wxPoint&** *pos* = *wxDefaultPosition*, **const wxSize& size** *size* = *wxDefaultSize*, **long** *style* = *wxDEFAULT_FRAME_STYLE*, **const wxString&** *name* = *"frame"*)

Constructor. Pass a print preview object plus other normal frame arguments. The print preview object will be destroyed by the frame when it closes.

wxPreviewFrame::~wxPreviewFrame

 ~wxPreviewFrame()

Destructor.

wxPreviewFrame::CreateControlBar

void CreateControlBar()

Creates a wxPreviewControlBar. Override this function to allow a user-defined preview control bar object to be created.

wxPreviewFrame::CreateCanvas

void CreateCanvas()

Creates a wxPreviewCanvas. Override this function to allow a user-defined preview canvas object to be created.

wxPreviewFrame::Initialize

void Initialize()

Creates the preview canvas and control bar, and calls wxWindow::MakeModal(true) to disable other top-level windows in the application.

This function should be called by the application prior to showing the frame.

wxPreviewFrame::OnCloseWindow

void OnCloseWindow(**wxCloseEvent&** *event*)

Enables the other frames in the application, and deletes the print preview object, implicitly deleting any printout objects associated with the print preview object.

wxPrintData

This class holds a variety of information related to printers and printer device contexts. This class is used to create a wxPrinterDC and a wxPostScriptDC. It is also used as a data member of wxPrintDialogData and wxPageSetupDialogData, as part of the mechanism for transferring data between the print dialogs and the application.

Derived from

wxObject (p. 1012)

Include files

<wx/cmndata.h>

See also

wxPrintDialog (p. 1057), *wxPageSetupDialog* (p. 1018), *wxPrintDialogData* (p. 1058), *wxPageSetupDialogData* (p. 1019), *wxPrintDialog Overview* (p. 1800), *wxPrinterDC* (p. 1065), *wxPostScriptDC* (p. 1046)

Remarks

The following functions are specific to PostScript printing and have not yet been documented:

```
const wxString& GetPrinterCommand() const ;
const wxString& GetPrinterOptions() const ;
const wxString& GetPreviewCommand() const ;
const wxString& GetFilename() const ;
const wxString& GetFontMetricPath() const ;
double GetPrinterScaleX() const ;
double GetPrinterScaleY() const ;
long GetPrinterTranslateX() const ;
long GetPrinterTranslateY() const ;
// wxPRINT_MODE_PREVIEW, wxPRINT_MODE_FILE, wxPRINT_MODE_PRINTER
wxPrintMode GetPrintMode() const ;

void SetPrinterCommand(const wxString& command) ;
void SetPrinterOptions(const wxString& options) ;
void SetPreviewCommand(const wxString& command) ;
void SetFilename(const wxString& filename) ;
void SetFontMetricPath(const wxString& path) ;
void SetPrinterScaleX(double x) ;
void SetPrinterScaleY(double y) ;
void SetPrinterScaling(double x, double y) ;
void SetPrinterTranslateX(long x) ;
void SetPrinterTranslateY(long y) ;
void SetPrinterTranslation(long x, long y) ;
void SetPrintMode(wxPrintMode printMode) ;
```

wxPrintData::wxPrintData

wxPrintData()

Default constructor.

wxPrintData(**const wxPrintData&** *data*)

Copy constructor.

wxPrintData::~wxPrintData

~wxPrintData()

Destructor.

wxPrintData::GetCollate

bool GetCollate() **const**

Returns true if collation is on.

wxPrintData::GetBin

wxPrintBin GetBin() **const**

Returns the current bin (papersource). By default, the system is left to select the bin (wxPRINTBIN_DEFAULT is returned).

See *SetBin()* (p. 1054) for the full list of bin values.

wxPrintData::GetColour

bool GetColour() **const**

Returns true if colour printing is on.

wxPrintData::GetDuplex

wxDuplexMode GetDuplex() **const**

Returns the duplex mode. One of wxDUPLEX_SIMPLEX, wxDUPLEX_HORIZONTAL, wxDUPLEX_VERTICAL.

wxPrintData::GetNoCopies

int GetNoCopies() **const**

Returns the number of copies requested by the user.

wxPrintData::GetOrientation

int GetOrientation() **const**

Gets the orientation. This can be wxLANDSCAPE or wxPORTRAIT.

wxPrintData::GetPaperId

wxPaperSize GetPaperId() **const**

Returns the paper size id. For more information, see *wxPrintData::SetPaperId* (p. 1055).

wxPrintData::GetPrinterName

const wxString& GetPrinterName() **const**

Returns the printer name. If the printer name is the empty string, it indicates that the default printer should be used.

wxPrintData::GetQuality

wxPrintQuality GetQuality() **const**

Returns the current print quality. This can be a positive integer, denoting the number of dots per inch, or one of the following identifiers:

```
wxPRINT_QUALITY_HIGH
wxPRINT_QUALITY_MEDIUM
wxPRINT_QUALITY_LOW
wxPRINT_QUALITY_DRAFT
```

On input you should pass one of these identifiers, but on return you may get back a positive integer indicating the current resolution setting.

wxPrintData::Ok

bool Ok() **const**

Returns true if the print data is valid for using in print dialogs. This can return false on Windows if the current printer is not set, for example. On all other platforms, it returns true.

wxPrintData::SetBin

void SetBin(**wxPrintBin** *flag*)

Sets the current bin. Possible values are:

```
enum wxPrintBin
{
    wxPRINTBIN_DEFAULT,

    wxPRINTBIN_ONLYONE,
    wxPRINTBIN_LOWER,
    wxPRINTBIN_MIDDLE,
    wxPRINTBIN_MANUAL,
    wxPRINTBIN_ENVELOPE,
    wxPRINTBIN_ENVMANUAL,
    wxPRINTBIN_AUTO,
    wxPRINTBIN_TRACTOR,
    wxPRINTBIN_SMALLFMT,
    wxPRINTBIN_LARGEFMT,
    wxPRINTBIN_LARGECAPACITY,
    wxPRINTBIN_CASSETTE,
    wxPRINTBIN_FORMSOURCE,

    wxPRINTBIN_USER,
};
```

wxPrintData::SetCollate

void SetCollate(**bool** *flag*)

Sets collation to on or off.

wxPrintData::SetColour

void SetColour(**bool** *flag*)

Sets colour printing on or off.

wxPrintData::SetDuplex

void SetDuplex(**wxDuplexMode** *mode*)

Returns the duplex mode. One of wxDUPLEX_SIMPLEX, wxDUPLEX_HORIZONTAL, wxDUPLEX_VERTICAL.

wxPrintData::SetNoCopies

void SetNoCopies(**int** *n*)

Sets the default number of copies to be printed out.

wxPrintData::SetOrientation

void SetOrientation(**int** *orientation*)

Sets the orientation. This can be wxLANDSCAPE or wxPORTRAIT.

wxPrintData::SetPaperId

void SetPaperId(**wxPaperSize** *paperId*)

Sets the paper id. This indicates the type of paper to be used. For a mapping between paper id, paper size and string name, see wxPrintPaperDatabase in `paper.h` (not yet documented).

paperId can be one of:

```
wxPAPER_NONE,            // Use specific dimensions
wxPAPER_LETTER,          // Letter, 8 1/2 by 11 inches
wxPAPER_LEGAL,           // Legal, 8 1/2 by 14 inches
wxPAPER_A4,              // A4 Sheet, 210 by 297 millimeters
wxPAPER_CSHEET,          // C Sheet, 17 by 22 inches
wxPAPER_DSHEET,          // D Sheet, 22 by 34 inches
wxPAPER_ESHEET,          // E Sheet, 34 by 44 inches
wxPAPER_LETTERSMALL,     // Letter Small, 8 1/2 by 11 inches
wxPAPER_TABLOID,         // Tabloid, 11 by 17 inches
wxPAPER_LEDGER,          // Ledger, 17 by 11 inches
wxPAPER_STATEMENT,       // Statement, 5 1/2 by 8 1/2 inches
wxPAPER_EXECUTIVE,       // Executive, 7 1/4 by 10 1/2 inches
wxPAPER_A3,              // A3 sheet, 297 by 420 millimeters
wxPAPER_A4SMALL,         // A4 small sheet, 210 by 297 millimeters
wxPAPER_A5,              // A5 sheet, 148 by 210 millimeters
wxPAPER_B4,              // B4 sheet, 250 by 354 millimeters
wxPAPER_B5,              // B5 sheet, 182-by-257-millimeter paper
wxPAPER_FOLIO,           // Folio, 8-1/2-by-13-inch paper
wxPAPER_QUARTO,          // Quarto, 215-by-275-millimeter paper
wxPAPER_10X14,           // 10-by-14-inch sheet
wxPAPER_11X17,           // 11-by-17-inch sheet
wxPAPER_NOTE,            // Note, 8 1/2 by 11 inches
wxPAPER_ENV_9,           // #9 Envelope, 3 7/8 by 8 7/8 inches
wxPAPER_ENV_10,          // #10 Envelope, 4 1/8 by 9 1/2 inches
wxPAPER_ENV_11,          // #11 Envelope, 4 1/2 by 10 3/8 inches
wxPAPER_ENV_12,          // #12 Envelope, 4 3/4 by 11 inches
wxPAPER_ENV_14,          // #14 Envelope, 5 by 11 1/2 inches
wxPAPER_ENV_DL,          // DL Envelope, 110 by 220 millimeters
wxPAPER_ENV_C5,          // C5 Envelope, 162 by 229 millimeters
wxPAPER_ENV_C3,          // C3 Envelope, 324 by 458 millimeters
wxPAPER_ENV_C4,          // C4 Envelope, 229 by 324 millimeters
wxPAPER_ENV_C6,          // C6 Envelope, 114 by 162 millimeters
wxPAPER_ENV_C65,         // C65 Envelope, 114 by 229 millimeters
wxPAPER_ENV_B4,          // B4 Envelope, 250 by 353 millimeters
wxPAPER_ENV_B5,          // B5 Envelope, 176 by 250 millimeters
wxPAPER_ENV_B6,          // B6 Envelope, 176 by 125 millimeters
wxPAPER_ENV_ITALY,       // Italy Envelope, 110 by 230 millimeters
wxPAPER_ENV_MONARCH,     // Monarch Envelope, 3 7/8 by 7 1/2 inches
wxPAPER_ENV_PERSONAL,    // 6 3/4 Envelope, 3 5/8 by 6 1/2 inches
```

```
    wxPAPER_FANFOLD_US,          // US Std Fanfold, 14 7/8 by 11 inches
    wxPAPER_FANFOLD_STD_GERMAN,  // German Std Fanfold, 8 1/2 by 12 inches
    wxPAPER_FANFOLD_LGL_GERMAN,  // German Legal Fanfold, 8 1/2 by 13
inches

Windows 95 only:
    wxPAPER_ISO_B4,              // B4 (ISO) 250 x 353 mm
    wxPAPER_JAPANESE_POSTCARD,   // Japanese Postcard 100 x 148 mm
    wxPAPER_9X11,                // 9 x 11 in
    wxPAPER_10X11,               // 10 x 11 in
    wxPAPER_15X11,               // 15 x 11 in
    wxPAPER_ENV_INVITE,          // Envelope Invite 220 x 220 mm
    wxPAPER_LETTER_EXTRA,        // Letter Extra 9 \275 x 12 in
    wxPAPER_LEGAL_EXTRA,         // Legal Extra 9 \275 x 15 in
    wxPAPER_TABLOID_EXTRA,       // Tabloid Extra 11.69 x 18 in
    wxPAPER_A4_EXTRA,            // A4 Extra 9.27 x 12.69 in
    wxPAPER_LETTER_TRANSVERSE,   // Letter Transverse 8 \275 x 11 in
    wxPAPER_A4_TRANSVERSE,       // A4 Transverse 210 x 297 mm
    wxPAPER_LETTER_EXTRA_TRANSVERSE, // Letter Extra Transverse 9\275 x 12
in
    wxPAPER_A_PLUS,              // SuperA/SuperA/A4 227 x 356 mm
    wxPAPER_B_PLUS,              // SuperB/SuperB/A3 305 x 487 mm
    wxPAPER_LETTER_PLUS,         // Letter Plus 8.5 x 12.69 in
    wxPAPER_A4_PLUS,             // A4 Plus 210 x 330 mm
    wxPAPER_A5_TRANSVERSE,       // A5 Transverse 148 x 210 mm
    wxPAPER_B5_TRANSVERSE,       // B5 (JIS) Transverse 182 x 257 mm
    wxPAPER_A3_EXTRA,            // A3 Extra 322 x 445 mm
    wxPAPER_A5_EXTRA,            // A5 Extra 174 x 235 mm
    wxPAPER_B5_EXTRA,            // B5 (ISO) Extra 201 x 276 mm
    wxPAPER_A2,                  // A2 420 x 594 mm
    wxPAPER_A3_TRANSVERSE,       // A3 Transverse 297 x 420 mm
    wxPAPER_A3_EXTRA_TRANSVERSE  // A3 Extra Transverse 322 x 445 mm
```

wxPrintData::SetPrinterName

void SetPrinterName(**const wxString&** *printerName*)

Sets the printer name. This can be the empty string to indicate that the default printer should be used.

wxPrintData::SetQuality

void SetQuality(**wxPrintQuality** *quality*)

Sets the desired print quality. This can be a positive integer, denoting the number of dots per inch, or one of the following identifiers:

```
wxPRINT_QUALITY_HIGH
wxPRINT_QUALITY_MEDIUM
wxPRINT_QUALITY_LOW
wxPRINT_QUALITY_DRAFT
```

On input you should pass one of these identifiers, but on return you may get back a positive integer indicating the current resolution setting.

wxPrintData::operator =

void operator =(**const wxPrintData&** *data*)

Assigns print data to this object.

void operator =(**const wxPrintSetupData&** *data*)

Assigns print setup data to this object. wxPrintSetupData is deprecated, but retained for backward compatibility.

wxPrintDialog

This class represents the print and print setup common dialogs. You may obtain a *wxPrinterDC* (p. 1065) device context from a successfully dismissed print dialog.

Derived from

wxDialog (p. 406)
wxWindow (p. 1490)
wxEvtHandler (p. 486)
wxObject (p. 1012)

Include files

<wx/printdlg.h>

See also

wxPrintDialog Overview (p. 1800)

wxPrintDialog::wxPrintDialog

wxPrintDialog(**wxWindow*** *parent*, **wxPrintDialogData*** *data* = *NULL*)

Constructor. Pass a parent window, and optionally a pointer to a block of print data, which will be copied to the print dialog's print data.

See also

wxPrintDialogData (p. 1058)

wxPrintDialog::~wxPrintDialog

~wxPrintDialog()

Destructor. If wxPrintDialog::GetPrintDC has *not* been called, the device context obtained by the dialog (if any) will be deleted.

wxPrintDialog::GetPrintDialogData

wxPrintDialogData& GetPrintDialogData()

Returns the *print dialog data* (p. 1058) associated with the print dialog.

wxPrintDialog::GetPrintDC

wxDC* GetPrintDC()

Returns the device context created by the print dialog, if any. When this function has been called, the ownership of the device context is transferred to the application, so it must then be deleted explicitly.

wxPrintDialog::ShowModal

int ShowModal()

Shows the dialog, returning wxID_OK if the user pressed OK, and wxID_CANCEL otherwise. After this function is called, a device context may be retrievable using *wxPrintDialog::GetPrintDC* (p. 1058).

wxPrintDialogData

This class holds information related to the visual characteristics of wxPrintDialog. It contains a wxPrintData object with underlying printing settings.

Derived from

wxObject (p. 1012)

Include files

<wx/cmndata.h>

See also

wxPrintDialog (p. 1057), *wxPrintDialog Overview* (p. 1800)

wxPrintDialogData::wxPrintDialogData

wxPrintDialogData()

Default constructor.

wxPrintDialogData(**wxPrintDialogData&** *dialogData*)

Copy constructor.

wxPrintDialogData(**wxPrintData&** *printData*)

Construct an object from a print dialog data object.

wxPrintDialogData::~wxPrintDialogData

~wxPrintDialogData()

Destructor.

wxPrintDialogData::EnableHelp

void EnableHelp(**bool** *flag*)

Enables or disables the 'Help' button.

wxPrintDialogData::EnablePageNumbers

void EnablePageNumbers(**bool** *flag*)

Enables or disables the 'Page numbers' controls.

wxPrintDialogData::EnablePrintToFile

void EnablePrintToFile(**bool** *flag*)

Enables or disables the 'Print to file' checkbox.

wxPrintDialogData::EnableSelection

void EnableSelection(**bool** *flag*)

Enables or disables the 'Selection' radio button.

wxPrintDialogData::GetAllPages

bool GetAllPages() **const**

Returns true if the user requested that all pages be printed.

wxPrintDialogData::GetCollate

bool GetCollate() **const**

Returns true if the user requested that the document(s) be collated.

wxPrintDialogData::GetFromPage

int GetFromPage() **const**

Returns the *from* page number, as entered by the user.

wxPrintDialogData::GetMaxPage

int GetMaxPage() **const**

Returns the *maximum* page number.

wxPrintDialogData::GetMinPage

int GetMinPage() **const**

Returns the *minimum* page number.

wxPrintDialogData::GetNoCopies

int GetNoCopies() **const**

Returns the number of copies requested by the user.

wxPrintDialogData::GetPrintData

wxPrintData& GetPrintData()

Returns a reference to the internal wxPrintData object.

wxPrintDialogData::GetPrintToFile

bool GetPrintToFile() **const**

Returns true if the user has selected printing to a file.

wxPrintDialogData::GetSelection

bool GetSelection() **const**

Returns true if the user requested that the selection be printed (where 'selection' is a concept specific to the application).

wxPrintDialogData::GetToPage

int GetToPage() **const**

Returns the *to* page number, as entered by the user.

wxPrintDialogData::Ok

bool Ok() **const**

Returns true if the print data is valid for using in print dialogs. This can return false on Windows if the current printer is not set, for example. On all other platforms, it returns true.

wxPrintDialogData::SetCollate

void SetCollate(**bool** *flag*)

Sets the 'Collate' checkbox to true or false.

wxPrintDialogData::SetFromPage

void SetFromPage(**int** *page*)

Sets the *from* page number.

wxPrintDialogData::SetMaxPage

void SetMaxPage(**int** *page*)

Sets the *maximum* page number.

wxPrintDialogData::SetMinPage

void SetMinPage(**int** *page*)

Sets the *minimum* page number.

wxPrintDialogData::SetNoCopies

void SetNoCopies(**int** *n*)

Sets the default number of copies the user has requested to be printed out.

wxPrintDialogData::SetPrintData

void SetPrintData(**const wxPrintData&** *printData*)

Sets the internal wxPrintData.

wxPrintDialogData::SetPrintToFile

void SetPrintToFile(**bool** *flag*)

Sets the 'Print to file' checkbox to true or false.

wxPrintDialogData::SetSelection

void SetSelection(**bool** *flag*)

Selects the 'Selection' radio button. The effect of printing the selection depends on how the application implements this command, if at all.

wxPrintDialogData::SetSetupDialog

void SetSetupDialog(**bool** *flag*)

Determines whether the dialog to be shown will be the Print dialog (pass false) or Print Setup dialog (pass true).

This function has been deprecated since version 2.5.4.

wxPrintDialogData::SetToPage

void SetToPage(**int** *page*)

Sets the *to* page number.

wxPrintDialogData::operator =

void operator =(**const wxPrintData&** *data*)

Assigns print data to this object.

void operator =(**const wxPrintDialogData&** *data*)

Assigns another print dialog data object to this object.

wxPrinter

This class represents the Windows or PostScript printer, and is the vehicle through which printing may be launched by an application. Printing can also be achieved through

using of lower functions and classes, but this and associated classes provide a more convenient and general method of printing.

Derived from

wxObject (p. 1012)

Include files

<wx/print.h>

See also

Printing framework overview (p. 1815), *wxPrinterDC* (p. 1065), *wxPrintDialog* (p. 1057), *wxPrintout* (p. 1065), *wxPrintPreview* (p. 1069).

wxPrinter::wxPrinter

 wxPrinter(**wxPrintDialogData*** *data* = *NULL*)

Constructor. Pass an optional pointer to a block of print dialog data, which will be copied to the printer object's local data.

See also

wxPrintDialogData (p. 1058),*wxPrintData* (p. 1051)

wxPrinter::CreateAbortWindow

void CreateAbortWindow(**wxWindow*** *parent*, **wxPrintout*** *printout*)

Creates the default printing abort window, with a cancel button.

wxPrinter::GetAbort

bool GetAbort()

Returns true if the user has aborted the print job.

wxPrinter::GetLastError

static wxPrinterError GetLastError()

Return last error. Valid after calling *Print* (p. 1064),*PrintDialog* (p. 1064) or *wxPrintPreview::Print* (p. 1072). These functions set last error to **wxPRINTER_NO_ERROR** if no error happened.

Returned value is one of the following:

wxPRINTER_NO_ERROR   No error happened.

wxPRINTER_CANCELLED   The user cancelled printing.

wxPRINTER_ERROR   There was an error during printing.

wxPrinter::GetPrintDialogData

wxPrintDialogData& GetPrintDialogData()

Returns the *print data* (p. 1051) associated with the printer object.

wxPrinter::Print

bool Print(**wxWindow ****parent*, **wxPrintout ****printout*, **bool** *prompt=true*)

Starts the printing process. Provide a parent window, a user-defined wxPrintout object which controls the printing of a document, and whether the print dialog should be invoked first.

Print could return false if there was a problem initializing the printer device context (current printer not set, for example) or the user cancelled printing. Call*wxPrinter::GetLastError* (p. 1063) to get detailed information about the kind of the error.

wxPrinter::PrintDialog

wxDC* PrintDialog(**wxWindow ****parent*)

Invokes the print dialog. If successful (the user did not press Cancel and no error occurred), a suitable device context will be returned (otherwise NULL is returned -- call*wxPrinter::GetLastError* (p. 1063) to get detailed information about the kind of the error).

The application must delete this device context to avoid a memory leak.

wxPrinter::ReportError

void ReportError(**wxWindow ****parent*, **wxPrintout ****printout*, **const wxString&** *message*)

Default error-reporting function.

wxPrinter::Setup

bool Setup(**wxWindow** **parent*)

Invokes the print setup dialog. Note that the setup dialog is obsolete from Windows 95, though retained for backward compatibility.

wxPrinterDC

A printer device context is specific to Windows, and allows access to any printer with a Windows driver. See *wxDC* (p. 365) for further information on device contexts, and *wxDC::GetSize* (p. 378) for advice on achieving the correct scaling for the page.

Derived from

wxDC (p. 365)
wxObject (p. 365)

Include files

<wx/dcprint.h>

See also

wxDC (p. 365), *Printing framework overview* (p. 1815)

wxPrinterDC::wxPrinterDC

wxPrinterDC(**const wxPrintData&** *printData*)

Pass a *wxPrintData* (p. 1051) object with information necessary for setting up a suitable printer device context. This is the recommended way to construct a wxPrinterDC. Make sure you specify a reference to a *wxPrintData* (p. 1051) object, not a pointer - you may not even get a warning if you pass a pointer instead.

wxPrinterDC(**const wxString&** *driver*, **const wxString&** *device*, **const wxString&** *output*, **const bool** *interactive = true*, **int** *orientation = wxPORTRAIT*)

Constructor. With empty strings for the first three arguments, the default printer dialog is displayed. *device* indicates the type of printer and *output* is an optional file for printing to. The *driver* parameter is currently unused. Use the *Ok* member to test whether the constructor was successful in creating a usable device context.

This constructor is deprecated and retained only for backward compatibility.

wxPrintout

This class encapsulates the functionality of printing out an application document. A new class must be derived and members overridden to respond to calls such as OnPrintPage

and HasPage. Instances of this class are passed to wxPrinter::Print or a wxPrintPreview object to initiate printing or previewing.

Derived from

wxObject (p. 1012)

Include files

<wx/print.h>

See also

Printing framework overview (p. 1815), *wxPrinterDC* (p. 1065), *wxPrintDialog* (p. 1057), *wxPrinter* (p. 1062), *wxPrintPreview* (p. 1069)

wxPrintout::wxPrintout

wxPrintout(**const wxString&** *title* = *"Printout"*)

Constructor. Pass an optional title argument - the current filename would be a good idea. This will appear in the printing list (at least in MSW)

wxPrintout::~wxPrintout

~wxPrintout()

Destructor.

wxPrintout::GetDC

wxDC * GetDC()

Returns the device context associated with the printout (given to the printout at start of printing or previewing). This will be a wxPrinterDC if printing under Windows, a wxPostScriptDC if printing on other platforms, and a wxMemoryDC if previewing.

wxPrintout::GetPageInfo

void GetPageInfo(**int ***minPage*, **int ***maxPage*, **int ***pageFrom*, **int ***pageTo*)

Called by the framework to obtain information from the application about minimum and maximum page values that the user can select, and the required page range to be printed. By default this returns 1, 32000 for the page minimum and maximum values, and 1, 1 for the required page range.

If *minPage* is zero, the page number controls in the print dialog will be disabled.

wxPython note: When this method is implemented in a derived Python class, it should be designed to take no parameters (other than the self reference) and to return a tuple of four integers.

wxPerl note: When this method is overridden in a derived class, it must not take any parameters, and returns a 4-element list.

wxPrintout::GetPageSizeMM

void GetPageSizeMM(**int ***w, **int ***h)

Returns the size of the printer page in millimetres.

wxPython note: This method returns the output-only parameters as a tuple.

wxPerl note: In wxPerl this method takes no arguments and returns a 2-element list (w, h)

wxPrintout::GetPageSizePixels

void GetPageSizePixels(**int ***w, **int ***h)

Returns the size of the printer page in pixels. These may not be the same as the values returned from *wxDC::GetSize* (p. 378) if the printout is being used for previewing, since in this case, a memory device context is used, using a bitmap size reflecting the current preview zoom. The application must take this discrepancy into account if previewing is to be supported.

wxPython note: This method returns the output-only parameters as a tuple.

wxPerl note: In wxPerl this method takes no arguments and returns a 2-element list (w, h)

wxPrintout::GetPPIPrinter

void GetPPIPrinter(**int ***w, **int ***h)

Returns the number of pixels per logical inch of the printer device context. Dividing the printer PPI by the screen PPI can give a suitable scaling factor for drawing text onto the printer. Remember to multiply this by a scaling factor to take the preview DC size into account.

wxPython note: This method returns the output-only parameters as a tuple.

wxPerl note: In wxPerl this method takes no arguments and returns a 2-element list (w, h)

wxPrintout::GetPPIScreen

void GetPPIScreen(**int ****w*, **int ****h*)

Returns the number of pixels per logical inch of the screen device context. Dividing the printer PPI by the screen PPI can give a suitable scaling factor for drawing text onto the printer. Remember to multiply this by a scaling factor to take the preview DC size into account.

wxPrintout::GetTitle

wxString GetTitle()

Returns the title of the printout

wxPython note: This method returns the output-only parameters as a tuple.

wxPerl note: In wxPerl this method takes no arguments and returns a 2-element list (`w, h`)

wxPrintout::HasPage

bool HasPage(**int** *pageNum*)

Should be overridden to return true if the document has this page, or false if not. Returning false signifies the end of the document. By default, HasPage behaves as if the document has only one page.

wxPrintout::IsPreview

bool IsPreview()

Returns true if the printout is currently being used for previewing.

wxPrintout::OnBeginDocument

bool OnBeginDocument(**int** *startPage*, **int** *endPage*)

Called by the framework at the start of document printing. Return false from this function cancels the print job. OnBeginDocument is called once for every copy printed.

The base wxPrintout::OnBeginDocument *must* be called (and the return value checked) from within the overridden function, since it calls wxDC::StartDoc.

wxPython note: If this method is overridden in a Python class then the base class version can be called by using the method `base_OnBeginDocument(startPage, endPage)`.

wxPrintout::OnEndDocument

void OnEndDocument()

Called by the framework at the end of document printing. OnEndDocument is called once for every copy printed.

The base wxPrintout::OnEndDocument *must* be called from within the overridden function, since it calls wxDC::EndDoc.

wxPrintout::OnBeginPrinting

void OnBeginPrinting()

Called by the framework at the start of printing. OnBeginPrinting is called once for every print job (regardless of how many copies are being printed).

wxPrintout::OnEndPrinting

void OnEndPrinting()

Called by the framework at the end of printing. OnEndPrinting is called once for every print job (regardless of how many copies are being printed).

wxPrintout::OnPreparePrinting

void OnPreparePrinting()

Called once by the framework before any other demands are made of the wxPrintout object. This gives the object an opportunity to calculate the number of pages in the document, for example.

wxPrintout::OnPrintPage

bool OnPrintPage(**int** *pageNum*)

Called by the framework when a page should be printed. Returning false cancels the print job. The application can use wxPrintout::GetDC to obtain a device context to draw on.

wxPrintPreview

Objects of this class manage the print preview process. The object is passed a wxPrintout object, and the wxPrintPreview object itself is passed to a wxPreviewFrame object. Previewing is started by initializing and showing the preview frame. Unlike wxPrinter::Print, flow of control returns to the application immediately after the frame is shown.

Derived from

wxObject (p. 1012)

Include files

<wx/print.h>

See also

Printing framework overview (p. 1815), *wxPrinterDC* (p. 1065), *wxPrintDialog* (p. 1057), *wxPrintout* (p. 1065), *wxPrinter* (p. 1062), *wxPreviewCanvas* (p. 1047), *wxPreviewControlBar* (p. 1048), *wxPreviewFrame* (p. 1049).

wxPrintPreview::wxPrintPreview

 wxPrintPreview(**wxPrintout*** *printout*, **wxPrintout*** *printoutForPrinting*,**wxPrintData*** *data=NULL*)

Constructor. Pass a printout object, an optional printout object to be used for actual printing, and the address of an optional block of printer data, which will be copied to the print preview object's print data.

If *printoutForPrinting* is non-NULL, a **Print...** button will be placed on the preview frame so that the user can print directly from the preview interface.

Do not explicitly delete the printout objects once this destructor has been called, since they will be deleted in the wxPrintPreview constructor. The same does not apply to the *data* argument.

Test the Ok member to check whether the wxPrintPreview object was created correctly. Ok could return false if there was a problem initializing the printer device context (current printer not set, for example).

wxPrintPreview::~wxPrintPreview

 ~wxPrinter()

Destructor. Deletes both print preview objects, so do not destroy these objects in your application.

wxPrintPreview::GetCanvas

wxPreviewCanvas* GetCanvas()

Gets the preview window used for displaying the print preview image.

wxPrintPreview::GetCurrentPage

int GetCurrentPage()

Gets the page currently being previewed.

wxPrintPreview::GetFrame

wxFrame * GetFrame()

Gets the frame used for displaying the print preview canvas and control bar.

wxPrintPreview::GetMaxPage

int GetMaxPage()

Returns the maximum page number.

wxPrintPreview::GetMinPage

int GetMinPage()

Returns the minimum page number.

wxPrintPreview::GetPrintout

wxPrintout * GetPrintout()

Gets the preview printout object associated with the wxPrintPreview object.

wxPrintPreview::GetPrintoutForPrinting

wxPrintout * GetPrintoutForPrinting()

Gets the printout object to be used for printing from within the preview interface, or NULL if none exists.

wxPrintPreview::Ok

bool Ok()

Returns true if the wxPrintPreview is valid, false otherwise. It could return false if there was a problem initializing the printer device context (current printer not set, for example).

wxPrintPreview::PaintPage

bool PaintPage(**wxPreviewCanvas ***canvas*, **wxDC&***dc*)

This refreshes the preview window with the preview image. It must be called from the preview window's OnPaint member.

The implementation simply blits the preview bitmap onto the canvas, creating a new preview bitmap if none exists.

wxPrintPreview::Print

bool Print(**bool** *prompt*)

Invokes the print process using the second wxPrintout object supplied in the wxPrintPreview constructor. Will normally be called by the **Print...** panel item on the preview frame's control bar.

Returns false in case of error -- call*wxPrinter::GetLastError* (p. 1063) to get detailed information about the kind of the error.

wxPrintPreview::RenderPage

bool RenderPage(**int** *pageNum*)

Renders a page into a wxMemoryDC. Used internally by wxPrintPreview.

wxPrintPreview::SetCanvas

void SetCanvas(**wxPreviewCanvas*** *window*)

Sets the window to be used for displaying the print preview image.

wxPrintPreview::SetCurrentPage

void SetCurrentPage(**int** *pageNum*)

Sets the current page to be previewed.

wxPrintPreview::SetFrame

void SetFrame(**wxFrame ****frame*)

Sets the frame to be used for displaying the print preview canvas and control bar.

wxPrintPreview::SetPrintout

void SetPrintout(**wxPrintout ****printout*)

Associates a printout object with the wxPrintPreview object.

wxPrintPreview::SetZoom

void SetZoom(**int** *percent*)

Sets the percentage preview zoom, and refreshes the preview canvas accordingly.

wxProcess

The objects of this class are used in conjunction with the*wxExecute* (p. 1592) function. When a wxProcess object is passed to wxExecute(), its *OnTerminate()* (p. 1077) virtual method is called when the process terminates. This allows the program to be (asynchronously) notified about the process termination and also retrieve its exit status which is unavailable from wxExecute() in the case of asynchronous execution.

Please note that if the process termination notification is processed by the parent, it is responsible for deleting the wxProcess object which sent it. However, if it is not processed, the object will delete itself and so the library users should only delete those objects whose notifications have been processed (and call *Detach()* (p. 1074) for others).

wxProcess also supports IO redirection of the child process. For this, you have to call its *Redirect* (p. 1078) method before passing it to*wxExecute* (p. 1592). If the child process was launched successfully,*GetInputStream* (p. 1075),*GetOutputStream* (p. 1075) and*GetErrorStream* (p. 1075) can then be used to retrieve the streams corresponding to the child process standard output, input and error output respectively.

wxPerl note: In wxPerl this class has an additional `Destroy` method, for explicit destruction.

Derived from

wxEvtHandler (p. 486)

Include files

<wx/process.h>

See also

wxExecute (p. 1592)
exec sample (p. 1711)

wxProcess::wxProcess

wxProcess(**wxEvtHandler** * *parent = NULL*, **int** *id = -1*)

wxProcess(**int** *flags*)

Constructs a process object. *id* is only used in the case you want to use wxWidgets

events. It identifies this object, or another window that will receive the event.

If the *parent* parameter is different from NULL, it will receive a wxEVT_END_PROCESS notification event (you should insert EVT_END_PROCESS macro in the event table of the parent to handle it) with the given *id*.

The second constructor creates an object without any associated parent (and hence no id neither) but allows to specify the *flags* which can have the value of wxPROCESS_DEFAULT or wxPROCESS_REDIRECT. Specifying the former value has no particular effect while using the latter one is equivalent to calling *Redirect* (p. 1078).

Parameters

parent

> The event handler parent.

id

> id of an event.

flags

> either wxPROCESS_DEFAULT or wxPROCESS_REDIRECT

wxProcess::~wxProcess

~wxProcess()

Destroys the wxProcess object.

wxProcess::CloseOutput

void CloseOutput()

Closes the output stream (the one connected to the stdin of the child process). This function can be used to indicate to the child process that there is no more data to be read - usually, a filter program will only terminate when the input stream is closed.

wxProcess::Detach

void Detach()

Normally, a wxProcess object is deleted by its parent when it receives the notification about the process termination. However, it might happen that the parent object is destroyed before the external process is terminated (e.g. a window from which this external process was launched is closed by the user) and in this case it **should not delete** the wxProcess object, but**should call Detach()** instead. After the wxProcess object is detached from its parent, no notification events will be sent to the parent and the object will delete itself upon reception of the process termination notification.

wxProcess::GetErrorStream

wxInputStream* GetErrorStream() **const**

Returns an input stream which corresponds to the standard error output (stderr) of the child process.

wxProcess::GetInputStream

wxInputStream* GetInputStream() **const**

It returns an input stream corresponding to the standard output stream of the subprocess. If it is NULL, you have not turned on the redirection. See *wxProcess::Redirect* (p. 1078).

wxProcess::GetOutputStream

wxOutputStream* GetOutputStream() **const**

It returns an output stream correspoding to the input stream of the subprocess. If it is NULL, you have not turned on the redirection. See *wxProcess::Redirect* (p. 1078).

wxProcess::IsErrorAvailable

bool IsErrorAvailable() **const**

Returns `true` if there is data to be read on the child process standard error stream.

See also

IsInputAvailable (p. 1075)

wxProcess::IsInputAvailable

bool IsInputAvailable() **const**

Returns `true` if there is data to be read on the child process standard output stream. This allows to write simple (and extremely inefficient) polling-based code waiting for a better mechanism in future wxWidgets versions.

See the *exec sample* (p. 1711) for an example of using this function.

See also

IsInputOpened (p. 1075)

wxProcess::IsInputOpened

bool IsInputOpened() **const**

Returns `true` if the child process standard output stream is opened.

wxProcess::Kill

static wxKillError Kill(**int** *pid*, **wxSignal** *signal* = *wxSIGNONE*, **int** *flags* =
wxKILL_NOCHILDREN)

Send the specified signal to the given process. Possible signal values are:

```
enum wxSignal
{
    wxSIGNONE = 0,  // verify if the process exists under Unix
    wxSIGHUP,
    wxSIGINT,
    wxSIGQUIT,
    wxSIGILL,
    wxSIGTRAP,
    wxSIGABRT,
    wxSIGEMT,
    wxSIGFPE,
    wxSIGKILL,      // forcefully kill, dangerous!
    wxSIGBUS,
    wxSIGSEGV,
    wxSIGSYS,
    wxSIGPIPE,
    wxSIGALRM,
    wxSIGTERM       // terminate the process gently
};
```

`wxSIGNONE`, `wxSIGKILL` and `wxSIGTERM` have the same meaning under both Unix and
Windows but all the other signals are equivalent to`wxSIGTERM` under Windows.

The *flags* parameter can be wxKILL_NOCHILDREN (the default), or
wxKILL_CHILDREN, in which case the child processes of this process will be killed too.
Note that under Unix, for wxKILL_CHILDREN to work you should have created the
process passing wxEXEC_MAKE_GROUP_LEADER.

Returns the element of `wxKillError` enum:

```
enum wxKillError
{
    wxKILL_OK,              // no error
    wxKILL_BAD_SIGNAL,      // no such signal
    wxKILL_ACCESS_DENIED,   // permission denied
    wxKILL_NO_PROCESS,      // no such process
    wxKILL_ERROR            // another, unspecified error
};
```

See also

wxProcess::Exists (p. 1076), *wxKill* (p. 1594), *Exec sample* (p. 1711)

wxProcess::Exists

static bool Exists(**int** *pid*)

Returns `true` if the given process exists in the system.

See also

wxProcess::Kill (p. 1076), *Exec sample* (p. 1711)

wxProcess::OnTerminate

void OnTerminate(**int** *pid*, **int** *status*)

It is called when the process with the pid *pid* finishes. It raises a wxWidgets event when it isn't overridden.

pid

> The pid of the process which has just terminated.

status

> The exit code of the process.

wxProcess::Open

static wxProcess * Open(**const wxString&** *cmd*, **int** *flags* = *wxEXEC_ASYNC*)

This static method replaces the standard `popen()` function: it launches the process specified by the *cmd* parameter and returns the wxProcess object which can be used to retrieve the streams connected to the standard input, output and error output of the child process.

If the process couldn't be launched, `NULL` is returned. Note that in any case the returned pointer should **not** be deleted, rather the process object will be destroyed automatically when the child process terminates. This does mean that the child process should be told to quit before the main program exits to avoid memory leaks.

Parameters

cmd

> The command to execute, including optional arguments.

flags

> The flags to pass to *wxExecute* (p. 1592). NOTE: wxEXEC_SYNC should not be used.

Return value

A pointer to new wxProcess object or `NULL` on error.

See also

wxExecute (p. 1592)

wxProcess::Redirect

void Redirect()

Turns on redirection. wxExecute will try to open a couple of pipes to catch the subprocess stdio. The caught input stream is returned by GetOutputStream() as a non-seekable stream. The caught output stream is returned by GetInputStream() as a non-seekable stream.

wxProcessEvent

A process event is sent when a process is terminated.

Derived from

wxEvent (p. 483)
wxObject (p. 1012)

Include files

<wx/process.h>

Event table macros

To process a wxProcessEvent, use these event handler macros to direct input to a member function that takes a wxProcessEvent argument.

EVT_END_PROCESS(id, func)	Process a wxEVT_END_PROCESS event.*id* is the identifier of the process object (the id passed to the wxProcess constructor) or a window to receive the event.

See also
wxProcess (p. 1073), *Event handling overview* (p. 1751)

wxProcessEvent::wxProcessEvent

wxProcessEvent(**int** *id = 0*, **int** *pid = 0*, **int** *exitcode = 0*)

Constructor. Takes a wxProcessObject or window id, a process id and an exit status.

wxProcessEvent::GetPid

int GetPid() **const**

Returns the process id.

wxProcessEvent::GetExitCode

int GetExitCode()

Returns the exist status.

wxProgressDialog

This class represents a dialog that shows a short message and a progress bar. Optionally, it can display an ABORT button.

Derived from

wxFrame (p. 579)
wxWindow (p. 1490)
wxEvtHandler (p. 486)
wxObject (p. 1012)

Include files

<wx/progdlg.h>

wxProgressDialog::wxProgressDialog

 wxProgressDialog(**const wxString&** *title*, **const wxString&** *message*, **int** *maximum = 100*, **wxWindow *** *parent = NULL*, **int** *style = wxPD_AUTO_HIDE | wxPD_APP_MODAL*)

Constructor. Creates the dialog, displays it and disables user input for other windows, or, if wxPD_APP_MODAL flag is not given, for its parent window only.

Parameters

title

Dialog title to show in titlebar.

message

Message displayed above the progress bar.

maximum

Maximum value for the progress bar.

parent

> Parent window.

style

> The dialog style. This is the combination of the following bitmask constants defined in wx/defs.h:

> | | |
> |---|---|
> | **wxPD_APP_MODAL** | Make the progress dialog modal. If this flag is not given, it is only "locally" modal - that is the input to the parent window is disabled, but not to the other ones. |
> | **wxPD_AUTO_HIDE** | Causes the progress dialog to disappear from screen as soon as the maximum value of the progress meter has been reached. |
> | **wxPD_SMOOTH** | Causes smooth progress of the gauge control. |
> | **wxPD_CAN_ABORT** | This flag tells the dialog that it should have a "Cancel" button which the user may press. If this happens, the next call to *Update()* (p. 1081) will return false. |
> | **wxPD_CAN_SKIP** | This flag tells the dialog that it should have a "Skip" button which the user may press. If this happens, the next call to *Update()* (p. 1081) will return true in its skip parameter. |
> | **wxPD_ELAPSED_TIME** | This flag tells the dialog that it should show elapsed time (since creating the dialog). |
> | **wxPD_ESTIMATED_TIME** | This flag tells the dialog that it should show estimated time. |
> | **wxPD_REMAINING_TIME** | This flag tells the dialog that it should show remaining time. |

wxProgressDialog::~wxProgressDialog

~wxMessageDialog()

Destructor. Deletes the dialog and enables all top level windows.

wxProgressDialog::Resume

void Resume()

Can be used to continue with the dialog, after the user had chosen ABORT.

wxProgressDialog::Update

virtual bool Update(**int** *value*, **const wxString&** *newmsg = "",* **bool** **skip = NULL*)

Updates the dialog, setting the progress bar to the new value and, if given changes the message above it. Returns true unless the Cancel button has been pressed.

If false is returned, the application can either immediately destroy the dialog or ask the user for the confirmation and if the abort is not confirmed the dialog may be resumed with *Resume* (p. 1080) function.

Parameters

value

> The new value of the progress meter. It should be less than or equal to the maximum value given to the constructor and the dialog is closed if it is equal to the maximum.

newmsg

> The new messages for the progress dialog text, if it is empty (which is the default) the message is not changed.

skip

> If "Skip" button was pressed since last *Update* (p. 1081) call the skip is true.

wxPropertySheetDialog

This class represents a property sheet dialog: a tabbed dialog for showing settings. It is optimized to show with flat tabs on PocketPC devices.

To use this class, call *wxPropertySheetDialog::Create* (p. 1082) from your own Create function. Then call *CreateButtons* (p. 1083), and create pages, adding them to the book control. Finally call *LayoutDialog* (p. 1083).

For example:

```
bool MyPropertySheetDialog::Create(...)
{
    if (!wxPropertySheetDialog::Create(...))
        return false;

    CreateButtons(wxOK|wxCANCEL|wxHELP);

    // Add page
    wxPanel* panel = new wxPanel(GetBookCtrl(), ...);
    GetBookCtrl()->AddPage(panel, wxT("General"));
```

```
        LayoutDialog();
        return true;
    }
```

If necessary, override CreateBookCtrl and AddBookCtrl to create and add a different kind of book control. You would then need to use two-step construction for the dialog.

Derived from

wxDialog (p. 406)
wxWindow (p. 1490)
wxEvtHandler (p. 486)
wxObject (p. 1012)

Include files

<wx/propdlg.h> <wx/generic/propdlg.h>

wxPropertySheetDialog::wxPropertySheetDialog

 wxPropertySheetDialog(**wxWindow*** *parent*, **wxWindowID** *id*, **const wxString&** *title*, **const wxPoint&** *pos* = *wxDefaultPosition*, **const wxSize&** *size* = *wxDefaultSize*, **long** *style* = *wxDEFAULT_DIALOG_STYLE*, **const wxString&** *name* = *"dialogBox"*)

Constructor.

wxPropertySheetDialog::AddBookCtrl

virtual void AddBookCtrl(**wxSizer*** *sizer*)

Override this if you wish to add the book control in a way different from the standard way (for example, using different spacing).

wxPropertySheetDialog::Create

bool Create(**wxWindow*** *parent*, **wxWindowID** *id*, **const wxString&** *title*, **const wxPoint&** *pos* = *wxDefaultPosition*, **const wxSize&** *size* = *wxDefaultSize*, **long** *style* = *wxDEFAULT_DIALOG_STYLE*, **const wxString&** *name* = *"dialogBox"*)

Call this from your own Create function, before adding buttons and pages.

wxPropertySheetDialog::CreateBookCtrl

virtual wxBookCtrlBase* CreateBookCtrl()

Override this if you wish to create a different kind of book control; by default, a

wxNotebook is created.

wxPropertySheetDialog::CreateButtons

void CreateButtons(**int** *flags=wxOK|wxCANCEL*)

Call this to create the buttons for the dialog. This calls *wxDialog::CreateButtonSizer* (p. 409), and the flags are the same. On PocketPC, no buttons are created.

wxPropertySheetDialog::GetBookCtrl

wxBookCtrlBase* GetBookCtrl() **const**

Returns the book control that will contain your settings pages.

wxPropertySheetDialog::GetInnerSizer

wxSizer* GetInnerSizer() **const**

Returns the inner sizer that contains the book control and button sizer.

wxPropertySheetDialog::LayoutDialog

void LayoutDialog()

Call this to lay out the dialog. On PocketPC, this does nothing, since the dialog will be shown full-screen, and the layout will be done when the dialog receives a size event.

wxPropertySheetDialog::SetBookCtrl

void SetBookCtrl(**wxBookCtrlBase*** *bookCtrl*)

Sets the book control used for the dialog. You will normally not need to use this.

wxPropertySheetDialog::SetInnerSizer

void SetInnerSizer(**wxSizer*** *sizer*)

Sets the inner sizer that contains the book control and button sizer. You will normally not need to use this.

wxProtocol

Derived from

wxSocketClient (p. 1219)

Include files

<wx/protocol/protocol.h>

See also

wxSocketBase (p. 1203)*, wxURL* (p. 1459)

wxProtocol::Reconnect

bool Reconnect()

Tries to reestablish a previous opened connection (close and renegotiate connection).

Return value

true, if the connection is established, else false.

wxProtocol::GetInputStream

wxInputStream * GetInputStream(**const wxString&** *path*)

Creates a new input stream on the specified path. You can use all but seek functionality of wxStream. Seek isn't available on all stream. For example, http or ftp streams doesn't deal with it. Other functions like StreamSize and Tell aren't available for the moment for this sort of stream. You will be notified when the EOF is reached by an error.

Return value

Returns the initialized stream. You will have to delete it yourself once you don't use it anymore. The destructor closes the network connection.

See also

wxInputStream (p. 814)

wxProtocol::Abort

bool Abort()

Abort the current stream.

Warning

It is advised to destroy the input stream instead of aborting the stream this way.

Return value

Returns true, if successful, else false.

wxProtocol::GetError

wxProtocolError GetError()

Returns the last occurred error.

wxPROTO_NOERR	No error.
wxPROTO_NETERR	A generic network error occurred.
wxPROTO_PROTERR	An error occurred during negotiation.
wxPROTO_CONNERR	The client failed to connect the server.
wxPROTO_INVVAL	Invalid value.
wxPROTO_NOHNDLR	.
wxPROTO_NOFILE	The remote file doesn't exist.
wxPROTO_ABRT	Last action aborted.
wxPROTO_RCNCT	An error occurred during reconnection.
wxPROTO_STREAM	Someone tried to send a command during a transfer.

wxProtocol::GetContentType

wxString GetContentType()

Returns the type of the content of the last opened stream. It is a mime-type.

wxProtocol::SetUser

void SetUser(**const wxString&** *user*)

Sets the authentication user. It is mainly useful when FTP is used.

wxProtocol::SetPassword

void SetPassword(**const wxString&** *user*)

Sets the authentication password. It is mainly useful when FTP is used.

wxQuantize

Performs quantization, or colour reduction, on a wxImage.

Functions in this class are static and so a wxQuantize object need not be created.

Derived from

wxObject (p. 1012)

Include files

<wx/quantize.h>

wxQuantize::wxQuantize

 wxQuantize()

Constructor. You do not need to construct a wxQuantize object since its functions are static.

wxQuantize::Quantize

bool Quantize(**const wxImage&** *src*, **wxImage&** *dest*, **wxPalette**** *pPalette*, **int** *desiredNoColours = 236*, **unsigned char**** *eightBitData = 0*, **int** *flags = wxQUANTIZE_INCLUDE_WINDOWS_COLOURS|wxQUANTIZE_FILL_DESTINATION_ IMAGE|wxQUANTIZE_RETURN_8BIT_DATA*)

Reduce the colours in the source image and put the result into the destination image. Both images may be the same, to overwrite the source image. Specify an optional palette pointer to receive the resulting palette. This palette may be passed to ConvertImageToBitmap, for example.

If you pass a palette pointer, you must free the palette yourself.

bool Quantize(**const wxImage&** *src*, **wxImage&** *dest*, **int** *desiredNoColours = 236*, **unsigned char**** *eightBitData = 0*, **int** *flags = wxQUANTIZE_INCLUDE_WINDOWS_COLOURS|wxQUANTIZE_FILL_DESTINATION_ IMAGE|wxQUANTIZE_RETURN_8BIT_DATA*)

This version sets a palette in the destination image so you don't have to manage it yourself.

wxQuantize::DoQuantize

void DoQuantize(**unsigned** *w*, **unsigned** *h*, **unsigned char**** *in_rows*, **unsigned char**** *out_rows*, **unsigned char*** *palette*, **int** *desiredNoColours*)

Converts input bitmap(s) into 8bit representation with custom palette.

in_rows and out_rows are arrays [0..h-1] of pointer to rows (in_rows contains w * 3 bytes per row, out_rows w bytes per row).

Fills out_rows with indexes into palette (which is also stored into palette variable).

wxQueryLayoutInfoEvent

This event is sent when *wxLayoutAlgorithm* (p. 835) wishes to get the size, orientation and alignment of a window. More precisely, the event is sent by the OnCalculateLayout handler which is itself invoked by wxLayoutAlgorithm.

Derived from

wxEvent (p. 483)
wxObject (p. 1012)

Include files

<wx/laywin.h>

Event table macros

EVT_QUERY_LAYOUT_INFO(func) Process a wxEVT_QUERY_LAYOUT_INFO event, to get size, orientation and alignment from a window.

Data structures

```
enum wxLayoutOrientation {
    wxLAYOUT_HORIZONTAL,
    wxLAYOUT_VERTICAL
};

enum wxLayoutAlignment {
    wxLAYOUT_NONE,
    wxLAYOUT_TOP,
    wxLAYOUT_LEFT,
    wxLAYOUT_RIGHT,
    wxLAYOUT_BOTTOM,
};
```

See also

wxCalculateLayoutEvent (p. 120), *wxSashLayoutWindow* (p. 1129), *wxLayoutAlgorithm* (p. 835).

wxQueryLayoutInfoEvent::wxQueryLayoutInfoEvent

wxQueryLayoutInfoEvent(**wxWindowID** *id = 0*)

Constructor.

wxQueryLayoutInfoEvent::GetAlignment

void GetAlignment() **const**

Specifies the alignment of the window (which side of the remaining parent client area the window sticks to). One of wxLAYOUT_TOP, wxLAYOUT_LEFT, wxLAYOUT_RIGHT, wxLAYOUT_BOTTOM.

wxQueryLayoutInfoEvent::GetFlags

int GetFlags() **const**

Returns the flags associated with this event. Not currently used.

wxQueryLayoutInfoEvent::GetOrientation

wxLayoutOrientation GetOrientation() **const**

Returns the orientation that the event handler specified to the event object. May be one of wxLAYOUT_HORIZONTAL, wxLAYOUT_VERTICAL.

wxQueryLayoutInfoEvent::GetRequestedLength

int GetRequestedLength() **const**

Returns the requested length of the window in the direction of the window orientation. This information is not yet used.

wxQueryLayoutInfoEvent::GetSize

wxSize GetSize() **const**

Returns the size that the event handler specified to the event object as being the requested size of the window.

wxQueryLayoutInfoEvent::SetAlignment

void SetAlignment(**wxLayoutAlignment** *alignment*)

Call this to specify the alignment of the window (which side of the remaining parent client area the window sticks to). May be one of wxLAYOUT_TOP, wxLAYOUT_LEFT, wxLAYOUT_RIGHT, wxLAYOUT_BOTTOM.

wxQueryLayoutInfoEvent::SetFlags

void SetFlags(**int** *flags*)

Sets the flags associated with this event. Not currently used.

wxQueryLayoutInfoEvent::SetOrientation

void SetOrientation(**wxLayoutOrientation** *orientation*)

Call this to specify the orientation of the window. May be one of
wxLAYOUT_HORIZONTAL, wxLAYOUT_VERTICAL.

wxQueryLayoutInfoEvent::SetRequestedLength

void SetRequestedLength(**int** *length*)

Sets the requested length of the window in the direction of the window orientation. This
information is not yet used.

wxQueryLayoutInfoEvent::SetSize

void SetSize(**const wxSize&** *size*)

Call this to let the calling code know what the size of the window is.

wxRadioBox

A radio box item is used to select one of number of mutually exclusive choices. It is
displayed as a vertical column or horizontal row of labelled buttons.

Derived from

wxControl (p. 215)
wxWindow (p. 1490)
wxEvtHandler (p. 486)
wxObject (p. 1012)

Include files

<wx/radiobox.h>

Window styles

wxRA_SPECIFY_ROWS	The major dimension parameter refers to the maximum number of rows.
wxRA_SPECIFY_COLS	The major dimension parameter refers to the maximum

number of columns.

wxRA_USE_CHECKBOX	Use of the checkbox controls instead of radio buttons (currently supported only on PalmOS)

See also *window styles overview* (p. 1762).

Event handling

EVT_RADIOBOX(id, func)	Process a wxEVT_COMMAND_RADIOBOX_SELECTED event, when a radiobutton is clicked.

See also

Event handling overview (p. 1751), *wxRadioButton* (p. 1096), *wxCheckBox* (p. 134)

wxRadioBox::wxRadioBox

wxRadioBox()

Default constructor.

wxRadioBox(**wxWindow*** *parent*, **wxWindowID** *id*, **const wxString&** *label*, **const wxPoint&** *point = wxDefaultPosition*, **const wxSize&** *size = wxDefaultSize*, **int** *n = 0*, **const wxString** *choices[] = NULL*, **int** *majorDimension = 0*, **long** *style = wxRA_SPECIFY_COLS*, **const wxValidator&** *validator = wxDefaultValidator*, **const wxString&** *name = "radioBox"*)

wxRadioBox(**wxWindow*** *parent*, **wxWindowID** *id*, **const wxString&** *label*, **const wxPoint&** *point*, **const wxSize&** *size*, **const wxArrayString&** *choices*, **int** *majorDimension = 0*, **long** *style = wxRA_SPECIFY_COLS*, **const wxValidator&** *validator = wxDefaultValidator*, **const wxString&** *name = "radioBox"*)

Constructor, creating and showing a radiobox.

Parameters

parent

Parent window. Must not be NULL.

id

Window identifier. A value of -1 indicates a default value.

label

Label for the static box surrounding the radio buttons.

pos

Window position. If the position (-1, -1) is specified then a default position is chosen.

size

Window size. If the default size (-1, -1) is specified then a default size is chosen.

n

Number of choices with which to initialize the radiobox.

choices

An array of choices with which to initialize the radiobox.

majorDimension

Specifies the maximum number of rows (if style contains wxRA_SPECIFY_ROWS) or columns (if style contains wxRA_SPECIFY_COLS) for a two-dimensional radiobox.

style

Window style. See *wxRadioBox* (p. 1089).

validator

Window validator.

name

Window name.

See also

wxRadioBox::Create (p. 1091), *wxValidator* (p. 1461)

wxPython note: The wxRadioBox constructor in wxPython reduces the `n` and `choices` arguments are to a single argument, which is a list of strings.

wxPerl note: In wxPerl there is just an array reference in place of `n` and `choices`.

wxRadioBox::~wxRadioBox

~wxRadioBox()

Destructor, destroying the radiobox item.

wxRadioBox::Create

bool Create(**wxWindow*** *parent*, **wxWindowID** *id*, **const wxString&** *label*, **const wxPoint&** *point* = *wxDefaultPosition*, **const wxSize&** *size* = *wxDefaultSize*, **int** *n = 0*,

const wxString *choices[] = NULL*, **int** *majorDimension = 0*, **long** *style =* *wxRA_SPECIFY_COLS*, **const wxValidator&** *validator = wxDefaultValidator*, **const** **wxString&** *name = "radioBox"*)

bool Create(**wxWindow*** *parent*, **wxWindowID** *id*, **const wxString&** *label*, **const** **wxPoint&** *point*, **const wxSize&** *size*, **const wxArrayString&** *choices*, **int** *majorDimension = 0*, **long** *style = wxRA_SPECIFY_COLS*, **const wxValidator&** *validator = wxDefaultValidator*, **const wxString&** *name = "radioBox"*)

Creates the radiobox for two-step construction. See *wxRadioBox::wxRadioBox* (p. 1090) for further details.

wxRadioBox::Enable

virtual bool Enable(**bool** *enable* = `true`)

Enables or disables the entire radiobox.

virtual bool Enable(**int** *n*, **bool** *enable* = `true`)

Enables or disables an individual button in the radiobox.

Parameters

enable

   true to enable, false to disable.

n

   The zero-based button to enable or disable.

wxPython note: In place of a single overloaded method name, wxPython implements the following methods:

Enable(flag)	Enables or disables the entire radiobox.
EnableItem(n, flag)	Enables or disables an individual button in the radiobox.

See also

wxWindow::Enable (p. 1500)

wxRadioBox::FindString

int FindString(**const wxString&** *string*) **const**

Finds a button matching the given string, returning the position if found, or -1 if not found.

Parameters

string

>   The string to find.

wxRadioBox::GetCount

int GetCount() **const**

Returns the number of items in the radiobox.

wxRadioBox::GetLabel

wxString GetLabel() **const**

Returns the radiobox label.

Parameters

n

>   The zero-based button index.

See also

wxRadioBox::SetLabel (p. 1094)

wxPython note: In place of a single overloaded method name, wxPython implements the following methods:

GetLabel()	Returns the radiobox label.
GetItemLabel(n)	Returns the label for the given button.

wxRadioBox::GetSelection

int GetSelection() **const**

Returns the zero-based position of the selected button.

wxRadioBox::GetStringSelection

wxString GetStringSelection() **const**

Returns the selected string.

wxRadioBox::GetString

wxString GetString(**int** *n*) **const**

Returns the label for the button at the given position.

Parameters

n

> The zero-based button position.

wxRadioBox::Number

int Number() **const**

Obsolescence note: This method is obsolete and was replaced with*GetCount* (p. 1093), please use the new method in the new code. This method is only available if wxWidgets was compiled withWXWIN_COMPATIBILITY_2_2 defined and will disappear completely in future versions.

Returns the number of buttons in the radiobox.

wxRadioBox::SetLabel

void SetLabel(**const wxString&** *label*)

Sets the radiobox label.

Parameters

label

> The label to set.

n

> The zero-based button index.

wxPython note: In place of a single overloaded method name, wxPython implements the following methods:

SetLabel(string)	Sets the radiobox label.
SetItemLabel(n, string)	Sets a label for a radio button.

wxRadioBox::SetSelection

void SetSelection(**int** *n*)

Sets a button by passing the desired string position. This does not cause a wxEVT_COMMAND_RADIOBOX_SELECTED event to get emitted.

Parameters

n

> The zero-based button position.

wxRadioBox::SetStringSelection

void SetStringSelection(**const wxString&** *string*)

Sets the selection to a button by passing the desired string. This does not cause a wxEVT_COMMAND_RADIOBOX_SELECTED event to get emitted.

Parameters

string

> The label of the button to select.

wxRadioBox::Show

virtual bool Show(**const bool** *show* = `true`)

Shows or hides the entire radiobox.

virtual bool Show(**int** *item*, **const bool** *show* = `true`)

Shows or hides individual buttons.

Parameters

show

> true to show, false to hide.

item

> The zero-based position of the button to show or hide.

Return value

`true` if the box or item has been shown or hidden or `false` if nothing was done because it already was in the requested state.

See also

wxWindow::Show (p. 1544)

wxPython note: In place of a single overloaded method name, wxPython implements the following methods:

Show(flag)	Shows or hides the entire radiobox.
ShowItem(n, flag)	Shows or hides individual buttons.

wxRadioButton

A radio button item is a button which usually denotes one of several mutually exclusive options. It has a text label next to a (usually) round button.

You can create a group of mutually-exclusive radio buttons by specifying `wxRB_GROUP` for the first in the group. The group ends when another radio button group is created, or there are no more radio buttons.

Derived from

wxControl (p. 215)
wxWindow (p. 1490)
wxEvtHandler (p. 486)
wxObject (p. 1012)

Include files

<wx/radiobut.h>

Window styles

wxRB_GROUP	Marks the beginning of a new group of radio buttons.
wxRB_SINGLE	In some circumstances, radio buttons that are not consecutive siblings trigger a hang bug in Windows (only). If this happens, add this style to mark the button as not belonging to a group, and implement the mutually-exclusive group behaviour yourself.
wxRB_USE_CHECKBOX	Use a checkbox button instead of radio button (currently supported only on PalmOS).

See also *window styles overview* (p. 1762).

Event handling

EVT_RADIOBUTTON(id, func)	Process a wxEVT_COMMAND_RADIOBUTTON_SELECTED event, when the radiobutton is clicked.

See also

Event handling overview (p. 1751), *wxRadioBox* (p. 1089), *wxCheckBox* (p. 134)

wxRadioButton::wxRadioButton

wxRadioButton()

Default constructor.

wxRadioButton(**wxWindow*** *parent*, **wxWindowID** *id*, **const wxString&** *label*, **const wxPoint&** *pos* = *wxDefaultPosition*, **const wxSize&** *size* = *wxDefaultSize*, **long** *style* = *0*, **const wxValidator&** *validator* = *wxDefaultValidator*, **const wxString&** *name* = *"radioButton"*)

Constructor, creating and showing a radio button.

Parameters

parent

> Parent window. Must not be NULL.

id

> Window identifier. A value of -1 indicates a default value.

label

> Label for the radio button.

pos

> Window position. If the position (-1, -1) is specified then a default position is chosen.

size

> Window size. If the default size (-1, -1) is specified then a default size is chosen.

style

> Window style. See *wxRadioButton* (p. 1096).

validator

> Window validator.

name

> Window name.

See also

wxRadioButton::Create (p. 1098), *wxValidator* (p. 1461)

wxRadioButton::~wxRadioButton

void ~wxRadioButton()

Destructor, destroying the radio button item.

wxRadioButton::Create

bool Create(**wxWindow*** *parent*, **wxWindowID** *id*, **const wxString&** *label*, **const wxPoint&** *pos = wxDefaultPosition*, **const wxSize&** *size = wxDefaultSize*, **long** *style = 0*, **const wxValidator&** *validator = wxDefaultValidator*, **const wxString&** *name = "radioButton"*)

Creates the choice for two-step construction. See *wxRadioButton::wxRadioButton* (p. 1097) for further details.

wxRadioButton::GetValue

bool GetValue() **const**

Returns true if the radio button is depressed, false otherwise.

wxRadioButton::SetValue

void SetValue(**const bool** *value*)

Sets the radio button to selected or deselected status. This does not cause a wxEVT_COMMAND_RADIOBUTTON_SELECTED event to get emitted.

Parameters

value

 true to select, false to deselect.

wxRealPoint

A **wxRealPoint** is a useful data structure for graphics operations. It contains floating point point *x* and *y* members. See also *wxPoint* (p. 1045) for an integer version.

Derived from

None

Include files

<wx/gdicmn.h>

See also

wxPoint (p. 1045)

wxRealPoint::wxRealPoint

 wxRealPoint()

 wxRealPoint(**double** *x*, **double** *y*)

Create a point.

double x

double y

Members of the **wxRealPoint** object.

wxRect

A class for manipulating rectangles.

Derived from

None

Include files

<wx/gdicmn.h>

See also

wxPoint (p. 1045), *wxSize* (p. 1173)

wxRect::wxRect

 wxRect()

Default constructor.

 wxRect(**int** *x*, **int** *y*, **int** *width*, **int** *height*)

Creates a wxRect object from x, y, width and height values.

 wxRect(**const wxPoint&** *topLeft*, **const wxPoint&** *bottomRight*)

Creates a wxRect object from top-left and bottom-right points.

 wxRect(**const wxPoint&** *pos*, **const wxSize&** *size*)

Creates a wxRect object from position and size values.

wxRect(**const wxSize&** *size*)

Creates a wxRect object from size values at the origin.

wxRect::x

int x

x coordinate of the top-level corner of the rectangle.

wxRect::y

int y

y coordinate of the top-level corner of the rectangle.

wxRect::width

int width

Width member.

wxRect::height

int height

Height member.

wxRect::Deflate

void Deflate(**wxCoord** *dx*, **wxCoord** *dy*)

void Deflate(**wxCoord** *diff*)

wxRect Deflate(**wxCoord** *dx*, **wxCoord** *dy*) **const**

Decrease the rectangle size.

This method is the opposite from *Inflate* (p. 1102): Deflate(a, b) is equivalent to Inflate(-a, -b). Please refer to *Inflate* (p. 1102) for full description.

See also

Inflate (p. 1102)

wxRect::GetBottom

int GetBottom() **const**

Gets the bottom point of the rectangle.

wxRect::GetHeight

int GetHeight() **const**

Gets the height member.

wxRect::GetLeft

int GetLeft() **const**

Gets the left point of the rectangle (the same as *wxRect::GetX* (p. 1102)).

wxRect::GetPosition

wxPoint GetPosition() **const**

Gets the position.

wxRect::GetTopLeft

wxPoint GetTopLeft() **const**

Gets the topleft position of the rectangle. (Same as GetPosition).

wxRect::GetBottomRight

wxPoint GetBottomRight() **const**

Gets the bottom right position. Returns the bottom right point inside the rectangle.

wxRect::GetRight

int GetRight() **const**

Gets the right point of the rectangle.

wxRect::GetSize

wxSize GetSize() **const**

Gets the size.

See also

wxRect::SetSize (p. 1104)

wxRect::GetTop

int GetTop() **const**

Gets the top point of the rectangle (the same as *wxRect::GetY* (p. 1102)).

wxRect::GetWidth

int GetWidth() **const**

Gets the width member.

wxRect::GetX

int GetX() **const**

Gets the x member.

wxRect::GetY

int GetY() **const**

Gets the y member.

wxRect::Inflate

void Inflate(**wxCoord** *dx*, **wxCoord** *dy*)

void Inflate(**wxCoord** *diff*)

wxRect Inflate(**wxCoord** *dx*, **wxCoord** *dy*) **const**

Increases the size of the rectangle.

The second form uses the same *diff* for both *dx* and *dy*.

The first two versions modify the rectangle in place, the last one returns a new rectangle leaving this one unchanged.

The left border is moved farther left and the right border is moved farther right by *dx*. The upper border is moved farther up and the bottom border is moved farther down by *dy*. (Note the the width and height of the rectangle thus change by 2*dx and 2*dy, respectively.) If one or both of *dx* and *dy* are negative, the opposite happens: the rectangle size decreases in the respective direction.

Inflating and deflating behaves "naturally". Defined more precisely, that means:

1. "Real" inflates (that is, *dx* and/or *dy* >= 0) are not constrained. Thus inflating a rectangle can cause its upper left corner to move into the negative numbers. (the versions prior to 2.5.4 forced the top left coordinate to not fall below (0, 0), which implied a forced move of the rectangle.)

2. Deflates are clamped to not reduce the width or height of the rectangle below zero. In such cases, the top-left corner is nonetheless handled properly. For example, a rectangle at (10, 10) with size (20, 40) that is inflated by (-15, -15) will become located at (20, 25) at size (0, 10). Finally, observe that the width and height are treated independently. In the above example, the width is reduced by 20, whereas the height is reduced by the full 30 (rather than also stopping at 20, when the width reached zero).

See also

Deflate (p. 1100)

wxRect::Inside

bool Inside(**int** *x*, **int** *y*) **const**

bool Inside(**const wxPoint&** *pt*) **const**

Returns `true` if the given point is inside the rectangle (or on its boundary) and `false` otherwise.

wxRect::Intersects

bool Intersects(**const wxRect&** *rect*) **const**

Returns `true` if this rectangle has a non empty intersection with the rectangle *rect* and `false` otherwise.

wxRect::IsEmpty

bool IsEmpty() **const**

Returns `true` if this rectangle has a width or height less than or equal to 0 and `false` otherwise.

wxRect::Offset

void Offset(**wxCoord** *dx*, **wxCoord** *dy*)

void Offset(**const wxPoint&** *pt*)

Moves the rectangle by the specified offset. If *dx* is positive, the rectangle is moved to the right, if *dy* is positive, it is moved to the bottom, otherwise it is moved to the left or top respectively.

wxRect::SetHeight

void SetHeight(**int** *height*)

Sets the height.

wxRect::SetSize

void SetSize(**const wxSize&** *s*)

Sets the size.

See also

wxRect::GetSize (p. 1101)

wxRect::SetWidth

void SetWidth(**int** *width*)

Sets the width.

wxRect::SetX

void SetX(**int** *x*)

Sets the x position.

wxRect::SetY

void SetY(**int** *y*)

Sets the y position.

wxRect::Union

wxRect Union(**const wxRect&** *rect*) **const**

wxRect& Union(**const wxRect&** *rect*)

Modifies the rectangle to contain the bounding box of this rectangle and the one passed in as parameter. The const version returns the new rectangle, the other one modifies this rectangle in place.

wxRect::operator =

void operator =(const wxRect& *rect*)

Assignment operator.

wxRect::operator ==

bool operator ==(const wxRect& *rect*)

Equality operator.

wxRect::operator !=

bool operator !=(const wxRect& *rect*)

Inequality operator.

wxRecursionGuard

wxRecursionGuard is a very simple class which can be used to prevent reentrancy problems in a function. It is not thread-safe and so should be used only in the single-threaded programs or in combination with some thread synchronization mechanisms.

```
wxRecursionGuard is always used together with the wxRecursionGuardFlag (p.
1106) like in this example:    void Foo()
{
    static wxRecursionGuardFlag s_flag;
    wxRecursionGuard guard(s_flag);
    if ( guard.IsInside() )
    {
        // don't allow reentrancy
        return;
    }

    ...
}
```

As you can see, wxRecursionGuard simply tests the flag value and sets it to true if it hadn't been already set. *IsInside()* (p. 1106) allows testing the old flag value. The advantage of using this class compared to directly manipulating the flag is that the flag is always reset in the wxRecursionGuard destructor and so you don't risk to forget to do it even if the function returns in an unexpected way (for example because an exception has been thrown).

Derived from

No base class

<wx/recguard.h>

wxRecursionGuard::wxRecursionGuard

wxRecursionGuard(**wxRecursionGuardFlag&** *flag*)

A wxRecursionGuard object must always be initialized with a (static) *wxRecursionGuardFlag* (p. 1106). The constructor saves the value of the flag to be able to return the correct value from *IsInside* (p. 1106).

wxRecursionGuard::~wxRecursionGuard

~wxRecursionGuard()

The destructor resets the flag value so that the function can be entered again the next time.

Note that it is not virtual and so this class is not meant to be derived from (besides, there is absolutely no reason to do it anyhow).

wxRecursionGuard::IsInside

bool IsInside() **const**

Returns `true` if we're already inside the code block "protected" by this wxRecursionGuard (i.e. between this line and the end of current scope). Usually the function using wxRecursionGuard takes some specific actions in such case (may be simply returning) to prevent reentrant calls to itself.

If this method returns `false`, it is safe to continue.

wxRecursionGuardFlag

This is a completely opaque class which exists only to be used with *wxRecursionGuard* (p. 1105), please see the example in that class documentation.

Please notice that wxRecursionGuardFlag object *must* be declared `static` or the recursion would never be detected.

No base class

Include files

<wx/recguard.h>

wxRegEx

wxRegEx represents a regular expression. This class provides support for regular expressions matching and also replacement.

It is built on top of either the system library (if it has support for POSIX regular expressions - which is the case of the most modern Unices) or uses the built in Henry Spencer's library. Henry Spencer would appreciate being given credit in the documentation of software which uses his library, but that is not a requirement.

Regular expressions, as defined by POSIX, come in two flavours: *extended*and *basic*. The builtin library also adds a third flavour of expression *advanced* (p. 1866), which is not available when using the system library.

Unicode is fully supported only when using the builtin library. When using the system library in Unicode mode, the expressions and data are translated to the default 8-bit encoding before being passed to the library.

On platforms where a system library is available, the default is to use the builtin library for Unicode builds, and the system library otherwise. It is possible to use the other if preferred by selecting it when building the wxWidgets.

Derived from

No base class

Data structures

Flags for regex compilation to be used with *Compile()* (p. 1109):

```
enum
{
    // use extended regex syntax
    wxRE_EXTENDED = 0,

    // use advanced RE syntax (built-in regex only)
#ifdef wxHAS_REGEX_ADVANCED
    wxRE_ADVANCED = 1,
#endif

    // use basic RE syntax
    wxRE_BASIC    = 2,

    // ignore case in match
    wxRE_ICASE    = 4,

    // only check match, don't set back references
    wxRE_NOSUB    = 8,

    // if not set, treat '\n' as an ordinary character, otherwise it is
    // special: it is not matched by '.' and '^' and '$' always match
    // after/before it regardless of the setting of wxRE_NOT[BE]OL
```

```
        wxRE_NEWLINE  = 16,

        // default flags
        wxRE_DEFAULT  = wxRE_EXTENDED
    }
```

Flags for regex matching to be used with *Matches()* (p. 1110).

These flags are mainly useful when doing several matches in a long string to prevent erroneous matches for '' and '$':

```
    enum
    {
        // '^' doesn't match at the start of line
        wxRE_NOTBOL = 32,

        // '$' doesn't match at the end of line
        wxRE_NOTEOL = 64
    }
```

Examples

A bad example of processing some text containing email addresses (the example is bad because the real email addresses can have more complicated form than user@host.net):

```
    wxString text;
    ...
    wxRegEx reEmail = wxT("([^@]+)@([[:alnum:].-_].)+([[:alnum:]]+)");
    if ( reEmail.Matches(text) )
    {
        wxString text = reEmail.GetMatch(email);
        wxString username = reEmail.GetMatch(email, 1);
        if ( reEmail.GetMatch(email, 3) == wxT("com") ) // .com TLD?
        {
            ...
        }
    }

    // or we could do this to hide the email address
    size_t count = reEmail.ReplaceAll(text, wxT("HIDDEN@\\2\\3"));
    printf("text now contains %u hidden addresses", count);
```

Include files

<wx/regex.h>

wxRegEx::wxRegEx

wxRegEx()

Default ctor: use *Compile()* (p. 1109) later.

wxRegEx(**const wxString&** *expr*, **int** *flags = wxRE_DEFAULT*)

Create and compile the regular expression, use *IsValid* (p. 1109) to test for compilation errors.

wxRegEx::~wxRegEx

~wxRegEx()

dtor not virtual, don't derive from this class

wxRegEx::Compile

bool Compile(**const wxString&** *pattern*, **int** *flags = wxRE_DEFAULT*)

Compile the string into regular expression, return `true` if ok or `false` if string has a syntax error.

wxRegEx::IsValid

bool IsValid() **const**

Return `true` if this is a valid compiled regular expression, `false` otherwise.

wxRegEx::GetMatch

bool GetMatch(**size_t*** *start*, **size_t*** *len*, **size_t** *index = 0*) **const**

Get the start index and the length of the match of the expression (if *index* is 0) or a bracketed subexpression (*index* different from 0).

May only be called after successful call to *Matches()* (p. 1110) and only if `wxRE_NOSUB` was **not** used in *Compile()* (p. 1109).

Returns `false` if no match or if an error occurred.

wxString GetMatch(**const wxString&** *text*, **size_t** *index = 0*) **const**

Returns the part of string corresponding to the match where *index* is interpreted as above. Empty string is returned if match failed

May only be called after successful call to *Matches()* (p. 1110) and only if `wxRE_NOSUB` was **not** used in *Compile()* (p. 1109).

wxRegEx::GetMatchCount

size_t GetMatchCount() **const**

Returns the size of the array of matches, i.e. the number of bracketed subexpressions

plus one for the expression itself, or 0 on error.

May only be called after successful call to *Compile()* (p. 1109). and only if `wxRE_NOSUB` was **not** used.

wxRegEx::Matches

bool Matches(**const wxChar*** *text*, **int** *flags = 0*) **const**

Matches the precompiled regular expression against the string *text*, returns `true` if matches and `false` otherwise.

Flags may be combination of `wxRE_NOTBOL` and `wxRE_NOTEOL`.

May only be called after successful call to *Compile()* (p. 1109).

wxRegEx::Replace

int Replace(**wxString*** *text*, **const wxString&** *replacement*, **size_t** *maxMatches = 0*) **const**

Replaces the current regular expression in the string pointed to by *text*, with the text in *replacement* and return number of matches replaced (maybe 0 if none found) or -1 on error.

The replacement text may contain back references `\number` which will be replaced with the value of the corresponding subexpression in the pattern match. `\0` corresponds to the entire match and `&` is a synonym for it. Backslash may be used to quote itself or `&` character.

maxMatches may be used to limit the number of replacements made, setting it to 1, for example, will only replace first occurrence (if any) of the pattern in the text while default value of 0 means replace all.

wxRegEx::ReplaceAll

int ReplaceAll(**wxString*** *text*, **const wxString&** *replacement*) **const**

Replace all occurrences: this is actually a synonym for *Replace()* (p. 1110).

See also

ReplaceFirst (p. 1110)

wxRegEx::ReplaceFirst

int ReplaceFirst(**wxString*** *text*, **const wxString&** *replacement*) **const**

Replace the first occurrence.

See also

Replace (p. 1110)

wxRegion

A wxRegion represents a simple or complex region on a device context or window. It uses reference counting, so copying and assignment operations are fast.

Derived from

wxGDIObject (p. 607)
wxObject (p. 1012)

Include files

<wx/region.h>

See also

wxRegionIterator (p. 1115)

wxRegion::wxRegion

wxRegion()

Default constructor.

wxRegion(**wxCoord** *x*, **wxCoord** *y*, **wxCoord** *width*, **wxCoord** *height*)

Constructs a rectangular region with the given position and size.

wxRegion(**const wxPoint&** *topLeft*, **const wxPoint&** *bottomRight*)

Constructs a rectangular region from the top left point and the bottom right point.

wxRegion(**const wxRect&** *rect*)

Constructs a rectangular region a wxRect object.

wxRegion(**const wxRegion&** *region*)

Constructs a region by copying another region.

wxRegion(**size_t** *n*, **const wxPoint** **points*, **int** *fillStyle = wxWINDING_RULE*)

Constructs a region corresponding to the polygon made of *n* points in the provided array. *fillStyle* parameter may have values `wxWINDING_RULE` or `wxODDEVEN_RULE`.

NB: This constructor is only implemented for Win32 and GTK+ wxWidgets ports.

wxRegion(**const wxBitmap&** *bmp*)

wxRegion(**const wxBitmap&** *bmp*, **const wxColour&** *transColour*, **int** *tolerance = 0*)

Constructs a region using the non-transparent pixels of a bitmap. See *Union* (p. 1114) for more details.

wxRegion::~wxRegion

~wxRegion()

Destructor.

wxRegion::Clear

void Clear()

Clears the current region.

wxRegion::Contains

wxRegionContain Contains(**long&** *x*, **long&** *y*) **const**

Returns a value indicating whether the given point is contained within the region.

wxRegionContain Contains(**const wxPoint&** *pt*) **const**

Returns a value indicating whether the given point is contained within the region.

wxRegionContain Contains(**long&** *x*, **long&** *y*, **long&** *width*, **long&** *height*) **const**

Returns a value indicating whether the given rectangle is contained within the region.

wxRegionContain Contains(**const wxRect&** *rect*) **const**

Returns a value indicating whether the given rectangle is contained within the region.

Return value

The return value is one of wxOutRegion, wxPartRegion and wxInRegion.

On Windows, only wxOutRegion and wxInRegion are returned; a value wxInRegion then indicates that all or some part of the region is contained in this region.

wxRegion::ConvertToBitmap

wxBitmap ConvertToBitmap() **const**

Convert the region to a black and white bitmap with the white pixels being inside the region.

wxRegion::GetBox

void GetBox(**wxCoord&** *x*, **wxCoord&** *y*, **wxCoord&** *width*, **wxCoord&** *height*) **const**

Returns the outer bounds of the region.

wxRect GetBox() **const**

Returns the outer bounds of the region.

wxRegion::Intersect

bool Intersect(**wxCoord** *x*, **wxCoord** *y*, **wxCoord** *width*, **wxCoord** *height*)

Finds the intersection of this region and another, rectangular region, specified using position and size.

bool Intersect(**const wxRect&** *rect*)

Finds the intersection of this region and another, rectangular region.

bool Intersect(**const wxRegion&** *region*)

Finds the intersection of this region and another region.

Return value

true if successful, false otherwise.

Remarks

Creates the intersection of the two regions, that is, the parts which are in both regions. The result is stored in this region.

wxRegion::IsEmpty

bool IsEmpty() **const**

Returns true if the region is empty, false otherwise.

wxRegion::Subtract

bool Subtract(**const wxRect&** *rect*)

Subtracts a rectangular region from this region.

bool Subtract(**const wxRegion&** *region*)

Subtracts a region from this region.

Return value

`true` if successful, `false` otherwise.

Remarks

This operation combines the parts of 'this' region that are not part of the second region. The result is stored in this region.

wxRegion::Offset

bool Offset(**wxCoord** *x*, **wxCoord** *y*)

Moves the region by the specified offsets in horizontal and vertical directions.

Return value

`true` if successful, `false` otherwise (the region is unchanged then).

wxRegion::Union

bool Union(**wxCoord** *x*, **wxCoord** *y*, **wxCoord** *width*, **wxCoord** *height*)

Finds the union of this region and another, rectangular region, specified using position and size.

bool Union(**const wxRect&** *rect*)

Finds the union of this region and another, rectangular region.

bool Union(**const wxRegion&** *region*)

Finds the union of this region and another region.

bool Union(**const wxBitmap&** *bmp*)

Finds the union of this region and the non-transparent pixels of a bitmap. Bitmap's mask is used to determine transparency. If the bitmap doesn't have a mask, solid rectangle of bitmap's dimensions is used.

bool Union(**const wxBitmap&** *bmp*,   **const wxColour&** *transColour*,   **int** *tolerance = 0*)

Finds the union of this region and the non-transparent pixels of a bitmap. Colour to be treated as transparent is specified in the*transColour* argument, along with an optional colour tolerance value.

Return value

true if successful, false otherwise.

Remarks

This operation creates a region that combines all of this region and the second region. The result is stored in this region.

wxRegion::Xor

bool Xor(**wxCoord** *x*, **wxCoord** *y*, **wxCoord** *width*, **wxCoord** *height*)

Finds the Xor of this region and another, rectangular region, specified using position and size.

bool Xor(**const wxRect&** *rect*)

Finds the Xor of this region and another, rectangular region.

bool Xor(**const wxRegion&** *region*)

Finds the Xor of this region and another region.

Return value

true if successful, false otherwise.

Remarks

This operation creates a region that combines all of this region and the second region, except for any overlapping areas. The result is stored in this region.

wxRegion::operator =

void operator =(**const wxRegion&** *region*)

Copies *region* by reference counting.

wxRegionIterator

This class is used to iterate through the rectangles in a region, typically when examining the damaged regions of a window within an OnPaint call.

To use it, construct an iterator object on the stack and loop through the regions, testing the object and incrementing the iterator at the end of the loop.

See *wxPaintEvent* (p. 1025) for an example of use.

Derived from

wxObject (p. 1012)

Include files

<wx/region.h>

See also

wxPaintEvent (p. 1025)

wxRegionIterator::wxRegionIterator

wxRegionIterator()

Default constructor.

wxRegionIterator(**const wxRegion&** *region*)

Creates an iterator object given a region.

wxRegionIterator::GetX

wxCoord GetX() **const**

Returns the x value for the current region.

wxRegionIterator::GetY

wxCoord GetY() **const**

Returns the y value for the current region.

wxRegionIterator::GetW

wxCoord GetW() **const**

An alias for GetWidth.

wxRegionIterator::GetHeight

wxCoord GetHeight() **const**

Returns the height value for the current region.

wxRegionIterator::GetH

wxCoord GetH() **const**

An alias for GetHeight.

wxRegionIterator::GetRect

wxRect GetRect() **const**

Returns the current rectangle.

wxRegionIterator::GetWidth

wxCoord GetWidth() **const**

Returns the width value for the current region.

wxRegionIterator::HaveRects

bool HaveRects() **const**

Returns `true` if there are still some rectangles; otherwise returns `false`.

wxRegionIterator::Reset

void Reset()

Resets the iterator to the beginning of the rectangles.

void Reset(**const wxRegion&** *region*)

Resets the iterator to the given region.

wxRegionIterator::operator ++

void operator ++()

Increment operator. Increments the iterator to the next region.

wxPython note: A wxPython alias for this operator is called `Next`.

wxRegionIterator::operator bool

operator bool() **const**

Returns `true` if there are still some rectangles; otherwise returns `false`.

You can use this to test the iterator object as if it were of type bool.

wxRegKey

wxRegKey is a class representing the Windows registry. One can create, query and delete registry keys using this class.

The Windows registry is easy to understand. There are five registry keys, namely:

1. HKEY_CLASSES_ROOT (HKCR)

2. HKEY_CURRENT_USER (HKCU)

3. HKEY_LOCAL_MACHINE (HKLM)

4. HKEY_CURRENT_CONFIG (HKCC)

5. HKEY_USERS (HKU)

After creating a key, it can hold a value. The values can be:

1. String Value

2. Binary Value

3. DWORD Value

4. Multi String Value

5. Expandable String Value

Derived from

None

Include files

<wx/config.h>

Example

```
wxRegKey *pRegKey = new wxRegKey("HKEY_LOCAL_MACHINE\\Software\\MyKey");

//will create the Key if it does not exist
if( !pRegKey->Exists() )
    pRegKey->Create();

//will create a new value MYVALUE and set it to 12
pRegKey->SetValue("MYVALUE",12);

//Query for the Value and Retrieve it
long lMyVal;
wxString strTemp;
pRegKey->QueryValue("MYVALUE",&lMyVal);
strTemp.Printf("%d",lMyVal);
wxMessageBox(strTemp,"Registry Value",0,this);

//Retrive the number of SubKeys and enumerate them
size_t nSubKeys;
pRegKey->GetKeyInfo(&nSubKeys,NULL,NULL,NULL);
```

```
pRegKey->GetFirstKey(strTemp,1);
for(int i=0;i<nSubKeys;i++)
{
    wxMessageBox(strTemp,"SubKey Name",0,this);
    pRegKey->GetNextKey(strTemp,1);
}
```

wxRegKey::wxRegKey

wxRegKey()

The Constructor to set to HKCR

wxRegKey(**const wxString&** *strKey*)

The constructor to set the full name of the key.

wxRegKey(**const wxRegKey&** *keyParent*, **const wxString&** *strKey*)

The constructor to set the full name of the key under a previously created parent.

wxRegKey::Close

void Close()

Closes the key.

wxRegKey::Create

bool Create(**bool** *bOkIfExists = true*)

Creates the key. Will fail if the key already exists and *bOkIfExists* is false.

wxRegKey::DeleteSelf

void DeleteSelf()

Deletes this key and all of its subkeys and values recursively.

wxRegKey::DeleteKey

void DeleteKey(**const wxChar** **szKey*)

Deletes the subkey with all of its subkeys/values recursively.

wxRegKey::DeleteValue

void DeleteValue(**const wxChar** **szKey*)

Deletes the named value.

wxRegKey::Exists

static bool Exists() **const**

Returns true if the key exists.

wxRegKey::GetName

wxString GetName(**bool** *bShortPrefix* = *true*) **const**

Gets the name of the registry key.

wxRegKey::GetFirstKey

bool GetKeyValue(**wxString&** *strKeyName*, **long&** *lIndex*)

Gets the first key.

wxRegKey::GetFirstValue

bool GetFirstValue(**wxString&** *strValueName*, **long&** *lIndex*)

Gets the first value of this key.

wxRegKey::GetKeyInfo

bool Exists(**size_t** **pnSubKeys*, **size_t** **pnValues*, **size_t** **pnMaxValueLen*) **const**

Gets information about the key.

Parameters

pnSubKeys

> The number of subkeys.

pnMaxKeyLen

> The maximum length of the subkey name.

pnValues

The number of values.

wxRegKey::GetNextKey

bool GetNextKey(**wxString&** *strKeyName*, **long&** *lIndex*) **const**

Gets the next key.

wxRegKey::GetNextValue

bool GetNextValue(**wxString&** *strValueName*, **long&** *lIndex*) **const**

Gets the next key value for this key.

wxRegKey::HasValue

bool HasValue(**const wxChar ***szValue*) **const**

Returns true if the value exists.

wxRegKey::HasValues

bool HasValues() **const**

Returns true if any values exist.

wxRegKey::HasSubKey

bool HasSubKey(**const wxChar ***szKey*) **const**

Returns true if given subkey exists.

wxRegKey::HasSubKeys

bool HasSubKeys() **const**

Returns true if any subkeys exist.

wxRegKey::IsEmpty

bool IsEmpty() **const**

Returns true if this key is empty, nothing under this key.

wxRegKey::IsOpened

bool IsOpened() **const**

Returns true if the key is opened.

wxRegKey::Open

bool Open()

Explicitly opens the key to be opened.

wxRegKey::QueryValue

bool QueryValue(**const wxChar ***szValue*, **wxString&** *strValue*) **const**

Retrieves the string value.

bool QueryValue(**const wxChar ***szValue*, **long ***plValue*) **const**

Retrieves the numeric value.

wxRegKey::Rename

bool Rename(**const wxChar *** *szNewName*)

Renames the key.

wxRegKey::RenameValue

bool RenameValue(**const wxChar ***szValueOld*, **const wxChar ***szValueNew*)

Renames a value.

wxRegKey::SetValue

bool SetValue(**const wxChar ***szValue*, **long** *lValue*)

Sets the numeric value.

wxRendererNative

First, a brief introduction to wxRenderer and why it is needed.

Usually wxWidgets uses the underlying low level GUI system to draw all the controls - this is what we mean when we say that it is a "native" framework. However not all

controls exist under all (or even any) platforms and in this case wxWidgets provides a default, generic, implementation of them written in wxWidgets itself.

These controls don't have the native appearance if only the standard line drawing and other graphics primitives are used, because the native appearance is different under different platforms while the lines are always drawn in the same way.

This is why we have renderers: wxRenderer is a class which virtualizes the drawing, i.e. it abstracts the drawing operations and allows you to draw say, a button, without caring about exactly how this is done. Of course, as we can draw the button differently in different renderers, this also allows us to emulate the native look and feel.

So the renderers work by exposing a large set of high-level drawing functions which are used by the generic controls. There is always a default global renderer but it may be changed or extended by the user, see *Render sample* (p. 1713).

All drawing functions take some standard parameters:

- *win* is the window being drawn. It is normally not used and when it is it should only be used as a generic *wxWindow* (p. 1490) (in order to get its low level handle, for example), but you should*not* assume that it is of some given type as the same renderer function may be reused for drawing different kinds of control.

- *dc* is the *wxDC* (p. 365) to draw on. Only this device context should be used for drawing. It is not necessary to restore pens and brushes for it on function exit but, on the other hand, you shouldn't assume that it is in any specific state on function entry: the rendering functions should always prepare it.

- *rect* the bounding rectangle for the element to be drawn.

- *flags* the optional flags (none by default) which can be a combination of the wxCONTROL_XXX constants below.

Constants

The following rendering flags are defined:

```
enum
{
    wxCONTROL_DISABLED   = 0x00000001,  // control is disabled
    wxCONTROL_FOCUSED    = 0x00000002,  // currently has keyboard focus
    wxCONTROL_PRESSED    = 0x00000004,  // (button) is pressed
    wxCONTROL_ISDEFAULT  = 0x00000008,  // only applies to the buttons
    wxCONTROL_ISSUBMENU  = wxCONTROL_ISDEFAULT, // only for menu items
    wxCONTROL_EXPANDED   = wxCONTROL_ISDEFAULT, // only for the tree items
    wxCONTROL_CURRENT    = 0x00000010,  // mouse is currently over the
control
    wxCONTROL_SELECTED   = 0x00000020,  // selected item in e.g. listbox
    wxCONTROL_CHECKED    = 0x00000040,  // (check/radio button) is checked
    wxCONTROL_CHECKABLE  = 0x00000080   // (menu) item can be checked
};
```

Derived from

No base class

Include files

<wx/renderer.h>

wxRendererNative::~wxRendererNative

 ~wxRendererNative()

Virtual destructor as for any base class.

wxRendererNative::DrawComboBoxDropButton

void DrawComboBoxDropButton(**wxWindow ****win*, **wxDC&** *dc*, **const wxRect&** *rect*, **int** *flags*)

Draw a button like the one used by *wxComboBox* (p. 172) to show a drop down window. The usual appearance is a downwards pointing arrow.

flags may have the wxCONTROL_PRESSED or wxCONTROL_CURRENT bit set.

wxRendererNative::DrawDropArrow

void DrawDropArrow(**wxWindow ****win*, **wxDC&** *dc*, **const wxRect&** *rect*, **int** *flags*)

Draw a drop down arrow that is suitable for use outside a combo box. Arrow will have transparent background.

rect is not entirely filled by the arrow. Instead, you should use bounding rectangle of a drop down button which arrow matches the size you need.*flags* may have the wxCONTROL_PRESSED or wxCONTROL_CURRENT bit set.

wxRendererNative::DrawHeaderButton

void DrawHeaderButton(**wxWindow*** *win*, **wxDC&** *dc*, **const wxRect&** *rect*, **int** *flags = 0*)

Draw the header control button (used by *wxListCtrl* (p. 852)).

wxRendererNative::DrawSplitterBorder

void DrawSplitterBorder(**wxWindow*** *win*, **wxDC&** *dc*, **const wxRect&** *rect*, **int** *flags = 0*)

Draw the border for sash window: this border must be such that the sash drawn by *DrawSash* (p. 1125) blends into it well.

wxRendererNative::DrawSplitterSash

void DrawSplitterSash(**wxWindow*** *win*, **wxDC&** *dc*, **const wxSize&** *size*, **wxCoord** *position*, **wxOrientation** *orient*, **int** *flags = 0*)

Draw a sash. The *orient* parameter defines whether the sash should be vertical or horizontal and how the *position* should be interpreted.

wxRendererNative::DrawTreeItemButton

void DrawTreeItemButton(**wxWindow*** *win*, **wxDC&** *dc*, **const wxRect&** *rect*, **int** *flags = 0*)

Draw the expanded/collapsed icon for a tree control item. To draw an expanded button the *flags* parameter must contain `wxCONTROL_EXPANDED` bit.

wxRendererNative::Get

wxRendererNative& Get()

Return the currently used renderer.

wxRendererNative::GetDefault

wxRendererNative& GetDefault()

Return the default (native) implementation for this platform -- this is also the one used by default but this may be changed by calling *Set* (p. 1126) in which case the return value of this method may be different from the return value of *Get* (p. 1125).

wxRendererNative::GetGeneric

wxRendererNative& GetGeneric()

Return the generic implementation of the renderer. Under some platforms, this is the default renderer implementation, others have platform-specific default renderer which can be retrieved by calling *GetDefault* (p. 1125).

wxRendererNative::GetSplitterParams

wxSplitterRenderParams GetSplitterParams(**const wxWindow*** *win*)

Get the splitter parameters, see *wxSplitterRenderParams* (p. 1250).

wxRendererNative::GetVersion

wxRendererVersion GetVersion() **const**

This function is used for version checking: *Load* (p. 1126) refuses to load any shared libraries implementing an older or incompatible version.

The implementation of this method is always the same in all renderers (simply construct *wxRendererVersion* (p. 1126) using the `wxRendererVersion::Current_XXX` values), but it has to be in the derived, not base, class, to detect mismatches between the renderers versions and so you have to implement it anew in all renderers.

wxRendererNative::Load

wxRendererNative* Load(**const wxString&** *name*)

Load the renderer from the specified DLL, the returned pointer must be deleted by caller if not `NULL` when it is not used any more.

The *name* should be just the base name of the renderer and not the full name of the DLL file which is constructed differently (using *wxDynamicLibrary::CanonicalizePluginName* (p. 475)) on different systems.

wxRendererNative::Set

wxRendererNative* Set(**wxRendererNative*** *renderer*)

Set the renderer to use, passing `NULL` reverts to using the default renderer (the global renderer must always exist).

Return the previous renderer used with Set() or `NULL` if none.

wxRendererVersion

This simple struct represents the *wxRendererNative* (p. 1122) interface version and is only used as the return value of *wxRendererNative::GetVersion* (p. 1125).

The version has two components: the version itself and the age. If the main program and the renderer have different versions they are never compatible with each other because the version is only changed when an existing virtual function is modified or removed. The age, on the other hand, is incremented each time a new virtual method is added and so, at least for the compilers using a common C++ object model, the calling program is compatible with any renderer which has the age greater or equal to its age. This verification is done by *IsCompatible* (p. 1127) method.

Derived from

No base class

Include files

<wx/renderer.h>

wxRendererVersion::IsCompatible

static bool IsCompatible(**const wxRendererVersion&** *ver*)

Checks if the main program is compatible with the renderer having the version *ver*, returns `true` if it is and `false` otherwise.

This method is used by *wxRendererNative::Load* (p. 1126) to determine whether a renderer can be used.

wxRendererVersion::version

const int version

The version component.

wxRendererVersion::age

const int age

The age component.

wxSashEvent

A sash event is sent when the sash of a *wxSashWindow* (p. 1132) has been dragged by the user.

Derived from

wxCommandEvent (p. 180)
wxEvent (p. 483)
wxObject (p. 1012)

Include files

<wx/sashwin.h>

Event table macros

To process an activate event, use these event handler macros to direct input to a member function that takes a wxSashEvent argument.

EVT_SASH_DRAGGED(id, func)	Process a wxEVT_SASH_DRAGGED event, when the user has finished dragging a sash.

EVT_SASH_DRAGGED_RANGE(id1, id2, func)   Process a
wxEVT_SASH_DRAGGED_RANGE event,
when the user has finished dragging a sash.
The event handler is called when windows with
ids in the given range have their sashes
dragged.

Data structures

```
enum wxSashDragStatus
{
    wxSASH_STATUS_OK,
    wxSASH_STATUS_OUT_OF_RANGE
};
```

Remarks

When a sash belonging to a sash window is dragged by the user, and then released, this
event is sent to the window, where it may be processed by an event table entry in a
derived class, a plug-in event handler or an ancestor class.

Note that the wxSashWindow doesn't change the window's size itself. It relies on the
application's event handler to do that. This is because the application may have to
handle other consequences of the resize, or it may wish to veto it altogether. The event
handler should look at the drag rectangle: see *wxSashEvent::GetDragRect* (p. 1129) to
see what the new size of the window would be if the resize were to be applied. It should
also call *wxSashEvent::GetDragStatus* (p. 1129) to see whether the drag was OK or out
of the current allowed range.

See also

wxSashWindow (p. 1132), *Event handling overview* (p. 1751)

wxSashEvent::wxSashEvent

 wxSashEvent(**int** *id = 0*, **wxSashEdgePosition** *edge = wxSASH_NONE*)

Constructor.

wxSashEvent::GetEdge

wxSashEdgePosition GetEdge() **const**

Returns the dragged edge. The return value is one of wxSASH_TOP, wxSASH_RIGHT,
wxSASH_BOTTOM, wxSASH_LEFT.

wxSashEvent::GetDragRect

wxRect GetDragRect() **const**

Returns the rectangle representing the new size the window would be if the resize was applied. It is up to the application to set the window size if required.

wxSashEvent::GetDragStatus

wxSashDragStatus GetDragStatus() **const**

Returns the status of the sash: one of wxSASH_STATUS_OK, wxSASH_STATUS_OUT_OF_RANGE. If the drag caused the notional bounding box of the window to flip over, for example, the drag will be out of rage.

wxSashLayoutWindow

wxSashLayoutWindow responds to OnCalculateLayout events generated by *wxLayoutAlgorithm* (p. 835). It allows the application to use simple accessors to specify how the window should be laid out, rather than having to respond to events. The fact that the class derives from wxSashWindow allows sashes to be used if required, to allow the windows to be user-resizable.

The documentation for *wxLayoutAlgorithm* (p. 835) explains the purpose of this class in more detail.

Derived from

wxSashWindow (p. 1132)
wxWindow (p. 1490)
wxEvtHandler (p. 486)
wxObject (p. 1012)

Include files

<wx/laywin.h>

Window styles

See *wxSashWindow* (p. 1132).

Event handling

This class handles the EVT_QUERY_LAYOUT_INFO and EVT_CALCULATE_LAYOUT events for you. However, if you use sashes, see *wxSashWindow* (p. 1132) for relevant event information.

See also *wxLayoutAlgorithm* (p. 835) for information about the layout events.

See also

wxLayoutAlgorithm (p. 835), *wxSashWindow* (p. 1132), *Event handling overview* (p. 1751)

wxSashLayoutWindow::wxSashLayoutWindow

wxSashLayoutWindow()

Default constructor.

wxSashLayoutWindow(**wxSashLayoutWindow*** *parent*, **wxWindowID** *id*, **const wxPoint&** *pos* = *wxDefaultPosition*, **const wxSize&** *size* = *wxDefaultSize*, **long** *style* = *wxCLIP_CHILDREN | wxSW_3D*, **const wxString&** *name* = *"layoutWindow"*)

Constructs a sash layout window, which can be a child of a frame, dialog or any other non-control window.

Parameters

parent

> Pointer to a parent window.

id

> Window identifier. If -1, will automatically create an identifier.

pos

> Window position. wxDefaultPosition is (-1, -1) which indicates that wxSashLayoutWindows should generate a default position for the window. If using the wxSashLayoutWindow class directly, supply an actual position.

size

> Window size. wxDefaultSize is (-1, -1) which indicates that wxSashLayoutWindows should generate a default size for the window.

style

> Window style. For window styles, please see *wxSashLayoutWindow* (p. 1129).

name

> Window name.

wxSashLayoutWindow::Create

bool Create(**wxSashLayoutWindow*** *parent*, **wxWindowID** *id*, **const wxPoint&** *pos* = *wxDefaultPosition*, **const wxSize&** *size* = *wxDefaultSize*, **long** *style* = *wxCLIP_CHILDREN | wxSW_3D*, **const wxString&** *name* = *"layoutWindow"*)

Initializes a sash layout window, which can be a child of a frame, dialog or any other non-control window.

Parameters

parent

> Pointer to a parent window.

id

> Window identifier. If -1, will automatically create an identifier.

pos

> Window position. wxDefaultPosition is (-1, -1) which indicates that wxSashLayoutWindows should generate a default position for the window. If using the wxSashLayoutWindow class directly, supply an actual position.

size

> Window size. wxDefaultSize is (-1, -1) which indicates that wxSashLayoutWindows should generate a default size for the window.

style

> Window style. For window styles, please see *wxSashLayoutWindow* (p. 1129).

name

> Window name.

wxSashLayoutWindow::GetAlignment

wxLayoutAlignment GetAlignment() **const**

Returns the alignment of the window: one of wxLAYOUT_TOP, wxLAYOUT_LEFT, wxLAYOUT_RIGHT, wxLAYOUT_BOTTOM.

wxSashLayoutWindow::GetOrientation

wxLayoutOrientation GetOrientation() **const**

Returns the orientation of the window: one of wxLAYOUT_HORIZONTAL, wxLAYOUT_VERTICAL.

wxSashLayoutWindow::OnCalculateLayout

void OnCalculateLayout(**wxCalculateLayoutEvent&** *event*)

The default handler for the event that is generated by wxLayoutAlgorithm. The

implementation of this function calls wxCalculateLayoutEvent::SetRect to shrink the provided size according to how much space this window takes up. For further details, see *wxLayoutAlgorithm* (p. 835) and *wxCalculateLayoutEvent* (p. 120).

wxSashLayoutWindow::OnQueryLayoutInfo

void OnQueryLayoutInfo(**wxQueryLayoutInfoEvent&** *event*)

The default handler for the event that is generated by OnCalculateLayout to get size, alignment and orientation information for the window. The implementation of this function uses member variables as set by accessors called by the application. For further details, see *wxLayoutAlgorithm* (p. 835) and *wxQueryLayoutInfoEvent* (p. 1087).

wxSashLayoutWindow::SetAlignment

void SetAlignment(**wxLayoutAlignment** *alignment*)

Sets the alignment of the window (which edge of the available parent client area the window is attached to). *alignment* is one of wxLAYOUT_TOP, wxLAYOUT_LEFT, wxLAYOUT_RIGHT, wxLAYOUT_BOTTOM.

wxSashLayoutWindow::SetDefaultSize

void SetDefaultSize(**const wxSize&** *size*)

Sets the default dimensions of the window. The dimension other than the orientation will be fixed to this value, and the orientation dimension will be ignored and the window stretched to fit the available space.

wxSashLayoutWindow::SetOrientation

void SetOrientation(**wxLayoutOrientation** *orientation*)

Sets the orientation of the window (the direction the window will stretch in, to fill the available parent client area). *orientation* is one of wxLAYOUT_HORIZONTAL, wxLAYOUT_VERTICAL.

wxSashWindow

wxSashWindow allows any of its edges to have a sash which can be dragged to resize the window. The actual content window will be created by the application as a child of wxSashWindow. The window (or an ancestor) will be notified of a drag via a *wxSashEvent* (p. 1127) notification.

Derived from

wxWindow (p. 1490)
wxEvtHandler (p. 486)
wxObject (p. 1012)

Include files

<wx/sashwin.h>

Window styles

The following styles apply in addition to the normal wxWindow styles.

wxSW_3D	Draws a 3D effect sash and border.
wxSW_3DSASH	Draws a 3D effect sash.
wxSW_3DBORDER	Draws a 3D effect border.
wxSW_BORDER	Draws a thin black border.

See also *window styles overview* (p. 1762).

Event handling

EVT_SASH_DRAGGED(id, func)	Process a wxEVT_SASH_DRAGGED event, when the user has finished dragging a sash.
EVT_SASH_DRAGGED_RANGE(id1, id2, func)	Process a wxEVT_SASH_DRAGGED_RANGE event, when the user has finished dragging a sash. The event handler is called when windows with ids in the given range have their sashes dragged.

Data types

```
enum wxSashEdgePosition {
    wxSASH_TOP = 0,
    wxSASH_RIGHT,
    wxSASH_BOTTOM,
    wxSASH_LEFT,
    wxSASH_NONE = 100
};
```

See also

wxSashEvent (p. 1127), *wxSashLayoutWindow* (p. 1129), *Event handling overview* (p. 1751)

wxSashWindow::wxSashWindow

wxSashWindow()

Default constructor.

wxSashWindow(**wxWindow*** *parent*, **wxWindowID** *id*, **const wxPoint&** *pos* = *wxDefaultPosition*, **const wxSize&** *size* = *wxDefaultSize*, **long** *style* = *wxCLIP_CHILDREN | wxSW_3D*, **const wxString&** *name* = *"sashWindow"*)

Constructs a sash window, which can be a child of a frame, dialog or any other non-control window.

Parameters

parent

> Pointer to a parent window.

id

> Window identifier. If -1, will automatically create an identifier.

pos

> Window position. wxDefaultPosition is (-1, -1) which indicates that wxSashWindows should generate a default position for the window. If using the wxSashWindow class directly, supply an actual position.

size

> Window size. wxDefaultSize is (-1, -1) which indicates that wxSashWindows should generate a default size for the window.

style

> Window style. For window styles, please see *wxSashWindow* (p. 1132).

name

> Window name.

wxSashWindow::~wxSashWindow

~wxSashWindow()

Destructor.

wxSashWindow::GetSashVisible

bool GetSashVisible(**wxSashEdgePosition** *edge*) **const**

Returns true if a sash is visible on the given edge, false otherwise.

Parameters

edge

>   Edge. One of wxSASH_TOP, wxSASH_RIGHT, wxSASH_BOTTOM,
>   wxSASH_LEFT.

See also

wxSashWindow::SetSashVisible (p. 1136)

wxSashWindow::GetMaximumSizeX

int GetMaximumSizeX() **const**

Gets the maximum window size in the x direction.

wxSashWindow::GetMaximumSizeY

int GetMaximumSizeY() **const**

Gets the maximum window size in the y direction.

wxSashWindow::GetMinimumSizeX

int GetMinimumSizeX()

Gets the minimum window size in the x direction.

wxSashWindow::GetMinimumSizeY

int GetMinimumSizeY() **const**

Gets the minimum window size in the y direction.

wxSashWindow::HasBorder

bool HasBorder(**wxSashEdgePosition** *edge*) **const**

Returns true if the sash has a border, false otherwise.

Parameters

edge

>   Edge. One of wxSASH_TOP, wxSASH_RIGHT, wxSASH_BOTTOM,
>   wxSASH_LEFT.

See also

wxSashWindow::SetMaximumSizeX

void SetMaximumSizeX(**int** *min*)

Sets the maximum window size in the x direction.

wxSashWindow::SetMaximumSizeY

void SetMaximumSizeY(**int** *min*)

Sets the maximum window size in the y direction.

wxSashWindow::SetMinimumSizeX

void SetMinimumSizeX(**int** *min*)

Sets the minimum window size in the x direction.

wxSashWindow::SetMinimumSizeY

void SetMinimumSizeY(**int** *min*)

Sets the minimum window size in the y direction.

wxSashWindow::SetSashVisible

void SetSashVisible(**wxSashEdgePosition** *edge*, **bool** *visible*)

Call this function to make a sash visible or invisible on a particular edge.

Parameters

edge

> Edge to change. One of wxSASH_TOP, wxSASH_RIGHT, wxSASH_BOTTOM, wxSASH_LEFT.

visible

> true to make the sash visible, false to make it invisible.

See also

wxSashWindow::SetSashBorder

void SetSashBorder(**wxSashEdgePosition** *edge*, **bool** *hasBorder*)

Call this function to give the sash a border, or remove the border.

Parameters

edge

> Edge to change. One of wxSASH_TOP, wxSASH_RIGHT, wxSASH_BOTTOM, wxSASH_LEFT.

hasBorder

> true to give the sash a border visible, false to remove it.

See also

wxSashWindow::HasBorder (p. 1135)

wxScopedArray

This is a simple scoped smart pointer array implementation that is similar to the Boost (`http://www.boost.org`) smart pointers but rewritten to use macros instead.

Example

Below is an example of using a wxWidgets scoped smart pointer and pointer array.

```
class MyClass { /* ... */ };

// declare a smart pointer to a MyClass called wxMyClassPtr
wxDECLARE_SCOPED_PTR(MyClass, wxMyClassPtr)
// declare a smart pointer to an array of chars
wxDECLARE_SCOPED_ARRAY(char, wxCharArray)

...

// define the first pointer class, must be complete
wxDEFINE_SCOPED_PTR(MyClass, wxMyClassPtr)
// define the second pointer class
wxDEFINE_SCOPED_ARRAY(char, wxCharArray)

// create an object with a new pointer to MyClass
wxMyClassPtr theObj(new MyClass());
// reset the pointer (deletes the previous one)
theObj.reset(new MyClass());

// access the pointer
theObj->MyFunc();

// create an object with a new array of chars
wxCharArray theCharObj(new char[100]);

// access the array
theCharObj[0] = "!";
```

Declaring new smart pointer types

```
wxDECLAR_SCOPED_ARRAY( TYPE,      // type of the values
                        CLASSNAME ); // name of the class
```

A smart pointer holds a pointer to an object (which must be complete when wxDEFINE_SCOPED_ARRAY() is called). The memory used by the object is deleted when the smart pointer goes out of scope. The first argument of the macro is the pointer type, the second is the name of the new smart pointer class being created. Below we will use wxScopedArray to represent the scoped pointer array class, but the user may create the class with any legal name.

Include files

<wx/ptr_scpd.h>

See also

wxScopedPtr (p. 1139)

wxScopedArray::wxScopedArray

wxScopedArray(**type** * *T* = *NULL*)

Creates the smart pointer with the given pointer or none if NULL. On compilers that support it, this uses the explicit keyword.

wxScopedArray::reset

reset(**T** *p* * = *NULL*)

Deletes the currently held pointer and sets it to 'p' or to NULL if no arguments are specified. This function does check to make sure that the pointer you are assigning is not the same pointer that is already stored.

wxScopedArray::operator []

const T & operator [](**long int***i*)

This operator acts like the standard [] indexing operator for C++ arrays. The function does not do bounds checking.

wxScopedArray::get

const T* get()

This operator gets the pointer stored in the smart pointer or returns NULL if there is none.

wxScopedArray::swap

swap(**wxScopedPtr** *& ot*)

Swap the pointer inside the smart pointer with 'ot'. The pointer being swapped must be of the same type (hence the same class name).

wxScopedPtr

This is a simple scoped smart pointer implementation that is similar to the Boost (http://www.boost.org/) smart pointers but rewritten to use macros instead.

A smart pointer holds a pointer to an object. The memory used by the object is deleted when the smart pointer goes out of scope. This class is different from the std::auto_ptr<> in so far as it doesn't provide copy constructor nor assignment operator. This limits what you can do with it but is much less surprizing than the "destructive copy" behaviour of the standard class.

Example

Below is an example of using a wxWidgets scoped smart pointer and pointer array.

```
class MyClass { /* ... */ };

// declare a smart pointer to a MyClass called wxMyClassPtr
wxDECLARE_SCOPED_PTR(MyClass, wxMyClassPtr)
// declare a smart pointer to an array of chars
wxDECLARE_SCOPED_ARRAY(char, wxCharArray)

...

// define the first pointer class, must be complete
wxDEFINE_SCOPED_PTR(MyClass, wxMyClassPtr)
// define the second pointer class
wxDEFINE_SCOPED_ARRAY(char, wxCharArray)

// create an object with a new pointer to MyClass
wxMyClassPtr theObj(new MyClass());
// reset the pointer (deletes the previous one)
theObj.reset(new MyClass());

// access the pointer
theObj->MyFunc();

// create an object with a new array of chars
wxCharArray theCharObj(new char[100]);

// access the array
theCharObj[0] = "!";
```

Declaring new smart pointer types

To declare the smart pointer class `CLASSNAME` containing pointes to a (possibly incomplete) type `TYPE` you should use

```
wxDECLARE_SCOPED_PTR( TYPE,      // type of the values
                          CLASSNAME ); // name of the class
```

And later, when `TYPE` is fully defined, you must also use

```
wxDEFINE_SCOPED_PTR( TYPE, CLASSNAME );
```

to implement the scoped pointer class.

The first argument of these macro is the pointer type, the second is the name of the new smart pointer class being created. Below we will use wxScopedPtr to represent the scoped pointer class, but the user may create the class with any legal name.

Alternatively, if you don't have to separate the point of declaration and definition of this class and if you accept the standard naming convention, that is that the scoped pointer for the class `Foo` is called `FooPtr`, you can use a single macro which replaces two macros above:

```
wxDEFINE_SCOPED_PTR_TYPE( TYPE );
```

Once again, in this cass `CLASSNAME` will be `TYPEPtr`.

Include files

<wx/ptr_scpd.h>

See also

wxScopedArray (p. 1137)

wxScopedPtr::wxScopedPtr

 explicit wxScopedPtr(**type** * *T = NULL*)

Creates the smart pointer with the given pointer or none if `NULL`. On compilers that support it, this uses the explicit keyword.

wxScopedPtr::~wxScopedPtr

 ~wxScopedPtr()

Destructor frees the pointer help by this object if it is not `NULL`.

wxScopedPtr::release

T * release()

Returns the currently hold pointer and resets the smart pointer object to `NULL`. After a call to this function the caller is responsible for deleting the pointer.

wxScopedPtr::reset

 reset(**T** *p* * = *NULL*)

Deletes the currently held pointer and sets it to *p* or to NULL if no arguments are specified. This function does check to make sure that the pointer you are assigning is not the same pointer that is already stored.

wxScopedPtr::operator *

const T& operator *()

This operator works like the standard C++ pointer operator to return the object being pointed to by the pointer. If the pointer is NULL or invalid this will crash.

wxScopedPtr::operator ->

const T* operator ->() This operator works like the standard C++ pointer operator to return the pointer in the smart pointer or NULL if it is empty.

wxScopedPtr::get

const T* get()

This operator gets the pointer stored in the smart pointer or returns NULL if there is none.

wxScopedPtr::swap

 swap(**wxScopedPtr** *& other*)

Swap the pointer inside the smart pointer with *other*. The pointer being swapped must be of the same type (hence the same class name).

wxScopedTiedPtr

This is a variation on the topic of *wxScopedPtr* (p. 1139). This class is also a smart pointer but in addition it "ties" the pointer value to another variable. In other words, during the life time of this class the value of that variable is set to be the same as the value of the pointer itself and it is reset to its old value when the object is destroyed. This

class is especially useful when converting the existing code (which may already store the pointers value in some variable) to the smart pointers.

Example

Derives from

wxScopedPtr (p. 1139)

Include files

<wx/ptr_scpd.h>

wxScopedTiedPtr::wxScopedTiedPtr

wxScopedTiedPtr(**T** **ppTie, **T** *ptr)

Constructor creates a smart pointer initialized with *ptr* and stores *ptr* in the location specified by *ppTie* which must not be NULL.

wxScopedTiedPtr::~wxScopedTiedPtr

~wxScopedTiedPtr()

Destructor frees the pointer help by this object and restores the value stored at the tied location (as specified in the *constructor* (p. 1142)) to the old value.

Warning: this location may now contain an uninitialized value if it hadn't been initialized previously, in particular don't count on it magically being NULL!

wxScreenDC

A wxScreenDC can be used to paint on the screen. This should normally be constructed as a temporary stack object; don't store a wxScreenDC object.

Derived from

wxDC (p. 365)

Include files

<wx/dcscreen.h>

See also

wxDC (p. 365), *wxMemoryDC* (p. 935), *wxPaintDC* (p. 1024), *wxClientDC* (p. 147), *wxWindowDC* (p. 1547)

wxScreenDC::wxScreenDC

wxScreenDC()

Constructor.

wxScreenDC::StartDrawingOnTop

bool StartDrawingOnTop(**wxWindow*** *window*)

bool StartDrawingOnTop(**wxRect*** *rect = NULL*)

Use this in conjunction with *EndDrawingOnTop* (p. 1143) to ensure that drawing to the screen occurs on top of existing windows. Without this, some window systems (such as X) only allow drawing to take place underneath other windows.

By using the first form of this function, an application is specifying that the area that will be drawn on coincides with the given window.

By using the second form, an application can specify an area of the screen which is to be drawn on. If NULL is passed, the whole screen is available.

It is recommended that an area of the screen is specified because with large regions, flickering effects are noticeable when destroying the temporary transparent window used to implement this feature.

You might use this pair of functions when implementing a drag feature, for example as in the *wxSplitterWindow* (p. 1239) implementation.

Remarks

This function is probably obsolete since the X implementations allow drawing directly on the screen now. However, the fact that this function allows the screen to be refreshed afterwards, may be useful to some applications.

wxScreenDC::EndDrawingOnTop

bool EndDrawingOnTop()

Use this in conjunction with *StartDrawingOnTop* (p. 1143).

This function destroys the temporary window created to implement on-top drawing (X only).

wxScrollBar

A wxScrollBar is a control that represents a horizontal or vertical scrollbar. It is distinct from the two scrollbars that some windows provide automatically, but the two types of scrollbar share the way events are received.

Derived from

wxControl (p. 215)
wxWindow (p. 1490)
wxEvtHandler (p. 486)
wxObject (p. 1012)

Include files

<wx/scrolbar.h>

Remarks

A scrollbar has the following main attributes: *range*, *thumb size*, *page size*, and *position*.

The range is the total number of units associated with the view represented by the scrollbar. For a table with 15 columns, the range would be 15.

The thumb size is the number of units that are currently visible. For the table example, the window might be sized so that only 5 columns are currently visible, in which case the application would set the thumb size to 5. When the thumb size becomes the same as or greater than the range, the scrollbar will be automatically hidden on most platforms.

The page size is the number of units that the scrollbar should scroll by, when 'paging' through the data. This value is normally the same as the thumb size length, because it is natural to assume that the visible window size defines a page.

The scrollbar position is the current thumb position.

Most applications will find it convenient to provide a function called **AdjustScrollbars** which can be called initially, from an **OnSize** event handler, and whenever the application data changes in size. It will adjust the view, object and page size according to the size of the window and the size of the data.

Window styles

wxSB_HORIZONTAL   Specifies a horizontal scrollbar.

wxSB_VERTICAL   Specifies a vertical scrollbar.

See also *window styles overview* (p. 1762).

Event table macros

To process a scroll event, use these event handler macros to direct input to member functions that take a wxScrollEvent argument. You can use EVT_COMMAND_SCROLL... macros with window IDs for when intercepting scroll events from controls, or EVT_SCROLL... macros without window IDs for intercepting scroll events from the receiving window - except for this, the macros behave exactly the same.

EVT_SCROLL(func)	Process all scroll events.
EVT_SCROLL_TOP(func)	Process wxEVT_SCROLL_TOP scroll-to-top events (minimum position).
EVT_SCROLL_BOTTOM(func)	Process wxEVT_SCROLL_TOP scroll-to-bottom events (maximum position).
EVT_SCROLL_LINEUP(func)	Process wxEVT_SCROLL_LINEUP line up events.
EVT_SCROLL_LINEDOWN(func)	Process wxEVT_SCROLL_LINEDOWN line down events.
EVT_SCROLL_PAGEUP(func)	Process wxEVT_SCROLL_PAGEUP page up events.
EVT_SCROLL_PAGEDOWN(func)	Process wxEVT_SCROLL_PAGEDOWN page down events.
EVT_SCROLL_THUMBTRACK(func)	Process wxEVT_SCROLL_THUMBTRACK thumbtrack events (frequent events sent as the user drags the thumbtrack).
EVT_SCROLL_THUMBRELEASE(func)	Process wxEVT_SCROLL_THUMBRELEASE thumb release events.
EVT_SCROLL_ENDSCROLL(func)	Process wxEVT_SCROLL_ENDSCROLL end of scrolling events (MSW only).
EVT_COMMAND_SCROLL(id, func)	Process all scroll events.
EVT_COMMAND_SCROLL_TOP(id, func)	Process wxEVT_SCROLL_TOP scroll-to-top events (minimum position).
EVT_COMMAND_SCROLL_BOTTOM(id, func)	Process wxEVT_SCROLL_TOP scroll-to-bottom events (maximum position).
EVT_COMMAND_SCROLL_LINEUP(id, func)	Process wxEVT_SCROLL_LINEUP line up events.
EVT_COMMAND_SCROLL_LINEDOWN(id, func)	Process wxEVT_SCROLL_LINEDOWN line

down events.

EVT_COMMAND_SCROLL_PAGEUP(id, func)	Process wxEVT_SCROLL_PAGEUP page up events.
EVT_COMMAND_SCROLL_PAGEDOWN(id, func)	Process wxEVT_SCROLL_PAGEDOWN page down events.
EVT_COMMAND_SCROLL_THUMBTRACK(id, func)	Process wxEVT_SCROLL_THUMBTRACK thumbtrack events (frequent events sent as the user drags the thumbtrack).
EVT_COMMAND_SCROLL_THUMBRELEASE(func)	Process wxEVT_SCROLL_THUMBRELEASE thumb release events.
EVT_COMMAND_SCROLL_ENDSCROLL(func)	Process wxEVT_SCROLL_ENDSCROLL end of scrolling events (MSW only).

The difference between `EVT_SCROLL_THUMBRELEASE` and `EVT_SCROLL_ENDSCROLL`

The `EVT_SCROLL_THUMBRELEASE` event is only emitted when actually dragging the thumb using the mouse and releasing it (This `EVT_SCROLL_THUMBRELEASE`event is also followed by an `EVT_SCROLL_ENDSCROLL` event).

The `EVT_SCROLL_ENDSCROLL` event also occurs when using the keyboard to change the thumb position, and when clicking next to the thumb (In all these cases the `EVT_SCROLL_THUMBRELEASE` event does not happen).

In short, the `EVT_SCROLL_ENDSCROLL` event is triggered when scrolling/ moving has finished. The only exception (unfortunately) is that changing the thumb position using the mousewheel does give a `EVT_SCROLL_THUMBRELEASE`event but NOT an `EVT_SCROLL_ENDSCROLL` event.

Please see the widgets sample ("Slider" page) to see the difference between `EVT_SCROLL_THUMBRELEASE` and `EVT_SCROLL_ENDSCROLL` in action.

See also

Scrolling overview (p. 1786), *Event handling overview* (p. 1751), *wxScrolledWindow* (p. 1150)

wxScrollBar::wxScrollBar

wxScrollBar()

Default constructor.

wxScrollBar(**wxWindow*** *parent*, **wxWindowID** *id*, **const wxPoint&** *pos* = *wxDefaultPosition*, **const wxSize&** *size* = *wxDefaultSize*, **long** *style* = *wxSB_HORIZONTAL*, **const wxValidator&** *validator* = *wxDefaultValidator*, **const wxString&** *name* = *"scrollBar"*)

Constructor, creating and showing a scrollbar.

Parameters

parent

> Parent window. Must be non-NULL.

id

> Window identifier. A value of -1 indicates a default value.

pos

> Window position. If the position (-1, -1) is specified then a default position is chosen.

size

> Window size. If the default size (-1, -1) is specified then a default size is chosen.

style

> Window style. See *wxScrollBar* (p. 1143).

validator

> Window validator.

name

> Window name.

See also

wxScrollBar::Create (p. 1147), *wxValidator* (p. 1461)

wxScrollBar::~wxScrollBar

void ~wxScrollBar()

Destructor, destroying the scrollbar.

wxScrollBar::Create

bool Create(**wxWindow*** *parent*, **wxWindowID** *id*, **const wxPoint&** *pos* = *wxDefaultPosition*, **const wxSize&** *size* = *wxDefaultSize*, **long** *style* = *wxSB_HORIZONTAL*, **const wxValidator&** *validator* = *wxDefaultValidator*, **const wxString&** *name* = *"scrollBar"*)

Scrollbar creation function called by the scrollbar constructor. See *wxScrollBar::wxScrollBar* (p. 1146) for details.

wxScrollBar::GetRange

int GetRange() **const**

Returns the length of the scrollbar.

See also

wxScrollBar::SetScrollbar (p. 1149)

wxScrollBar::GetPageSize

int GetPageSize() **const**

Returns the page size of the scrollbar. This is the number of scroll units that will be scrolled when the user pages up or down. Often it is the same as the thumb size.

See also

wxScrollBar::SetScrollbar (p. 1149)

wxScrollBar::GetThumbPosition

int GetThumbPosition() **const**

Returns the current position of the scrollbar thumb.

See also

wxScrollBar::SetThumbPosition (p. 1149)

wxScrollBar::GetThumbSize

int GetThumbSize() **const**

Returns the thumb or 'view' size.

See also

wxScrollBar::SetScrollbar (p. 1149)

wxScrollBar::SetThumbPosition

void SetThumbPosition(**int** *viewStart*)

Sets the position of the scrollbar.

Parameters

viewStart

> The position of the scrollbar thumb.

See also

wxScrollBar::GetThumbPosition (p. 1148)

wxScrollBar::SetScrollbar

virtual void SetScrollbar(**int** *position*, **int** *thumbSize*, **int** *range*, **int** *pageSize*, **const bool** *refresh* = *true*)

Sets the scrollbar properties.

Parameters

position

> The position of the scrollbar in scroll units.

thumbSize

> The size of the thumb, or visible portion of the scrollbar, in scroll units.

range

> The maximum position of the scrollbar.

pageSize

> The size of the page size in scroll units. This is the number of units the scrollbar will scroll when it is paged up or down. Often it is the same as the thumb size.

refresh

> true to redraw the scrollbar, false otherwise.

Remarks

Let's say you wish to display 50 lines of text, using the same font. The window is sized so that you can only see 16 lines at a time.

You would use:

```
scrollbar->SetScrollbar(0, 16, 50, 15);
```

The page size is 1 less than the thumb size so that the last line of the previous page will be visible on the next page, to help orient the user.

Note that with the window at this size, the thumb position can never go above 50 minus 16, or 34.

You can determine how many lines are currently visible by dividing the current view size by the character height in pixels.

When defining your own scrollbar behaviour, you will always need to recalculate the scrollbar settings when the window size changes. You could therefore put your scrollbar calculations and SetScrollbar call into a function named AdjustScrollbars, which can be called initially and also from a *wxSizeEvent* (p. 1175) event handler function.

See also

Scrolling overview (p. 1786), *wxWindow::SetScrollbar* (p. 1536), *wxScrolledWindow* (p. 1150)

wxScrolledWindow

The wxScrolledWindow class manages scrolling for its client area, transforming the coordinates according to the scrollbar positions, and setting the scroll positions, thumb sizes and ranges according to the area in view.

Starting from version 2.4 of wxWidgets, there are several ways to use a wxScrolledWindow. In particular, there are now three ways to set the size of the scrolling area:

One way is to set the scrollbars directly using a call to*wxScrolledWindow::SetScrollbars* (p. 1157). This is the way it used to be in any previous version of wxWidgets and it will be kept for backwards compatibility.

An additional method of manual control, which requires a little less computation of your own, is to set the total size of the scrolling area by calling either *wxWindow::SetVirtualSize* (p. 1542), or *wxWindow::FitInside* (p. 1502), and setting the scrolling increments for it by calling *wxScrolledWindow::SetScrollRate* (p. 1159). Scrolling in some orientation is enabled by setting a non zero increment for it.

The most automatic and newest way is to simply let sizers determine the scrolling area. This is now the default when you set an interior sizer into a wxScrolledWindow with *wxWindow::SetSizer* (p. 1540). The scrolling area will be set to the size requested by the sizer and the scrollbars will be assigned for each orientation according to the need for them and the scrolling increment set by *wxScrolledWindow::SetScrollRate* (p. 1159). As above, scrolling is only enabled in orientations with a non-zero increment. You can influence the minimum size of the scrolled area controlled by a sizer by calling*wxWindow::SetVirtualSizeHints* (p. 1542). (calling *wxScrolledWindow::SetScrollbars* (p. 1157) has analogous effects in wxWidgets 2.4 -- in

later versions it may not continue to override the sizer)

Note: if Maximum size hints are still supported by SetVirtualSizeHints, use them at your own dire risk. They may or may not have been removed for 2.4, but it really only makes sense to set minimum size hints here. We should probably replace SetVirtualSizeHints with SetMinVirtualSize or similar and remove it entirely in future.

As with all windows, an application can draw onto a wxScrolledWindow using a *device context* (p. 1791).

You have the option of handling the OnPaint handler or overriding the *OnDraw* (p. 1157) function, which is passed a pre-scrolled device context (prepared by *DoPrepareDC* (p. 1156)).

If you don't wish to calculate your own scrolling, you must call DoPrepareDC when not drawing from within OnDraw, to set the device origin for the device context according to the current scroll position.

A wxScrolledWindow will normally scroll itself and therefore its child windows as well. It might however be desired to scroll a different window than itself: e.g. when designing a spreadsheet, you will normally only have to scroll the (usually white) cell area, whereas the (usually grey) label area will scroll very differently. For this special purpose, you can call *SetTargetWindow* (p. 1159) which means that pressing the scrollbars will scroll a different window.

Note that the underlying system knows nothing about scrolling coordinates, so that all system functions (mouse events, expose events, refresh calls etc) as well as the position of subwindows are relative to the "physical" origin of the scrolled window. If the user insert a child window at position (10,10) and scrolls the window down 100 pixels (moving the child window out of the visible area), the child window will report a position of (10,-90).

Derived from

wxPanel (p. 1030)
wxWindow (p. 1490)
wxEvtHandler (p. 486)
wxObject (p. 1012)

Include files

<wx/scrolwin.h>

Window styles

wxRETAINED   Uses a backing pixmap to speed refreshes. Motif only.

See also *window styles overview* (p. 1762).

Remarks

Use wxScrolledWindow for applications where the user scrolls by a fixed amount, and where a 'page' can be interpreted to be the current visible portion of the window. For more sophisticated applications, use the wxScrolledWindow implementation as a guide

to build your own scroll behaviour.

See also

wxScrollBar (p. 1143), *wxClientDC* (p. 147),
wxPaintDC (p. 1024), *wxVScrolledWindow* (p. 1484)

wxScrolledWindow::wxScrolledWindow

wxScrolledWindow()

Default constructor.

wxScrolledWindow(**wxWindow*** *parent*, **wxWindowID** *id = -1*, **const wxPoint&** *pos = wxDefaultPosition*, **const wxSize&** *size = wxDefaultSize*, **long** *style = wxHSCROLL | wxVSCROLL*, **const wxString&** *name = "scrolledWindow"*)

Constructor.

Parameters

parent

> Parent window.

id

> Window identifier. A value of -1 indicates a default value.

pos

> Window position. If a position of (-1, -1) is specified then a default position is chosen.

size

> Window size. If a size of (-1, -1) is specified then the window is sized appropriately.

style

> Window style. See *wxScrolledWindow* (p. 1150).

name

> Window name.

Remarks

The window is initially created without visible scrollbars. Call *wxScrolledWindow::SetScrollbars* (p. 1157) to specify how big the virtual window size should be.

wxScrolledWindow::~wxScrolledWindow

~wxScrolledWindow()

Destructor.

wxScrolledWindow::CalcScrolledPosition

void CalcScrolledPosition(**int** *x*, **int** *y*, **int ****xx* **int ****yy*) **const**

Translates the logical coordinates to the device ones. For example, if a window is scrolled 10 pixels to the bottom, the device coordinates of the origin are (0, 0) (as always), but the logical coordinates are (0, 10) and so the call to CalcScrolledPosition(0, 10, &xx, &yy) will return 0 in yy.

See also

CalcUnscrolledPosition (p. 1153)

wxPython note: The wxPython version of this methods accepts only two parameters and returns xx and yy as a tuple of values.

wxPerl note: In wxPerl this method takes two parameters and returns a 2-element list (xx, yy).

wxScrolledWindow::CalcUnscrolledPosition

void CalcUnscrolledPosition(**int** *x*, **int** *y*, **int ****xx* **int ****yy*) **const**

Translates the device coordinates to the logical ones. For example, if a window is scrolled 10 pixels to the bottom, the device coordinates of the origin are (0, 0) (as always), but the logical coordinates are (0, 10) and so the call to CalcUnscrolledPosition(0, 0, &xx, &yy) will return 10 in yy.

See also

CalcScrolledPosition (p. 1153)

wxPython note: The wxPython version of this methods accepts only two parameters and returns xx and yy as a tuple of values.

wxPerl note: In wxPerl this method takes two parameters and returns a 2-element list (xx, yy).

wxScrolledWindow::Create

bool Create(**wxWindow*** *parent*, **wxWindowID** *id = -1*, **const wxPoint&** *pos = wxDefaultPosition*, **const wxSize&** *size = wxDefaultSize*, **long** *style = wxHSCROLL | wxVSCROLL*, **const wxString&** *name = "scrolledWindow"*)

Creates the window for two-step construction. Derived classes should call or replace this function. See *wxScrolledWindow::wxScrolledWindow* (p. 1152) for details.

wxScrolledWindow::EnableScrolling

void EnableScrolling(**const bool** *xScrolling*, **const bool** *yScrolling*)

Enable or disable physical scrolling in the given direction. Physical scrolling is the physical transfer of bits up or down the screen when a scroll event occurs. If the application scrolls by a variable amount (e.g. if there are different font sizes) then physical scrolling will not work, and you should switch it off. Note that you will have to reposition child windows yourself, if physical scrolling is disabled.

Parameters

xScrolling

 If true, enables physical scrolling in the x direction.

yScrolling

 If true, enables physical scrolling in the y direction.

Remarks

Physical scrolling may not be available on all platforms. Where it is available, it is enabled by default.

wxScrolledWindow::GetScrollPixelsPerUnit

void GetScrollPixelsPerUnit(**int*** *xUnit*, **int*** *yUnit*) **const**

Get the number of pixels per scroll unit (line), in each direction, as set by *wxScrolledWindow::SetScrollbars* (p. 1157). A value of zero indicates no scrolling in that direction.

Parameters

xUnit

 Receives the number of pixels per horizontal unit.

yUnit

 Receives the number of pixels per vertical unit.

See also

wxScrolledWindow::SetScrollbars (p. 1157), *wxScrolledWindow::GetVirtualSize* (p. 1155)

wxPython note: The wxPython version of this methods accepts no parameters and

returns a tuple of values for xUnit and yUnit.

wxPerl note: In wxPerl this method takes no parameters and returns a 2-element list (
`xUnit, yUnit )`.

wxScrolledWindow::GetViewStart

void GetViewStart(**int*** *x*, **int*** *y*) **const**

Get the position at which the visible portion of the window starts.

Parameters

x

> Receives the first visible x position in scroll units.

y

> Receives the first visible y position in scroll units.

Remarks

If either of the scrollbars is not at the home position, *x* and/or *y* will be greater than zero. Combined with *wxWindow::GetClientSize* (p. 1506), the application can use this function to efficiently redraw only the visible portion of the window. The positions are in logical scroll units, not pixels, so to convert to pixels you will have to multiply by the number of pixels per scroll increment.

See also

wxScrolledWindow::SetScrollbars (p. 1157)

wxPython note: The wxPython version of this methods accepts no parameters and returns a tuple of values for x and y.

wxPerl note: In wxPerl this method takes no parameters and returns a 2-element list (
`x, y )`.

wxScrolledWindow::GetVirtualSize

void GetVirtualSize(**int*** *x*, **int*** *y*) **const**

Gets the size in device units of the scrollable window area (as opposed to the client size, which is the area of the window currently visible).

Parameters

x

> Receives the length of the scrollable window, in pixels.

y

Receives the height of the scrollable window, in pixels.

Remarks

Use *wxDC::DeviceToLogicalX* (p. 369) and *wxDC::DeviceToLogicalY* (p. 370) to translate these units to logical units.

See also

wxScrolledWindow::SetScrollbars (p. 1157), *wxScrolledWindow::GetScrollPixelsPerUnit* (p. 1154)

wxPython note: The wxPython version of this methods accepts no parameters and returns a tuple of values for x and y.

wxPerl note: In wxPerl this method takes no parameters and returns a 2-element list (`x, y`).

wxScrolledWindow::IsRetained

bool IsRetained() const

Motif only: true if the window has a backing bitmap.

wxScrolledWindow::DoPrepareDC

void DoPrepareDC(wxDC& *dc*)

Call this function to prepare the device context for drawing a scrolled image. It sets the device origin according to the current scroll position.

DoPrepareDC is called automatically within the default wxScrolledWindow::OnPaint event handler, so your *wxScrolledWindow::OnDraw* (p. 1157) override will be passed a 'pre-scrolled' device context. However, if you wish to draw from outside of OnDraw (via OnPaint), or you wish to implement OnPaint yourself, you must call this function yourself. For example:

```
void MyWindow::OnEvent(wxMouseEvent& event)
{
  wxClientDC dc(this);
  DoPrepareDC(dc);

  dc.SetPen(*wxBLACK_PEN);
  float x, y;
  event.Position(&x, &y);
  if (xpos > -1 && ypos > -1 && event.Dragging())
  {
    dc.DrawLine(xpos, ypos, x, y);
  }
  xpos = x;
  ypos = y;
}
```

wxScrolledWindow::OnDraw

virtual void OnDraw(**wxDC&** *dc*)

Called by the default paint event handler to allow the application to define painting behaviour without having to worry about calling *wxScrolledWindow::DoPrepareDC* (p. 1156).

Instead of overriding this function you may also just process the paint event in the derived class as usual, but then you will have to call DoPrepareDC() yourself.

wxScrolledWindow::PrepareDC

void PrepareDC(**wxDC&** *dc*)

This function is for backwards compatibility only and simply calls *DoPrepareDC* (p. 1156) now. Notice that it is *not* called by the default paint event handle (DoPrepareDC() is), so overriding this method in your derived class is useless.

wxScrolledWindow::Scroll

void Scroll(**int** *x*, **int** *y*)

Scrolls a window so the view start is at the given point.

Parameters

x

      The x position to scroll to, in scroll units.

y

      The y position to scroll to, in scroll units.

Remarks

The positions are in scroll units, not pixels, so to convert to pixels you will have to multiply by the number of pixels per scroll increment. If either parameter is -1, that position will be ignored (no change in that direction).

See also

wxScrolledWindow::SetScrollbars (p. 1157), *wxScrolledWindow::GetScrollPixelsPerUnit* (p. 1154)

wxScrolledWindow::SetScrollbars

void SetScrollbars(**int** *pixelsPerUnitX*, **int** *pixelsPerUnitY*, **int** *noUnitsX*, **int** *noUnitsY*,

int *xPos = 0*, **int** *yPos = 0*, **bool** *noRefresh = false*)

Sets up vertical and/or horizontal scrollbars.

Parameters

pixelsPerUnitX

> Pixels per scroll unit in the horizontal direction.

pixelsPerUnitY

> Pixels per scroll unit in the vertical direction.

noUnitsX

> Number of units in the horizontal direction.

noUnitsY

> Number of units in the vertical direction.

xPos

> Position to initialize the scrollbars in the horizontal direction, in scroll units.

yPos

> Position to initialize the scrollbars in the vertical direction, in scroll units.

noRefresh

> Will not refresh window if true.

Remarks

The first pair of parameters give the number of pixels per 'scroll step', i.e. amount moved when the up or down scroll arrows are pressed. The second pair gives the length of scrollbar in scroll steps, which sets the size of the virtual window.

xPos and *yPos* optionally specify a position to scroll to immediately.

For example, the following gives a window horizontal and vertical scrollbars with 20 pixels per scroll step, and a size of 50 steps (1000 pixels) in each direction.

```
window->SetScrollbars(20, 20, 50, 50);
```

wxScrolledWindow manages the page size itself, using the current client window size as the page size.

Note that for more sophisticated scrolling applications, for example where scroll steps may be variable according to the position in the document, it will be necessary to derive a new class from wxWindow, overriding **OnSize** and adjusting the scrollbars appropriately.

See also

wxWindow::SetVirtualSize (p. 1542)

wxScrolledWindow::SetScrollRate

void SetScrollRate(**int** *xstep,* **int** *ystep*)

Set the horizontal and vertical scrolling increment only. See the pixelsPerUnit parameter in SetScrollbars.

wxScrolledWindow::SetTargetWindow

void SetTargetWindow(**wxWindow*** *window*)

Call this function to tell wxScrolledWindow to perform the actual scrolling on a different window (and not on itself).

wxScrollEvent

A scroll event holds information about events sent from stand-alone*scrollbars* (p. 1143) and *sliders* (p. 1192). Note that starting from wxWidgets 2.1, scrolled windows send the *wxScrollWinEvent* (p. 1162) which does not derive from wxCommandEvent, but from wxEvent directly - don't confuse these two kinds of events and use the event table macros mentioned below only for the scrollbar-like controls.

Derived from

wxCommandEvent (p. 180)
wxEvent (p. 483)
wxObject (p. 1012)

Include files

<wx/event.h>

Event table macros

To process a scroll event, use these event handler macros to direct input to member functions that take a wxScrollEvent argument. You can useEVT_COMMAND_SCROLL... macros with window IDs for when intercepting scroll events from controls, or EVT_SCROLL... macros without window IDs for intercepting scroll events from the receiving window - except for this, the macros behave exactly the same.

EVT_SCROLL(func)	Process all scroll events.
EVT_SCROLL_TOP(func)	Process wxEVT_SCROLL_TOP scroll-to-top events (minimum position).
EVT_SCROLL_BOTTOM(func)	Process wxEVT_SCROLL_TOP

	scroll-to-bottom events (maximum position).
EVT_SCROLL_LINEUP(func)	Process wxEVT_SCROLL_LINEUP line up events.
EVT_SCROLL_LINEDOWN(func)	Process wxEVT_SCROLL_LINEDOWN line down events.
EVT_SCROLL_PAGEUP(func)	Process wxEVT_SCROLL_PAGEUP page up events.
EVT_SCROLL_PAGEDOWN(func)	Process wxEVT_SCROLL_PAGEDOWN page down events.
EVT_SCROLL_THUMBTRACK(func)	Process wxEVT_SCROLL_THUMBTRACK thumbtrack events (frequent events sent as the user drags the thumbtrack).
EVT_SCROLL_THUMBRELEASE(func)	Process wxEVT_SCROLL_THUMBRELEASE thumb release events.
EVT_SCROLL_ENDSCROLL(func)	Process wxEVT_SCROLL_ENDSCROLL end of scrolling events (MSW only).
EVT_COMMAND_SCROLL(id, func)	Process all scroll events.
EVT_COMMAND_SCROLL_TOP(id, func)	Process wxEVT_SCROLL_TOP scroll-to-top events (minimum position).
EVT_COMMAND_SCROLL_BOTTOM(id, func)	Process wxEVT_SCROLL_TOP scroll-to-bottom events (maximum position).
EVT_COMMAND_SCROLL_LINEUP(id, func)	Process wxEVT_SCROLL_LINEUP line up events.
EVT_COMMAND_SCROLL_LINEDOWN(id, func)	Process wxEVT_SCROLL_LINEDOWN line down events.
EVT_COMMAND_SCROLL_PAGEUP(id, func)	Process wxEVT_SCROLL_PAGEUP page up events.
EVT_COMMAND_SCROLL_PAGEDOWN(id, func)	Process

	wxEVT_SCROLL_PAGEDOWN page down events.
EVT_COMMAND_SCROLL_THUMBTRACK(id, func)	Process wxEVT_SCROLL_THUMBTRACK thumbtrack events (frequent events sent as the user drags the thumbtrack).
EVT_COMMAND_SCROLL_THUMBRELEASE(func)	Process wxEVT_SCROLL_THUMBRELEASE thumb release events.
EVT_COMMAND_SCROLL_ENDSCROLL(func)	Process wxEVT_SCROLL_ENDSCROLL end of scrolling events (MSW only).

The difference between `EVT_SCROLL_THUMBRELEASE` and `EVT_SCROLL_ENDSCROLL`

The `EVT_SCROLL_THUMBRELEASE` event is only emitted when actually dragging the thumb using the mouse and releasing it (This `EVT_SCROLL_THUMBRELEASE`event is also followed by an `EVT_SCROLL_ENDSCROLL` event).

The `EVT_SCROLL_ENDSCROLL` event also occurs when using the keyboard to change the thumb position, and when clicking next to the thumb (In all these cases the `EVT_SCROLL_THUMBRELEASE` event does not happen).

In short, the `EVT_SCROLL_ENDSCROLL` event is triggered when scrolling/ moving has finished. The only exception (unfortunately) is that changing the thumb position using the mousewheel does give a `EVT_SCROLL_THUMBRELEASE`event but NOT an `EVT_SCROLL_ENDSCROLL` event.

Please see the widgets sample ("Slider" page) to see the difference between `EVT_SCROLL_THUMBRELEASE` and `EVT_SCROLL_ENDSCROLL` in action.

Remarks

Note that unless specifying a scroll control identifier, you will need to test for scrollbar orientation with *wxScrollEvent::GetOrientation* (p. 1162), since horizontal and vertical scroll events are processed using the same event handler.

See also

wxScrollBar (p. 1143), *wxSlider* (p. 1192), *wxSpinButton* (p. 1227), *wxScrollWinEvent* (p. 1162), *Event handling overview* (p. 1751)

wxScrollEvent::wxScrollEvent

wxScrollEvent(**WXTYPE** *commandType = 0*, **int** *id = 0*, **int** *pos = 0*, **int** *orientation = 0*)

Constructor.

wxScrollEvent::GetOrientation

int GetOrientation() **const**

Returns wxHORIZONTAL or wxVERTICAL, depending on the orientation of the scrollbar.

wxScrollEvent::GetPosition

int GetPosition() **const**

Returns the position of the scrollbar.

wxScrollWinEvent

A scroll event holds information about events sent from scrolling windows.

Derived from

wxEvent (p. 483)
wxObject (p. 1012)

Include files

<wx/event.h>

Event table macros

To process a scroll window event, use these event handler macros to direct input to member functions that take a wxScrollWinEvent argument. You can use the EVT_SCROLLWIN... macros for intercepting scroll window events from the receiving window.

EVT_SCROLLWIN(func)	Process all scroll events.
EVT_SCROLLWIN_TOP(func)	Process wxEVT_SCROLLWIN_TOP scroll-to-top events.
EVT_SCROLLWIN_BOTTOM(func)	Process wxEVT_SCROLLWIN_TOP scroll-to-bottom events.
EVT_SCROLLWIN_LINEUP(func)	Process wxEVT_SCROLLWIN_LINEUP line up events.
EVT_SCROLLWIN_LINEDOWN(func)	Process wxEVT_SCROLLWIN_LINEDOWN line down events.
EVT_SCROLLWIN_PAGEUP(func)	Process wxEVT_SCROLLWIN_PAGEUP page

up events.

EVT_SCROLLWIN_PAGEDOWN(func) Process wxEVT_SCROLLWIN_PAGEDOWN
page down events.

EVT_SCROLLWIN_THUMBTRACK(func) Process
wxEVT_SCROLLWIN_THUMBTRACK
thumbtrack events (frequent events sent as the
user drags the thumbtrack).

EVT_SCROLLWIN_THUMBRELEASE(func)   Process
wxEVT_SCROLLWIN_THUMBRELEASE
thumb release events.

See also
wxScrollEvent (p. 1159), *Event handling overview* (p. 1751)

wxScrollWinEvent::wxScrollWinEvent

 wxScrollWinEvent(**WXTYPE** *commandType = 0*, **int** *pos = 0*, **int** *orientation = 0*)

Constructor.

wxScrollWinEvent::GetOrientation

int GetOrientation() **const**

Returns wxHORIZONTAL or wxVERTICAL, depending on the orientation of the
scrollbar.

wxScrollWinEvent::GetPosition

int GetPosition() **const**

Returns the position of the scrollbar for the thumb track and release events. Note that
this field can't be used for the other events, you need to query the window itself for the
current position in that case.

wxSemaphore

wxSemaphore is a counter limiting the number of threads concurrently accessing a
shared resource. This counter is always between 0 and the maximum value specified
during the semaphore creation. When the counter is strictly greater than 0, a call to *Wait*
(p. 1165) returns immediately and decrements the counter. As soon as it reaches 0, any
subsequent calls to*Wait* (p. 1165) block and only return when the semaphore counter
becomes strictly positive again as the result of calling *Post* (p. 1164) which increments

the counter.

In general, the semaphores are useful to restrict access to a shared resource which can only be accessed by some fixed number of clients at once. For example, when modeling a hotel reservation system a semaphore with the counter equal to the total number of available rooms could be created. Each time a room is reserved, the semaphore should be acquired by calling *Wait* (p. 1165) and each time a room is freed it should be released by calling *Post* (p. 1164).

Derived from

No base class

Include files

<wx/thread.h>

wxSemaphore::wxSemaphore

wxSemaphore(**int** *initialcount = 0*, **int** *maxcount = 0*)

Specifying a *maxcount* of 0 actually makes wxSemaphore behave as if there is no upper limit. If maxcount is 1 the semaphore behaves exactly as a mutex.

initialcount is the initial value of the semaphore which must be between0 and *maxcount* (if it is not set to 0).

wxSemaphore::~wxSemaphore

~wxSemaphore()

Destructor is not virtual, don't use this class polymorphically.

wxSemaphore::Post

wxSemaError Post()

Increments the semaphore count and signals one of the waiting threads in an atomic way. Returns wxSEMA_OVERFLOW if the count would increase the counter past the maximum.

Return value

One of:

wxSEMA_NO_ERROR	There was no error.
wxSEMA_INVALID	Semaphore hasn't been initialized successfully.

wxSEMA_OVERFLOW	Post() would increase counter past the max.
wxSEMA_MISC_ERROR	Miscellaneous error.

wxSemaphore::TryWait

wxSemaError TryWait()

Same as *Wait()* (p. 1165), but returns immediately.

Return value

One of:

wxSEMA_NO_ERROR	There was no error.
wxSEMA_INVALID	Semaphore hasn't been initialized successfully.
wxSEMA_BUSY	Returned by TryWait() if Wait() would block, i.e. the count is zero.
wxSEMA_MISC_ERROR	Miscellaneous error.

wxSemaphore::Wait

wxSemaError Wait()

Wait indefinitely until the semaphore count becomes strictly positive and then decrement it and return.

Return value

One of:

wxSEMA_NO_ERROR	There was no error.
wxSEMA_INVALID	Semaphore hasn't been initialized successfully.
wxSEMA_MISC_ERROR	Miscellaneous error.

wxSemaphore::WaitTimeout

wxSemaError WaitTimeout(**unsigned long***timeout_millis*)

Same as *Wait()* (p. 1165), but with a timeout limit.

Return value

One of:

wxSEMA_NO_ERROR	There was no error.

wxSEMA_INVALID	Semaphore hasn't been initialized successfully.
wxSEMA_TIMEOUT	Timeout occurred without receiving semaphore.
wxSEMA_MISC_ERROR	Miscellaneous error.

wxSetCursorEvent

A SetCursorEvent is generated when the mouse cursor is about to be set as a result of mouse motion. This event gives the application the chance to perform specific mouse cursor processing based on the current position of the mouse within the window. Use *SetCursor* (p. 1167) to specify the cursor you want to be displayed.

Derived from

wxEvent (p. 483)
wxObject (p. 1012)

Include files

<wx/event.h>

See also

::wxSetCursor (p. 1625)*wxWindow::wxSetCursor* (p. 1530)

Event table macros

To process a set cursor event, use this event handler macro to direct input to a member function that takes a wxSetCursorEvent argument.

EVT_SET_CURSOR(func)   Process a wxEVT_SET_CURSOR event.

wxSetCursorEvent::wxSetCursorEvent

 wxSetCursorEvent(**wxCoord***x = 0*, **wxCoord***y = 0*)

Constructor, used by the library itself internally to initialize the event object.

wxSetCursorEvent::GetCursor

wxCursor& GetCursor() **const**

Returns a reference to the cursor specified by this event.

wxSetCursorEvent::GetX

wxCoord GetX() **const**

Returns the X coordinate of the mouse in client coordinates.

wxSetCursorEvent::GetY

wxCoord GetY() **const**

Returns the Y coordinate of the mouse in client coordinates.

wxSetCursorEvent::HasCursor

bool HasCursor() **const**

Returns `true` if the cursor specified by this event is a valid cursor.

Remarks

You cannot specify `wxNullCursor` with this event, as it is not considered a valid cursor.

wxSetCursorEvent::SetCursor

void SetCursor(**const wxCursor&***cursor*)

Sets the cursor associated with this event.

wxServer

A wxServer object represents the server part of a client-server DDE-like (Dynamic Data Exchange) conversation. The actual DDE-based implementation using wxDDEServer is available on Windows only, but a platform-independent, socket-based version of this API is available using wxTCPServer, which has the same API.

To create a server which can communicate with a suitable client, you need to derive a class from wxConnection and another from wxServer. The custom wxConnection class will intercept communications in a 'conversation' with a client, and the custom wxServer is required so that a user-overridden *wxServer::OnAcceptConnection* (p. 1168) member can return a wxConnection of the required class, when a connection is made. Look at the IPC sample and the *Interprocess communications overview* (p. 1841) for an example of how to do this.

Derived from

wxServerBase

Include files

<wx/ipc.h>

See also

wxClient (p. 145),*wxConnection* (p. 387), *IPC overview* (p. 1841)

wxServer::wxServer

 wxServer()

Constructs a server object.

wxServer::Create

bool Create(**const wxString&** *service*)

Registers the server using the given service name. Under Unix, the service name may be either an integer port identifier in which case an Internet domain socket will be used for the communications, or a valid file name (which shouldn't exist and will be deleted afterwards) in which case a Unix domain socket is created. false is returned if the call failed (for example, the port number is already in use).

wxServer::OnAcceptConnection

virtual wxConnectionBase * OnAcceptConnection(**const wxString&** *topic*)

When a client calls **MakeConnection**, the server receives the message and this member is called. The application should derive a member to intercept this message and return a connection object of either the standard wxConnection type, or (more likely) of a user-derived type.

If the topic is **STDIO**, the application may wish to refuse the connection. Under UNIX, when a server is created the OnAcceptConnection message is always sent for standard input and output, but in the context of DDE messages it doesn't make a lot of sense.

wxSimpleHelpProvider

wxSimpleHelpProvider is an implementation of *wxHelpProvider* (p. 703) which supports only plain text help strings, and shows the string associated with the control (if any) in a tooltip.

Derived from

wxHelpProvider (p. 703)

Include files

<wx/cshelp.h>

wxHelpProvider (p. 703), *wxHelpControllerHelpProvider* (p. 701), *wxContextHelp* (p. 212), *wxWindow::SetHelpText* (p. 1535), *wxWindow::GetHelpText* (p. 1509)

wxSingleChoiceDialog

This class represents a dialog that shows a list of strings, and allows the user to select one. Double-clicking on a list item is equivalent to single-clicking and then pressing OK.

Derived from

wxDialog (p. 406)
wxWindow (p. 1490)
wxEvtHandler (p. 486)
wxObject (p. 1012)

Include files

<wx/choicdlg.h>

See also

wxSingleChoiceDialog overview (p. 1801),*wxMultiChoiceDialog* (p. 993)

wxSingleChoiceDialog::wxSingleChoiceDialog

 wxSingleChoiceDialog(**wxWindow*** *parent*, **const wxString&** *message*, **const wxString&** *caption*, **int** *n*, **const wxString*** *choices*, **void**** *clientData* = *NULL*, **long** *style* = *wxCHOICEDLG_STYLE*, **const wxPoint&** *pos* = *wxDefaultPosition*)

 wxSingleChoiceDialog(**wxWindow*** *parent*, **const wxString&** *message*, **const wxString&** *caption*, **const wxArrayString&** *choices*, **void**** *clientData* = *NULL*, **long** *style* = *wxCHOICEDLG_STYLE*, **const wxPoint&** *pos* = *wxDefaultPosition*)

Constructor, taking an array of wxString choices and optional client data.

Parameters

parent

> Parent window.

message

> Message to show on the dialog.

caption

> The dialog caption.

n

> The number of choices.

choices

> An array of strings, or a string list, containing the choices.

clientData

> An array of client data to be associated with the items. See *GetSelectionClientData* (p. 1170).

style

> A dialog style (bitlist) containing flags chosen from standard dialog styles and the following:

wxOK	Show an OK button.
wxCANCEL	Show a Cancel button.
wxCENTRE	Centre the message. Not Windows.

The default value is equivalent to **wxDEFAULT_DIALOG_STYLE | wxRESIZE_BORDER | wxOK | wxCANCEL | wxCENTRE**.

pos

> Dialog position. Not Windows.

Remarks

Use *wxSingleChoiceDialog::ShowModal* (p. 1171) to show the dialog.

wxPython note: For Python the two parameters `n` and `choices` are collapsed into a single parameter `choices` which is expected to be a Python list of strings.

wxPerl note: In wxPerl there is just an array reference in place of `n`and `choices`, and the client data array, if present, must have the same length as the choices array.

wxSingleChoiceDialog::GetSelection

int GetSelection() **const**

Returns the index of selected item.

wxSingleChoiceDialog::GetSelectionClientData

char* GetSelectionClientData() **const**

Returns the client data associated with the selection.

wxSingleChoiceDialog::GetStringSelection

wxString GetStringSelection() **const**

Returns the selected string.

wxSingleChoiceDialog::SetSelection

void SetSelection(**int** *selection*) **const**

Sets the index of the initially selected item.

wxSingleChoiceDialog::ShowModal

int ShowModal()

Shows the dialog, returning either wxID_OK or wxID_CANCEL.

wxSingleInstanceChecker

wxSingleInstanceChecker class allows to check that only a single instance of a program is running. To do it, you should create an object of this class. As long as this object is alive, calls to *IsAnotherRunning()* (p. 1173) from other processes will return true.

As the object should have the life span as big as possible, it makes sense to create it either as a global or in *wxApp::OnInit* (p. 38). For example:

```
bool MyApp::OnInit()
{
    const wxString name = wxString::Format("MyApp-%s",
wxGetUserId().c_str());
    m_checker = new wxSingleInstanceChecker(name);
    if ( m_checker->IsAnotherRunning() )
    {
        wxLogError(_("Another program instance is already running,
aborting."));

        return false;
    }

    ... more initializations ...

    return true;
}

int MyApp::OnExit()
{
    delete m_checker;

    return 0;
}
```

Note using *wxGetUserId()* (p. 1609) to construct the name: this allows different user to

run the application concurrently which is usually the intended goal. If you don't use the user name in the wxSingleInstanceChecker name, only one user would be able to run the application at a time.

This class is implemented for Win32 and Unix platforms (supporting `fcntl()` system call, but almost all of modern Unix systems do) only.

Derived from

No base class

Include files

<wx/snglinst.h>

wxSingleInstanceChecker::wxSingleInstanceChecker

wxSingleInstanceChecker()

Default ctor, use *Create()* (p. 1172) after it.

wxSingleInstanceChecker::wxSingleInstanceChecker

wxSingleInstanceChecker(**const wxString&** *name*, **const wxString&** *path* = *wxEmptyString*)

Like *Create()* (p. 1172) but without error checking.

wxSingleInstanceChecker::Create

bool Create(**const wxString&** *name*, **const wxString&** *path* = *wxEmptyString*)

Initialize the object if it had been created using the default constructor. Note that you can't call Create() more than once, so calling it if the *non default ctor* (p. 1172) had been used is an error.

Parameters

name

> must be given and be as unique as possible. It is used as the mutex name under Win32 and the lock file name under Unix. *GetAppName()* (p. 33) and *wxGetUserId()* (p. 1609) are commonly used to construct this parameter.

path

> is optional and is ignored under Win32 and used as the directory to create the lock file in under Unix (default is *wxGetHomeDir()* (p. 1606))

Return value

Returns `false` if initialization failed, it doesn't mean that another instance is running - use *IsAnotherRunning()* (p. 1173) to check for it.

Note

One of possible reasons while Create may fail on Unix is that the lock file used for checking already exists but was not created by the user. Therefore applications shouldn't treat failure of this function as fatal condition, because doing so would open them to the possibility of a Denial of Service attack. Instead, they should alert the user about the problem and offer to continue execution without checking if another instance is running.

wxSingleInstanceChecker::IsAnotherRunning

bool IsAnotherRunning() **const**

Returns `true` if another copy of this program is already running, `false` otherwise.

wxSingleInstanceChecker::~wxSingleInstanceChecker

~wxSingleInstanceChecker()

Destructor frees the associated resources.

Note that it is not virtual, this class is not meant to be used polymorphically

wxSize

A **wxSize** is a useful data structure for graphics operations. It simply contains integer *width* and *height* members.

wxSize is used throughout wxWidgets as well as wxPoint which, although almost equivalent to wxSize, has a different meaning: wxPoint represents a position while wxSize - the size.

wxPython note: wxPython defines aliases for the `x` and `y` members named `width` and `height` since it makes much more sense for sizes.

Derived from

None

Include files

<wx/gdicmn.h>

See also

wxPoint (p. 1045), *wxRealPoint* (p. 1098)

wxSize::wxSize

wxSize()

wxSize(**int** *width*, **int** *height*)

Creates a size object.

wxSize::DecTo

void DecTo(**const wxSize&** *size*)

Decrements this object so that both of its dimensions are not greater than the corresponding dimensions of the *size*.

See also

IncTo (p. 1174)

wxSize::IsFullySpecified

bool IsFullySpecified() **const**

Returns `true` if neither of the size object components is equal to -1, which is used as default for the size values in wxWidgets (hence the predefined `wxDefaultSize` has both of its components equal to -1).

This method is typically used before calling *SetDefaults* (p. 1175).

wxSize::GetWidth

int GetWidth() **const**

Gets the width member.

wxSize::GetHeight

int GetHeight() **const**

Gets the height member.

wxSize::IncTo

void IncTo(**const wxSize&** *size*)

Increments this object so that both of its dimensions are not less than the corresponding dimensions of the *size*.

See also

DecTo (p. 1174)

wxSize::Set

void Set(**int** *width*, **int** *height*)

Sets the width and height members.

wxSize::SetDefaults

void SetDefaults(**const wxSize&** *sizeDefault*)

```
Combine this size object with another one replacing the default (i.e.
equal to -1) components of this object with those of the other. It is
typically used like this:    if ( !size.IsFullySpecified() )
    {
        size.SetDefaults(GetDefaultSize());
    }
```

See also

IsFullySpecified (p. 1174)

wxSize::SetHeight

void SetHeight(**int** *height*)

Sets the height.

wxSize::SetWidth

void SetWidth(**int** *width*)

Sets the width.

wxSize::operator =

void operator =(**const wxSize&** *sz*)

Assignment operator.

wxSizeEvent

A size event holds information about size change events.

The EVT_SIZE handler function will be called when the window has been resized.

You may wish to use this for frames to resize their child windows as appropriate.

Note that the size passed is of the whole window: call *wxWindow::GetClientSize* (p. 1506) for the area which may be used by the application.

When a window is resized, usually only a small part of the window is damaged and you may only need to repaint that area. However, if your drawing depends on the size of the window, you may need to clear the DC explicitly and repaint the whole window. In which case, you may need to call *wxWindow::Refresh* (p. 1522) to invalidate the entire window.

Derived from

wxEvent (p. 483)
wxObject (p. 1012)

Include files

<wx/event.h>

Event table macros

To process a size event, use this event handler macro to direct input to a member function that takes a wxSizeEvent argument.

EVT_SIZE(func)   Process a wxEVT_SIZE event.

See also
wxSize (p. 1173), *Event handling overview* (p. 1751)

wxSizeEvent::wxSizeEvent

 wxSizeEvent(**const wxSize&** *sz,* **int** *id = 0*)

Constructor.

wxSizeEvent::GetSize

wxSize GetSize() **const**

Returns the entire size of the window generating the size change event.

wxSizer

wxSizer is the abstract base class used for laying out subwindows in a window. You

cannot use wxSizer directly; instead, you will have to use one of the sizer classes derived from it. Currently there are *wxBoxSizer* (p. 102), *wxStaticBoxSizer* (p. 1262),*wxGridSizer* (p. 683) *wxFlexGridSizer* (p. 554) and *wxGridBagSizer* (p. 656).

The layout algorithm used by sizers in wxWidgets is closely related to layout in other GUI toolkits, such as Java's AWT, the GTK toolkit or the Qt toolkit. It is based upon the idea of the individual subwindows reporting their minimal required size and their ability to get stretched if the size of the parent window has changed. This will most often mean that the programmer does not set the original size of a dialog in the beginning, rather the dialog will be assigned a sizer and this sizer will be queried about the recommended size. The sizer in turn will query its children, which can be normal windows, empty space or other sizers, so that a hierarchy of sizers can be constructed. Note that wxSizer does not derive from wxWindow and thus does not interfere with tab ordering and requires very little resources compared to a real window on screen.

What makes sizers so well fitted for use in wxWidgets is the fact that every control reports its own minimal size and the algorithm can handle differences in font sizes or different window (dialog item) sizes on different platforms without problems. If e.g. the standard font as well as the overall design of Motif widgets requires more space than on Windows, the initial dialog size will automatically be bigger on Motif than on Windows.

Sizers may also be used to control the layout of custom drawn items on the window. The Add, Insert, and Prepend functions return a pointer to the newly added wxSizerItem. Just add empty space of the desired size and attributes, and then use the wxSizerItem::GetRect method to determine where the drawing operations should take place.

wxPython note: If you wish to create a sizer class in wxPython you should derive the class from `wxPySizer` in order to get Python-aware capabilities for the various virtual methods.

Derived from

wxObject (p. 1012)
wxClientDataContainer (p. 149)

Include files

<wx/sizer.h>

See also

Sizer overview (p. 1771)

wxSizer::wxSizer

wxSizer()

The constructor. Note that wxSizer is an abstract base class and may not be instantiated.

wxSizer::~wxSizer

~wxSizer()

The destructor.

wxSizer::Add

wxSizerItem* Add(**wxWindow*** *window*, **const wxSizerFlags&** *flags*)

wxSizerItem* Add(**wxWindow*** *window*, **int** *proportion = 0*,**int** *flag = 0*, **int** *border = 0*, **wxObject*** *userData = NULL*)

wxSizerItem* Add(**wxSizer*** *sizer*, **const wxSizerFlags&** *flags*)

wxSizerItem* Add(**wxSizer*** *sizer*, **int** *proportion = 0*, **int** *flag = 0*, **int** *border = 0*, **wxObject*** *userData = NULL*)

wxSizerItem* Add(**int** *width*, **int** *height*, **int** *proportion = 0*, **int** *flag = 0*, **int** *border = 0*, **wxObject*** *userData = NULL*)

Appends a child to the sizer. wxSizer itself is an abstract class, but the parameters are equivalent in the derived classes that you will instantiate to use it so they are described here:

window

> The window to be added to the sizer. Its initial size (either set explicitly by the user or calculated internally when using wxDefaultSize) is interpreted as the minimal and in many cases also the initial size. This is particularly useful in connection with *SetSizeHints* (p. 1185).

sizer

> The (child-)sizer to be added to the sizer. This allows placing a child sizer in a sizer and thus to create hierarchies of sizers (typically a vertical box as the top sizer and several horizontal boxes on the level beneath).

width and height

> The dimension of a spacer to be added to the sizer. Adding spacers to sizers gives more flexibility in the design of dialogs; imagine for example a horizontal box with two buttons at the bottom of a dialog: you might want to insert a space between the two buttons and make that space stretchable using the *proportion* flag and the result will be that the left button will be aligned with the left side of the dialog and the right button with the right side - the space in between will shrink and grow with the dialog.

proportion

> Although the meaning of this parameter is undefined in wxSizer, it is used in wxBoxSizer to indicate if a child of a sizer can change its size in the main orientation of the wxBoxSizer - where 0 stands for not changeable and a value of

more than zero is interpreted relative to the value of other children of the same wxBoxSizer. For example, you might have a horizontal wxBoxSizer with three children, two of which are supposed to change their size with the sizer. Then the two stretchable windows would get a value of 1 each to make them grow and shrink equally with the sizer's horizontal dimension.

flag

This parameter can be used to set a number of flags which can be combined using the binary OR operator |. Two main behaviours are defined using these flags. One is the border around a window: the *border* parameter determines the border width whereas the flags given here determine which side(s) of the item that the border will be added. The other flags determine how the sizer item behaves when the space allotted to the sizer changes, and is somewhat dependent on the specific kind of sizer used.

wxTOP

wxBOTTOM	
wxLEFT	
wxRIGHT	
wxALL	These flags are used to specify which side(s) of the sizer item the *border* width will apply to.
wxEXPAND	The item will be expanded to fill the space assigned to the item.
wxSHAPED	The item will be expanded as much as possible while also maintaining its aspect ratio
wxFIXED_MINSIZE	Normally wxSizers will use *GetAdjustedBestSize* (p. 1503) to determine what the minimal size of window items should be, and will use that size to calculate the layout. This allows layouts to adjust when an item changes and its *best size* becomes different. If you would rather have a window item stay the size it started with then use wxFIXED_MINSIZE.

wxALIGN_CENTER

wxALIGN_LEFT	
wxALIGN_RIGHT	
wxALIGN_TOP	
wxALIGN_BOTTOM	
wxALIGN_CENTER_VERTICAL	
wxALIGN_CENTER_HORIZONTAL	The wxALIGN flags allow you to specify the alignment of the item within the space allotted to it by the sizer, adjusted for the border if any.

border

Determines the border width, if the *flag* parameter is set to include any border flag.

userData

> Allows an extra object to be attached to the sizer item, for use in derived classes when sizing information is more complex than the *proportion* and *flag* will allow for.

flags

> A *wxSizerFlags* (p. 1185) object that enables you to specify most of the above parameters more conveniently.

wxSizer::AddSpacer

wxSizerItem* AddSpacer(**int** *size*)

Adds non-stretchable space to the sizer. More readable way of calling*Add* (p. 1178)(size, size, 0).

wxSizer::AddStretchSpacer

wxSizerItem* AddStretchSpacer(**int** *prop = 1*)

Adds stretchable space to the sizer. More readable way of calling*Add* (p. 1178)(0, 0, prop).

wxSizer::CalcMin

wxSize CalcMin()

This method is abstract and has to be overwritten by any derived class. Here, the sizer will do the actual calculation of its children minimal sizes.

wxSizer::Detach

bool Detach(**wxWindow*** *window*)

bool Detach(**wxSizer*** *sizer*)

bool Detach(**size_t** *index*)

Detach a child from the sizer without destroying it. *window* is the window to be detached, *sizer* is the equivalent sizer and *index* is the position of the child in the sizer, typically 0 for the first item. This method does not cause any layout or resizing to take place, call *wxSizer::Layout* (p. 1182)to update the layout "on screen" after detaching a child from the sizer.

Returns true if the child item was found and detached, false otherwise.

See also

wxSizer::Remove (p. 1183)

wxSizer::Fit

wxSize Fit(**wxWindow*** *window*)

Tell the sizer to resize the *window* to match the sizer's minimal size. This is commonly done in the constructor of the window itself, see sample in the description of *wxBoxSizer* (p. 102). Returns the new size.

For a top level window this is the total window size, not client size.

wxSizer::FitInside

void FitInside(**wxWindow*** *window*)

Tell the sizer to resize the virtual size of the *window* to match the sizer's minimal size. This will not alter the on screen size of the window, but may cause the addition/removal/alteration of scrollbars required to view the virtual area in windows which manage it.

See also

wxScrolledWindow::SetScrollbars (p. 1157)*, wxSizer::SetVirtualSizeHints* (p. 1185)

wxSizer::GetItem

wxSizerItem * GetItem(**wxWindow*** *window*, **bool** *recursive = false*)

wxSizerItem * GetItem(**wxSizer*** *sizer*, **bool** *recursive = false*)

wxSizerItem * GetItem(**size_t** *index*)

Finds item of the sizer which holds given *window*, *sizer* or is located in sizer at position *index*. Use parameter *recursive* to search in subsizers too.

Returns pointer to item or NULL.

wxSizer::GetSize

wxSize GetSize()

Returns the current size of the sizer.

wxSizer::GetPosition

wxPoint GetPosition()

Returns the current position of the sizer.

wxSizer::GetMinSize

wxSize GetMinSize()

Returns the minimal size of the sizer. This is either the combined minimal size of all the children and their borders or the minimal size set by *SetMinSize* (p. 1184), depending on which is bigger.

wxSizer::Insert

wxSizerItem* Insert(**size_t** *index*, **wxWindow*** *window*, **const wxSizerFlags&** *flags*)

wxSizerItem* Insert(**size_t** *index*, **wxWindow*** *window*, **int** *proportion = 0*, **int** *flag = 0*, **int** *border = 0*, **wxObject*** *userData = NULL*)

wxSizerItem* Insert(**size_t** *index*, **wxSizer*** *sizer*, **const wxSizerFlags&** *flags*)

wxSizerItem* Insert(**size_t** *index*, **wxSizer*** *sizer*, **int** *proportion = 0*, **int** *flag = 0*, **int** *border = 0*, **wxObject*** *userData = NULL*)

wxSizerItem* Insert(**size_t** *index*, **int** *width*, **int** *height*, **int** *proportion = 0*, **int** *flag = 0*, **int** *border = 0*, **wxObject*** *userData = NULL*)

Insert a child into the sizer before any existing item at *index*.

index

> The position this child should assume in the sizer.

See *wxSizer::Add* (p. 1178) for the meaning of the other parameters.

wxSizer::InsertSpacer

wxSizerItem* InsertSpacer(**size_t** *index*, **int** *size*)

Inserts non-stretchable space to the sizer. More readable way of calling*Insert* (p. 1182)(size, size, 0).

wxSizer::InsertStretchSpacer

wxSizerItem* InsertStretchSpacer(**size_t** *index*, **int** *prop = 1*)

Inserts stretchable space to the sizer. More readable way of calling*Insert* (p. 1182)(0, 0, prop).

wxSizer::Layout

void Layout()

Call this to force layout of the children anew, e.g. after having added a child to or removed a child (window, other sizer or space) from the sizer while keeping the current dimension.

wxSizer::Prepend

wxSizerItem* Prepend(**wxWindow*** *window*, **const wxSizerFlags&** *flags*)

wxSizerItem* Prepend(**wxWindow*** *window*, **int** *proportion = 0*, **int** *flag = 0*, **int** *border = 0*, **wxObject*** *userData = NULL*)

wxSizerItem* Prepend(**wxSizer*** *sizer*, **const wxSizerFlags&** *flags*)

wxSizerItem* Prepend(**wxSizer*** *sizer*, **int** *proportion = 0*, **int** *flag = 0*, **int** *border = 0*, **wxObject*** *userData = NULL*)

wxSizerItem* Prepend(**int** *width*, **int** *height*, **int** *proportion = 0*, **int** *flag = 0*, **int** *border= 0*, **wxObject*** *userData = NULL*)

Same as *wxSizer::Add* (p. 1178), but prepends the items to the beginning of the list of items (windows, subsizers or spaces) owned by this sizer.

wxSizer::PrependSpacer

wxSizerItem* PrependSpacer(**int** *size*)

Prepends non-stretchable space to the sizer. More readable way of calling*Prepend* (p. 1183)(size, size, 0).

wxSizer::PrependStretchSpacer

wxSizerItem* PrependStretchSpacer(**int** *prop = 1*)

Prepends stretchable space to the sizer. More readable way of calling*Prepend* (p. 1183)(0, 0, prop).

wxSizer::RecalcSizes

void RecalcSizes()

This method is abstract and has to be overwritten by any derived class. Here, the sizer will do the actual calculation of its children's positions and sizes.

wxSizer::Remove

bool Remove(**wxWindow*** *window*)

bool Remove(**wxSizer*** *sizer*)

bool Remove(**size_t** *index*)

Removes a child from the sizer and destroys it. *sizer* is the wxSizer to be removed,*index* is the position of the child in the sizer, typically 0 for the first item. This method does not cause any layout or resizing to take place, call*wxSizer::Layout* (p. 1182) to update the layout "on screen" after removing a child from the sizer.

NB: The method taking a wxWindow* parameter is deprecated. For historical reasons it does not destroy the window as would usually be expected from Remove. You should use*wxSizer::Detach* (p. 1180) in new code instead. There is currently no wxSizer method that will both detach and destroy a wxWindow item.

Returns true if the child item was found and removed, false otherwise.

wxSizer::SetDimension

void SetDimension(**int** *x*, **int** *y*, **int** *width*, **int** *height*)

Call this to force the sizer to take the given dimension and thus force the items owned by the sizer to resize themselves according to the rules defined by the parameter in the *Add* (p. 1178) and *Prepend* (p. 1183) methods.

wxSizer::SetMinSize

void SetMinSize(**int** *width*, **int** *height*)

void SetMinSize(**wxSize** *size*)

Call this to give the sizer a minimal size. Normally, the sizer will calculate its minimal size based purely on how much space its children need. After calling this method *GetMinSize* (p. 1182) will return either the minimal size as requested by its children or the minimal size set here, depending on which is bigger.

wxSizer::SetItemMinSize

void SetItemMinSize(**wxWindow*** *window*, **int** *width*, **int** *height*)

void SetItemMinSize(**wxSizer*** *sizer*, **int** *width*, **int** *height*)

void SetItemMinSize(**size_t** *index*, **int** *width*, **int** *height*)

Set an item's minimum size by window, sizer, or position. The item will be found recursively in the sizer's descendants. This function enables an application to set the size of an item after initial creation.

wxSizer::SetSizeHints

void SetSizeHints(**wxWindow*** *window*)

Tell the sizer to set (and *Fit* (p. 1181)) the minimal size of the *window* to match the sizer's minimal size. This is commonly done in the constructor of the window itself, see sample in the description of *wxBoxSizer* (p. 102) if the window is resizable (as are many dialogs under Unix and frames on probably all platforms).

wxSizer::SetVirtualSizeHints

void SetVirtualSizeHints(**wxWindow*** *window*)

Tell the sizer to set the minimal size of the *window* virtual area to match the sizer's minimal size. For windows with managed scrollbars this will set them appropriately.

See also

wxScrolledWindow::SetScrollbars (p. 1157)

wxSizer::Show

bool Show(**wxWindow*** *window*, **bool** *show = true*, **bool** *recursive = false*)

bool Show(**wxSizer*** *sizer*, **bool** *show = true*, **bool** *recursive = false*)

bool Show(**size_t** *index*, **bool** *show = true*)

Shows or hides the *window*, *sizer*, or item at *index*. To make a sizer item disappear or reappear, use Show() followed by Layout(). Use parameter *recursive* to show or hide elements found in subsizers.

Returns true if the child item was found, false otherwise.

Note that this only works with wxBoxSizer and wxFlexGridSizer, since they are the only two sizer classes that can size rows/columns independently.

wxSizerFlags

PRELIMINARY.

Normally, when you add something to a sizer via *wxSizer::Add* (p. 1178), you have to specify a lot of flags and parameters. This can be unwieldy.

This is where wxSizerFlags comes in. Instead of a bunch of flags and other stuff, you can use wxSizerFlags, which is a convenient class for doing so.

Note that by specification, all methods of wxSizerFlags return the wxSizerFlags object itself to ease the calling of multiple methods at a time.

wxSizerFlags::wxSizerFlags

wxSizerFlags(**int** *proportion = 0*)

Creates the wxSizer with the proportion specified by `proportion`.

wxSizerFlags::Align

wxSizerFlags& Align(**int** *align = 0*)

Sets the alignment of this wxSizerFlags to `align`.

Note that if this method is not called, the wxSizerFlags has no specified alignment.

See also

Left (p. 1187),
Right (p. 1187),
Centre (p. 1186)

wxSizerFlags::Border

wxSizerFlags& Border(**int** *direction*, **int** *borderinpixels*)

Sets the wxSizerFlags to have a border of a number of pixels specified by `borderinpixels` with the directions specified by `direction`.

wxSizerFlags& Border(**int** *direction = wxALL*)

Sets the wxSizerFlags to have a border of a default size with the directions specified by `direction`.

wxSizerFlags::Center

wxSizerFlags& Center()

Sets the object of the wxSizerFlags to center itself in the area it is given.

wxSizerFlags::Centre

wxSizerFlags& Centre()

wxSizerFlags::Center (p. 1186) for people with the other dialect of english.

wxSizerFlags::Expand

wxSizerFlags& Expand()

Sets the object of the wxSizerFlags to expand to fill as much area as it can.

wxSizerFlags::Left

wxSizerFlags& Left()

Aligns the object to the left, shortcut for `Align(wxALIGN_LEFT)`

See also

Align (p. 1186)

wxSizerFlags::Proportion

wxSizerFlags& Proportion(**int** *proportion = 0*)

Sets the proportion of this wxSizerFlags to `proportion`

wxSizerFlags::Right

wxSizerFlags& Right()

Aligns the object to the right, shortcut for `Align(wxALIGN_RIGHT)`

See also

Align (p. 1186)

wxSizerItem

The wxSizerItem class is used to track the position, size and other attributes of each item managed by a *wxSizer* (p. 1176). In normal usage user code should never need to deal directly with a wxSizerItem, but derived sizer classes will.

Derived from

wxObject (p. 1012)

Include files

<wx/sizer.h>

wxSizerItem::wxSizerItem

wxSizerItem(**int** *width*, **int** *height*, **int** *proportion*, **int** *flag*, **int** *border*, **wxObject***

userData)

Construct a sizer item for tracking a spacer.

wxSizerItem(**wxWindow*** *window*, **const wxSizerFlags&** *flags*)

wxSizerItem(**wxWindow*** *window*, **int** *proportion*, **int** *flag*, **int** *border*, **wxObject*** *userData*)

Construct a sizer item for tracking a window.

wxSizerItem(**wxSizer*** *window*, **const wxSizerFlags&** *flags*)

wxSizerItem(**wxSizer*** *sizer*, **int** *proportion*, **int** *flag*, **int** *border*, **wxObject*** *userData*)

Construct a sizer item for tracking a subsizer.

wxSizerItem::~wxSizerItem

~wxSizerItem()

Deletes the user data and subsizer, if any.

wxSizerItem::CalcMin

wxSize CalcMin()

Calculates the minimum desired size for the item, including any space needed by borders.

wxSizerItem::DeleteWindows

void DeleteWindows()

Destroy the window or the windows in a subsizer, depending on the type of item.

wxSizerItem::DetachSizer

void DetachSizer()

Enable deleting the SizerItem without destroying the contained sizer.

wxSizerItem::GetBorder

int GetBorder() **const**

Return the border attribute.

wxSizerItem::GetFlag

int GetFlag() **const**

Return the flags attribute.

wxSizerItem::GetMinSize

wxSize GetMinSize() **const**

Get the minimum size needed for the item.

wxSizerItem::GetPosition

wxPoint GetPosition() **const**

What is the current position of the item, as set in the last Layout.

wxSizerItem::GetProportion

int GetProportion() **const**

Get the proportion item attribute.

wxSizerItem::GetRatio

float GetRatio() **const**

Get the ration item attribute.

wxSizerItem::GetRect

wxRect GetRect()

Get the rectangle of the item on the parent window, excluding borders.

wxSizerItem::GetSize

wxSize GetSize() **const**

Get the current size of the item, as set in the last Layout.

wxSizerItem::GetSizer

wxSizer* GetSizer() **const**

If this item is tracking a sizer, return it. NULL otherwise.

wxSizerItem::GetSpacer

const wxSize& GetSpacer() **const**

If this item is tracking a spacer, return its size.

wxSizerItem::GetUserData

wxObject* GetUserData() **const**

Get the userData item attribute.

wxSizerItem::GetWindow

wxWindow* GetWindow() **const**

If this item is trackign a window then return it. NULL otherwise.

wxSizerItem::IsShown

bool IsShown() **const**

Is this item shown?

wxSizerItem::IsSizer

bool IsSizer() **const**

Is this item a sizer?

wxSizerItem::IsSpacer

bool IsSpacer() **const**

Is this item a spacer?

wxSizerItem::IsWindow

bool IsWindow() **const**

Is this item a window?

wxSizerItem::SetBorder

void SetBorder(**int** *border*)

Set the border item attribute.

wxSizerItem::SetDimension

void SetDimension(**wxPoint** *pos*, **wxSize** *size*)

Set the position and size of the space allocated to the sizer, and adjust the position and size of the item to be within that space taking alignment and borders into account.

wxSizerItem::SetFlag

void SetFlag(**int** *flag*)

Set the flag item attribute.

wxSizerItem::SetInitSize

void SetInitSize(**int** *x*, **int** *y*)

wxSizerItem::SetProportion

void SetProportion(**int** *proportion*)

Set the proportion item attribute.

wxSizerItem::SetRatio

void SetRatio(**int** *width*, **int** *height*)

void SetRatio(**wxSize** *size*)

void SetRatio(**float** *ratio*)

Set the ratio item attribute.

wxSizerItem::SetSizer

void SetSizer(**wxSizer*** *sizer*)

Set the sizer tracked by this item.

wxSizerItem::SetSpacer

void SetSpacer(**const wxSize&** *size*)

Set the size of the spacer tracked by this item.

wxSizerItem::SetWindow

void SetWindow(**wxWindow*** *window*)

Set the window to be tracked by thsi item.

wxSizerItem::Show

void Show(**bool** *show*)

Set the show item attribute, which sizers use to determine if the item is to be made part of the layout or not. If the item is tracking a window then it is shown or hidden as needed.

wxSlider

A slider is a control with a handle which can be pulled back and forth to change the value.

On Windows, the track bar control is used.

Slider events are handled in the same way as a scrollbar.

Derived from

wxControl (p. 215)
wxWindow (p. 1490)
wxEvtHandler (p. 486)
wxObject (p. 1012)

Include files

<wx/slider.h>

Window styles

wxSL_HORIZONTAL	Displays the slider horizontally (this is the default).
wxSL_VERTICAL	Displays the slider vertically.
wxSL_AUTOTICKS	Displays tick marks.
wxSL_LABELS	Displays minimum, maximum and value labels.

wxSL_LEFT	Displays ticks on the left and forces the slider to be vertical.
wxSL_RIGHT	Displays ticks on the right and forces the slider to be vertical.
wxSL_TOP	Displays ticks on the top.
wxSL_BOTTOM	Displays ticks on the bottom (this is the default).
wxSL_SELRANGE	Allows the user to select a range on the slider. Windows only.
wxSL_INVERSE	Inverses the mininum and maximum endpoints on the slider. Not compatible with wxSL_SELRANGE.

See also *window styles overview* (p. 1762).

Event table macros

To process a scroll event, use these event handler macros to direct input to member functions that take a wxScrollEvent argument. You can use `EVT_COMMAND_SCROLL...` macros with window IDs for when intercepting scroll events from controls, or `EVT_SCROLL...` macros without window IDs for intercepting scroll events from the receiving window - except for this, the macros behave exactly the same.

EVT_SCROLL(func)	Process all scroll events.
EVT_SCROLL_TOP(func)	Process wxEVT_SCROLL_TOP scroll-to-top events (minimum position).
EVT_SCROLL_BOTTOM(func)	Process wxEVT_SCROLL_TOP scroll-to-bottom events (maximum position).
EVT_SCROLL_LINEUP(func)	Process wxEVT_SCROLL_LINEUP line up events.
EVT_SCROLL_LINEDOWN(func)	Process wxEVT_SCROLL_LINEDOWN line down events.
EVT_SCROLL_PAGEUP(func)	Process wxEVT_SCROLL_PAGEUP page up events.
EVT_SCROLL_PAGEDOWN(func)	Process wxEVT_SCROLL_PAGEDOWN page down events.
EVT_SCROLL_THUMBTRACK(func)	Process wxEVT_SCROLL_THUMBTRACK thumbtrack events (frequent events sent as the user drags the

thumbtrack).

EVT_SCROLL_THUMBRELEASE(func)	Process wxEVT_SCROLL_THUMBRELEASE thumb release events.
EVT_SCROLL_ENDSCROLL(func)	Process wxEVT_SCROLL_ENDSCROLL end of scrolling events (MSW only).
EVT_COMMAND_SCROLL(id, func)	Process all scroll events.
EVT_COMMAND_SCROLL_TOP(id, func)	Process wxEVT_SCROLL_TOP scroll-to-top events (minimum position).
EVT_COMMAND_SCROLL_BOTTOM(id, func)	Process wxEVT_SCROLL_TOP scroll-to-bottom events (maximum position).
EVT_COMMAND_SCROLL_LINEUP(id, func)	Process wxEVT_SCROLL_LINEUP line up events.
EVT_COMMAND_SCROLL_LINEDOWN(id, func)	Process wxEVT_SCROLL_LINEDOWN line down events.
EVT_COMMAND_SCROLL_PAGEUP(id, func)	Process wxEVT_SCROLL_PAGEUP page up events.
EVT_COMMAND_SCROLL_PAGEDOWN(id, func)	Process wxEVT_SCROLL_PAGEDOWN page down events.
EVT_COMMAND_SCROLL_THUMBTRACK(id, func)	Process wxEVT_SCROLL_THUMBTRACK thumbtrack events (frequent events sent as the user drags the thumbtrack).
EVT_COMMAND_SCROLL_THUMBRELEASE(func)	Process wxEVT_SCROLL_THUMBRELEASE thumb release events.
EVT_COMMAND_SCROLL_ENDSCROLL(func)	Process wxEVT_SCROLL_ENDSCROLL end of scrolling events (MSW only).

The difference between `EVT_SCROLL_THUMBRELEASE` and `EVT_SCROLL_ENDSCROLL`

The `EVT_SCROLL_THUMBRELEASE` event is only emitted when actually dragging the

thumb using the mouse and releasing it (This `EVT_SCROLL_THUMBRELEASE`event is also followed by an `EVT_SCROLL_ENDSCROLL` event).

The `EVT_SCROLL_ENDSCROLL` event also occurs when using the keyboard to change the thumb position, and when clicking next to the thumb (In all these cases the `EVT_SCROLL_THUMBRELEASE` event does not happen).

In short, the `EVT_SCROLL_ENDSCROLL` event is triggered when scrolling/ moving has finished. The only exception (unfortunately) is that changing the thumb position using the mousewheel does give a `EVT_SCROLL_THUMBRELEASE`event but NOT an `EVT_SCROLL_ENDSCROLL` event.

Please see the widgets sample ("Slider" page) to see the difference between `EVT_SCROLL_THUMBRELEASE` and `EVT_SCROLL_ENDSCROLL` in action.

See also

Event handling overview (p. 1751), *wxScrollBar* (p. 1143)

wxSlider::wxSlider

wxSlider()

Default slider.

wxSlider(**wxWindow*** *parent*, **wxWindowID** *id*, **int** *value* , **int** *minValue*, **int** *maxValue*, **const wxPoint&** *point = wxDefaultPosition*, **const wxSize&** *size = wxDefaultSize*, **long** *style = wxSL_HORIZONTAL*, **const wxValidator&** *validator = wxDefaultValidator*, **const wxString&** *name = "slider"*)

Constructor, creating and showing a slider.

Parameters

parent

      Parent window. Must not be NULL.

id

      Window identifier. A value of -1 indicates a default value.

value

      Initial position for the slider.

minValue

      Minimum slider position.

maxValue

Maximum slider position.

size

Window size. If the default size (-1, -1) is specified then a default size is chosen.

style

Window style. See *wxSlider* (p. 1192).

validator

Window validator.

name

Window name.

See also

wxSlider::Create (p. 1196), *wxValidator* (p. 1461)

wxSlider::~wxSlider

void ~wxSlider()

Destructor, destroying the slider.

wxSlider::ClearSel

void ClearSel()

Clears the selection, for a slider with the **wxSL_SELRANGE** style.

Remarks

Windows 95 only.

wxSlider::ClearTicks

void ClearTicks()

Clears the ticks.

Remarks

Windows 95 only.

wxSlider::Create

bool Create(**wxWindow*** *parent*, **wxWindowID** *id*, **int** *value* , **int** *minValue*, **int** *maxValue*, **const wxPoint&** *point* = *wxDefaultPosition*, **const wxSize&** *size* = *wxDefaultSize*, **long** *style* = *wxSL_HORIZONTAL*, **const wxValidator&** *validator* = *wxDefaultValidator*, **const wxString&** *name* = *"slider"*)

Used for two-step slider construction. See *wxSlider::wxSlider* (p. 1195) for further details.

wxSlider::GetLineSize

int GetLineSize() **const**

Returns the line size.

See also

wxSlider::SetLineSize (p. 1199)

wxSlider::GetMax

int GetMax() **const**

Gets the maximum slider value.

See also

wxSlider::GetMin (p. 1197), *wxSlider::SetRange* (p. 1199)

wxSlider::GetMin

int GetMin() **const**

Gets the minimum slider value.

See also

wxSlider::GetMin (p. 1197), *wxSlider::SetRange* (p. 1199)

wxSlider::GetPageSize

int GetPageSize() **const**

Returns the page size.

See also

wxSlider::SetPageSize (p. 1199)

wxSlider::GetSelEnd

int GetSelEnd() **const**

Returns the selection end point.

Remarks

Windows 95 only.

See also

wxSlider::GetSelStart (p. 1198), *wxSlider::SetSelection* (p. 1200)

wxSlider::GetSelStart

int GetSelStart() **const**

Returns the selection start point.

Remarks

Windows 95 only.

See also

wxSlider::GetSelEnd (p. 1197), *wxSlider::SetSelection* (p. 1200)

wxSlider::GetThumbLength

int GetThumbLength() **const**

Returns the thumb length.

Remarks

Windows 95 only.

See also

wxSlider::SetThumbLength (p. 1200)

wxSlider::GetTickFreq

int GetTickFreq() **const**

Returns the tick frequency.

Remarks

Windows 95 only.

See also

wxSlider::SetTickFreq (p. 1201)

wxSlider::GetValue

int GetValue() **const**

Gets the current slider value.

See also

wxSlider::GetMin (p. 1197), *wxSlider::GetMax* (p. 1197), *wxSlider::SetValue* (p. 1201)

wxSlider::SetLineSize

void SetLineSize(**int** *lineSize*)

Sets the line size for the slider.

Parameters

lineSize

> The number of steps the slider moves when the user moves it up or down a line.

See also

wxSlider::GetLineSize (p. 1197)

wxSlider::SetPageSize

void SetPageSize(**int** *pageSize*)

Sets the page size for the slider.

Parameters

pageSize

> The number of steps the slider moves when the user pages up or down.

See also

wxSlider::GetPageSize (p. 1197)

wxSlider::SetRange

void SetRange(**int** *minValue*, **int** *maxValue*)

Sets the minimum and maximum slider values.

wxSlider::GetMin (p. 1197), *wxSlider::GetMax* (p. 1197)

wxSlider::SetSelection

void SetSelection(**int** *startPos*, **int** *endPos*)

Sets the selection.

Parameters

startPos

> The selection start position.

endPos

> The selection end position.

Remarks

Windows 95 only.

See also

wxSlider::GetSelStart (p. 1198), *wxSlider::GetSelEnd* (p. 1197)

wxSlider::SetThumbLength

void SetThumbLength(**int** *len*)

Sets the slider thumb length.

Parameters

len

> The thumb length.

Remarks

Windows 95 only.

See also

wxSlider::GetThumbLength (p. 1198)

wxSlider::SetTick

void SetTick(**int** *tickPos*)

Sets a tick position.

Parameters

tickPos

>   The tick position.

Remarks

Windows 95 only.

See also

wxSlider::SetTickFreq (p. 1201)

wxSlider::SetTickFreq

void SetTickFreq(**int** *n,* **int** *pos*)

Sets the tick mark frequency and position.

Parameters

n

>   Frequency. For example, if the frequency is set to two, a tick mark is displayed for every other increment in the slider's range.

pos

>   Position. Must be greater than zero. TODO: what is this for?

Remarks

Windows 95 only.

See also

wxSlider::GetTickFreq (p. 1198)

wxSlider::SetValue

void SetValue(**int** *value*)

Sets the slider position.

Parameters

value

>   The slider position.

wxSockAddress

You are unlikely to need to use this class: only wxSocketBase uses it.

Derived from

wxObject (p. 1012)

Include files

<wx/socket.h>

See also

wxSocketBase (p. 1203)*wxIPaddress* (p. 817)*wxIPV4address* (p. 819)

wxSockAddress::wxSockAddress

wxSockAddress()

Default constructor.

wxSockAddress::~wxSockAddress

~wxSockAddress()

Default destructor.

wxSockAddress::Clear

void Clear()

Delete all informations about the address.

wxSockAddress::SockAddrLen

int SockAddrLen()

Returns the length of the socket address.

wxSocketBase

wxSocketBase is the base class for all socket-related objects, and it defines all basic IO functionality.

Note: (Workaround for implementation limitation for wxWidgets up to 2.5.x) If you want to use sockets or derived classes such as wxFTP in a secondary thread, call wxSocketBase::Initialize() (undocumented) from the main thread before creating any sockets - in wxApp::OnInit for example. See http://wiki.wxwidgets.org/wiki.pl?WxSocket or http://www.litwindow.com/knowhow/knowhow.html for more details.

Derived from

wxObject (p. 1012)

Include files

<wx/socket.h>

wxSocket errors

wxSOCKET_NOERROR	No error happened.
wxSOCKET_INVOP	Invalid operation.
wxSOCKET_IOERR	Input/Output error.
wxSOCKET_INVADDR	Invalid address passed to wxSocket.
wxSOCKET_INVSOCK	Invalid socket (uninitialized).
wxSOCKET_NOHOST	No corresponding host.
wxSOCKET_INVPORT	Invalid port.
wxSOCKET_WOULDBLOCK	The socket is non-blocking and the operation would block.
wxSOCKET_TIMEDOUT	The timeout for this operation expired.
wxSOCKET_MEMERR	Memory exhausted.

wxSocket events

wxSOCKET_INPUT	There is data available for reading.
wxSOCKET_OUTPUT	The socket is ready to be written to.
wxSOCKET_CONNECTION	Incoming connection request (server), or successful connection establishment (client).
wxSOCKET_LOST	The connection has been closed.

A brief note on how to use these events:

The **wxSOCKET_INPUT** event will be issued whenever there is data available for reading. This will be the case if the input queue was empty and new data arrives, or if the application has read some data yet there is still more data available. This means that the application does not need to read all available data in response to a **wxSOCKET_INPUT** event, as more events will be produced as necessary.

The **wxSOCKET_OUTPUT** event is issued when a socket is first connected with *Connect* (p. 1219) or accepted with *Accept* (p. 1224). After that, new events will be generated only after an output operation fails with **wxSOCKET_WOULDBLOCK** and buffer space becomes available again. This means that the application should assume that it can write data to the socket until an **wxSOCKET_WOULDBLOCK** error occurs; after this, whenever the socket becomes writable again the application will be notified with another **wxSOCKET_OUTPUT** event.

The **wxSOCKET_CONNECTION** event is issued when a delayed connection request completes successfully (client) or when a new connection arrives at the incoming queue (server).

The **wxSOCKET_LOST** event is issued when a close indication is received for the socket. This means that the connection broke down or that it was closed by the peer. Also, this event will be issued if a connection request fails.

Event handling

To process events coming from a socket object, use the following event handler macro to direct events to member functions that take a *wxSocketEvent* (p. 1221) argument.

EVT_SOCKET(id, func)   Process a wxEVT_SOCKET event.

See also

wxSocketEvent (p. 1221), *wxSocketClient* (p. 1219), *wxSocketServer* (p. 1223), *Sockets sample* (p. 1714)

Construction and destruction

wxSocketBase (p. 1206)
~wxSocketBase (p. 1206)
Destroy (p. 1206)

Socket state

Functions to retrieve current state and miscellaneous info.

Error (p. 1207)
GetLocal (p. 1207)
GetPeer (p. 1207)*IsConnected* (p. 1208)
IsData (p. 1208)
IsDisconnected (p. 1208)

Basic IO

Functions that perform basic IO functionality.

Functions that perform a timed wait on a certain IO condition.

and also:

Functions that allow applications to customize socket IO as needed.

Handling socket events

Functions that allow applications to receive socket events.

wxSocketBase::wxSocketBase

wxSocketBase()

Default constructor. Don't use it directly; instead, use *wxSocketClient* (p. 1219) to construct a socket client, or *wxSocketServer* (p. 1223) to construct a socket server.

wxSocketBase::~wxSocketBase

~wxSocketBase()

Destructor. Do not destroy a socket using the delete operator directly; use *Destroy* (p. 1206) instead. Also, do not create socket objects in the stack.

wxSocketBase::Close

void Close()

This function shuts down the socket, disabling further transmission and reception of data; it also disables events for the socket and frees the associated system resources. Upon socket destruction, Close is automatically called, so in most cases you won't need to do it yourself, unless you explicitly want to shut down the socket, typically to notify the peer that you are closing the connection.

Remark/Warning

Although Close immediately disables events for the socket, it is possible that event messages may be waiting in the application's event queue. The application must therefore be prepared to handle socket event messages even after calling Close.

wxSocketBase::Destroy

bool Destroy()

Destroys the socket safely. Use this function instead of the delete operator, since otherwise socket events could reach the application even after the socket has been destroyed. To prevent this problem, this function appends the wxSocket to a list of object to be deleted on idle time, after all events have been processed. For the same reason, you should avoid creating socket objects in the stack.

Destroy calls *Close* (p. 1206) automatically.

Return value

Always true.

wxSocketBase::Discard

wxSocketBase& Discard()

This function simply deletes all bytes in the incoming queue. This function always returns immediately and its operation is not affected by IO flags.

Use *LastCount* (p. 1208) to verify the number of bytes actually discarded.

If you use *Error* (p. 1207), it will always return false.

wxSocketBase::Error

bool Error() **const**

Returns true if an error occurred in the last IO operation.

Use this function to check for an error condition after one of the following calls: Discard, Peek, Read, ReadMsg, Unread, Write, WriteMsg.

wxSocketBase::GetClientData

void * GetClientData() **const**

Returns a pointer of the client data for this socket, as set with *SetClientData* (p. 1210)

wxSocketBase::GetLocal

bool GetLocal(**wxSockAddress&** *addr*) **const**

This function returns the local address field of the socket. The local address field contains the complete local address of the socket (local address, local port, ...).

Return value

true if no error happened, false otherwise.

wxSocketBase::GetFlags

wxSocketFlags GetFlags() **const**

Returns current IO flags, as set with *SetFlags* (p. 1210)

wxSocketBase::GetPeer

bool GetPeer(**wxSockAddress&** *addr*) **const**

This function returns the peer address field of the socket. The peer address field contains the complete peer host address of the socket (address, port, ...).

true if no error happened, false otherwise.

wxSocketBase::InterruptWait

void InterruptWait()

Use this function to interrupt any wait operation currently in progress. Note that this is not intended as a regular way to interrupt a Wait call, but only as an escape mechanism for exceptional situations where it is absolutely necessary to use it, for example to abort an operation due to some exception or abnormal problem. InterruptWait is automatically called when you *Close* (p. 1206) a socket (and thus also upon socket destruction), so you don't need to use it in these cases.

wxSocketBase::Wait (p. 1215), *wxSocketServer::WaitForAccept* (p. 1225), *wxSocketBase::WaitForLost* (p. 1216), *wxSocketBase::WaitForRead* (p. 1216), *wxSocketBase::WaitForWrite* (p. 1217), *wxSocketClient::WaitOnConnect* (p. 1220)

wxSocketBase::IsConnected

bool IsConnected() **const**

Returns true if the socket is connected.

wxSocketBase::IsData

bool IsData() **const**

This function waits until the socket is readable. This might mean that queued data is available for reading or, for streamed sockets, that the connection has been closed, so that a read operation will complete immediately without blocking (unless the **wxSOCKET_WAITALL** flag is set, in which case the operation might still block).

wxSocketBase::IsDisconnected

bool IsDisconnected() **const**

Returns true if the socket is not connected.

wxSocketBase::LastCount

wxUint32 LastCount() **const**

Returns the number of bytes read or written by the last IO call.

Use this function to get the number of bytes actually transferred after using one of the

following IO calls: Discard, Peek, Read, ReadMsg, Unread, Write, WriteMsg.

wxSocketBase::LastError

wxSocketError LastError() const

Returns the last wxSocket error. See *wxSocket errors* (p. 1203).

Please note that this function merely returns the last error code, but it should not be used to determine if an error has occurred (this is because successful operations do not change the LastError value). Use *Error* (p. 1207) first, in order to determine if the last IO call failed. If this returns true, use LastError to discover the cause of the error.

wxSocketBase::Notify

void Notify(bool *notify*)

According to the *notify* value, this function enables or disables socket events. If *notify* is true, the events configured with *SetNotify* (p. 1212) will be sent to the application. If *notify* is false; no events will be sent.

wxSocketBase::Ok

bool Ok() const

Returns true if the socket is initialized and ready and false in other cases.

Remark/Warning

For *wxSocketClient* (p. 1219), Ok won't return true unless the client is connected to a server.

For *wxSocketServer* (p. 1223), Ok will return true if the server could bind to the specified address and is already listening for new connections.

Ok does not check for IO errors; use *Error* (p. 1207) instead for that purpose.

wxSocketBase::RestoreState

void RestoreState()

This function restores the previous state of the socket, as saved with *SaveState* (p. 1210)

Calls to SaveState and RestoreState can be nested.

See also

wxSocketBase::SaveState (p. 1210)

wxSocketBase::SaveState

void SaveState()

This function saves the current state of the socket in a stack. Socket state includes flags, as set with *SetFlags* (p. 1210), event mask, as set with *SetNotify* (p. 1212) and *Notify* (p. 1209), user data, as set with *SetClientData* (p. 1210).

Calls to SaveState and RestoreState can be nested.

See also

wxSocketBase::RestoreState (p. 1209)

wxSocketBase::SetClientData

void SetClientData(**void ****data*)

Sets user-supplied client data for this socket. All socket events will contain a pointer to this data, which can be retrieved with the *wxSocketEvent::GetClientData* (p. 1221) function.

wxSocketBase::SetEventHandler

void SetEventHandler(**wxEvtHandler&** *handler*, **int** *id = -1*)

Sets an event handler to be called when a socket event occurs. The handler will be called for those events for which notification is enabled with *SetNotify* (p. 1212) and *Notify* (p. 1209).

Parameters

handler

> Specifies the event handler you want to use.

id

> The id of socket event.

See also

wxSocketBase::SetNotify (p. 1212), *wxSocketBase::Notify* (p. 1209), *wxSocketEvent* (p. 1221), *wxEvtHandler* (p. 486)

wxSocketBase::SetFlags

void SetFlags(**wxSocketFlags** *flags*)

Use SetFlags to customize IO operation for this socket. The *flags* parameter may be a

combination of flags ORed together. The following flags can be used:

wxSOCKET_NONE	Normal functionality.
wxSOCKET_NOWAIT	Read/write as much data as possible and return immediately.
wxSOCKET_WAITALL	Wait for all required data to be read/written unless an error occurs.
wxSOCKET_BLOCK	Block the GUI (do not yield) while reading/writing data.
wxSOCKET_REUSEADDR	Allows the use of an in-use port (wxServerSocket only)

A brief overview on how to use these flags follows.

If no flag is specified (this is the same as **wxSOCKET_NONE**), IO calls will return after some data has been read or written, even when the transfer might not be complete. This is the same as issuing exactly one blocking low-level call to recv() or send(). Note that *blocking* here refers to when the function returns, not to whether the GUI blocks during this time.

If **wxSOCKET_NOWAIT** is specified, IO calls will return immediately. Read operations will retrieve only available data. Write operations will write as much data as possible, depending on how much space is available in the output buffer. This is the same as issuing exactly one nonblocking low-level call to recv() or send(). Note that *nonblocking* here refers to when the function returns, not to whether the GUI blocks during this time.

If **wxSOCKET_WAITALL** is specified, IO calls won't return until ALL the data has been read or written (or until an error occurs), blocking if necessary, and issuing several low level calls if necessary. This is the same as having a loop which makes as many blocking low-level calls to recv() or send() as needed so as to transfer all the data. Note that *blocking* here refers to when the function returns, not to whether the GUI blocks during this time.

The **wxSOCKET_BLOCK** flag controls whether the GUI blocks during IO operations. If this flag is specified, the socket will not yield during IO calls, so the GUI will remain blocked until the operation completes. If it is not used, then the application must take extra care to avoid unwanted reentrance.

The **wxSOCKET_REUSEADDR** flag controls the use of the SO_REUSEADDR standard setsockopt() flag. This flag allows the socket to bind to a port that is already in use. This is mostly used on UNIX-based systems to allow rapid starting and stopping of a server - otherwise you may have to wait several minutes for the port to become available. This option can have suprising platform dependent behavior, check the documentation for your platforms implementation of setsockopt().

So:

wxSOCKET_NONE will try to read at least SOME data, no matter how much.

wxSOCKET_NOWAIT will always return immediately, even if it cannot read or write

ANY data.

wxSOCKET_WAITALL will only return when it has read or written ALL the data.

wxSOCKET_BLOCK has nothing to do with the previous flags and it controls whether the GUI blocks.

wxSOCKET_REUSEADDR controls special platform-specific behavior for wxServerSocket.

wxSocketBase::SetNotify

void SetNotify(**wxSocketEventFlags** *flags*)

SetNotify specifies which socket events are to be sent to the event handler. The *flags* parameter may be combination of flags ORed together. The following flags can be used:

wxSOCKET_INPUT_FLAG   to receive wxSOCKET_INPUT

wxSOCKET_OUTPUT_FLAG   to receive wxSOCKET_OUTPUT

wxSOCKET_CONNECTION_FLAG   to receive wxSOCKET_CONNECTION

wxSOCKET_LOST_FLAG   to receive wxSOCKET_LOST

For example:

```
sock.SetNotify(wxSOCKET_INPUT_FLAG | wxSOCKET_LOST_FLAG);
sock.Notify(true);
```

In this example, the user will be notified about incoming socket data and whenever the connection is closed.

For more information on socket events see *wxSocket events* (p. 1203).

wxSocketBase::SetTimeout

void SetTimeout(**int** *seconds*)

This function sets the default socket timeout in seconds. This timeout applies to all IO calls, and also to the *Wait* (p. 1215) family of functions if you don't specify a wait interval. Initially, the default timeout is 10 minutes.

wxSocketBase::Peek

wxSocketBase& Peek(**void *** *buffer*, **wxUint32** *nbytes*)

This function peeks a buffer of *nbytes* bytes from the socket. Peeking a buffer doesn't delete it from the socket input queue.

Use *LastCount* (p. 1208) to verify the number of bytes actually peeked.

Use *Error* (p. 1207) to determine if the operation succeeded.

Parameters

buffer

> Buffer where to put peeked data.

nbytes

> Number of bytes.

Return value

Returns a reference to the current object.

Remark/Warning

The exact behaviour of wxSocketBase::Peek depends on the combination of flags being used. For a detailed explanation, see *wxSocketBase::SetFlags* (p. 1210)

See also

wxSocketBase::Error (p. 1207), *wxSocketBase::LastError* (p. 1209), *wxSocketBase::LastCount* (p. 1208), *wxSocketBase::SetFlags* (p. 1210)

wxSocketBase::Read

wxSocketBase& Read(**void *** *buffer*, **wxUint32** *nbytes*)

This function reads a buffer of *nbytes* bytes from the socket.

Use *LastCount* (p. 1208) to verify the number of bytes actually read.

Use *Error* (p. 1207) to determine if the operation succeeded.

Parameters

buffer

> Buffer where to put read data.

nbytes

> Number of bytes.

Return value

Returns a reference to the current object.

Remark/Warning

The exact behaviour of wxSocketBase::Read depends on the combination of flags being used. For a detailed explanation, see *wxSocketBase::SetFlags* (p. 1210).

wxSocketBase::ReadMsg

wxSocketBase& ReadMsg(**void *** *buffer*, **wxUint32** *nbytes*)

This function reads a buffer sent by *WriteMsg* (p. 1218) on a socket. If the buffer passed
to the function isn't big enough, the remaining bytes will be discarded. This function
always waits for the buffer to be entirely filled, unless an error occurs.

Use *LastCount* (p. 1208) to verify the number of bytes actually read.

Use *Error* (p. 1207) to determine if the operation succeeded.

Parameters

buffer

> Buffer where to put read data.

nbytes

> Size of the buffer.

Return value

Returns a reference to the current object.

Remark/Warning

wxSocketBase::ReadMsg will behave as if the **wxSOCKET_WAITALL** flag was always
set and it will always ignore the **wxSOCKET_NOWAIT** flag. The exact behaviour of
ReadMsg depends on the **wxSOCKET_BLOCK** flag. For a detailed explanation, see
wxSocketBase::SetFlags (p. 1210).

See also

wxSocketBase::Unread

wxSocketBase& Unread(**const void *** *buffer*, **wxUint32** *nbytes*)

This function unreads a buffer. That is, the data in the buffer is put back in the incoming
queue. This function is not affected by wxSocket flags.

If you use *LastCount* (p. 1208), it will always return *nbytes*.

If you use *Error* (p. 1207), it will always return false.

Parameters

buffer

Buffer to be unread.

nbytes

Number of bytes.

Return value

Returns a reference to the current object.

See also

wxSocketBase::Error (p. 1207), *wxSocketBase::LastCount* (p. 1208),
wxSocketBase::LastError (p. 1209)

wxSocketBase::Wait

bool Wait(**long** *seconds = -1*, **long** *millisecond = 0*)

This function waits until any of the following conditions is true:

- The socket becomes readable.

- The socket becomes writable.

- An ongoing connection request has completed (*wxSocketClient* (p. 1219) only)

- An incoming connection request has arrived (*wxSocketServer* (p. 1223) only)

- The connection has been closed.

Note that it is recommended to use the individual Wait functions to wait for the required condition, instead of this one.

Parameters

seconds

Number of seconds to wait. If -1, it will wait for the default timeout, as set with *SetTimeout* (p. 1212).

millisecond

Number of milliseconds to wait.

Return value

Returns true when any of the above conditions is satisfied, false if the timeout was

reached.

See also

wxSocketBase::InterruptWait (p. 1208), *wxSocketServer::WaitForAccept* (p. 1225),
wxSocketBase::WaitForLost (p. 1216), *wxSocketBase::WaitForRead* (p. 1216),
wxSocketBase::WaitForWrite (p. 1217), *wxSocketClient::WaitOnConnect* (p. 1220)

wxSocketBase::WaitForLost

bool Wait(**long** *seconds = -1*, **long** *millisecond = 0*)

This function waits until the connection is lost. This may happen if the peer gracefully
closes the connection or if the connection breaks.

Parameters

seconds

Number of seconds to wait. If -1, it will wait for the default timeout, as set with
SetTimeout (p. 1212).

millisecond

Number of milliseconds to wait.

Return value

Returns true if the connection was lost, false if the timeout was reached.

See also

wxSocketBase::InterruptWait (p. 1208),*wxSocketBase::Wait* (p. 1215)

wxSocketBase::WaitForRead

bool WaitForRead(**long** *seconds = -1*, **long** *millisecond = 0*)

This function waits until the socket is readable. This might mean that queued data is
available for reading or, for streamed sockets, that the connection has been closed, so
that a read operation will complete immediately without blocking (unless the
wxSOCKET_WAITALL flag is set, in which case the operation might still block).

Parameters

seconds

Number of seconds to wait. If -1, it will wait for the default timeout, as set with
SetTimeout (p. 1212).

millisecond

Number of milliseconds to wait.

Return value

Returns true if the socket becomes readable, false on timeout.

See also

wxSocketBase::InterruptWait (p. 1208), *wxSocketBase::Wait* (p. 1215)

wxSocketBase::WaitForWrite

bool WaitForWrite(**long** *seconds = -1*, **long** *millisecond = 0*)

This function waits until the socket becomes writable. This might mean that the socket is ready to send new data, or for streamed sockets, that the connection has been closed, so that a write operation is guaranteed to complete immediately (unless the **wxSOCKET_WAITALL** flag is set, in which case the operation might still block).

Parameters

seconds

Number of seconds to wait. If -1, it will wait for the default timeout, as set with *SetTimeout* (p. 1212).

millisecond

Number of milliseconds to wait.

Return value

Returns true if the socket becomes writable, false on timeout.

See also

wxSocketBase::InterruptWait (p. 1208), *wxSocketBase::Wait* (p. 1215)

wxSocketBase::Write

wxSocketBase& Write(**const void *** *buffer*, **wxUint32** *nbytes*)

This function writes a buffer of *nbytes* bytes to the socket.

Use *LastCount* (p. 1208) to verify the number of bytes actually written.

Use *Error* (p. 1207) to determine if the operation succeeded.

Parameters

buffer

Buffer with the data to be sent.

nbytes

> Number of bytes.

Return value

Returns a reference to the current object.

Remark/Warning

The exact behaviour of wxSocketBase::Write depends on the combination of flags being used. For a detailed explanation, see *wxSocketBase::SetFlags* (p. 1210).

See also

wxSocketBase::Error (p. 1207), *wxSocketBase::LastError* (p. 1209), *wxSocketBase::LastCount* (p. 1208), *wxSocketBase::SetFlags* (p. 1210)

wxSocketBase::WriteMsg

wxSocketBase& WriteMsg(**const void *** *buffer*, **wxUint32** *nbytes*)

This function writes a buffer of *nbytes* bytes from the socket, but it writes a short header before so that *ReadMsg* (p. 1214) knows how much data should it actually read. So, a buffer sent with WriteMsg **must** be read with ReadMsg. This function always waits for the entire buffer to be sent, unless an error occurs.

Use *LastCount* (p. 1208) to verify the number of bytes actually written.

Use *Error* (p. 1207) to determine if the operation succeeded.

Parameters

buffer

> Buffer with the data to be sent.

nbytes

> Number of bytes to send.

Return value

Returns a reference to the current object.

Remark/Warning

wxSocketBase::WriteMsg will behave as if the **wxSOCKET_WAITALL** flag was always set and it will always ignore the **wxSOCKET_NOWAIT** flag. The exact behaviour of WriteMsg depends on the **wxSOCKET_BLOCK** flag. For a detailed explanation, see *wxSocketBase::SetFlags* (p. 1210).

See also

wxSocketBase::Error (p. 1207), *wxSocketBase::LastError* (p. 1209),
wxSocketBase::LastCount (p. 1208), *wxSocketBase::SetFlags* (p. 1210),
wxSocketBase::ReadMsg (p. 1214)

wxSocketClient

Derived from

wxSocketBase (p. 1203)

Include files

<wx/socket.h>

wxSocketClient::wxSocketClient

wxSocketClient(**wxSocketFlags** *flags* = *wxSOCKET_NONE*)

Constructor.

Parameters

flags

      Socket flags (See *wxSocketBase::SetFlags* (p. 1210))

wxSocketClient::~wxSocketClient

~wxSocketClient()

Destructor. Please see *wxSocketBase::Destroy* (p. 1206).

wxSocketClient::Connect

bool Connect(**wxSockAddress&** *address*, **bool** *wait* = *true*)

Connects to a server using the specified address.

If *wait* is true, Connect will wait until the connection completes. **Warning:** This will block the GUI.

If *wait* is false, Connect will try to establish the connection and return immediately, without blocking the GUI. When used this way, even if Connect returns false, the connection request can be completed later. To detect this, use *WaitOnConnect* (p. 1220), or catch **wxSOCKET_CONNECTION** events (for successful establishment) and **wxSOCKET_LOST** events (for connection failure).

Parameters

address

> Address of the server.

wait

> If true, waits for the connection to complete.

Return value

Returns true if the connection is established and no error occurs.

If *wait* was true, and Connect returns false, an error occurred and the connection failed.

If *wait* was false, and Connect returns false, you should still be prepared to handle the completion of this connection request, either with *WaitOnConnect* (p. 1220) or by watching **wxSOCKET_CONNECTION** and **wxSOCKET_LOST** events.

See also

wxSocketClient::WaitOnConnect (p. 1220), *wxSocketBase::SetNotify* (p. 1212), *wxSocketBase::Notify* (p. 1209)

wxSocketClient::WaitOnConnect

bool WaitOnConnect(**long** *seconds = -1*, **long** *milliseconds = 0*)

Wait until a connection request completes, or until the specified timeout elapses. Use this function after issuing a call to *Connect* (p. 1219) with *wait* set to false.

Parameters

seconds

> Number of seconds to wait. If -1, it will wait for the default timeout, as set with *SetTimeout* (p. 1212).

millisecond

> Number of milliseconds to wait.

Return value

WaitOnConnect returns true if the connection request completes. This does not necessarily mean that the connection was successfully established; it might also happen that the connection was refused by the peer. Use *IsConnected* (p. 1208) to distinguish between these two situations.

If the timeout elapses, WaitOnConnect returns false.

These semantics allow code like this:

```
// Issue the connection request
client->Connect(addr, false);

// Wait until the request completes or until we decide to give up
bool waitmore = true;
while ( !client->WaitOnConnect(seconds, millis) && waitmore )
{
    // possibly give some feedback to the user,
    // and update waitmore as needed.
}
bool success = client->IsConnected();
```

See also

wxSocketClient::Connect (p. 1219), *wxSocketBase::InterruptWait* (p. 1208), *wxSocketBase::IsConnected* (p. 1208)

wxSocketEvent

This event class contains information about socket events.

Derived from

wxEvent (p. 483)

Include files

<wx/socket.h>

Event table macros

To process a socket event, use these event handler macros to direct input to member functions that take a wxSocketEvent argument.

EVT_SOCKET(id, func)   Process a socket event, supplying the member function.

See also

wxSocketBase (p. 1203), *wxSocketClient* (p. 1219), *wxSocketServer* (p. 1223)

wxSocketEvent::wxSocketEvent

 wxSocketEvent(**int** *id = 0*)

Constructor.

wxSocketEvent::GetClientData

void * GetClientData()

Gets the client data of the socket which generated this event, as set with
wxSocketBase::SetClientData (p. 1210).

wxSocketEvent::GetSocket

wxSocketBase * GetSocket() const

Returns the socket object to which this event refers to. This makes it possible to use the
same event handler for different sockets.

wxSocketEvent::GetSocketEvent

wxSocketNotify GetSocketEvent() const

Returns the socket event type.

wxSocketInputStream

This class implements an input stream which reads data from a connected socket. Note
that this stream is purely sequential and it does not support seeking.

Derived from

wxInputStream (p. 814)

Include files

<wx/sckstrm.h>

See also

wxSocketBase (p. 1203)

wxSocketInputStream::wxSocketInputStream

wxSocketInputStream(wxSocketBase& *s*)

Creates a new read-only socket stream using the specified initialized socket connection.

wxSocketOutputStream

This class implements an output stream which writes data from a connected socket.
Note that this stream is purely sequential and it does not support seeking.

Derived from

wxOutputStream (p. 1016)

Include files

<wx/sckstrm.h>

See also

wxSocketBase (p. 1203)

wxSocketOutputStream::wxSocketOutputStream

wxSocketOutputStream(**wxSocketBase&** *s*)

Creates a new write-only socket stream using the specified initialized socket connection.

wxSocketServer

Derived from

wxSocketBase (p. 1203)

Include files

<wx/socket.h>

wxSocketServer::wxSocketServer

wxSocketServer(**wxSockAddress&** *address*, **wxSocketFlags** *flags* = *wxSOCKET_NONE*)

Constructs a new server and tries to bind to the specified *address*. Before trying to accept new connections, test whether it succeeded with *wxSocketBase::Ok* (p. 1209).

Parameters

address

> Specifies the local address for the server (e.g. port number).

flags

> Socket flags (See *wxSocketBase::SetFlags* (p. 1210))

wxSocketServer::~wxSocketServer

~wxSocketServer()

Destructor (it doesn't close the accepted connections).

wxSocketServer::Accept

wxSocketBase * Accept(**bool** *wait* = *true*)

Accepts an incoming connection request, and creates a new *wxSocketBase* (p. 1203) object which represents the server-side of the connection.

If *wait* is true and there are no pending connections to be accepted, it will wait for the next incoming connection to arrive. **Warning:** This will block the GUI.

If *wait* is false, it will try to accept a pending connection if there is one, but it will always return immediately without blocking the GUI. If you want to use Accept in this way, you can either check for incoming connections with *WaitForAccept* (p. 1225) or catch **wxSOCKET_CONNECTION** events, then call Accept once you know that there is an incoming connection waiting to be accepted.

Return value

Returns an opened socket connection, or NULL if an error occurred or if the *wait* parameter was false and there were no pending connections.

See also

wxSocketServer::WaitForAccept (p. 1225), *wxSocketBase::SetNotify* (p. 1212), *wxSocketBase::Notify* (p. 1209), *wxSocketServer::AcceptWith* (p. 1224)

wxSocketServer::AcceptWith

bool AcceptWith(**wxSocketBase&** *socket*, **bool** *wait* = *true*)

Accept an incoming connection using the specified socket object.

Parameters

socket

    Socket to be initialized

Return value

Returns true on success, or false if an error occurred or if the*wait* parameter was false and there were no pending connections.

wxSocketServer::WaitForAccept (p. 1225), *wxSocketBase::SetNotify* (p. 1212), *wxSocketBase::Notify* (p. 1209), *wxSocketServer::Accept* (p. 1224)

wxSocketServer::WaitForAccept

bool WaitForAccept(**long** *seconds = -1*, **long** *millisecond = 0*)

This function waits for an incoming connection. Use it if you want to call *Accept* (p. 1224) or *AcceptWith* (p. 1224) with *wait* set to false, to detect when an incoming connection is waiting to be accepted.

Parameters

seconds

> Number of seconds to wait. If -1, it will wait for the default timeout, as set with *SetTimeout* (p. 1212).

millisecond

> Number of milliseconds to wait.

Return value

Returns true if an incoming connection arrived, false if the timeout elapsed.

See also

wxSocketServer::Accept (p. 1224), *wxSocketServer::AcceptWith* (p. 1224),*wxSocketBase::InterruptWait* (p. 1208)

wxSound

This class represents a short sound (loaded from Windows WAV file), that can be stored in memory and played. Currently this class is implemented on Windows and Unix (and uses either Open Sound System (`http://www.opensound.com/oss.html`) or Simple DirectMedia Layer (`http://www.libsdl.org/`)).

Derived from

wxObject (p. 1012)

Include files

<wx/sound.h>

wxSound::wxSound

 wxSound()

Default constructor.

 wxSound(**const wxString&** *fileName*, **bool** *isResource = false*)

Constructs a wave object from a file or resource. Call *wxSound::IsOk* (p. 1226) to determine whether this succeeded.

Parameters

fileName

The filename or Windows resource.

isResource

`true` if *fileName* is a resource, `false` if it is a filename.

wxSound::~wxSound

~wxSound()

Destroys the wxSound object.

wxSound::Create

bool Create(**const wxString&** *fileName*, **bool** *isResource* = `false`)

Constructs a wave object from a file or resource.

Parameters

fileName

The filename or Windows resource.

isResource

`true` if *fileName* is a resource, `false` if it is a filename.

Return value

`true` if the call was successful, `false` otherwise.

wxSound::IsOk

bool IsOk() **const**

Returns `true` if the object contains a successfully loaded file or resource, `false` otherwise.

wxSound::IsPlaying

static bool IsPlaying() **const**

Returns `true` if a sound is played at the moment.

wxSound::Play

bool Play(**unsigned** *flags* = *wxSOUND_ASYNC*) **const**

static bool Play(**const wxString&** *filename*, **unsigned** *flags* = *wxSOUND_ASYNC*)

Plays the sound file. If another sound is playing, it will be interrupted. Returns `true` on success, `false` otherwise.

The possible values for *flags* are:

wxSOUND_SYNC   `Play` will block and wait until the sound is replayed.

wxSOUND_ASYNC   Sound is played asynchronously, `Play` returns immediately

wxSOUND_ASYNC | wxSOUND_LOOP   Sound is played asynchronously and loops until another sound is played, *wxSound::Stop* (p. 1227) is called or the program terminates.

The static form is shorthand for this code:

```
wxSound(filename).Play(flags);
```

wxSound::Stop

static void Stop()

If a sound is played, this function stops it.

wxSpinButton

A wxSpinButton has two small up and down (or left and right) arrow buttons. It is often used next to a text control for increment and decrementing a value. Portable programs should try to use *wxSpinCtrl* (p. 1231) instead as wxSpinButton is not implemented for all platforms.

NB: the range supported by this control (and wxSpinCtrl) depends on the platform but is at least $-0x8000$ to $0x7fff$. Under GTK and Win32 with sufficiently new version of `comctrl32.dll` (at least 4.71 is required, 5.80 is recommended) the full 32 bit range is supported.

Derived from

wxControl (p. 215)
wxWindow (p. 1490)

wxEvtHandler (p. 486)
wxObject (p. 1012)

See also

wxSpinCtrl (p. 1231)

Include files

<wx/spinbutt.h>

Window styles

wxSP_HORIZONTAL	Specifies a horizontal spin button (note that this style is not supported in wxGTK).
wxSP_VERTICAL	Specifies a vertical spin button.
wxSP_ARROW_KEYS	The user can use arrow keys to change the value.
wxSP_WRAP	The value wraps at the minimum and maximum.

See also *window styles overview* (p. 1762).

Event handling

To process input from a spin button, use one of these event handler macros to direct input to member functions that take a *wxSpinEvent* (p. 1234) argument:

EVT_SPIN(id, func)	Generated whenever an arrow is pressed.
EVT_SPIN_UP(id, func)	Generated when left/up arrow is pressed.
EVT_SPIN_DOWN(id, func)	Generated when right/down arrow is pressed.

Note that if you handle both SPIN and UP or DOWN events, you will be notified about each of them twice: first the UP/DOWN event will be receieved and then, if it wasn't vetoed, the SPIN event will be sent.**See also**
Event handling overview (p. 1751)

wxSpinButton::wxSpinButton

wxSpinButton()

Default constructor.

wxSpinButton(**wxWindow*** *parent*, **wxWindowID** *id*, **const wxPoint&** *pos =*
wxDefaultPosition, **const wxSize&** *size = wxDefaultSize*, **long** *style =*
wxSP_HORIZONTAL, **const wxString&** *name = "spinButton"*)

Constructor, creating and showing a spin button.

Parameters

parent

>   Parent window. Must not be NULL.

id

>   Window identifier. A value of -1 indicates a default value.

pos

>   Window position. If the position (-1, -1) is specified then a default position is chosen.

size

>   Window size. If the default size (-1, -1) is specified then a default size is chosen.

style

>   Window style. See *wxSpinButton* (p. 1227).

validator

>   Window validator.

name

>   Window name.

See also

wxSpinButton::Create (p. 1229), *wxValidator* (p. 1461)

wxSpinButton::~wxSpinButton

void ~wxSpinButton()

Destructor, destroys the spin button control.

wxSpinButton::Create

bool Create(**wxWindow*** *parent*, **wxWindowID** *id*, **const wxPoint&** *pos* = *wxDefaultPosition*, **const wxSize&** *size* = *wxDefaultSize*, **long** *style* = *wxSP_HORIZONTAL*, **const wxString&** *name* = *"spinButton"*)

Scrollbar creation function called by the spin button constructor. See *wxSpinButton::wxSpinButton* (p. 1228) for details.

wxSpinButton::GetMax

int GetMax() **const**

Returns the maximum permissible value.

See also

wxSpinButton::SetRange (p. 1230)

wxSpinButton::GetMin

int GetMin() **const**

Returns the minimum permissible value.

See also

wxSpinButton::SetRange (p. 1230)

wxSpinButton::GetValue

int GetValue() **const**

Returns the current spin button value.

See also

wxSpinButton::SetValue (p. 1230)

wxSpinButton::SetRange

void SetRange(**int** *min*, **int** *max*)

Sets the range of the spin button.

Parameters

min

The minimum value for the spin button.

max

The maximum value for the spin button.

See also

wxSpinButton::GetMin (p. 1230), *wxSpinButton::GetMax* (p. 1229)

wxSpinButton::SetValue

void SetValue(**int** *value*)

Sets the value of the spin button.

Parameters

value

The value for the spin button.

See also

wxSpinButton::GetValue (p. 1230)

wxSpinCtrl

wxSpinCtrl combines *wxTextCtrl* (p. 1340) and *wxSpinButton* (p. 1227) in one control.

Derived from

wxControl (p. 215)
wxWindow (p. 1490)
wxEvtHandler (p. 486)
wxObject (p. 1012)

Include files

<wx/spinctrl.h>

Window styles

wxSP_ARROW_KEYS	The user can use arrow keys to change the value.
wxSP_WRAP	The value wraps at the minimum and maximum.

Event handling

To process input from a spin button, use one of these event handler macros to direct input to member functions that take a *wxSpinEvent* (p. 1234) argument:

EVT_SPINCTRL(id, func)	Generated whenever the numeric value of the spinctrl is updated

You may also use the *wxSpinButton* (p. 1227) event macros, however the corresponding events will not be generated under all platforms. Finally, if the user modifies the text in the edit part of the spin control directly, the EVT_TEXT is generated, like for the *wxTextCtrl* (p. 1340).

See also

Event handling overview (p. 1751),*wxSpinButton* (p. 1227),*wxControl* (p. 215)

wxSpinCtrl::wxSpinCtrl

wxSpinCtrl()

Default constructor.

wxSpinCtrl(**wxWindow*** *parent*, **wxWindowID** *id = -1*, **const wxString&** *value = wxEmptyString*, **const wxPoint&** *pos = wxDefaultPosition*, **const wxSize&** *size = wxDefaultSize*, **long** *style = wxSP_ARROW_KEYS*, **int** *min = 0*, **int** *max = 100*, **int** *initial = 0*, **const wxString&** *name = _T("wxSpinCtrl")*)

Constructor, creating and showing a spin control.

Parameters

parent

> Parent window. Must not be NULL.

value

> Default value.

id

> Window identifier. A value of -1 indicates a default value.

pos

> Window position. If the position (-1, -1) is specified then a default position is chosen.

size

> Window size. If the default size (-1, -1) is specified then a default size is chosen.

style

> Window style. See *wxSpinButton* (p. 1227).

min

> Minimal value.

max

> Maximal value.

initial

> Initial value.

name

> Window name.

See also

wxSpinCtrl::Create (p. 1233)

wxSpinCtrl::Create

bool Create(**wxWindow*** *parent*, **wxWindowID** *id = -1*, **const wxString&** *value = wxEmptyString*, **const wxPoint&** *pos = wxDefaultPosition*, **const wxSize&** *size = wxDefaultSize*, **long** *style = wxSP_ARROW_KEYS*, **int** *min = 0*, **int** *max = 100*, **int** *initial = 0*, **const wxString&** *name = _T("wxSpinCtrl")*)

Creation function called by the spin control constructor.

See *wxSpinCtrl::wxSpinCtrl* (p. 1232) for details.

wxSpinCtrl::SetValue

void SetValue(**const wxString&** *text*)

void SetValue(**int** *value*)

Sets the value of the spin control.

wxSpinCtrl::GetValue

int GetValue() **const**

Gets the value of the spin control.

wxSpinCtrl::SetRange

void SetRange(**int** *minVal*, **int** *maxVal*)

Sets range of allowable values.

wxSpinCtrl::SetSelection

void SetSelection(**long** *from*, **long** *to*)

Select the text in the text part of the control between positions *from* (inclusive) and *to* (exclusive). This is similar to *wxTextCtrl::SetSelection* (p. 1356).

NB: this is currently only implemented for Windows and generic versions of the control.

wxSpinCtrl::GetMin

int GetMin() **const**

Gets minimal allowable value.

wxSpinCtrl::GetMax

int GetMax() **const**

Gets maximal allowable value.

wxSpinEvent

This event class is used for the events generated by *wxSpinButton* (p. 1227) and *wxSpinCtrl* (p. 1231).

Derived from

wxNotifyEvent (p. 1011)
wxCommandEvent (p. 180)
wxEvent (p. 483)
wxObject (p. 1012)

Include files

<wx/spinbutt.h> or <wx/spinctrl.h>

Event handling

To process input from a spin button, use one of these event handler macros to direct input to member functions that take a *wxSpinEvent* (p. 1234) argument:

EVT_SPIN(id, func)   Generated whenever an arrow is pressed.

EVT_SPIN_UP(id, func)   Generated when left/up arrow is pressed.

EVT_SPIN_DOWN(id, func)   Generated when right/down arrow is pressed.

Note that if you handle both SPIN and UP or DOWN events, you will be notified about each of them twice: first the UP/DOWN event will be receieved and then, if it wasn't vetoed, the SPIN event will be sent.**See also**
wxSpinButton (p. 1227) and *wxSpinCtrl* (p. 1231)

wxSpinEvent::wxSpinEvent

 wxSpinEvent(**wxEventType** *commandType = wxEVT_NULL*, **int** *id = 0*)

The constructor is not normally used by the user code.

wxSpinEvent::GetPosition

int GetPosition() **const**

Retrieve the current spin button or control value.

wxSpinEvent::SetPosition

void SetPosition(**int** *pos*)

Set the value associated with the event.

wxSplashScreen

wxSplashScreen shows a window with a thin border, displaying a bitmap describing your application. Show it in application initialisation, and then either explicitly destroy it or let it time-out.

Example usage:

```
wxBitmap bitmap;
if (bitmap.LoadFile("splash16.png", wxBITMAP_TYPE_PNG))
{
    wxSplashScreen* splash = new wxSplashScreen(bitmap,
        wxSPLASH_CENTRE_ON_SCREEN|wxSPLASH_TIMEOUT,
        6000, NULL, -1, wxDefaultPosition, wxDefaultSize,
        wxSIMPLE_BORDER|wxSTAY_ON_TOP);
}
wxYield();
```

Derived from

wxFrame (p. 579)
wxWindow (p. 1490)
wxEvtHandler (p. 486)
wxObject (p. 1012)

Include files

<wx/splash.h>

wxSplashScreen::wxSplashScreen

 wxSplashScreen(**const wxBitmap&** *bitmap*, **long** *splashStyle*, **int** *milliseconds*, **wxWindow*** *parent*, **wxWindowID** *id*, **const wxPoint&** *pos* = *wxDefaultPosition*, **const wxSize&** *size* = *wxDefaultSize*, **long** *style* = *wxSIMPLE_BORDER|wxFRAME_NO_TASKBAR|wxSTAY_ON_TOP*)

Construct the splash screen passing a bitmap, a style, a timeout, a window id, optional

position and size, and a window style.

splashStyle is a bitlist of some of the following:

- wxSPLASH_CENTRE_ON_PARENT

- wxSPLASH_CENTRE_ON_SCREEN

- wxSPLASH_NO_CENTRE

- wxSPLASH_TIMEOUT

- wxSPLASH_NO_TIMEOUT

milliseconds is the timeout in milliseconds.

wxSplashScreen::~wxSplashScreen

~wxSplashScreen()

Destroys the splash screen.

wxSplashScreen::OnCloseWindow

void OnCloseWindow(**wxCloseEvent&** *event*)

Reimplement this event handler if you want to set an application variable on window destruction, for example.

wxSplashScreen::GetSplashStyle

long GetSplashStyle() **const**

Returns the splash style (see *wxSplashScreen::wxSplashScreen* (p. 1235) for details).

wxSplashScreen::GetSplashWindow

wxSplashScreenWindow* GetSplashWindow() **const**

Returns the window used to display the bitmap.

wxSplashScreen::GetTimeout

int GetTimeout() **const**

Returns the timeout in milliseconds.

wxSplitterEvent

This class represents the events generated by a splitter control. Also there is only one event class, the data associated to the different events is not the same and so not all accessor functions may be called for each event. The documentation mentions the kind of event(s) for which the given accessor function makes sense: calling it for other types of events will result in assert failure (in debug mode) and will return meaningless results.

Derived from

wxNotifyEvent (p. 1011)
wxCommandEvent (p. 180)
wxEvent (p. 483)
wxObject (p. 1012)

Include files

<wx/splitter.h>

Event table macros

To process a splitter event, use these event handler macros to direct input to member functions that take a wxSplitterEvent argument.

EVT_SPLITTER_SASH_POS_CHANGING(id, func)	The sash position is in the process of being changed. You may prevent this change from happening by calling *Veto* (p. 1012) or you may also modify the position of the tracking bar to properly reflect the position that would be set if the drag were to be completed at this point. Processes a wxEVT_COMMAND_SPLITTER_SASH_POS_CHANGING event.
EVT_SPLITTER_SASH_POS_CHANGED(id, func)	The sash position was changed. This event is generated after the user releases the mouse after dragging the splitter. Processes a wxEVT_COMMAND_SPLITTER_SASH_POS_CHANGED event.
EVT_SPLITTER_UNSPLIT(id, func)	The splitter has been just unsplit. Processes a wxEVT_COMMAND_SPLITT

	ER_UNSPLIT event. This event can't be vetoed.
EVT_SPLITTER_DCLICK(id, func)	The sash was double clicked. The default behaviour is to unsplit the window when this happens (unless the minimum pane size has been set to a value greater than zero). This won't happen if you veto this event. Processes a wxEVT_COMMAND_SPLITT ER_DOUBLECLICKED event.

See also
wxSplitterWindow (p. 1239), *Event handling overview* (p. 1751)

wxSplitterEvent::wxSplitterEvent

 wxSplitterEvent(**wxEventType** *eventType = wxEVT_NULL*,
wxSplitterWindow * *splitter = NULL*)

Constructor. Used internally by wxWidgets only.

wxSplitterEvent::GetSashPosition

int GetSashPosition() **const**

Returns the new sash position.

May only be called while processing
wxEVT_COMMAND_SPLITTER_SASH_POS_CHANGING and
wxEVT_COMMAND_SPLITTER_SASH_POS_CHANGED events.

wxSplitterEvent::GetX

int GetX() **const**

Returns the x coordinate of the double-click point.

May only be called while processing
wxEVT_COMMAND_SPLITTER_DOUBLECLICKED events.

wxSplitterEvent::GetY

int GetY() **const**

Returns the y coordinate of the double-click point.

May only be called while processing
wxEVT_COMMAND_SPLITTER_DOUBLECLICKED events.

wxSplitterEvent::GetWindowBeingRemoved

wxWindow* GetWindowBeingRemoved() **const**

Returns a pointer to the window being removed when a splitter window is unsplit.

May only be called while processing wxEVT_COMMAND_SPLITTER_UNSPLIT events.

wxSplitterEvent::SetSashPosition

void SetSashPosition(**int** *pos*)

In the case of wxEVT_COMMAND_SPLITTER_SASH_POS_CHANGED events, sets the new sash position. In the case of wxEVT_COMMAND_SPLITTER_SASH_POS_CHANGING events, sets the new tracking bar position so visual feedback during dragging will represent that change that will actually take place. Set to -1 from the event handler code to prevent repositioning.

May only be called while processing
wxEVT_COMMAND_SPLITTER_SASH_POS_CHANGING and
wxEVT_COMMAND_SPLITTER_SASH_POS_CHANGED events.

Parameters

pos

New sash position.

wxSplitterWindow

wxSplitterWindow overview (p. 1794)

This class manages up to two subwindows. The current view can be split into two programmatically (perhaps from a menu command), and unsplit either programmatically or via the wxSplitterWindow user interface.

Window styles

wxSP_3D	Draws a 3D effect border and sash.
wxSP_3DSASH	Draws a 3D effect sash.

wxSP_3DBORDER	Synonym for wxSP_BORDER.
wxSP_BORDER	Draws a standard border.
wxSP_NOBORDER	No border (default).
wxSP_NO_XP_THEME	Under Windows XP, switches off the attempt to draw the splitter using Windows XP theming, so the borders and sash will take on the pre-XP look.
wxSP_PERMIT_UNSPLIT	Always allow to unsplit, even with the minimum pane size other than zero.
wxSP_LIVE_UPDATE	Don't draw XOR line but resize the child windows immediately.

See also *window styles overview* (p. 1762).

Derived from

wxWindow (p. 1490)
wxEvtHandler (p. 486)
wxObject (p. 1012)

Include files

<wx/splitter.h>

Event handling

To process input from a splitter control, use the following event handler macros to direct input to member functions that take a *wxSplitterEvent* (p. 1237) argument.

EVT_SPLITTER_SASH_POS_CHANGING(id, func)	The sash position is in the process of being changed. May be used to modify the position of the tracking bar to properly reflect the position that would be set if the drag were to be completed at this point. Processes a wxEVT_COMMAND_SPLITTER_SASH_POS_CHANGING event.
EVT_SPLITTER_SASH_POS_CHANGED(id, func)	The sash position was changed. May be used to modify the sash position

	before it is set, or to prevent the change from taking place. Processes a wxEVT_COMMAND_SPLITTER_SASH_POS_CHANGED event.
EVT_SPLITTER_UNSPLIT(id, func)	The splitter has been just unsplit. Processes a wxEVT_COMMAND_SPLITTER_UNSPLIT event.
EVT_SPLITTER_DCLICK(id, func)	The sash was double clicked. The default behaviour is to unsplit the window when this happens (unless the minimum pane size has been set to a value greater than zero). Processes a wxEVT_COMMAND_SPLITTER_DOUBLECLICKED event.

See also
wxSplitterEvent (p. 1237)

wxSplitterWindow::wxSplitterWindow

wxSplitterWindow()

Default constructor.

wxSplitterWindow(**wxWindow*** *parent*, **wxWindowID** *id*, **const wxPoint&** *point* = *wxDefaultPosition*, **const wxSize&** *size* = *wxDefaultSize*, **long** *style=wxSP_3D*, **const wxString&** *name* = *"splitterWindow"*)

Constructor for creating the window.

Parameters

parent

> The parent of the splitter window.

id

> The window identifier.

pos

> The window position.

size

> The window size.

style

> The window style. See *wxSplitterWindow* (p. 1239).

name

> The window name.

Remarks

After using this constructor, you must create either one or two subwindows with the splitter window as parent, and then call one of *wxSplitterWindow::Initialize* (p. 1243), *wxSplitterWindow::SplitVertically* (p. 1248) and *wxSplitterWindow::SplitHorizontally* (p. 1248) in order to set the pane(s).

You can create two windows, with one hidden when not being shown; or you can create and delete the second pane on demand.

See also

wxSplitterWindow::Initialize (p. 1243), *wxSplitterWindow::SplitVertically* (p. 1248), *wxSplitterWindow::SplitHorizontally* (p. 1248), *wxSplitterWindow::Create* (p. 1242)

wxSplitterWindow::~wxSplitterWindow

~wxSplitterWindow()

Destroys the wxSplitterWindow and its children.

wxSplitterWindow::Create

bool Create(**wxWindow*** *parent*, **wxWindowID** *id*, **const wxPoint&** *point* = *wxDefaultPosition*, **const wxSize&** *size* = *wxDefaultSize*, **long** *style=wxSP_3D*, **const wxString&** *name* = *"splitterWindow"*)

Creation function, for two-step construction. See *wxSplitterWindow::wxSplitterWindow* (p. 1241) for details.

wxSplitterWindow::GetMinimumPaneSize

int GetMinimumPaneSize() **const**

Returns the current minimum pane size (defaults to zero).

See also

wxSplitterWindow::SetMinimumPaneSize (p. 1247)

wxSplitterWindow::GetSashGravity

double GetSashGravity()

Returns the current sash gravity.

See also

wxSplitterWindow::SetSashGravity (p. 1246)

wxSplitterWindow::GetSashPosition

int GetSashPosition()

Returns the current sash position.

See also

wxSplitterWindow::SetSashPosition (p. 1246)

wxSplitterWindow::GetSplitMode

int GetSplitMode() **const**

Gets the split mode.

See also

wxSplitterWindow::SetSplitMode (p. 1247), *wxSplitterWindow::SplitVertically* (p. 1248), *wxSplitterWindow::SplitHorizontally* (p. 1248).

wxSplitterWindow::GetWindow1

wxWindow* GetWindow1() **const**

Returns the left/top or only pane.

wxSplitterWindow::GetWindow2

wxWindow* GetWindow2() **const**

Returns the right/bottom pane.

wxSplitterWindow::Initialize

void Initialize(**wxWindow*** *window*)

Initializes the splitter window to have one pane. The child window is shown if it is

currently hidden.

Parameters

window

> The pane for the unsplit window.

Remarks

This should be called if you wish to initially view only a single pane in the splitter window.

See also

wxSplitterWindow::SplitVertically (p. 1248), *wxSplitterWindow::SplitHorizontally* (p. 1248)

wxSplitterWindow::IsSplit

bool IsSplit() **const**

Returns true if the window is split, false otherwise.

wxSplitterWindow::OnDoubleClickSash

virtual void OnDoubleClickSash(**int** *x*, **int** *y*)

Application-overridable function called when the sash is double-clicked with the left mouse button.

Parameters

x

> The x position of the mouse cursor.

y

> The y position of the mouse cursor.

Remarks

The default implementation of this function calls *Unsplit* (p. 1249) if the minimum pane size is zero.

See also

wxSplitterWindow::Unsplit (p. 1249)

wxSplitterWindow::OnUnsplit

virtual void OnUnsplit(**wxWindow*** *removed*)

Application-overridable function called when the window is unsplit, either programmatically or using the wxSplitterWindow user interface.

Parameters

removed

The window being removed.

Remarks

The default implementation of this function simply hides *removed*. You may wish to delete the window.

wxSplitterWindow::OnSashPositionChange

virtual bool OnSashPositionChange(**int** *newSashPosition*)

Application-overridable function called when the sash position is changed by user. It may return false to prevent the change or true to allow it.

Parameters

newSashPosition

The new sash position (always positive or zero)

Remarks

The default implementation of this function verifies that the sizes of both panes of the splitter are greater than minimum pane size.

wxSplitterWindow::ReplaceWindow

bool ReplaceWindow(**wxWindow *** *winOld*, **wxWindow *** *winNew*)

This function replaces one of the windows managed by the wxSplitterWindow with another one. It is in general better to use it instead of calling Unsplit() and then resplitting the window back because it will provoke much less flicker (if any). It is valid to call this function whether the splitter has two windows or only one.

Both parameters should be non-NULL and *winOld* must specify one of the windows managed by the splitter. If the parameters are incorrect or the window couldn't be replaced, false is returned. Otherwise the function will return true, but please notice that it will not delete the replaced window and you may wish to do it yourself.

See also

wxSplitterWindow::GetMinimumPaneSize (p. 1242)

See also

wxSplitterWindow::SetSashGravity

void SetSashGravity(**double** *gravity*)

Sets the sash gravity.

Parameters

gravity

> The sash gravity. Value between 0.0 and 1.0.

RemarksGravity is real factor which controls position of sash while resizing wxSplitterWindow. Gravity tells wxSplitterWindow how much will left/top window grow while resizing.

Example values:

- 0.0 - only the bottom/right window is automaticaly resized

- 0.5 - both windows grow by equal size

- 1.0 - only left/top window grows

Gravity should be real value betwwen 0.0 and 1.0.

Default value of sash gravity is 0.0. That value is compatible with previous (before gravity was introduced) behaviour of wxSplitterWindow.

See also

wxSplitterWindow::GetSashGravity (p. 1243)

wxSplitterWindow::SetSashPosition

void SetSashPosition(**int** *position*, **const bool** *redraw = true*)

Sets the sash position.

Parameters

position

> The sash position in pixels.

redraw

> If true, resizes the panes and redraws the sash and border.

Remarks

Does not currently check for an out-of-range value.

See also

wxSplitterWindow::GetSashPosition (p. 1243)

wxSplitterWindow::SetSashSize

void SetSashSize(**int** *size*)

Sets the sash size. Normally, the sash size is determined according to the metrics of each platform, but the application can override this, for example to show a thin sash that the user is not expected to drag. If *size* is more -1, the custom sash size will be used.

wxSplitterWindow::SetMinimumPaneSize

void SetMinimumPaneSize(**int** *paneSize*)

Sets the minimum pane size.

Parameters

paneSize

Minimum pane size in pixels.

Remarks

The default minimum pane size is zero, which means that either pane can be reduced to zero by dragging the sash, thus removing one of the panes. To prevent this behaviour (and veto out-of-range sash dragging), set a minimum size, for example 20 pixels. If the wxSP_PERMIT_UNSPLIT style is used when a splitter window is created, the window may be unsplit even if minimum size is non-zero.

See also

wxSplitterWindow::GetMinimumPaneSize (p. 1242)

wxSplitterWindow::SetSplitMode

void SetSplitMode(**int** *mode*)

Sets the split mode.

Parameters

mode

Can be wxSPLIT_VERTICAL or wxSPLIT_HORIZONTAL.

Remarks

Only sets the internal variable; does not update the display.

See also

wxSplitterWindow::GetSplitMode (p. 1243), *wxSplitterWindow::SplitVertically* (p. 1248), *wxSplitterWindow::SplitHorizontally* (p. 1248).

wxSplitterWindow::SplitHorizontally

bool SplitHorizontally(**wxWindow*** *window1*, **wxWindow*** *window2*, **int** *sashPosition = 0*)

Initializes the top and bottom panes of the splitter window. The child windows are shown if they are currently hidden.

Parameters

window1

> The top pane.

window2

> The bottom pane.

sashPosition

> The initial position of the sash. If this value is positive, it specifies the size of the upper pane. If it is negative, it is absolute value gives the size of the lower pane. Finally, specify 0 (default) to choose the default position (half of the total window height).

Return value

true if successful, false otherwise (the window was already split).

Remarks

This should be called if you wish to initially view two panes. It can also be called at any subsequent time, but the application should check that the window is not currently split using *IsSplit* (p. 1244).

See also

wxSplitterWindow::SplitVertically (p. 1248), *wxSplitterWindow::IsSplit* (p. 1244), *wxSplitterWindow::Unsplit* (p. 1249)

wxSplitterWindow::SplitVertically

bool SplitVertically(**wxWindow*** *window1*, **wxWindow*** *window2*, **int** *sashPosition = 0*)

Initializes the left and right panes of the splitter window. The child windows are shown if they are currently hidden.

Parameters

window1

> The left pane.

window2

> The right pane.

sashPosition

> The initial position of the sash. If this value is positive, it specifies the size of the left pane. If it is negative, it is absolute value gives the size of the right pane. Finally, specify 0 (default) to choose the default position (half of the total window width).

Return value

true if successful, false otherwise (the window was already split).

Remarks

This should be called if you wish to initially view two panes. It can also be called at any subsequent time, but the application should check that the window is not currently split using *IsSplit* (p. 1244).

See also

wxSplitterWindow::SplitHorizontally (p. 1248), *wxSplitterWindow::IsSplit* (p. 1244), *wxSplitterWindow::Unsplit* (p. 1249).

wxSplitterWindow::Unsplit

bool Unsplit(**wxWindow*** *toRemove = NULL*)

Unsplits the window.

Parameters

toRemove

> The pane to remove, or NULL to remove the right or bottom pane.

Return value

true if successful, false otherwise (the window was not split).

Remarks

This call will not actually delete the pane being removed; it calls *OnUnsplit* (p. 1244)

which can be overridden for the desired behaviour. By default, the pane being removed is hidden.

See also

wxSplitterWindow::SplitHorizontally (p. 1248), *wxSplitterWindow::SplitVertically* (p. 1248), *wxSplitterWindow::IsSplit* (p. 1244), *wxSplitterWindow::OnUnsplit* (p. 1244)

wxSplitterWindow::UpdateSize

void UpdateSize()

Causes any pending sizing of the sash and child panes to take place immediately.

Such resizing normally takes place in idle time, in order to wait for layout to be completed. However, this can cause unacceptable flicker as the panes are resized after the window has been shown. To work around this, you can perform window layout (for example by sending a size event to the parent window), and then call this function, before showing the top-level window.

wxSplitterRenderParams

This is just a simple `struct` used as a return value of *wxRendererNative::GetSplitterParams* (p. 1125).

It doesn't have any methods and all of its fields are constant and so can be only examined but not modified.

Include files

<wx/renderer.h>

wxEvent::widthSash

const wxCoord widthSash

The width of the splitter sash.

wxSplitterRenderParams::border

const wxCoord border

The width of the border drawn by the splitter inside it, may be 0.

wxSplitterRenderParams::isHotSensitive

const bool isHotSensitive

`true` if the sash changes appearance when the mouse passes over it, `false`otherwise.

wxStackFrame

wxStackFrame represents a single stack frame, or a single function in the call stack, and is used exclusively together with *wxStackWalker* (p. 1253), see there for a more detailed discussion.

Derived from

No base class

Include files

<wx/stackwalk.h>

Only available if `wxUSE_STACKWALKER` is 1, currently only implemented for Win32 and Unix versions using recent version of GNU libc.

See also

wxStackWalker (p. 1253)

wxStackFrame::GetAddress

void* GetAddress() **const**

Return the address of this frame.

wxStackFrame::GetFileName

wxString GetFileName() **const**

Return the name of the file containing this frame, empty if unavailable (typically because debug info is missing).

Use *HasSourceLocation* (p. 1253) to check whether the file name is available.

wxStackFrame::GetLevel

size_t GetLevel() **const**

Get the level of this frame (deepest/innermost one is 0).

wxStackFrame::GetLine

size_t GetLine() **const**

Return the line number of this frame, 0 if unavailable.

See also

GetFileName (p. 1251)

wxStackFrame::GetModule

wxString GetModule() **const**

Get the module this function belongs to (empty if not available).

wxStackFrame::GetName

wxString GetName() **const**

Return the unmangled (if possible) name of the function containing this frame.

wxStackFrame::GetOffset

size_t GetOffset() **const**

Return the return address of this frame.

wxStackFrame::GetParam

bool GetParam(**size_t** *n*, **wxString *** *type*, **wxString *** *name*, **wxString *** *value*) **const**

Get the name, type and value (in text form) of the given parameter. Any pointer may be NULL if you're not interested in the corresponding value.

Return true if at least some values could be retrieved.

This function currently is only implemented under Win32 and requires a PDB file.

wxStackFrame::GetParamCount

size_t GetParamCount() **const**

Return the number of parameters of this function (may return 0 if we can't retrieve the parameters info even although the function does have parameters).

wxStackFrame::HasSourceLocation

bool HasSourceLocation() **const**

Return `true` if we have the file name and line number for this frame.

wxStackWalker

wxStackWalker allows an application to enumerate, or walk, the stack frames (the function callstack). It is mostly useful in only two situations: inside *wxApp::OnFatalException* (p. 37) function to programmatically get the location of the crash and, in debug builds, in *wxApp::OnAssert* (p. 35) to report the caller of the failed assert.

wxStackWalker works by repeatedly calling the *OnStackFrame* (p. 1254) method for each frame in the stack, so to use it you must derive your own class from it and override this method.

This class will not return anything except raw stack frame addresses if the debug information is not available. Under Win32 this means that the PDB file matching the program being executed should be present. Note that if you use Microsoft Visual C++ compiler, you can create PDB files even for the programs built in release mode and it doesn't affect the program size (at least if you don't forget to add `/opt:ref` option which is suppressed by using`/debug` linker option by default but should be always enabled for release builds). Under Unix, you need to compile your program with debugging information (usually using `-g` compiler and linker options) to get the file and line numbers information, however function names should be available even without it. Of course, all this is only true if you build using a recent enough version of GNU libc which provides the `backtrace()` function needed to walk the stack.

debugging overview (p. 1746) for how to make it available.

Derived from

No base class

Include files

<wx/stackwalk.h>

Only available if `wxUSE_STACKWALKER` is 1, currently only implemented for Win32 and Unix versions using recent version of GNU libc.

See also

wxStackFrame (p. 1251)

wxStackWalker::wxStackWalker

wxStackWalker()

Constructor does nothing, use *Walk()* (p. 1254) to walk the stack.

wxStackWalker::~wxStackWalker

~wxStackWalker()

Destructor does nothing neither but should be virtual as this class is used as a base one.

wxStackWalker::OnStackFrame

void OnStackFrame(**const wxStackFrame&** *frame*)

This function must be overridden to process the given frame.

wxStackWalker::Walk

void Walk(**size_t** *skip = 1*)

Enumerate stack frames from the current location, skipping the initial number of them (this can be useful when Walk() is called from some known location and you don't want to see the first few frames anyhow; also notice that Walk() frame itself is not included if skip $\geq$ 1).

wxStackWalker::WalkFromException

void WalkFromException()

Enumerate stack frames from the location of uncaught exception. This method can only be called from *wxApp::OnFatalException()* (p. 37).

wxStandardPaths

wxStandardPaths returns the standard locations in the file system and should be used by applications to find their data files in a portable way.

Please note that this is not a real class because object of this type are never created but more a namespace containing the class methods which are all static, so to use wxStandardPaths simply call its methods directly.

In the description of the methods below, the example return values are given for the Unix, Windows and Mac OS X systems, however please note that these are just the examples and the actual values may differ. For example, under Windows: the system administrator may change the standard directories locations, i.e. the Windows directory may be named `W:\Win2003` instead of the default `C:\Windows`.

The strings *appname* and *username* should be replaced with the value returned by *wxApp::GetAppName* (p. 33) and the name of the currently logged in user, respectively. The string *prefix* is only used under Unix and is `/usr/local` by default but may be changed using *SetInstallPrefix* (p. 1257).

The directories returned by the methods of this class may or may not exist. If they don't exist, it's up to the caller to create them, wxStandardPaths doesn't do it.

Finally note that these functions only work with standardly packaged applications. I.e. under Unix you should follow the standard installation conventions and under Mac you should create your application bundle according to the Apple guidelines. Again, this class doesn't help you to do it.

This class is MT-safe: its methods may be called concurrently from different threads without additional locking.

Derived from

No base class

Include files

<wx/stdpaths.h>

wxStandardPaths::GetConfigDir

static wxString GetConfigDir()

Return the directory containing the system config files.

Example return values:

- Unix: `/etc`

- Windows: `C:\Documents and Settings\All Users\Application Data`

- Mac: `/Library/Preferences`

See also

wxFileConfig (p. 509)

wxStandardPaths::GetDataDir

static wxString GetDataDir()

Return the location of the applications global, i.e. not user-specific, data files.

Example return values:

- Unix: *prefix*/share/*appname*

- Windows: the directory where the executable file is located

- Mac: *appname*.app/Contents/SharedSupport bundle subdirectory

See also

GetLocalDataDir (p. 1256)

wxStandardPaths::GetInstallPrefix

static wxString GetInstallPrefix()

Note: This function is only available under Unix.

Return the program installation prefix, e.g. /usr, /opt or /home/zeitlin.

If the prefix had been previously by *SetInstallPrefix* (p. 1257), returns that value, otherwise tries to determine it automatically (Linux only right now) and finally returns the default /usr/local value if it failed.

wxStandardPaths::GetLocalDataDir

static wxString GetLocalDataDir()

Return the location for application data files which are host-specific and can't, or shouldn't, be shared with the other machines.

This is the same as *GetDataDir()* (p. 1255) except under Unix where it returns /etc/*appname*.

wxStandardPaths::GetPluginsDir

static wxString GetPluginsDir()

Return the directory where the loadable modules (plugins) live.

Example return values:

- Unix: *prefix*/lib/*appname*

- Windows: the directory of the executable file

- Mac: *appname*.app/Contents/PlugIns bundle subdirectory

See also

wxDynamicLibrary (p. 474)

wxStandardPaths::GetUserConfigDir

static wxString GetUserConfigDir()

Return the directory for the user config files:

- Unix: ~ (the home directory)

- Windows: `C:\Documents and Settings\`*username*

- Mac: ~/Library/Preferences

Only use this method if you have a single configuration file to put in this directory, otherwise *GetUserDataDir()* (p. 1257) is more appropriate.

wxStandardPaths::GetUserDataDir

static wxString GetUserDataDir()

Return the directory for the user-dependent application data files:

- Unix: ~/.*appname*

- Windows: `C:\Documents and Settings\`*username*`\Application Data\`*appname*

- Mac: ~/Library/Application Support/*appname*

wxStandardPaths::GetUserLocalDataDir

static wxString GetUserLocalDataDir()

Return the directory for user data files which shouldn't be shared with the other machines.

This is the same as *GetUserDataDir()* (p. 1257) for all platforms except Windows where it returns `C:\Documents and Settings\`*username*`\Local Settings\Application Data\`*appname*

wxStandardPaths::SetInstallPrefix

static void SetInstallPrefix(const wxString& *prefix*)

Note: This function is only available under Unix.

Lets wxStandardPaths know about the real program installation prefix on a Unix system. By default, the value returned by *GetInstallPrefix* (p. 1256) is used.

Although under Linux systems the program prefix may usually be determined automatically, portable programs should call this function. Usually the prefix is set during

program configuration if using GNU autotools and so it is enough to pass its value defined in config.h to this function.

wxStaticBitmap

A static bitmap control displays a bitmap. It is meant for display of the small icons in the dialog boxes and is not meant to be a general purpose image display control. In particular, under Windows 9x the size of bitmap is limited to 64*64 pixels and thus you should use your own control if you want to display larger images portably.

Derived from

wxControl (p. 215)
wxWindow (p. 1490)
wxEvtHandler (p. 486)
wxObject (p. 1012)

Include files

<wx/statbmp.h>

Window styles

There are no special styles for this control.

See also *window styles overview* (p. 1762).

See also

wxStaticBitmap (p. 1258), *wxStaticBox* (p. 1260)

Remarks

The bitmap to be displayed should have a small number of colours, such as 16, to avoid palette problems.

wxStaticBitmap::wxStaticBitmap

wxStaticBitmap()

Default constructor.

wxStaticBitmap(**wxWindow*** *parent*, **wxWindowID** *id*, **const wxBitmap&** *label*, **const wxPoint&** *pos*, **const wxSize&** *size = wxDefaultSize*, **long** *style = 0*, **const wxString&** *name = "staticBitmap"*)

Constructor, creating and showing a text control.

Parameters

parent

Parent window. Should not be NULL.

id

Control identifier. A value of -1 denotes a default value.

label

Bitmap label.

pos

Window position.

size

Window size.

style

Window style. See *wxStaticBitmap* (p. 1258).

name

Window name.

See also

wxStaticBitmap::Create (p. 1259)

wxStaticBitmap::Create

bool Create(**wxWindow*** *parent*, **wxWindowID** *id*, **const wxBitmap&** *label*, **const wxPoint&** *pos*, **const wxSize&** *size* = *wxDefaultSize*, **long** *style* = *0*, **const wxString&** *name* = *"staticBitmap"*)

Creation function, for two-step construction. For details see *wxStaticBitmap::wxStaticBitmap* (p. 1258).

wxStaticBitmap::GetBitmap

wxBitmap& GetBitmap() **const**

Returns a reference to the label bitmap.

See also

wxStaticBitmap::SetBitmap (p. 1260)

wxStaticBitmap::SetBitmap

virtual void SetBitmap(**const wxBitmap&** *label*)

Sets the bitmap label.

Parameters

label

The new bitmap.

See also

wxStaticBitmap::GetBitmap (p. 1259)

wxStaticBox

A static box is a rectangle drawn around other panel items to denote a logical grouping of items.

Please note that a static box should **not** be used as the parent for the controls it contains, instead they should be siblings of each other. Although using a static box as a parent might work in some versions of wxWidgets, it results in a crash under, for example, wxGTK.

Also, please note that because of this, the order in which you create new controls is important. Create your wxStaticBox control **before** any siblings that are to appear inside the wxStaticBox in order to preserve the correct Z-Order of controls.

Derived from

wxControl (p. 215)
wxWindow (p. 1490)
wxEvtHandler (p. 486)
wxObject (p. 1012)

Include files

<wx/statbox.h>

Window styles

There are no special styles for this control.

See also *window styles overview* (p. 1762).

See also

wxStaticText (p. 1264)

wxStaticBox::wxStaticBox

wxStaticBox()

Default constructor.

wxStaticBox(**wxWindow*** *parent*, **wxWindowID** *id*, **const wxString&** *label*, **const wxPoint&** *pos* = *wxDefaultPosition*, **const wxSize&** *size* = *wxDefaultSize*, **long** *style* = *0*, **const wxString&** *name* = *"staticBox"*)

Constructor, creating and showing a static box.

Parameters

parent

> Parent window. Must not be NULL.

id

> Window identifier. A value of -1 indicates a default value.

label

> Text to be displayed in the static box, the empty string for no label.

pos

> Window position. If the position (-1, -1) is specified then a default position is chosen.

size

> Checkbox size. If the size (-1, -1) is specified then a default size is chosen.

style

> Window style. See *wxStaticBox* (p. 1260).

name

> Window name.

See also

wxStaticBox::Create (p. 1262)

wxStaticBox::~wxStaticBox

void ~wxStaticBox()

Destructor, destroying the group box.

wxStaticBox::Create

bool Create(**wxWindow*** *parent*, **wxWindowID** *id*, **const wxString&** *label*, **const wxPoint&** *pos* = *wxDefaultPosition*, **const wxSize&** *size* = *wxDefaultSize*, **long** *style* = *0*, **const wxString&** *name* = *"staticBox"*)

Creates the static box for two-step construction. See *wxStaticBox::wxStaticBox* (p. 1261) for further details.

wxStaticBoxSizer

wxStaticBoxSizer is a sizer derived from wxBoxSizer but adds a static box around the sizer. This static box has to be created independently or the sizer may create it itself as a convenience.

Derived from

wxBoxSizer (p. 102)
wxSizer (p. 1176)
wxObject (p. 1012)

Include files

<wx/sizer.h>

See also

wxSizer (p. 1176), *wxStaticBox* (p. 1260), *wxBoxSizer* (p. 102), *Sizer overview* (p. 1771)

wxStaticBoxSizer::wxStaticBoxSizer

 wxStaticBoxSizer(**wxStaticBox*** *box*, **int** *orient*)

 wxStaticBoxSizer(**int** *orient*, **wxWindow** **parent*, **const wxString&** *label* = *wxEmptyString*)

The first constructor uses an already existing static box. It takes the associated static box and the orientation *orient*, which can be either wxVERTICAL or wxHORIZONTAL as parameters.

The second one creates a new static box with the given label and parent window.

wxStaticBoxSizer::GetStaticBox

wxStaticBox* GetStaticBox()

Returns the static box associated with the sizer.

wxStaticLine

A static line is just a line which may be used in a dialog to separate the groups of controls. The line may be only vertical or horizontal.

Derived from

wxControl (p. 215)
wxWindow (p. 1490)
wxEvtHandler (p. 486)
wxObject (p. 1012)

Include files

<wx/statline.h>

Window styles

wxLI_HORIZONTAL   Creates a horizontal line.

wxLI_VERTICAL   Creates a vertical line.

See also

wxStaticBox (p. 1260)

wxStaticLine::wxStaticLine

wxStaticLine()

Default constructor.

wxStaticLine(**wxWindow*** *parent*, **wxWindowID** *id = wxID_ANY*, **const wxPoint&** *pos = wxDefaultPosition*, **const wxSize&** *size = wxDefaultSize*, **long** *style = wxLI_HORIZONTAL*, **const wxString&** *name = "staticLine"*)

Constructor, creating and showing a static line.

Parameters

parent

   Parent window. Must not be NULL.

id

   Window identifier. A value of -1 indicates a default value.

pos

   Window position. If the position (-1, -1) is specified then a default position is

chosen.

size

Size. Note that either the height or the width (depending on whether the line if horizontal or vertical) is ignored.

style

Window style (either wxLI_HORIZONTAL or wxLI_VERTICAL).

name

Window name.

See also

wxStaticLine::Create (p. 1264)

wxStaticLine::Create

bool Create(**wxWindow*** *parent*, **wxWindowID** *id = wxID_ANY*, **const wxPoint&** *pos = wxDefaultPosition*, **const wxSize&** *size = wxDefaultSize*, **long** *style = 0*, **const wxString&** *name = "staticLine"*)

Creates the static line for two-step construction. See *wxStaticLine::wxStaticLine* (p. 1263) for further details.

wxStaticLine::IsVertical

bool IsVertical() **const**

Returns true if the line is vertical, false if horizontal.

wxStaticLine::GetDefaultSize

int GetDefaultSize()

This static function returns the size which will be given to the smaller dimension of the static line, i.e. its height for a horizontal line or its width for a vertical one.

wxStaticText

A static text control displays one or more lines of read-only text.

Derived from

wxControl (p. 215)
wxWindow (p. 1490)

wxEvtHandler (p. 486)
wxObject (p. 1012)

Include files

<wx/stattext.h>

Window styles

wxALIGN_LEFT   Align the text to the left

wxALIGN_RIGHT   Align the text to the right

wxALIGN_CENTRE   Center the text (horizontally)

wxST_NO_AUTORESIZE   By default, the control will adjust its size to exactly fit to the size of the text when *SetLabel* (p. 1266) is called. If this style flag is given, the control will not change its size (this style is especially useful with controls which also have wxALIGN_RIGHT or CENTER style because otherwise they won't make sense any longer after a call to SetLabel)

See also *window styles overview* (p. 1762).

See also

wxStaticBitmap (p. 1258), *wxStaticBox* (p. 1260)

wxStaticText::wxStaticText

 wxStaticText()

Default constructor.

 wxStaticText(**wxWindow*** *parent*, **wxWindowID** *id*, **const wxString&** *label*, **const wxPoint&** *pos* = *wxDefaultPosition*, **const wxSize&** *size* = *wxDefaultSize*, **long** *style* = *0*, **const wxString&** *name* = *"staticText"*)

Constructor, creating and showing a text control.

Parameters

parent

   Parent window. Should not be NULL.

id

   Control identifier. A value of -1 denotes a default value.

label

Text label.

pos

Window position.

size

Window size.

style

Window style. See *wxStaticText* (p. 1264).

name

Window name.

See also

wxStaticText::Create (p. 1266)

wxStaticText::Create

bool Create(**wxWindow*** *parent*, **wxWindowID** *id*, **const wxString&** *label*, **const wxPoint&** *pos* = *wxDefaultPosition*, **const wxSize&** *size* = *wxDefaultSize*, **long** *style* = *0*, **const wxString&** *name* = *"staticText"*)

Creation function, for two-step construction. For details see *wxStaticText::wxStaticText* (p. 1265).

wxStaticText::GetLabel

wxString GetLabel() **const**

Returns the contents of the control.

wxStaticText::SetLabel

virtual void SetLabel(**const wxString&** *label*)

Sets the static text label and updates the controls size to exactly fit the label unless the control has wxST_NO_AUTORESIZE flag.

Parameters

label

The new label to set. It may contain newline characters.

wxStatusBar

A status bar is a narrow window that can be placed along the bottom of a frame to give small amounts of status information. It can contain one or more fields, one or more of which can be variable length according to the size of the window.

wxWindow (p. 1490)
wxEvtHandler (p. 486)
wxObject (p. 1012)

Derived from

wxWindow (p. 1490)
wxEvtHandler (p. 486)
wxObject (p. 1012)

Include files

<wx/statusbr.h>

Window styles

wxST_SIZEGRIP   On Windows 95, displays a gripper at right-hand side of the status bar.

See also *window styles overview* (p. 1762).

Remarks

It is possible to create controls and other windows on the status bar. Position these windows from an **OnSize** event handler.

See also

wxFrame (p. 579), *Status bar sample* (p. 1715)

wxStatusBar::wxStatusBar

 wxStatusBar()

Default constructor.

 wxStatusBar(**wxWindow*** *parent*, **wxWindowID** *id*, **long** *style* = *wxST_SIZEGRIP*, **const wxString&** *name* = *"statusBar"*)

Constructor, creating the window.

Parameters

parent

The window parent, usually a frame.

id

The window identifier. It may take a value of -1 to indicate a default value.

style

The window style. See *wxStatusBar* (p. 1267).

name

The name of the window. This parameter is used to associate a name with the item, allowing the application user to set Motif resource values for individual windows.

See also

wxStatusBar::Create (p. 1268)

wxStatusBar::~wxStatusBar

void ~wxStatusBar()

Destructor.

wxStatusBar::Create

bool Create(**wxWindow*** *parent*, **wxWindowID** *id*, **long** *style* = *wxST_SIZEGRIP*, **const wxString&** *name* = *"statusBar"*)

Creates the window, for two-step construction.

See *wxStatusBar::wxStatusBar* (p. 1267) for details.

wxStatusBar::GetFieldRect

virtual bool GetFieldRect(**int** *i*, **wxRect&** *rect*) **const**

Returns the size and position of a field's internal bounding rectangle.

Parameters

i

The field in question.

rect

The rectangle values are placed in this variable.

Return value

true if the field index is valid, false otherwise.

See also

wxRect (p. 1099)

wxPerl note: In wxPerl this function returns a `Wx::Rect` if the field index is valid, `undef` otherwise.

wxStatusBar::GetFieldsCount

int GetFieldsCount() **const**

Returns the number of fields in the status bar.

wxStatusBar::GetStatusText

virtual wxString GetStatusText(**int** *i = 0*) **const**

Returns the string associated with a status bar field.

Parameters

i

> The number of the status field to retrieve, starting from zero.

Return value

The status field string if the field is valid, otherwise the empty string.

See also

wxStatusBar::SetStatusText (p. 1270)

wxStatusBar::PopStatusText

void PopStatusText(**int** *field = 0*)

Sets the field text to the top of the stack, and pops the stack of saved strings.

See also

wxStatusBar::PushStatusText (p. 1269)

wxStatusBar::PushStatusText

void PushStatusText(**const wxString&** *string*, **int** *field = 0*)

Saves the current field text in a per field stack, and sets the field text to the string passed as argument.

wxStatusBar::SetFieldsCount

virtual void SetFieldsCount(**int** *number = 1*, **int*** *widths = NULL*)

Sets the number of fields, and optionally the field widths.

wxPython note: Only the first parameter is accepted. Use SetStatusWidths to set the widths of the fields.

wxPerl note: In wxPerl this function accepts only the `n` parameter. Use SetStatusWidths to set the field widths.

Parameters

number

> The number of fields.

widths

> An array of *n* integers interpreted in the same way as in *SetStatusWidths* (p. 1271)

wxStatusBar::SetMinHeight

void SetMinHeight(**int** *height*)

Sets the minimal possible height for the status bar. The real height may be bigger than the height specified here depending on the size of the font used by the status bar.

wxStatusBar::SetStatusText

virtual void SetStatusText(**const wxString&** *text*, **int** *i = 0*)

Sets the text for one field.

Parameters

text

> The text to be set. Use an empty string ("") to clear the field.

i

> The field to set, starting from zero.

See also

wxStatusBar::GetStatusText (p. 1269), *wxFrame::SetStatusText* (p. 589)

wxStatusBar::SetStatusWidths

virtual void SetStatusWidths(**int** *n*, **int ****widths*)

Sets the widths of the fields in the status line. There are two types of fields: fixed widths one and variable width fields. For the fixed width fields you should specify their (constant) width in pixels. For the variable width fields, specify a negative number which indicates how the field should expand: the space left for all variable width fields is divided between them according to the absolute value of this number. A variable width field with width of -2gets twice as much of it as a field with width -1 and so on.

For example, to create one fixed width field of width 50 in the right part of the status bar and two more fields which get 66% and 33% of the remaining space correspondingly, you should use an array containing -2, -1 and 100.

Parameters

n

> The number of fields in the status bar. Must be equal to the number passed to *SetFieldsCount* (p. 1270) the last time it was called.

widths

> Contains an array of *n* integers, each of which is either an absolute status field width in pixels if positive or indicates a variable width field if negative.

Remarks

The widths of the variable fields are calculated from the total width of all fields, minus the sum of widths of the non-variable fields, divided by the number of variable fields.

See also

wxStatusBar::SetFieldsCount (p. 1270), *wxFrame::SetStatusWidths* (p. 589)

wxPython note: Only a single parameter is required, a Python list of integers.

wxPerl note: In wxPerl this method takes as parameters the field widths.

wxStatusBar::SetStatusStyles

virtual void SetStatusStyles(**int** *n*, **int ****styles*)

Sets the styles of the fields in the status line which can make fields appear flat or raised instead of the standard sunken 3D border.

Parameters

n

> The number of fields in the status bar. Must be equal to the number passed to *SetFieldsCount* (p. 1270) the last time it was called.

styles

Contains an array of *n* integers with the styles for each field. There are three possible styles:

wxSB_NORMAL   (default) The field appears sunken with a standard 3D border.

wxSB_FLAT   No border is painted around the field so that it appears flat.

wxSB_RAISED   A raised 3D border is painted around the field.

wxStdDialogButtonSizer

This class creates button layouts which conform to the standard button spacing and ordering defined by the platform or toolkit's user interface guidelines (if such things exist). By using this class, you can ensure that all your standard dialogs look correct on all major platforms. Currently it conforms to the Windows, GTK+ and Mac OS X human interface guidelines.

When there aren't interface guidelines defined for a particular platform or toolkit, wxStdDialogButtonSizer reverts to the Windows implementation.

To use this class, first add buttons to the sizer by calling AddButton (or SetAffirmativeButton, SetNegativeButton, or SetCancelButton) and then call Realize in order to create the actual button layout used. Other than these special operations, this sizer works like any other sizer.

If you add a button with wxID_SAVE, on Mac OS X the button will be renamed to "Save" and the wxID_NO button will be renamed to "Don't Save" in accordance with the Mac OS X Human Interface Guidelines.

Derived from

wxBoxSizer (p. 102)
wxSizer (p. 1176)
wxObject (p. 1012)

Include files

<wx/sizer.h>

See also

wxSizer (p. 1176), *Sizer overview* (p. 1771), *wxDialog::CreateButtonSizer* (p. 409)

wxStdDialogButtonSizer::wxStdDialogButtonSizer

wxStdDialogButtonSizer()

Constructor for a wxStdDialogButtonSizer.

wxStdDialogButtonSizer::AddButton

void AddButton(**wxButton*** *button*)

Adds a button to the wxStdDialogButtonSizer. The button must have one of the following identifiers:

- wxID_OK
- wxID_YES
- wxID_SAVE
- wxID_APPLY
- wxID_NO
- wxID_CANCEL
- wxID_HELP
- wxID_CONTEXT_HELP

wxStdDialogButtonSizer::Realize

void Realize()

Rearranges the buttons and applies proper spacing between buttons to make them match the platform or toolkit's interface guidelines.

wxStdDialogButtonSizer::SetAffirmativeButton

void SetAffirmativeButton(**wxButton*** *button*)

Sets the affirmative button for the sizer. This allows you to use identifiers other than the standard identifiers outlined above.

wxStdDialogButtonSizer::SetCancelButton

void SetCancelButton(**wxButton*** *button*)

Sets the cancel button for the sizer. This allows you to use identifiers other than the standard identifiers outlined above.

wxStdDialogButtonSizer::SetNegativeButton

void SetNegativeButton(**wxButton*** *button*)

Sets the negative button for the sizer. This allows you to use identifiers other than the standard identifiers outlined above.

wxStopWatch

The wxStopWatch class allow you to measure time intervals. For example, you may use it to measure the time elapsed by some function:

```
wxStopWatch sw;
CallLongRunningFunction();
wxLogMessage("The long running function took %ldms to execute",
             sw.Time());
sw.Pause();
... stopwatch is stopped now ...
sw.Resume();
CallLongRunningFunction();
wxLogMessage("And calling it twice took $ldms in all", sw.Time());
```

Include files

<wx/timer.h>

See also

::wxStartTimer (p. 1656), *::wxGetElapsedTime* (p. 1654), *wxTimer* (p. 1386)

wxStopWatch::wxStopWatch

wxStopWatch()

Constructor. This starts the stop watch.

wxStopWatch::Pause

void Pause()

Pauses the stop watch. Call *wxStopWatch::Resume* (p. 1274) to resume time measuring again.

If this method is called several times, `Resume()` must be called the same number of times to really resume the stop watch. You may, however, call *Start* (p. 1275) to resume it unconditionally.

wxStopWatch::Resume

void Resume()

Resumes the stop watch which had been paused with *wxStopWatch::Pause* (p. 1274).

wxStopWatch::Start

void Start(**long** *milliseconds = 0*)

(Re)starts the stop watch with a given initial value.

wxStopWatch::Time

long Time() **const**

Returns the time in milliseconds since the start (or restart) or the last call of *wxStopWatch::Pause* (p. 1274).

wxStreamBase

This class is the base class of most stream related classes in wxWidgets. It must not be used directly.

Derived from

None

Include files

<wx/stream.h>

See also

wxStreamBuffer (p. 1277)

wxStreamBase::wxStreamBase

wxStreamBase()

Creates a dummy stream object. It doesn't do anything.

wxStreamBase::~wxStreamBase

~wxStreamBase()

Destructor.

wxStreamBase::GetLength

wxFileOffset GetLength() **const**

Returns the length of the stream in bytes. If the length cannot be determined (this is always the case for socket streams for example), returns `wxInvalidOffset.`

This function is new since wxWidgets version 2.5.4

wxStreamBase::GetLastError

wxStreamError GetLastError() **const**

This function returns the last error.

wxSTREAM_NO_ERROR   No error occurred.

wxSTREAM_EOF   An End-Of-File occurred.

wxSTREAM_WRITE_ERROR   A generic error occurred on the last write call.

wxSTREAM_READ_ERROR   A generic error occurred on the last read call.

wxStreamBase::GetSize

size_t GetSize() **const**

GetLength (p. 1276)

This function returns the size of the stream. For example, for a file it is the size of the file.

Warning

There are streams which do not have size by definition, such as socket streams. In that cases, GetSize returns 0 so you should always test its return value.

wxStreamBase::IsOk

bool IsOk() **const**

Returns true if no error occurred on the stream.

See also

GetLastError (p. 1276)

wxStreamBase::IsSeekable

bool IsSeekable() **const**

Returns true if the streams supports seeking to arbitrary offsets.

wxStreamBase::OnSysRead

size_t OnSysRead(**void*** *buffer*, **size_t** *bufsize*)

Internal function. It is called when the stream wants to read data of the specified size. It should return the size that was actually read.

wxStreamBase::OnSysSeek

off_t OnSysSeek(**off_t** *pos*, **wxSeekMode** *mode*)

Internal function. It is called when the stream needs to change the current position.

wxStreamBase::OnSysTell

off_t OnSysTell() **const**

Internal function. Is is called when the stream needs to know the real position.

wxStreamBase::OnSysWrite

size_t OnSysWrite(**void ****buffer*, **size_t** *bufsize*)

See *OnSysRead* (p. 1277).

wxStreamBuffer

Derived from

None

Include files

<wx/stream.h>

See also

wxStreamBase (p. 1275)

wxStreamBuffer::wxStreamBuffer

wxStreamBuffer(**wxStreamBase&** *stream*, **BufMode** *mode*)

Constructor, creates a new stream buffer using *stream* as a parent stream and *mode* as the IO mode. *mode* can be: wxStreamBuffer::read, wxStreamBuffer::write, wxStreamBuffer::read_write.

One stream can have many stream buffers but only one is used internally to pass IO call (e.g. wxInputStream::Read() -> wxStreamBuffer::Read()), but you can call directly wxStreamBuffer::Read without any problems. Note that all errors and messages linked to the stream are stored in the stream, not the stream buffers:

```
streambuffer.Read(...);
streambuffer2.Read(...); /* This call erases previous error messages set
by
                        ``streambuffer'' */
```

wxStreamBuffer(**BufMode** *mode*)

Constructor, creates a new empty stream buffer which won't flush any data to a stream. *mode* specifies the type of the buffer (read, write, read_write). This stream buffer has the advantage to be stream independent and to work only on memory buffers but it is still compatible with the rest of the wxStream classes. You can write, read to this special stream and it will grow (if it is allowed by the user) its internal buffer. Briefly, it has all functionality of a "normal" stream.

Warning

The "read_write" mode doesn't currently work for standalone stream buffers.

wxStreamBuffer(**const wxStreamBuffer&***buffer*)

Constructor. It initializes the stream buffer with the data of the specified stream buffer. The new stream buffer has the same attributes, size, position and they share the same buffer. This will cause problems if the stream to which the stream buffer belong is destroyed and the newly cloned stream buffer continues to be used, trying to call functions in the (destroyed) stream. It is advised to use this feature only in very local area of the program.

See also

wxStreamBuffer:SetBufferIO (p. 1280)

wxStreamBuffer::~wxStreamBuffer

wxStreamBuffer(~wxStreamBuffer)

Destructor. It finalizes all IO calls and frees all internal buffers if necessary.

wxStreamBuffer::Read

size_t Read(**void ****buffer*, **size_t** *size*)

Reads a block of the specified *size* and stores the data in *buffer*. This function tries to read from the buffer first and if more data has been requested, reads more data from the

associated stream and updates the buffer accordingly until all requested data is read.

Return value

It returns the size of the data read. If the returned size is different of the specified *size*, an error has occurred and should be tested using *GetLastError* (p. 1276).

size_t Read(**wxStreamBuffer** **buffer*)

Reads a *buffer*. The function returns when *buffer* is full or when there isn't data anymore in the current buffer.

See also

wxStreamBuffer::Write (p. 1279)

wxStreamBuffer::Write

size_t Write(**const void** **buffer*, **size_t** *size*)

Writes a block of the specified *size* using data of *buffer*. The data are cached in a buffer before being sent in one block to the stream.

size_t Write(**wxStreamBuffer** **buffer*)

See *Read* (p. 1278).

wxStreamBuffer::GetChar

char GetChar()

Gets a single char from the stream buffer. It acts like the Read call.

Problem

You aren't directly notified if an error occurred during the IO call.

See also

wxStreamBuffer::Read (p. 1278)

wxStreamBuffer::PutChar

void PutChar(**char** *c*)

Puts a single char to the stream buffer.

Problem

You aren't directly notified if an error occurred during the IO call.

See also

wxStreamBuffer::Read (p. 1279)

wxStreamBuffer::Tell

off_t Tell() **const**

Gets the current position in the stream. This position is calculated from the *real* position in the stream and from the internal buffer position: so it gives you the position in the *real* stream counted from the start of the stream.

Return value

Returns the current position in the stream if possible, wxInvalidOffset in the other case.

wxStreamBuffer::Seek

off_t Seek(**off_t** *pos*, **wxSeekMode** *mode*)

Changes the current position.

mode may be one of the following:

wxFromStart	The position is counted from the start of the stream.
wxFromCurrent	The position is counted from the current position of the stream.
wxFromEnd	The position is counted from the end of the stream.

Return value

Upon successful completion, it returns the new offset as measured in bytes from the beginning of the stream. Otherwise, it returns wxInvalidOffset.

wxStreamBuffer::ResetBuffer

void ResetBuffer()

Resets to the initial state variables concerning the buffer.

wxStreamBuffer::SetBufferIO

void SetBufferIO(**char*** *buffer_start*, **char*** *buffer_end*)

Specifies which pointers to use for stream buffering. You need to pass a pointer on the start of the buffer end and another on the end. The object will use this buffer to cache stream data. It may be used also as a source/destination buffer when you create an

empty stream buffer (See *wxStreamBuffer::wxStreamBuffer* (p. 1277)).

Remarks

When you use this function, you will have to destroy the IO buffers yourself after the stream buffer is destroyed or don't use it anymore. In the case you use it with an empty buffer, the stream buffer will not resize it when it is full.

See also

wxStreamBuffer constructor (p. 1277)
wxStreamBuffer::Fixed (p. 1282)
wxStreamBuffer::Flushable (p. 1282)

void SetBufferIO(**size_t** *bufsize*)

Destroys or invalidates the previous IO buffer and allocates a new one of the specified size.

Warning

All previous pointers aren't valid anymore.

Remark

The created IO buffer is growable by the object.

See also

wxStreamBuffer::Fixed (p. 1282)
wxStreamBuffer::Flushable (p. 1282)

wxStreamBuffer::GetBufferStart

void * GetBufferStart() **const**

Returns a pointer on the start of the stream buffer.

wxStreamBuffer::GetBufferEnd

void * GetBufferEnd() **const**

Returns a pointer on the end of the stream buffer.

wxStreamBuffer::GetBufferPos

void * GetBufferPos() **const**

Returns a pointer on the current position of the stream buffer.

wxStreamBuffer::GetIntPosition

off_t GetIntPosition() **const**

Returns the current position (counted in bytes) in the stream buffer.

wxStreamBuffer::SetIntPosition

void SetIntPosition(**size_t** *pos*)

Sets the current position (in bytes) in the stream buffer.

Warning

Since it is a very low-level function, there is no check on the position: specifying an invalid position can induce unexpected results.

wxStreamBuffer::GetLastAccess

size_t GetLastAccess() **const**

Returns the amount of bytes read during the last IO call to the parent stream.

wxStreamBuffer::Fixed

void Fixed(**bool** *fixed*)

Toggles the fixed flag. Usually this flag is toggled at the same time as *flushable*. This flag allows (when it has the false value) or forbids (when it has the true value) the stream buffer to resize dynamically the IO buffer.

See also

wxStreamBuffer::SetBufferIO (p. 1280)

wxStreamBuffer::Flushable

void Flushable(**bool** *flushable*)

Toggles the flushable flag. If *flushable* is disabled, no data are sent to the parent stream.

wxStreamBuffer::FlushBuffer

bool FlushBuffer()

Flushes the IO buffer.

wxStreamBuffer::FillBuffer

bool FillBuffer()

Fill the IO buffer.

wxStreamBuffer::GetDataLeft

size_t GetDataLeft()

Returns the amount of available data in the buffer.

wxStreamBuffer::Stream

wxStreamBase* Stream()

Returns the parent stream of the stream buffer.

wxStreamToTextRedirector

This class can be used to (temporarily) redirect all output sent to a C++ ostream object to a *wxTextCtrl* (p. 1340) instead.

NB: Some compilers and/or build configurations don't support multiply inheriting *wxTextCtrl* (p. 1340) from `std::streambuf` in which case this class is not compiled in. You also must have `wxUSE_STD_IOSTREAM` option on (i.e. set to 1) in your setup.h to be able to use it. Under Unix, specify `--enable-std_iostreams` switch when running configure for this.

```
Example of usage:  using namespace std;

  wxTextCtrl *text = new wxTextCtrl(...);

  {
    wxStreamToTextRedirector redirect(text);

    // this goes to the text control
    cout << "Hello, text!" << endl;
  }

  // this goes somewhere else, presumably to stdout
  cout << "Hello, console!" << endl;
```

Derived from

No base class

Include files

<wx/textctrl.h>

wxStreamToTextRedirector::wxStreamToTextRedirector

wxStreamToTextRedirector(**wxTextCtrl** *text*, **ostream** *ostr = NULL*)

The constructor starts redirecting output sent to *ostr* or *cout* for the default parameter value to the text control *text*.

Parameters

text

> The text control to append output too, must be non NULL

ostr

> The C++ stream to redirect, *cout* is used if it is NULL

wxStreamToTextRedirector::~wxStreamToTextRedirector

~wxStreamToTextRedirector()

When a wxStreamToTextRedirector object is destroyed, the redirection is ended and any output sent to the C++ ostream which had been specified at the time of the object construction will go to its original destination.

wxString

wxString is a class representing a character string. Please see the *wxString overview* (p. 1721) for more information about it. As explained there, wxString implements about 90% of methods of the std::string class (iterators are not supported, nor all methods which use them). These standard functions are not documented in this manual so please see the STL documentation. The behaviour of all these functions is identical to the behaviour described there (except that wxString is sensitive to null character).

You may notice that wxString sometimes has many functions which do the same thing like, for example, *Length()* (p. 1299), *Len()* (p. 1299) and `length()` which all return the string length. In all cases of such duplication the `std::string`-compatible method (`length()` in this case, always the lowercase version) should be used as it will ensure smoother transition to `std::string` when wxWidgets starts using it instead of wxString.

Derived from

None

Include files

<wx/string.h>

Predefined objects

Objects:

wxEmptyString

See also

Overview (p. 1721)

Constructors and assignment operators

A string may be constructed either from a C string, (some number of copies of) a single character or a wide (UNICODE) string. For all constructors (except the default which creates an empty string) there is also a corresponding assignment operator.

wxString (p. 1291)
operator = (p. 1305)
~wxString (p. 1292)

String length

These functions return the string length and check whether the string is empty or empty it.

Len (p. 1299)
IsEmpty (p. 1298)
operator! (p. 1305)
Empty (p. 1295)
Clear (p. 1294)

Character access

Many functions in this section take a character index in the string. As with C strings and/or arrays, the indices start from 0, so the first character of a string is string[0]. Attempt to access a character beyond the end of the string (which may be even 0 if the string is empty) will provoke an assert failure in *debug build* (p. 1746), but no checks are done in release builds.

This section also contains both implicit and explicit conversions to C style strings. Although implicit conversion is quite convenient, it is advised to use explicit *c_str()* (p. 1294) method for the sake of clarity. Also see *overview* (p. 1722) for the cases where it

is necessary to use it.

GetChar (p. 1297)
GetWritableChar (p. 1297)
SetChar (p. 1302)
Last (p. 1299)
operator [] (p. 1306)
c_str (p. 1294)
mb_str (p. 1300)
wc_str (p. 1305)
fn_str (p. 1296)
operator const char* (p. 1307)

Concatenation

Anything may be concatenated (appended to) with a string. However, you can't append something to a C string (including literal constants), so to do this it should be converted to a wxString first.

operator << (p. 1306)
operator += (p. 1306)
operator + (p. 1305)
Append (p. 1293)
Prepend (p. 1300)

Comparison

The default comparison function *Cmp* (p. 1294) is case-sensitive and so is the default version of *IsSameAs* (p. 1298). For case insensitive comparisons you should use *CmpNoCase* (p. 1294) or give a second parameter to IsSameAs. This last function is may be more convenient if only equality of the strings matters because it returns a boolean true value if the strings are the same and not 0 (which is usually false in C) as `Cmp()` does.

Matches (p. 1300) is a poor man's regular expression matcher: it only understands '*' and '?' metacharacters in the sense of DOS command line interpreter.

StartsWith (p. 1302) is helpful when parsing a line of text which should start with some predefined prefix and is more efficient than doing direct string comparison as you would also have to precalculate the length of the prefix then.

Cmp (p. 1294)
CmpNoCase (p. 1294)
IsSameAs (p. 1298)
Matches (p. 1300)
StartsWith (p. 1302)

Substring extraction

These functions allow to extract substring from this string. All of them don't modify the original string and return a new string containing the extracted substring.

Mid (p. 1300)
operator() (p. 1306)
Left (p. 1299)
Right (p. 1301)
BeforeFirst (p. 1293)
BeforeLast (p. 1294)
AfterFirst (p. 1293)
AfterLast (p. 1293)
StartsWith (p. 1302)

Case conversion

The MakeXXX() variants modify the string in place, while the other functions return a new string which contains the original text converted to the upper or lower case and leave the original string unchanged.

MakeUpper (p. 1300)
Upper (p. 1304)
MakeLower (p. 1299)
Lower (p. 1299)

Searching and replacing

These functions replace the standard *strchr()* and *strstr()* functions.

Find (p. 1295)
Replace (p. 1301)

Conversion to numbers

The string provides functions for conversion to signed and unsigned integer and floating point numbers. All three functions take a pointer to the variable to put the numeric value in and return true if the **entire** string could be converted to a number.

ToLong (p. 1303)
ToULong (p. 1303)
ToDouble (p. 1303)

Writing values into the string

Both formatted versions (*Printf* (p. 1301)) and stream-like insertion operators exist (for basic types only). Additionally, the *Format* (p. 1296) function allows to use simply append formatted value to a string:

```
// the following 2 snippets are equivalent
```

```
wxString s = "...";
s += wxString::Format("%d", n);

wxString s;
s.Printf("...%d", n);
```

Format (p. 1296)
FormatV (p. 1296)
Printf (p. 1301)
PrintfV (p. 1301)
operator << (p. 1306)

Memory management

These are "advanced" functions and they will be needed quite rarely. *Alloc* (p. 1292) and *Shrink* (p. 1302) are only interesting for optimization purposes. *GetWriteBuf* (p. 1297) may be very useful when working with some external API which requires the caller to provide a writable buffer, but extreme care should be taken when using it: before performing any other operation on the string *UngetWriteBuf* (p. 1304) **must** be called!

Alloc (p. 1292)
Shrink (p. 1302)
GetWriteBuf (p. 1297)
UngetWriteBuf (p. 1304)

Miscellaneous

Other string functions.

Trim (p. 1304)
Pad (p. 1300)
Truncate (p. 1304)

wxWidgets 1.xx compatibility functions

These functions are deprecated, please consider using new wxWidgets 2.0 functions instead of them (or, even better, std::string compatible variants).

SubString (p. 1302)
sprintf (p. 1302)
CompareTo (p. 1295)
Length (p. 1299)
Freq (p. 1296)
LowerCase (p. 1299)
UpperCase (p. 1304)
Strip (p. 1302)
Index (p. 1297)
Remove (p. 1301)

std::string compatibility functions

The supported functions are only listed here, please see any STL reference for their documentation.

```
    // take nLen chars starting at nPos
wxString(const wxString& str, size_t nPos, size_t nLen);
    // take all characters from pStart to pEnd (poor man's iterators)
wxString(const void *pStart, const void *pEnd);

// lib.string.capacity
    // return the length of the string
size_t size() const;
    // return the length of the string
size_t length() const;
    // return the maximum size of the string
size_t max_size() const;
    // resize the string, filling the space with c if c != 0
void resize(size_t nSize, char ch = '\0');
    // delete the contents of the string
void clear();
    // returns true if the string is empty
bool empty() const;

// lib.string.access
    // return the character at position n
char at(size_t n) const;
    // returns the writable character at position n
char& at(size_t n);

// lib.string.modifiers
    // append a string
wxString& append(const wxString& str);
    // append elements str[pos], ..., str[pos+n]
wxString& append(const wxString& str, size_t pos, size_t n);
    // append first n (or all if n == npos) characters of sz
wxString& append(const char *sz, size_t n = npos);

    // append n copies of ch
wxString& append(size_t n, char ch);

    // same as `this_string = str'
wxString& assign(const wxString& str);
    // same as ` = str[pos..pos + n]
wxString& assign(const wxString& str, size_t pos, size_t n);
    // same as `= first n (or all if n == npos) characters of sz'
wxString& assign(const char *sz, size_t n = npos);
    // same as `= n copies of ch'
wxString& assign(size_t n, char ch);

    // insert another string
```

```
  wxString& insert(size_t nPos, const wxString& str);
    // insert n chars of str starting at nStart (in str)
  wxString& insert(size_t nPos, const wxString& str, size_t nStart, size_t
n);

    // insert first n (or all if n == npos) characters of sz
  wxString& insert(size_t nPos, const char *sz, size_t n = npos);
    // insert n copies of ch
  wxString& insert(size_t nPos, size_t n, char ch);

    // delete characters from nStart to nStart + nLen
  wxString& erase(size_t nStart = 0, size_t nLen = npos);

    // replaces the substring of length nLen starting at nStart
  wxString& replace(size_t nStart, size_t nLen, const char* sz);
    // replaces the substring with nCount copies of ch
  wxString& replace(size_t nStart, size_t nLen, size_t nCount, char ch);
    // replaces a substring with another substring
  wxString& replace(size_t nStart, size_t nLen,
                    const wxString& str, size_t nStart2, size_t nLen2);
    // replaces the substring with first nCount chars of sz
    wxString& replace(size_t nStart, size_t nLen,
                      const char* sz, size_t nCount);

    // swap two strings
  void swap(wxString& str);

    // All find() functions take the nStart argument which specifies the
    // position to start the search on, the default value is 0. All
functions
    // return npos if there were no match.

    // find a substring
  size_t find(const wxString& str, size_t nStart = 0) const;

    // find first n characters of sz
  size_t find(const char* sz, size_t nStart = 0, size_t n = npos) const;

    // find the first occurrence of character ch after nStart
  size_t find(char ch, size_t nStart = 0) const;

    // rfind() family is exactly like find() but works right to left

    // as find, but from the end
  size_t rfind(const wxString& str, size_t nStart = npos) const;

    // as find, but from the end
  size_t rfind(const char* sz, size_t nStart = npos,
        size_t n = npos) const;
    // as find, but from the end
  size_t rfind(char ch, size_t nStart = npos) const;

    // find first/last occurrence of any character in the set

    //
  size_t find_first_of(const wxString& str, size_t nStart = 0) const;
    //
  size_t find_first_of(const char* sz, size_t nStart = 0) const;
    // same as find(char, size_t)
  size_t find_first_of(char c, size_t nStart = 0) const;
    //
  size_t find_last_of (const wxString& str, size_t nStart = npos) const;
    //
```

```
size_t find_last_of (const char* s, size_t nStart = npos) const;
  // same as rfind(char, size_t)
size_t find_last_of (char c, size_t nStart = npos) const;

  // find first/last occurrence of any character not in the set

  //
size_t find_first_not_of(const wxString& str, size_t nStart = 0) const;
  //
size_t find_first_not_of(const char* s, size_t nStart = 0) const;
  //
size_t find_first_not_of(char ch, size_t nStart = 0) const;
  //
size_t find_last_not_of(const wxString& str, size_t nStart=npos) const;
  //
size_t find_last_not_of(const char* s, size_t nStart = npos) const;
  //
size_t find_last_not_of(char ch, size_t nStart = npos) const;

  // All compare functions return a negative, zero or positive value
  // if the [sub]string is less, equal or greater than the compare()
argument.

  // just like strcmp()
int compare(const wxString& str) const;
  // comparison with a substring
int compare(size_t nStart, size_t nLen, const wxString& str) const;
  // comparison of 2 substrings
int compare(size_t nStart, size_t nLen,
            const wxString& str, size_t nStart2, size_t nLen2) const;
  // just like strcmp()
int compare(const char* sz) const;
  // substring comparison with first nCount characters of sz
int compare(size_t nStart, size_t nLen,
            const char* sz, size_t nCount = npos) const;

// substring extraction
wxString substr(size_t nStart = 0, size_t nLen = npos) const;
```

wxString::wxString

wxString()

Default constructor. Initializes the string to **" "** (empty string).

wxString(**const wxString&** *x*)

Copy constructor.

wxString(**char** *ch*, **size_t** *n = 1*)

Constructs a string of *n* copies of character *ch*.

wxString(**const char*** *psz*, **size_t** *nLength = wxSTRING_MAXLEN*)

Takes first *nLength* characters from the C string *psz*. The default value of

`wxSTRING_MAXLEN` means to take all the string.

Note that this constructor may be used even if *psz* points to a buffer with binary data (i.e. containing `NUL` characters) as long as you provide the correct value for *nLength*. However, the default form of it works only with strings without intermediate `NUL`s because it uses `strlen()` to calculate the effective length and it would not give correct results otherwise.

 wxString(**const unsigned char*** *psz*, **size_t** *nLength* = *wxSTRING_MAXLEN*)

For compilers using unsigned char: takes first *nLength* characters from the C string *psz*. The default value of `wxSTRING_MAXLEN` means take all the string.

Note: In Unicode build, all of the above constructors take `wchar_t` arguments instead of `char`.

Constructors with conversion

The following constructors allow you to construct wxString from wide string in ANSI build or from C string in Unicode build.

 wxString(**const wchar_t*** *psz*, **wxMBConv&** *conv*, **size_t** *nLength* = *wxSTRING_MAXLEN*)

Initializes the string from first *nLength* characters of wide string. The default value of `wxSTRING_MAXLEN` means take all the string. In ANSI build, *conv*'s *WC2MB* (p. 910) method is called to convert *psz* to wide string. It is ignored in Unicode build.

 wxString(**const char*** *psz*, **wxMBConv&** *conv*, **size_t** *nLength* = *wxSTRING_MAXLEN*)

Initializes the string from first *nLength* characters of C string. The default value of `wxSTRING_MAXLEN` means take all the string. In Unicode build, *conv*'s *MB2WC* (p. 909) method is called to convert *psz* to wide string. It is ignored in ANSI build.

See also

wxMBConv classes (p. 1733), *mb_str* (p. 1300), *wc_str* (p. 1305)

wxString::~wxString

 ~wxString()

String destructor. Note that this is not virtual, so wxString must not be inherited from.

wxString::Alloc

void Alloc(**size_t** *nLen*)

Preallocate enough space for wxString to store *nLen* characters. This function may be used to increase speed when the string is constructed by repeated concatenation as in

```
    // delete all vowels from the string
    wxString DeleteAllVowels(const wxString& original)
    {
        wxString result;

        size_t len = original.length();

        result.Alloc(len);

        for ( size_t n = 0; n < len; n++ )
        {
            if ( strchr("aeuio", tolower(original[n])) == NULL )
                result += original[n];
        }

        return result;
    }
```

because it will avoid the need to reallocate string memory many times (in case of long strings). Note that it does not set the maximal length of a string - it will still expand if more than *nLen* characters are stored in it. Also, it does not truncate the existing string (use *Truncate()* (p. 1304) for this) even if its current length is greater than *nLen*

wxString::Append

wxString& Append(**const char*** *psz*)

Concatenates *psz* to this string, returning a reference to it.

wxString& Append(**char** *ch*, **int** *count = 1*)

Concatenates character *ch* to this string, *count* times, returning a reference to it.

wxString::AfterFirst

wxString AfterFirst(**char** *ch*) **const**

Gets all the characters after the first occurrence of *ch*. Returns the empty string if *ch* is not found.

wxString::AfterLast

wxString AfterLast(**char** *ch*) **const**

Gets all the characters after the last occurrence of *ch*. Returns the whole string if *ch* is not found.

wxString::BeforeFirst

wxString BeforeFirst(**char** *ch*) **const**

Gets all characters before the first occurrence of *ch*. Returns the whole string if *ch* is not found.

wxString::BeforeLast

wxString BeforeLast(char *ch*) const

Gets all characters before the last occurrence of *ch*. Returns the empty string if *ch* is not found.

wxString::c_str

const wxChar * c_str() const

Returns a pointer to the string data (`const char*` in ANSI build,`const wchar_t*` in Unicode build).

See also

mb_str (p. 1300), *wc_str* (p. 1305),*fn_str* (p. 1296)

wxString::Clear

void Clear()

Empties the string and frees memory occupied by it.

See also: *Empty* (p. 1295)

wxString::Cmp

int Cmp(const char* *psz*) const

Case-sensitive comparison.

Returns a positive value if the string is greater than the argument, zero if it is equal to it or a negative value if it is less than the argument (same semantics as the standard *strcmp()* function).

See also *CmpNoCase* (p. 1294), *IsSameAs* (p. 1298).

wxString::CmpNoCase

int CmpNoCase(const char* *psz*) const

Case-insensitive comparison.

Returns a positive value if the string is greater than the argument, zero if it is equal to it

or a negative value if it is less than the argument (same semantics as the standard *strcmp()* function).

See also *Cmp* (p. 1294), *IsSameAs* (p. 1298).

wxString::CompareTo

```
#define NO_POS ((int)(-1)) // undefined position
enum caseCompare {exact, ignoreCase};
```

int CompareTo(**const char*** *psz*, **caseCompare** *cmp = exact*) **const**

Case-sensitive comparison. Returns 0 if equal, 1 if greater or -1 if less.

wxString::Contains

bool Contains(**const wxString&** *str*) **const**

Returns 1 if target appears anywhere in wxString; else 0.

wxString::Empty

void Empty()

Makes the string empty, but doesn't free memory occupied by the string.

See also: *Clear()* (p. 1294).

wxString::Find

int Find(**char** *ch*, **bool** *fromEnd = false*) **const**

Searches for the given character. Returns the starting index, or -1 if not found.

int Find(**const char*** *sz*) **const**

Searches for the given string. Returns the starting index, or -1 if not found.

wxString::First

int First(**char** *c*)

int First(**const char*** *psz*) **const**

int First(**const wxString&** *str*) **const**

Same as *Find* (p. 1295).

wxString::fn_str

const wchar_t* fn_str() **const**

const char* fn_str() **const**

const wxCharBuffer fn_str() **const**

Returns string representation suitable for passing to OS' functions for file handling. In ANSI build, this is same as *c_str* (p. 1294). In Unicode build, returned value can be either wide character string or C string in charset matching the `wxConvFileName` object, depending on the OS.

See also

wxMBConv (p. 909),*wc_str* (p. 1305), *mb_str* (p. 1305)

wxString::Format

static wxString Format(**const wxChar** **format*, *...*)

This static function returns the string containing the result of calling *Printf* (p. 1301) with the passed parameters on it.

See also

FormatV (p. 1296), *Printf* (p. 1301)

wxString::FormatV

static wxString FormatV(**const wxChar** **format*, **va_list** *argptr*)

This static function returns the string containing the result of calling *PrintfV* (p. 1301) with the passed parameters on it.

See also

Format (p. 1296), *PrintfV* (p. 1301)

wxString::Freq

int Freq(**char** *ch*) **const**

Returns the number of occurrences of *ch* in the string.

wxString::FromAscii

static wxString FromAscii(**const char*** *s*)

static wxString FromAscii(**const char** *c*)

Converts the string or character from an ASCII, 7-bit form to the native wxString representation. Most useful when using a Unicode build of wxWidgets. Use *wxString constructors* (p. 1291) if you need to convert from another charset.

wxString::GetChar

char GetChar(**size_t** *n*) **const**

Returns the character at position *n* (read-only).

wxString::GetData

const wxChar* GetData() **const**

wxWidgets compatibility conversion. Returns a constant pointer to the data in the string.

wxString::GetWritableChar

char& GetWritableChar(**size_t** *n*)

Returns a reference to the character at position *n*.

wxString::GetWriteBuf

wxChar* GetWriteBuf(**size_t** *len*)

Returns a writable buffer of at least *len* bytes. It returns a pointer to a new memory block, and the existing data will not be copied.

Call *wxString::UngetWriteBuf* (p. 1304) as soon as possible to put the string back into a reasonable state.

wxString::Index

size_t Index(**char** *ch*) **const**

size_t Index(**const char*** *sz*) **const**

Same as *wxString::Find* (p. 1295).

wxString::IsAscii

bool IsAscii() **const**

Returns true if the string contains only ASCII characters.

wxString::IsEmpty

bool IsEmpty() **const**

Returns true if the string is empty.

wxString::IsNull

bool IsNull() **const**

Returns true if the string is empty (same as *IsEmpty* (p. 1298)).

wxString::IsNumber

bool IsNumber() **const**

Returns true if the string is an integer (with possible sign).

wxString::IsSameAs

bool IsSameAs(**const char*** *psz*, **bool** *caseSensitive = true*) **const**

Test for string equality, case-sensitive (default) or not.

caseSensitive is true by default (case matters).

Returns true if strings are equal, false otherwise.

See also *Cmp* (p. 1294), *CmpNoCase* (p. 1294)

bool IsSameAs(**char** *c*, **bool** *caseSensitive = true*) **const**

Test whether the string is equal to the single character *c*. The test is case-sensitive if *caseSensitive* is true (default) or not if it is false.

Returns true if the string is equal to the character, false otherwise.

See also *Cmp* (p. 1294), *CmpNoCase* (p. 1294)

wxString::IsWord

bool IsWord() **const**

Returns true if the string is a word. TODO: what's the definition of a word?

wxString::Last

char Last() **const**

Returns the last character.

char& Last()

Returns a reference to the last character (writable).

wxString::Left

wxString Left(**size_t** *count*) **const**

Returns the first *count* characters of the string.

wxString::Len

size_t Len() **const**

Returns the length of the string.

wxString::Length

size_t Length() **const**

Returns the length of the string (same as Len).

wxString::Lower

wxString Lower() **const**

Returns this string converted to the lower case.

wxString::LowerCase

void LowerCase()

Same as MakeLower.

wxString::MakeLower

wxString& MakeLower()

Converts all characters to lower case and returns the result.

wxString::MakeUpper

wxString& MakeUpper()

Converts all characters to upper case and returns the result.

wxString::Matches

bool Matches(**const char*** *szMask*) **const**

Returns true if the string contents matches a mask containing '*' and '?'.

wxString::mb_str

const char* mb_str(**wxMBConv&** *conv*) **const**

const wxCharBuffer mb_str(**wxMBConv&** *conv*) **const**

Returns multibyte (C string) representation of the string. In Unicode build, converts using *conv*'s *cWC2MB* (p. 910)method and returns wxCharBuffer. In ANSI build, this function is same as *c_str* (p. 1294). The macro wxWX2MBbuf is defined as the correct return type (without const).

See also

wxMBConv (p. 909),*c_str* (p. 1294), *wc_str* (p. 1305),*fn_str* (p. 1296)

wxString::Mid

wxString Mid(**size_t** *first*, **size_t** *count = wxSTRING_MAXLEN*) **const**

Returns a substring starting at *first*, with length *count*, or the rest of the string if *count* is the default value.

wxString::Pad

wxString& Pad(**size_t** *count*, **char** *pad = ' '*, **bool** *fromRight = true*)

Adds *count* copies of *pad* to the beginning, or to the end of the string (the default).

Removes spaces from the left or from the right (default).

wxString::Prepend

wxString& Prepend(**const wxString&** *str*)

Prepends *str* to this string, returning a reference to this string.

wxString::Printf

int Printf(**const char*** *pszFormat*, *...*)

Similar to the standard function *sprintf()*. Returns the number of characters written, or an integer less than zero on error.

NB: This function will use a safe version of *vsprintf()* (usually called *vsnprintf()*) whenever available to always allocate the buffer of correct size. Unfortunately, this function is not available on all platforms and the dangerous *vsprintf()* will be used then which may lead to buffer overflows.

wxString::PrintfV

int PrintfV(**const char*** *pszFormat*, **va_list** *argPtr*)

Similar to vprintf. Returns the number of characters written, or an integer less than zero on error.

wxString::Remove

wxString& Remove(**size_t** *pos*)

Same as Truncate. Removes the portion from *pos* to the end of the string.

wxString& Remove(**size_t** *pos*, **size_t** *len*)

Removes the *len* characters from the string, starting at *pos*.

wxString::RemoveLast

wxString& RemoveLast()

Removes the last character.

wxString::Replace

size_t Replace(**const char*** *szOld*, **const char*** *szNew*, **bool** *replaceAll = true*)

Replace first (or all) occurrences of substring with another one.

replaceAll: global replace (default), or only the first occurrence.

Returns the number of replacements made.

wxString::Right

wxString Right(**size_t** *count*) **const**

Returns the last *count* characters.

wxString::SetChar

void SetChar(**size_t** *n*, **char***ch*)

Sets the character at position *n*.

wxString::Shrink

void Shrink()

Minimizes the string's memory. This can be useful after a call to *Alloc()* (p. 1292) if too much memory were preallocated.

wxString::sprintf

void sprintf(**const char*** *fmt*)

The same as Printf.

wxString::StartsWith

bool StartsWith(**const wxChar** **prefix*, **wxString** **rest = NULL*) **const**

This function can be used to test if the string starts with the specified *prefix*. If it does, the function will return `true` and put the rest of the string (i.e. after the prefix) into *rest* string if it is not `NULL`. Otherwise, the function returns `false` and doesn't modify the *rest*.

wxString::Strip

```
enum stripType {leading = 0x1, trailing = 0x2, both = 0x3};
```

wxString Strip(**stripType** *s = trailing*) **const**

Strip characters at the front and/or end. The same as Trim except that it doesn't change this string.

wxString::SubString

wxString SubString(**size_t** *from*, **size_t** *to*) **const**

Deprecated, use *Mid* (p. 1300) instead (but note that parameters have different meaning).

Returns the part of the string between the indices *from* and *to*inclusive.

wxString::ToAscii

const char* ToAscii() **const**

Converts the string to an ASCII, 7-bit string (ANSI builds only).

const wxCharBuffer ToAscii() **const**

Converts the string to an ASCII, 7-bit string in the form of a wxCharBuffer (Unicode builds only).

Note that this conversion only works if the string contains only ASCII characters. The *mb_str* (p. 1300) method provides more powerful means of converting wxString to C string.

wxString::ToDouble

bool ToDouble(**double** **val*) **const**

Attempts to convert the string to a floating point number. Returns true on success (the number is stored in the location pointed to by *val*) or false if the string does not represent such number.

See also

wxString::ToLong (p. 1303),
wxString::ToULong (p. 1303)

wxString::ToLong

bool ToLong(**long** **val*, **int** *base = 10*) **const**

Attempts to convert the string to a signed integer in base *base*. Returns `true` on success in which case the number is stored in the location pointed to by *val* or `false` if the string does not represent a valid number in the given base.

The value of *base* must be comprised between 2 and 36, inclusive, or be a special value 0 which means that the usual rules of `C` numbers are applied: if the number starts with `0x` it is considered to be in base16, if it starts with `0` - in base 8 and in base 10 otherwise. Note that you may not want to specify the base 0 if you are parsing the numbers which may have leading zeroes as they can yield unexpected (to the user not familiar with C) results.

See also

wxString::ToDouble (p. 1303),
wxString::ToULong (p. 1303)

wxString::ToULong

bool ToULong(**unsigned long** **val*, **int** *base = 10*) **const**

Attempts to convert the string to an unsigned integer in base *base*. Returns `true` on success in which case the number is stored in the location pointed to by *val* or `false` if the string does not represent a valid number in the given base.

See *wxString::ToLong* (p. 1303) for the more detailed description of the *base* parameter.

See also

wxString::ToDouble (p. 1303),
wxString::ToLong (p. 1303)

wxString::Trim

wxString& Trim(**bool** *fromRight = true*)

Removes spaces from the left or from the right (default).

wxString::Truncate

wxString& Truncate(**size_t** *len*)

Truncate the string to the given length.

wxString::UngetWriteBuf

void UngetWriteBuf()

void UngetWriteBuf(**size_t** *len*)

Puts the string back into a reasonable state (in which it can be used normally), after *wxString::GetWriteBuf* (p. 1297) was called.

The version of the function without the *len* parameter will calculate the new string length itself assuming that the string is terminated by the first`NUL` character in it while the second one will use the specified length and thus is the only version which should be used with the strings with embedded `NUL`s (it is also slightly more efficient as `strlen()` doesn't have to be called).

wxString::Upper

wxString Upper() **const**

Returns this string converted to upper case.

wxString::UpperCase

void UpperCase()

The same as MakeUpper.

wxString::wc_str

const wchar_t* wc_str(**wxMBConv&** *conv*) **const**

const wxWCharBuffer wc_str(**wxMBConv&** *conv*) **const**

Returns wide character representation of the string. In ANSI build, converts using *conv*'s *cMB2WC* (p. 910)method and returns wxWCharBuffer. In Unicode build, this function is same as *c_str* (p. 1294). The macro wxWX2WCbuf is defined as the correct return type (without const).

See also

wxMBConv (p. 909),*c_str* (p. 1294), *mb_str* (p. 1305),*fn_str* (p. 1296)

wxString::operator!

bool operator!() **const**

Empty string is false, so !string will only return true if the string is empty. This allows the tests for NULLness of a *const char ** pointer and emptiness of the string to look the same in the code and makes it easier to port old code to wxString.

See also *IsEmpty()* (p. 1298).

wxString::operator =

wxString& operator =(**const wxString&** *str*)

wxString& operator =(**const char*** *psz*)

wxString& operator =(**char** *c*)

wxString& operator =(**const unsigned char*** *psz*)

wxString& operator =(**const wchar_t*** *pwz*)

Assignment: the effect of each operation is the same as for the corresponding constructor (see *wxString constructors* (p. 1291)).

wxString::operator +

Concatenation: all these operators return a new string equal to the concatenation of the operands.

wxString operator +(const wxString& *x*, const wxString& *y*)

wxString operator +(const wxString& *x*, const char* *y*)

wxString operator +(const wxString& *x*, char *y*)

wxString operator +(const char* *x*, const wxString& *y*)

wxString::operator +=

void operator +=(const wxString& *str*)

void operator +=(const char* *psz*)

void operator +=(char *c*)

Concatenation in place: the argument is appended to the string.

wxString::operator []

char& operator [](size_t *i*)

char operator [](size_t *i*)

char operator [](int *i*)

Element extraction.

wxString::operator ()

wxString operator ()(size_t *start*, size_t *len*)

Same as Mid (substring extraction).

wxString::operator <<

wxString& operator <<(const wxString& *str*)

wxString& operator <<(const char* *psz*)

wxString& operator <<(char *ch*)

Same as +=.

wxString& operator <<(int *i*)

wxString& operator <<(float *f*)

wxString& operator <<(double *d*)

These functions work as C++ stream insertion operators: they insert the given value into the string. Precision or format cannot be set using them, you can use *Printf* (p. 1301) for this.

wxString::operator >>

friend istream& operator >>(istream& *is*, **wxString&** *str***)**

Extraction from a stream.

wxString::operator const char*

operator const char*() const

Implicit conversion to a C string.

Comparison operators

bool operator ==(const wxString& *x*, **const wxString&** *y***)**

bool operator ==(const wxString& *x*, **const char*** *t***)**

bool operator !=(const wxString& *x*, **const wxString&** *y***)**

bool operator !=(const wxString& *x*, **const char*** *t***)**

bool operator >(const wxString& *x*, **const wxString&** *y***)**

bool operator >(const wxString& *x*, **const char*** *t***)**

bool operator >=(const wxString& *x*, **const wxString&** *y***)**

bool operator >=(const wxString& *x*, **const char*** *t***)**

bool operator <(const wxString& *x*, **const wxString&** *y***)**

bool operator <(const wxString& *x*, **const char*** *t***)**

bool operator <=(const wxString& *x*, **const wxString&** *y***)**

bool operator <=(const wxString& *x*, **const char*** *t***)**

Remarks

These comparisons are case-sensitive.

wxStringBuffer

This tiny class allows to conveniently access the *wxString* (p. 1284) internal buffer as a

writable pointer without any risk of forgetting to restore the string to the usable state later.

For example, assuming you have a low-level OS function called `GetMeaningOfLifeAsString(char *)` returning the value in the provided buffer (which must be writable, of course) you might call it like this:

```
wxString theAnswer;
GetMeaningOfLifeAsString(wxStringBuffer(theAnswer, 1024));
if ( theAnswer != "42" )
{
    wxLogError("Something is very wrong!");
}
```

Note that the exact usage of this depends on whether on not wxUSE_STL is enabled. If wxUSE_STL is enabled, wxStringBuffer creates a seperate empty character buffer, and if wxUSE_STL is disabled, it uses GetWriteBuf() from wxString, keeping the same buffer wxString uses intact. In other words, relying on wxStringBuffer containing the old wxString data is probably not a good idea if you want to build your program in both with and without wxUSE_STL.

Derived from

None

Include files

<wx/string.h>

wxStringBuffer::wxStringBuffer

wxStringBuffer(**const wxString&** *str*, **size_t** *len*)

Constructs a writable string buffer object associated with the given string and containing enough space for at least *len* characters. Basically, this is equivalent to calling *GetWriteBuf* (p. 1297) and saving the result.

wxStringBuffer::~wxStringBuffer

~wxStringBuffer()

Restores the string passed to the constructor to the usable state by calling *UngetWriteBuf* (p. 1304) on it.

wxStringBuffer::operator wxChar *

wxChar * operator wxChar *()

Returns the writable pointer to a buffer of the size at least equal to the length specified in

the constructor.

wxStringBufferLength

This tiny class allows to conveniently access the *wxString* (p. 1284) internal buffer as a writable pointer without any risk of forgetting to restore the string to the usable state later, and allows the user to set the internal length of the string.

For example, assuming you have a low-level OS function called `int GetMeaningOfLifeAsString(char *)` copying the value in the provided buffer (which must be writable, of course), and returning the actual length of the string, you might call it like this:

```
wxString theAnswer;
wxStringBuffer theAnswerBuffer(theAnswer, 1024);
int nLength = GetMeaningOfLifeAsString(theAnswerBuffer);
theAnswerBuffer.SetLength(nLength);
if ( theAnswer != "42" )
{
    wxLogError("Something is very wrong!");
}
```

Note that the exact usage of this depends on whether on not wxUSE_STL is enabled. If wxUSE_STL is enabled, wxStringBuffer creates a seperate empty character buffer, and if wxUSE_STL is disabled, it uses GetWriteBuf() from wxString, keeping the same buffer wxString uses intact. In other words, relying on wxStringBuffer containing the old wxString data is probably not a good idea if you want to build your program in both with and without wxUSE_STL.

Note that SetLength `must` be called before wxStringBufferLength destructs.

Derived from

None

Include files

<wx/string.h>

wxStringBufferLength::wxStringBufferLength

 wxStringBufferLength(**const wxString&** *str,* **size_t** *len*)

Constructs a writable string buffer object associated with the given string and containing enough space for at least *len* characters. Basically, this is equivalent to calling *GetWriteBuf* (p. 1297) and saving the result.

wxStringBufferLength::~wxStringBufferLength

~wxStringBufferLength()

Restores the string passed to the constructor to the usable state by calling *UngetWriteBuf* (p. 1304) on it.

wxStringBufferLength::SetLength

void SetLength(**size_t** *nLength*)

Sets the internal length of the string referred to by wxStringBufferLength to *nLength* characters.

Must be called before wxStringBufferLength destructs.

wxStringBufferLength::operator wxChar *

wxChar * operator wxChar *()

Returns the writable pointer to a buffer of the size at least equal to the length specified in the constructor.

wxStringClientData

Predefined client data class for holding a string.

Derived from

wxClientData (p. 148)

Include files

<clntdata.h>

Data structures

wxStringClientData::wxStringClientData

wxStringClientData()

Empty constructor.

wxStringClientData(**const wxString&** *data*)

Create client data with string.

wxStringClientData::GetData

const wxString& GetData() **const**

Get string client data.

---

wxStringClientData::SetData

void SetData(**const wxString&** *data*)

Set string client data.

---

wxStringInputStream

This class implements an input stream which reads data from a string. It supports seeking.

Derived from

wxInputStream (p. 814)

Include files

<wx/sstream.h>

---

wxStringInputStream::wxStringInputStream

 wxStringInputStream(**const wxString&** *s*)

Creates a new read-only stream using the specified string. Note that the string is copied by the stream so if the original string is modified after using this constructor, changes to it are not reflected when reading from stream.

---

wxStringOutputStream

This class implements an output stream which writes data either to a user-provided or internally allocated string. Note that currently this stream does not support seeking but can tell its current position.

Derived from

wxOutputStream (p. 1016)

Include files

<wx/sstream.h>

wxStringOutputStream::wxStringOutputStream

wxStringOutputStream(**wxString** **str = NULL*)

If the provided pointer is non-NULL, data will be written to it. Otherwise, an internal string is used for the data written to this stream, use *GetString()* (p. 1312) to get access to it.

If *str* is used, data written to the stream is appended to the current contents of it, i.e. the string is not cleared here. However if it is not empty, the positions returned by *TellO* (p. 1017) will be offset by the initial string length, i.e. initial stream position will be the initial length of the string and not 0.

wxStringOutputStream::GetString

const wxString& GetString() **const**

Returns the string containing all the data written to the stream so far.

wxStringTokenizer

wxStringTokenizer helps you to break a string up into a number of tokens. It replaces the standard C function `strtok()` and also extends it in a number of ways.

To use this class, you should create a wxStringTokenizer object, give it the string to tokenize and also the delimiters which separate tokens in the string (by default, white space characters will be used).

Then *GetNextToken* (p. 1314) may be called repeatedly until it *HasMoreTokens* (p. 1314) returns `false`.

For example:

```
wxStringTokenizer tkz(wxT("first:second:third:fourth"), wxT(":"));
while ( tkz.HasMoreTokens() )
{
    wxString token = tkz.GetNextToken();

    // process token here
}
```

By default, wxStringTokenizer will behave in the same way as `strtok()` if the delimiters string only contains white space characters but, unlike the standard function, it will return empty tokens if this is not the case. This is helpful for parsing strictly formatted data where the number of fields is fixed but some of them may be empty (i.e. `TAB` or comma delimited text files).

The behaviour is governed by the last *constructor* (p. 1313)/*SetString* (p. 1314) parameter `mode` which may be one of the following:

wxTOKEN_DEFAULT   Default behaviour (as described above): same as
                         wxTOKEN_STRTOK if the delimiter string contains only

whitespaces, same as `wxTOKEN_RET_EMPTY` otherwise

`wxTOKEN_RET_EMPTY`	In this mode, the empty tokens in the middle of the string will be returned, i.e. `"a::b:"` will be tokenized in three tokens 'a', '' and 'b'.
`wxTOKEN_RET_EMPTY_ALL`	In this mode, empty trailing token (after the last delimiter character) will be returned as well. The string as above will contain four tokens: the already mentioned ones and another empty one as the last one.
`wxTOKEN_RET_DELIMS`	In this mode, the delimiter character after the end of the current token (there may be none if this is the last token) is returned appended to the token. Otherwise, it is the same mode as `wxTOKEN_RET_EMPTY`.
`wxTOKEN_STRTOK`	In this mode the class behaves exactly like the standard `strtok()` function. The empty tokens are never returned.

Derived from

wxObject (p. 1012)

Include files

<wx/tokenzr.h>

wxStringTokenizer::wxStringTokenizer

 wxStringTokenizer()

Default constructor. You must call *SetString* (p. 1314) before calling any other methods.

 wxStringTokenizer(**const wxString&** *str*, **const wxString&** *delims = " \t\r\n"*, **wxStringTokenizerMode** *mode = wxTOKEN_DEFAULT*)

Constructor. Pass the string to tokenize, a string containing delimiters and the mode specifying how the string should be tokenized.

wxStringTokenizer::CountTokens

int CountTokens() **const**

Returns the number of tokens remaining in the input string. The number of tokens returned by this function is decremented each time *GetNextToken* (p. 1314) is called and when it reaches 0 *HasMoreTokens* (p. 1314) returns`false`.

wxStringTokenizer::HasMoreTokens

bool HasMoreTokens() **const**

Returns `true` if the tokenizer has further tokens, `false` if none are left.

wxStringTokenizer::GetNextToken

wxString GetNextToken()

Returns the next token or empty string if the end of string was reached.

wxStringTokenizer::GetPosition

size_t GetPosition() **const**

Returns the current position (i.e. one index after the last returned token or 0 if GetNextToken() has never been called) in the original string.

wxStringTokenizer::GetString

wxString GetString() **const**

Returns the part of the starting string without all token already extracted.

wxStringTokenizer::SetString

void SetString(**const wxString&** *to_tokenize*, **const wxString&** *delims* = *" \t\r\n"*, **wxStringTokenizerMode** *mode* = *wxTOKEN_DEFAULT*)

Initializes the tokenizer.

Pass the string to tokenize, a string containing delimiters, and the mode specifying how the string should be tokenized.

wxSysColourChangedEvent

This class is used for system colour change events, which are generated when the user changes the colour settings using the control panel. This is only appropriate under Windows.

Derived from

wxEvent (p. 483)
wxObject (p. 1012)

Include files

<wx/event.h>

Event table macros

To process a system colour changed event, use this event handler macro to direct input to a member function that takes a wxSysColourChanged argument.

EVT_SYS_COLOUR_CHANGED(func) Process a wxEVT_SYS_COLOUR_CHANGED event.

Remarks
The default event handler for this event propagates the event to child windows, since Windows only sends the events to top-level windows. If intercepting this event for a top-level window, remember to call the base class handler, or to pass the event on to the window's children explicitly.

See also

Event handling overview (p. 1751)

wxSysColourChangedEvent::wxSysColourChangedEvent

wxSysColourChangedEvent()

Constructor.

wxSystemOptions

wxSystemOptions stores option/value pairs that wxWidgets itself or applications can use to alter behaviour at run-time. It can be used to optimize behaviour that doesn't deserve a distinct API, but is still important to be able to configure.

These options are currently recognised by wxWidgets.

Windows

Option	Value
no-maskblt	1 to never use WIN32's MaskBlt function, 0 to allow it to be used where possible. Default: 0. In some circumstances the MaskBlt function can be slower than using the fallback code, especially if using DC cacheing. By default, MaskBlt will be used where it is implemented by the operating system and driver.
msw.remap	If 1 (the default), wxToolBar bitmap colours will be remapped to the current theme's values. Set this to 0 to disable this functionality, for

	example if you're using more than 16 colours in your tool bitmaps.
msw.window.no-clip-children	If 1, windows will not automatically get the WS_CLIPCHILDREN style. This restores the way windows are refreshed back to the method used in versions of wxWidgets earlier than 2.5.4, and for some complex window hierarchies it can reduce apparent refresh delays. You may still specify wxCLIP_CHILDREN for individual windows.
msw.notebook.themed-background	If set to 0, globally disables themed backgrounds on notebook pages. Note that this won't disable the theme on the actual notebook background (noticeable only if there are no pages).
msw.staticbitmap.htclient	If set to 1, allows the static bitmap to respond to mouse events. The default is 0, since a value of 1 can interfere with refresh in static boxes. Note that once set, this option cannot be unset later in the application.

Mac

Option	Value
mac.window-plain-transition	If 1, uses a plainer transition when showing a window. You can also use the symbol wxMAC_WINDOW_PLAIN_TRANSITION.
window-default-variant	The default variant used by windows (cast to integer from the wxWindowVariant enum). Also known as wxWINDOW_DEFAULT_VARIANT.

MGL

Option	Value
mgl.aa-threshold	Set this integer option to point size below which fonts are not antialiased. Default: 10.
mgl.screen-refresh	Screen refresh rate in Hz. A reasonable default is used if not specified.

Motif

Option	Value
motif.largebuttons	If 1, uses a bigger default size for wxButtons.

The compile-time option to include or exclude this functionality is wxUSE_SYSTEM_OPTIONS.

wxObject (p. 1012)

Include files

<wx/sysopt.h>

wxSystemOptions::wxSystemOptions

wxSystemOptions()

Default constructor. You don't need to create an instance of wxSystemOptions since all of its functions are static.

wxSystemOptions::GetOption

wxString GetOption(**const wxString&** *name*) **const**

Gets an option. The function is case-insensitive to *name*.

Returns empty string if the option hasn't been set.

See also

wxSystemOptions::SetOption (p. 1318), *wxSystemOptions::GetOptionInt* (p. 1317), *wxSystemOptions::HasOption* (p. 1317)

wxSystemOptions::GetOptionInt

int GetOptionInt(**const wxString&** *name*) **const**

Gets an option as an integer. The function is case-insensitive to *name*.

If the option hasn't been set, this function returns 0.

See also

wxSystemOptions::SetOption (p. 1318), *wxSystemOptions::GetOption* (p. 1317), *wxSystemOptions::HasOption* (p. 1317)

wxSystemOptions::HasOption

bool HasOption(**const wxString&** *name*) **const**

Returns `true` if the given option is present. The function is case-insensitive to *name*.

See also

wxSystemOptions::IsFalse

bool IsFalse(**const wxString&** *name*) **const**

Returns `true` if the option with the given *name* had been set to 0value. This is mostly useful for boolean options for which you can't use`GetOptionInt(name) == 0` as this would also be true if the option hadn't been set at all.

wxSystemOptions::SetOption

void SetOption(**const wxString&** *name*, **const wxString&** *value*)

void SetOption(**const wxString&** *name*, **int** *value*)

Sets an option. The function is case-insensitive to *name*.

See also

wxSystemSettings

wxSystemSettings allows the application to ask for details about the system. This can include settings such as standard colours, fonts, and user interface element sizes.

Derived from

Include files

<wx/settings.h>

See also

wxSystemSettings::wxSystemSettings

 wxSystemSettings()

Default constructor. You don't need to create an instance of wxSystemSettings since all of its functions are static.

wxSystemSettings::GetColour

static wxColour GetColour(**wxSystemColour** *index*)

Returns a system colour.

index can be one of:

wxSYS_COLOUR_SCROLLBAR	The scrollbar grey area.
wxSYS_COLOUR_BACKGROUND	The desktop colour.
wxSYS_COLOUR_ACTIVECAPTION	Active window caption.
wxSYS_COLOUR_INACTIVECAPTION	Inactive window caption.
wxSYS_COLOUR_MENU	Menu background.
wxSYS_COLOUR_WINDOW	Window background.
wxSYS_COLOUR_WINDOWFRAME	Window frame.
wxSYS_COLOUR_MENUTEXT	Menu text.
wxSYS_COLOUR_WINDOWTEXT	Text in windows.
wxSYS_COLOUR_CAPTIONTEXT	Text in caption, size box and scrollbar arrow box.
wxSYS_COLOUR_ACTIVEBORDER	Active window border.
wxSYS_COLOUR_INACTIVEBORDER	Inactive window border.
wxSYS_COLOUR_APPWORKSPACE	Background colour MDI applications.
wxSYS_COLOUR_HIGHLIGHT	Item(s) selected in a control.
wxSYS_COLOUR_HIGHLIGHTTEXT	Text of item(s) selected in a control.
wxSYS_COLOUR_BTNFACE	Face shading on push buttons.
wxSYS_COLOUR_BTNSHADOW	Edge shading on push buttons.
wxSYS_COLOUR_GRAYTEXT	Greyed (disabled) text.
wxSYS_COLOUR_BTNTEXT	Text on push buttons.
wxSYS_COLOUR_INACTIVECAPTIONTEXT	Colour of text in active captions.
wxSYS_COLOUR_BTNHIGHLIGHT	Highlight colour for buttons (same as wxSYS_COLOUR_3DHILIGHT).
wxSYS_COLOUR_3DDKSHADOW	Dark shadow for three-dimensional display elements.
wxSYS_COLOUR_3DLIGHT	Light colour for three-dimensional display

elements.

wxSYS_COLOUR_INFOTEXT	Text colour for tooltip controls.
wxSYS_COLOUR_INFOBK	Background colour for tooltip controls.
wxSYS_COLOUR_DESKTOP	Same as wxSYS_COLOUR_BACKGROUND.
wxSYS_COLOUR_3DFACE	Same as wxSYS_COLOUR_BTNFACE.
wxSYS_COLOUR_3DSHADOW	Same as wxSYS_COLOUR_BTNSHADOW.
wxSYS_COLOUR_3DHIGHLIGHT	Same as wxSYS_COLOUR_BTNHIGHLIGHT.
wxSYS_COLOUR_3DHILIGHT	Same as wxSYS_COLOUR_BTNHIGHLIGHT.
wxSYS_COLOUR_BTNHILIGHT	Same as wxSYS_COLOUR_BTNHIGHLIGHT.

wxPython note: This static method is implemented in Python as a standalone function named `wxSystemSettings_GetColour`

wxSystemSettings::GetFont

static wxFont GetFont(**wxSystemFont** *index*)

Returns a system font.

index can be one of:

wxSYS_OEM_FIXED_FONT	Original equipment manufacturer dependent fixed-pitch font.
wxSYS_ANSI_FIXED_FONT	Windows fixed-pitch font.
wxSYS_ANSI_VAR_FONT	Windows variable-pitch (proportional) font.
wxSYS_SYSTEM_FONT	System font.
wxSYS_DEVICE_DEFAULT_FONT	Device-dependent font (Windows NT only).
wxSYS_DEFAULT_GUI_FONT	Default font for user interface objects such as menus and dialog boxes. Note that with modern GUIs nothing guarantees that the same font is used for all GUI elements, so some controls might use a different font by default.

wxPython note: This static method is implemented in Python as a standalone function named `wxSystemSettings_GetFont`

wxSystemSettings::GetMetric

static int GetMetric(**wxSystemMetric** *index*, **wxWindow*** *win* = *NULL*)

Returns the value of a system metric, or -1 if the metric is not supported on the current system. The value of *win* determines if the metric returned is a global value or a *wxWindow* (p. 1490) based value, in which case it might determine the widget, the display the window is on, or something similar. The window given should be as close to the metric as possible (e.g a wxTopLevelWindow in case of the wxSYS_CAPTION_Y metric).

index can be one of:

wxSYS_MOUSE_BUTTONS	Number of buttons on mouse, or zero if no mouse was installed.
wxSYS_BORDER_X	Width of single border.
wxSYS_BORDER_Y	Height of single border.
wxSYS_CURSOR_X	Width of cursor.
wxSYS_CURSOR_Y	Height of cursor.
wxSYS_DCLICK_X	Width in pixels of rectangle within which two successive mouse clicks must fall to generate a double-click.
wxSYS_DCLICK_Y	Height in pixels of rectangle within which two successive mouse clicks must fall to generate a double-click.
wxSYS_DRAG_X	Width in pixels of a rectangle centered on a drag point to allow for limited movement of the mouse pointer before a drag operation begins.
wxSYS_DRAG_Y	Height in pixels of a rectangle centered on a drag point to allow for limited movement of the mouse pointer before a drag operation begins.
wxSYS_EDGE_X	Width of a 3D border, in pixels.
wxSYS_EDGE_Y	Height of a 3D border, in pixels.
wxSYS_HSCROLL_ARROW_X	Width of arrow bitmap on horizontal scrollbar.
wxSYS_HSCROLL_ARROW_Y	Height of arrow bitmap on horizontal scrollbar.
wxSYS_HTHUMB_X	Width of horizontal scrollbar thumb.
wxSYS_ICON_X	The default width of an icon.
wxSYS_ICON_Y	The default height of an icon.
wxSYS_ICONSPACING_X	Width of a grid cell for items in large icon view, in pixels. Each item fits into a rectangle of this size when arranged.

wxSYS_ICONSPACING_Y	Height of a grid cell for items in large icon view, in pixels. Each item fits into a rectangle of this size when arranged.
wxSYS_WINDOWMIN_X	Minimum width of a window.
wxSYS_WINDOWMIN_Y	Minimum height of a window.
wxSYS_SCREEN_X	Width of the screen in pixels.
wxSYS_SCREEN_Y	Height of the screen in pixels.
wxSYS_FRAMESIZE_X	Width of the window frame for a wxTHICK_FRAME window.
wxSYS_FRAMESIZE_Y	Height of the window frame for a wxTHICK_FRAME window.
wxSYS_SMALLICON_X	Recommended width of a small icon (in window captions, and small icon view).
wxSYS_SMALLICON_Y	Recommended height of a small icon (in window captions, and small icon view).
wxSYS_HSCROLL_Y	Height of horizontal scrollbar in pixels.
wxSYS_VSCROLL_X	Width of vertical scrollbar in pixels.
wxSYS_VSCROLL_ARROW_X	Width of arrow bitmap on a vertical scrollbar.
wxSYS_VSCROLL_ARROW_Y	Height of arrow bitmap on a vertical scrollbar.
wxSYS_VTHUMB_Y	Height of vertical scrollbar thumb.
wxSYS_CAPTION_Y	Height of normal caption area.
wxSYS_MENU_Y	Height of single-line menu bar.
wxSYS_NETWORK_PRESENT	1 if there is a network present, 0 otherwise.
wxSYS_PENWINDOWS_PRESENT	1 if PenWindows is installed, 0 otherwise.
wxSYS_SHOW_SOUNDS	Non-zero if the user requires an application to present information visually in situations where it would otherwise present the information only in audible form; zero otherwise.
wxSYS_SWAP_BUTTONS	Non-zero if the meanings of the left and right mouse buttons are swapped; zero otherwise.

win is a pointer to the window for which the metric is requested. Specifying the *win* parameter is encouraged, because some metrics on some ports are not supported without one, or they might be capable of reporting better values if given one. If a window does not make sense for a metric, one should still be given, as for example it might determine which displays cursor width is requested with wxSYS_CURSOR_X.

wxPython note: This static method is implemented in Python as a standalone function named `wxSystemSettings_GetMetric`

wxSystemSettings::GetScreenType

static wxSystemScreenType GetScreenType()

Returns the screen type. The return value is one of:

wxSYS_SCREEN_NONE	Undefined screen type
wxSYS_SCREEN_TINY	Tiny screen, less than 320x240
wxSYS_SCREEN_PDA	PDA screen, 320x240 or more but less than 640x480
wxSYS_SCREEN_SMALL	Small screen, 640x480 or more but less than 800x600
wxSYS_SCREEN_DESKTOP	Desktop screen, 800x600 or more

wxTaskBarIcon

This class represents a taskbar icon. A taskbar icon is an icon that appears in the 'system tray' and responds to mouse clicks, optionally with a tooltip above it to help provide information.

X Window System Note

Under X Window System, the window manager must support either the System Tray Protocol by freedesktop.org (`http://freedesktop.org/Standards/systemtray-spec`)(WMs used by modern desktop environments such as GNOME >= 2, KDE >= 3 and XFCE >= 4 all do) or the older methods used in GNOME 1.2 and KDE 1 and 2. If it doesn't, the icon will appear as a toplevel window on user's desktop.

Because not all window managers have system tray, there's no guarantee that wxTaskBarIcon will work correctly under X Window System and so the applications should use it only as an optional component of their user interface. The user should be required to explicitly enable the taskbar icon on Unix, it shouldn't be on by default.

Derived from

wxEvtHandler (p. 486)
wxObject (p. 1012)

Include files

<wx/taskbar.h>

Event handling

To process input from a taskbar icon, use the following event handler macros to direct input to member functions that take a wxTaskBarIconEvent argument. Note that not all ports are required to send these events and so it's better to override*CreatePopupMenu* (p. 1324) if all that the application does is that it shows a popup menu in reaction to mouse click.

EVT_TASKBAR_MOVE(func)	Process a wxEVT_TASKBAR_MOVE event.
EVT_TASKBAR_LEFT_DOWN(func)	Process a wxEVT_TASKBAR_LEFT_DOWN event.
EVT_TASKBAR_LEFT_UP(func)	Process a wxEVT_TASKBAR_LEFT_UP event.
EVT_TASKBAR_RIGHT_DOWN(func)	Process a wxEVT_TASKBAR_RIGHT_DOWN event.
EVT_TASKBAR_RIGHT_UP(func)	Process a wxEVT_TASKBAR_RIGHT_UP event.
EVT_TASKBAR_LEFT_DCLICK(func)	Process a wxEVT_TASKBAR_LEFT_DCLICK event.
EVT_TASKBAR_RIGHT_DCLICK(func)	Process a wxEVT_TASKBAR_RIGHT_DCLICK event.

wxTaskBarIcon::wxTaskBarIcon

wxTaskBarIcon()

Default constructor.

wxTaskBarIcon::~wxTaskBarIcon

~wxTaskBarIcon()

Destroys the wxTaskBarIcon object, removing the icon if not already removed.

wxTaskBarIcon::CreatePopupMenu

virtual wxMenu* CreatePopupMenu()

This method is called by the library when the user requests popup menu (on Windows and Unix platforms, this is when the user right-clicks the icon). Override this function in order to provide popup menu associated with the icon.

If CreatePopupIcon returns NULL (this happens by default), no menu is shown, otherwise the menu is displayed and then deleted by the library as soon as the user dismisses it. The events can be handled by a class derived from wxTaskBarIcon.

wxTaskBarIcon::IsIconInstalled

bool IsIconInstalled()

Returns true if *SetIcon* (p. 1325) was called with no subsequent *RemoveIcon* (p. 1325).

wxTaskBarIcon::IsOk

bool IsOk()

Returns true if the object initialized successfully.

wxTaskBarIcon::PopupMenu

bool PopupMenu(**wxMenu*** *menu*)

Pops up a menu at the current mouse position. The events can be handled by a class derived from wxTaskBarIcon.

Note

It is recommended to override*CreatePopupIcon* (p. 1324)callback instead of calling this method from event handler, because some ports (e.g. wxCocoa) may not implement PopupMenu and mouse click events at all.

wxTaskBarIcon::RemoveIcon

bool RemoveIcon()

Removes the icon previously set with *SetIcon* (p. 1325).

wxTaskBarIcon::SetIcon

bool SetIcon(**const wxIcon&** *icon*, **const wxString&** *tooltip*)

Sets the icon, and optional tooltip text.

wxTCPClient

A wxTCPClient object represents the client part of a client-server conversation. It emulates a DDE-style protocol, but uses TCP/IP which is available on most platforms.

A DDE-based implementation for Windows is available using *wxDDEClient* (p. 386).

To create a client which can communicate with a suitable server, you need to derive a class from wxTCPConnection and another from wxTCPClient. The custom wxTCPConnection class will intercept communications in a 'conversation' with a server,

and the custom wxTCPServer is required so that a user-overridden
wxTCPClient::OnMakeConnection (p. 1326) member can return a wxTCPConnection of
the required class, when a connection is made.

Derived from

wxClientBase
wxObject (p. 1012)

Include files

<wx/sckipc.h>

See also

wxTCPServer (p. 1331), *wxTCPConnection* (p. 1327), *Interprocess communications
overview* (p. 1841)

wxTCPClient::wxTCPClient

wxTCPClient()

Constructs a client object.

wxTCPClient::MakeConnection

wxConnectionBase * MakeConnection(**const wxString&** *host*, **const wxString&**
service, **const wxString&** *topic*)

Tries to make a connection with a server specified by the host (a machine name under
Unix), service name (must contain an integer port number under Unix), and a topic
string. If the server allows a connection, a wxTCPConnection object will be returned. The
type of wxTCPConnection returned can be altered by overriding the
wxTCPClient::OnMakeConnection (p. 1326) member to return your own derived
connection object.

wxTCPClient::OnMakeConnection

wxConnectionBase * OnMakeConnection()

The type of *wxTCPConnection* (p. 1327) returned from a *wxTCPClient::MakeConnection*
(p. 1326) call can be altered by deriving the **OnMakeConnection** member to return your
own derived connection object. By default, a wxTCPConnection object is returned.

The advantage of deriving your own connection class is that it will enable you to
intercept messages initiated by the server, such as *wxTCPConnection::OnAdvise* (p.
1329). You may also want to store application-specific data in instances of the new
class.

wxTCPClient::ValidHost

bool ValidHost(**const wxString&** *host*)

Returns true if this is a valid host name, false otherwise.

wxTCPConnection

A wxTCPClient object represents the connection between a client and a server. It emulates a DDE-style protocol, but uses TCP/IP which is available on most platforms.

A DDE-based implementation for Windows is available using *wxDDEConnection* (p. 387).

A wxTCPConnection object can be created by making a connection using a *wxTCPClient* (p. 1325) object, or by the acceptance of a connection by a *wxTCPServer* (p. 1331) object. The bulk of a conversation is controlled by calling members in a **wxTCPConnection** object or by overriding its members.

An application should normally derive a new connection class from wxTCPConnection, in order to override the communication event handlers to do something interesting.

Derived from

wxConnectionBase
wxObject (p. 1012)

Include files

<wx/sckipc.h>

Types

wxIPCFormat is defined as follows:

```
enum wxIPCFormat
{
  wxIPC_INVALID =          0,
  wxIPC_TEXT =             1,  /* CF_TEXT */
  wxIPC_BITMAP =           2,  /* CF_BITMAP */
  wxIPC_METAFILE =         3,  /* CF_METAFILEPICT */
  wxIPC_SYLK =             4,
  wxIPC_DIF =              5,
  wxIPC_TIFF =             6,
  wxIPC_OEMTEXT =          7,  /* CF_OEMTEXT */
  wxIPC_DIB =              8,  /* CF_DIB */
  wxIPC_PALETTE =          9,
  wxIPC_PENDATA =          10,
  wxIPC_RIFF =             11,
  wxIPC_WAVE =             12,
  wxIPC_UNICODETEXT =      13,
  wxIPC_ENHMETAFILE =      14,
  wxIPC_FILENAME =         15, /* CF_HDROP */
  wxIPC_LOCALE =           16,
  wxIPC_PRIVATE =          20
};
```

wxTCPConnection::wxTCPConnection

wxTCPConnection()

wxTCPConnection(**char*** *buffer*, **int** *size*)

Constructs a connection object. If no user-defined connection object is to be derived from wxTCPConnection, then the constructor should not be called directly, since the default connection object will be provided on requesting (or accepting) a connection. However, if the user defines his or her own derived connection object, the *wxTCPServer::OnAcceptConnection* (p. 1331) and/or *wxTCPClient::OnMakeConnection* (p. 1326) members should be replaced by functions which construct the new connection object. If the arguments of the wxTCPConnection constructor are void, then a default buffer is associated with the connection. Otherwise, the programmer must provide a a buffer and size of the buffer for the connection object to use in transactions.

wxTCPConnection::Advise

bool Advise(**const wxString&** *item*, **char*** *data*, **int** *size = -1*, **wxIPCFormat** *format = wxCF_TEXT*)

Called by the server application to advise the client of a change in the data associated with the given item. Causes the client connection's *wxTCPConnection::OnAdvise* (p. 1329) member to be called. Returns true if successful.

wxTCPConnection::Execute

bool Execute(**char*** *data*, **int** *size = -1*, **wxIPCFormat** *format = wxCF_TEXT*)

Called by the client application to execute a command on the server. Can also be used to transfer arbitrary data to the server (similar to *wxTCPConnection::Poke* (p. 1330) in that respect). Causes the server connection's *wxTCPConnection::OnExecute* (p. 1329) member to be called. Returns true if successful.

wxTCPConnection::Disconnect

bool Disconnect()

Called by the client or server application to disconnect from the other program; it causes the *wxTCPConnection::OnDisconnect* (p. 1329) message to be sent to the

corresponding connection object in the other program. The default behaviour of
OnDisconnect is to delete the connection, but the calling application must explicitly
delete its side of the connection having called **Disconnect**. Returns true if successful.

wxTCPConnection::OnAdvise

virtual bool OnAdvise(**const wxString&** *topic*, **const wxString&** *item*, **char*** *data*, **int**
size, **wxIPCFormat** *format*)

Message sent to the client application when the server notifies it of a change in the data
associated with the given item.

wxTCPConnection::OnDisconnect

virtual bool OnDisconnect()

Message sent to the client or server application when the other application notifies it to
delete the connection. Default behaviour is to delete the connection object.

wxTCPConnection::OnExecute

virtual bool OnExecute(**const wxString&** *topic*, **char*** *data*, **int** *size*, **wxIPCFormat**
format)

Message sent to the server application when the client notifies it to execute the given
data. Note that there is no item associated with this message.

wxTCPConnection::OnPoke

virtual bool OnPoke(**const wxString&** *topic*, **const wxString&** *item*, **char*** *data*, **int**
size, **wxIPCFormat** *format*)

Message sent to the server application when the client notifies it to accept the given
data.

wxTCPConnection::OnRequest

virtual char* OnRequest(**const wxString&** *topic*, **const wxString&** *item*, **int ****size*,
wxIPCFormat *format*)

Message sent to the server application when the client calls *wxTCPConnection::Request*
(p. 1330). The server should respond by returning a character string from **OnRequest**,
or NULL to indicate no data.

wxTCPConnection::OnStartAdvise

virtual bool OnStartAdvise(**const wxString&** *topic*, **const wxString&** *item*)

Message sent to the server application by the client, when the client wishes to start an 'advise loop' for the given topic and item. The server can refuse to participate by returning false.

wxTCPConnection::OnStopAdvise

virtual bool OnStopAdvise(**const wxString&** *topic*, **const wxString&** *item*)

Message sent to the server application by the client, when the client wishes to stop an 'advise loop' for the given topic and item. The server can refuse to stop the advise loop by returning false, although this doesn't have much meaning in practice.

wxTCPConnection::Poke

bool Poke(**const wxString&** *item*, **char*** *data*, **int** *size = -1*, **wxIPCFormat** *format = wxCF_TEXT*)

Called by the client application to poke data into the server. Can be used to transfer arbitrary data to the server. Causes the server connection's *wxTCPConnection::OnPoke* (p. 1329) member to be called. Returns true if successful.

wxTCPConnection::Request

char* Request(**const wxString&** *item*, **int ****size*, **wxIPCFormat** *format = wxIPC_TEXT*)

Called by the client application to request data from the server. Causes the server connection's *wxTCPConnection::OnRequest* (p. 1329) member to be called. Returns a character string (actually a pointer to the connection's buffer) if successful, NULL otherwise.

wxTCPConnection::StartAdvise

bool StartAdvise(**const wxString&** *item*)

Called by the client application to ask if an advise loop can be started with the server. Causes the server connection's *wxTCPConnection::OnStartAdvise* (p. 1329) member to be called. Returns true if the server okays it, false otherwise.

wxTCPConnection::StopAdvise

bool StopAdvise(**const wxString&** *item*)

Called by the client application to ask if an advise loop can be stopped. Causes the server connection's *wxTCPConnection::OnStopAdvise* (p. 1330) member to be called. Returns true if the server okays it, false otherwise.

wxTCPServer

A wxTCPServer object represents the server part of a client-server conversation. It emulates a DDE-style protocol, but uses TCP/IP which is available on most platforms.

A DDE-based implementation for Windows is available using *wxDDEServer* (p. 391).

Derived from

wxServerBase
wxObject (p. 1012)

Include files

<wx/sckipc.h>

See also

wxTCPClient (p. 1325), *wxTCPConnection* (p. 1327), *IPC overview* (p. 1841)

wxTCPServer::wxTCPServer

wxTCPServer()

Constructs a server object.

wxTCPServer::Create

bool Create(**const wxString&** *service*)

Registers the server using the given service name. Under Unix, the string must contain an integer id which is used as an Internet port number. false is returned if the call failed (for example, the port number is already in use).

wxTCPServer::OnAcceptConnection

virtual wxConnectionBase * OnAcceptConnection(**const wxString&** *topic*)

When a client calls **MakeConnection**, the server receives the message and this member is called. The application should derive a member to intercept this message and return a connection object of either the standard wxTCPConnection type, or of a user-derived type. If the topic is "STDIO'', the application may wish to refuse the connection. Under Unix, when a server is created the OnAcceptConnection message is always sent for standard input and output.

wxTempFile

wxTempFile provides a relatively safe way to replace the contents of the existing file. The name is explained by the fact that it may be also used as just a temporary file if you don't replace the old file contents.

Usually, when a program replaces the contents of some file it first opens it for writing, thus losing all of the old data and then starts recreating it. This approach is not very safe because during the regeneration of the file bad things may happen: the program may find that there is an internal error preventing it from completing file generation, the user may interrupt it (especially if file generation takes long time) and, finally, any other external interrupts (power supply failure or a disk error) will leave you without either the original file or the new one.

wxTempFile addresses this problem by creating a temporary file which is meant to replace the original file - but only after it is fully written. So, if the user interrupts the program during the file generation, the old file won't be lost. Also, if the program discovers itself that it doesn't want to replace the old file there is no problem - in fact, wxTempFile will **not** replace the old file by default, you should explicitly call *Commit* (p. 1334) to do it. Calling *Discard* (p. 1334) explicitly discards any modifications: it closes and deletes the temporary file and leaves the original file unchanged. If you don't call neither of Commit() and Discard(), the destructor will call Discard() automatically.

To summarize: if you want to replace another file, create an instance of wxTempFile passing the name of the file to be replaced to the constructor (you may also use default constructor and pass the file name to *Open* (p. 1333)). Then you can *write* (p. 1333) to wxTempFile using *wxFile* (p. 502)-like functions and later call Commit() to replace the old file (and close this one) or call Discard() to cancel the modifications.

Derived from

No base class

Include files

<wx/file.h>

See also:

wxFile (p. 502)
wxTempFileOutputStream (p. 1334)

wxTempFile::wxTempFile

wxTempFile()

Default constructor - *Open* (p. 1333) must be used to open the file.

wxTempFile::wxTempFile

wxTempFile(**const wxString&** *strName*)

Associates wxTempFile with the file to be replaced and opens it. You should use *IsOpened* (p. 1333) to verify if the constructor succeeded.

wxTempFile::Open

bool Open(**const wxString&** *strName*)

Open the temporary file, returns `true` on success, `false` if an error occurred.

strName is the name of file to be replaced. The temporary file is always created in the directory where *strName* is. In particular, if *strName* doesn't include the path, it is created in the current directory and the program should have write access to it for the function to succeed.

wxTempFile::IsOpened

bool IsOpened() **const**

Returns `true` if the file was successfully opened.

wxTempFile::Length

wxFileOffset Length() **const**

Returns the length of the file.

wxTempFile::Seek

wxFileOffset Seek(**wxFileOffset** *ofs*, **wxSeekMode** *mode* = *wxFromStart*)

Seeks to the specified position.

wxTempFile::Tell

wxFileOffset Tell() **const**

Returns the current position or wxInvalidOffset if file is not opened or if another error occurred.

wxTempFile::Write

bool Write(**const void** **p*, **size_t** *n*)

Write to the file, return `true` on success, `false` on failure.

wxTempFile::Write

bool Write(**const wxString&** *str*, **wxMBConv&** *conv* = *wxConvLibc*)

Write to the file, return `true` on success, `false` on failure.

The second argument is only meaningful in Unicode build of wxWidgets when*conv* is used to convert *str* to multibyte representation.

wxTempFile::Commit

bool Commit()

Validate changes: deletes the old file of name m_strName and renames the new file to the old name. Returns `true` if both actions succeeded. If `false` is returned it may unfortunately mean two quite different things: either that either the old file couldn't be deleted or that the new file couldn't be renamed to the old name.

wxTempFile::Discard

void Discard()

Discard changes: the old file contents is not changed, temporary file is deleted.

wxTempFile::~wxTempFile

~wxTempFile()

Destructor calls *Discard()* (p. 1334) if temporary file is still opened.

wxTempFileOutputStream

wxTempFileOutputStream is an output stream based on *wxTempFile* (p. 1331). It provides a relatively safe way to replace the contents of the existing file.

Derived from

wxOutputStream (p. 1016)

Include files

<wx/wfstream.h>

See also

wxTempFile (p. 1331)

wxTempFileOutputStream::wxTempFileOutputStream

wxTempFileOutputStream(**const wxString&** *fileName*)

Associates wxTempFileOutputStream with the file to be replaced and opens it. You should use *IsOk* (p. 1276) to verify if the constructor succeeded.

Call *Commit()* (p. 1335) or *Close()* (p. 1017) to replace the old file and close this one. Calling *Discard()* (p. 1335) (or allowing the destructor to do it) will discard the changes.

wxTempFileOutputStream::Commit

bool Commit()

Validate changes: deletes the old file of the given name and renames the new file to the old name. Returns `true` if both actions succeeded. If `false` is returned it may unfortunately mean two quite different things: either that either the old file couldn't be deleted or that the new file couldn't be renamed to the old name.

wxTempFileOutputStream::Discard

void Discard()

Discard changes: the old file contents are not changed, the temporary file is deleted.

wxTextAttr

wxTextAttr represents the character and paragraph attributes, or style, for a range of text in a *wxTextCtrl* (p. 1340).

When setting up a wxTextAttr object, pass a bitlist mask to SetFlags to indicate which style elements should be changed. As a convenience, when you call a setter such as SetFont, the relevant bit will be set.

Derived from

No base class

Include files

<wx/textctrl.h>

Typedefs

`wxTextPos` is the type containing the index of a position in a text control. `wxTextCoord` contains the index of a column or a row in the control.

Note that although both of these types should probably have been unsigned, due to backwards compatibility reasons, are defined as `long` currently. Their use (instead of

plain `long`) is still encouraged as it makes the code more readable.

Constants

The following values can be passed to SetAlignment to determine paragraph alignment.

```
enum wxTextAttrAlignment
{
    wxTEXT_ALIGNMENT_DEFAULT,
    wxTEXT_ALIGNMENT_LEFT,
    wxTEXT_ALIGNMENT_CENTRE,
    wxTEXT_ALIGNMENT_CENTER = wxTEXT_ALIGNMENT_CENTRE,
    wxTEXT_ALIGNMENT_RIGHT,
    wxTEXT_ALIGNMENT_JUSTIFIED
};
```

These values are passed in a bitlist to SetFlags to determine what attributes will be considered when setting the attributes for a text control.

```
#define wxTEXT_ATTR_TEXT_COLOUR              0x0001
#define wxTEXT_ATTR_BACKGROUND_COLOUR        0x0002
#define wxTEXT_ATTR_FONT_FACE                0x0004
#define wxTEXT_ATTR_FONT_SIZE                0x0008
#define wxTEXT_ATTR_FONT_WEIGHT              0x0010
#define wxTEXT_ATTR_FONT_ITALIC              0x0020
#define wxTEXT_ATTR_FONT_UNDERLINE           0x0040
#define wxTEXT_ATTR_FONT \
  wxTEXT_ATTR_FONT_FACE | wxTEXT_ATTR_FONT_SIZE | wxTEXT_ATTR_FONT_WEIGHT
\
| wxTEXT_ATTR_FONT_ITALIC | wxTEXT_ATTR_FONT_UNDERLINE
#define wxTEXT_ATTR_ALIGNMENT                0x0080
#define wxTEXT_ATTR_LEFT_INDENT              0x0100
#define wxTEXT_ATTR_RIGHT_INDENT             0x0200
#define wxTEXT_ATTR_TABS                     0x0400
```

```
The values below are the possible return codes of theHitTest (p. 1350)
method:// the point asked is ...
enum wxTextCtrlHitTestResult
{
    wxTE_HT_UNKNOWN = -2,    // this means HitTest() is simply not
implemented
    wxTE_HT_BEFORE,          // either to the left or upper
    wxTE_HT_ON_TEXT,         // directly on
    wxTE_HT_BELOW,           // below [the last line]
    wxTE_HT_BEYOND           // after [the end of line]
};
// ... the character returned
```

wxTextAttr::wxTextAttr

wxTextAttr()

wxTextAttr(**const wxColour&** *colText*, **const wxColour&** *colBack = wxNullColour*, **const wxFont&** *font = wxNullFont*, **wxTextAttrAlignment** *alignment = wxTEXT_ALIGNMENT_DEFAULT*)

The constructors initialize one or more of the text foreground colour, background colour, font, and alignment. The values not initialized in the constructor can be set later, otherwise *wxTextCtrl::SetStyle* (p. 1356) will use the default values for them.

wxTextAttr::GetAlignment

wxTextAttrAlignment GetAlignment() **const**

Returns the paragraph alignment.

wxTextAttr::GetBackgroundColour

const wxColour& GetBackgroundColour() **const**

Return the background colour specified by this attribute.

wxTextAttr::GetFont

const wxFont& GetFont() **const**

Return the text font specified by this attribute.

wxTextAttr::GetLeftIndent

int GetLeftIndent() **const**

Returns the left indent in tenths of a millimetre.

wxTextAttr::GetLeftSubIndent

int GetLeftSubIndent() **const**

Returns the left sub indent for all lines but the first line in a paragraph in tenths of a millimetre.

wxTextAttr::GetRightIndent

int GetRightIndent() **const**

Returns the right indent in tenths of a millimetre.

wxTextAttr::GetTabs

const wxArrayInt& GetTabs() **const**

Returns the array of integers representing the tab stops. Each array element specifies the tab stop in tenths of a millimetre.

wxTextAttr::GetTextColour

const wxColour& GetTextColour() **const**

Return the text colour specified by this attribute.

wxTextAttr::HasBackgroundColour

bool HasBackgroundColour() **const**

Returns `true` if this style specifies the background colour to use.

wxTextAttr::HasFont

bool HasFont() **const**

Returns `true` if this style specifies the font to use.

wxTextAttr::HasTextColour

bool HasTextColour() **const**

Returns `true` if this style specifies the foreground colour to use.

wxTextAttr::GetFlags

long GetFlags()

Returns a bitlist indicating which attributes will be set.

wxTextAttr::IsDefault

bool IsDefault() **const**

Returns `true` if this style specifies any non-default attributes.

wxTextAttr::SetAlignment

void SetAlignment(**wxTextAttrAlignment** *alignment*)

Sets the paragraph alignment.

wxTextAttr::SetBackgroundColour

void SetBackgroundColour(**const wxColour&** *colour*)

Sets the background colour.

wxTextAttr::SetFlags

void SetFlags(**long** *flags*)

Pass a bitlist indicating which attributes will be set.

wxTextAttr::SetFont

void SetFont(**const wxFont&** *font*)

Sets the text font.

wxTextAttr::SetLeftIndent

void SetLeftIndent(**int** *indent*, **int** *subIndent = 0*)

Sets the left indent in tenths of a millimetre. subIndent sets the indent for all lines but the first line in a paragraph relative to the first line.

wxTextAttr::SetRightIndent

void SetRightIndent(**int** *indent*)

Sets the right indent in tenths of a millimetre.

wxTextAttr::SetTabs

void SetTabs(**const wxArrayInt&** *tabs*)

Sets the array of integers representing the tab stops. Each array element specifies the tab stop in tenths of a millimetre.

wxTextAttr::SetTextColour

void SetTextColour(**const wxColour&** *colour*)

Sets the text colour.

wxTextCtrl

A text control allows text to be displayed and edited. It may be single line or multi-line.

Derived from

streambuf
wxControl (p. 215)
wxWindow (p. 1490)
wxEvtHandler (p. 486)
wxObject (p. 1012)

Include files

<wx/textctrl.h>

Window styles

wxTE_PROCESS_ENTER	The control will generate the event wxEVT_COMMAND_TEXT_ENTER (otherwise pressing Enter key is either processed internally by the control or used for navigation between dialog controls).
wxTE_PROCESS_TAB	The control will receive wxEVT_CHAR events for TAB pressed - normally, TAB is used for passing to the next control in a dialog instead. For the control created with this style, you can still use Ctrl-Enter to pass to the next control from the keyboard.
wxTE_MULTILINE	The text control allows multiple lines.
wxTE_PASSWORD	The text will be echoed as asterisks.
wxTE_READONLY	The text will not be user-editable.
wxTE_RICH	Use rich text control under Win32, this allows to have more than 64KB of text in the control even under Win9x. This style is ignored under other platforms.
wxTE_RICH2	Use rich text control version 2.0 or 3.0 under Win32, this style is ignored under other platforms
wxTE_AUTO_URL	Highlight the URLs and generate the wxTextUrlEvents when mouse events occur over them. This style is only supported for wxTE_RICH Win32 and multi-line wxGTK2 text controls.
wxTE_NOHIDESEL	By default, the Windows text control doesn't show the selection when it doesn't have focus - use this style to force it to always show it. It doesn't do anything under other

platforms.

wxHSCROLL	A horizontal scrollbar will be created and used, so that text won't be wrapped. No effect under wxGTK1.
wxTE_LEFT	The text in the control will be left-justified (default).
wxTE_CENTRE	The text in the control will be centered (currently wxMSW and wxGTK2 only).
wxTE_RIGHT	The text in the control will be right-justified (currently wxMSW and wxGTK2 only).
wxTE_DONTWRAP	Same as `wxHSCROLL` style: don't wrap at all, show horizontal scrollbar instead.
wxTE_CHARWRAP	Wrap the lines too long to be shown entirely at any position (wxUniv and wxGTK2 only).
wxTE_WORDWRAP	Wrap the lines too long to be shown entirely at word boundaries (wxUniv and wxGTK2 only).
wxTE_BESTWRAP	Wrap the lines at word boundaries or at any other character if there are words longer than the window width (this is the default).
wxTE_CAPITALIZE	On PocketPC and Smartphone, causes the first letter to be capitalized.

See also *window styles overview* (p. 1762) and *wxTextCtrl::wxTextCtrl* (p. 1343).

wxTextCtrl text format

The multiline text controls always store the text as a sequence of lines separated by `\n` characters, i.e. in the Unix text format even on non-Unix platforms. This allows the user code to ignore the differences between the platforms but at a price: the indices in the control such as those returned by *GetInsertionPoint* (p. 1347) or*GetSelection* (p. 1349) can **not** be used as indices into the string returned by *GetValue* (p. 1350) as they're going to be slightly off for platforms using`\r\n` as separator (as Windows does), for example.

Instead, if you need to obtain a substring between the 2 indices obtained from the control with the help of the functions mentioned above, you should use *GetRange* (p. 1349). And the indices themselves can only be passed to other methods, for example*SetInsertionPoint* (p. 1355) or*SetSelection* (p. 1356).

To summarize: never use the indices returned by (multiline) wxTextCtrl as indices into the string it contains, but only as arguments to be passed back to the other wxTextCtrl methods.

wxTextCtrl styles

Multi-line text controls support the styles, i.e. provide a possibility to set colours and font for individual characters in it (note that under Windows `wxTE_RICH` style is required for

style support). To use the styles you can either call *SetDefaultStyle* (p. 1354) before inserting the text or call *SetStyle* (p. 1356) later to change the style of the text already in the control (the first solution is much more efficient).

In either case, if the style doesn't specify some of the attributes (for example you only want to set the text colour but without changing the font nor the text background), the values of the default style will be used for them. If there is no default style, the attributes of the text control itself are used.

So the following code correctly describes what it does: the second call to *SetDefaultStyle* (p. 1354) doesn't change the text foreground colour (which stays red) while the last one doesn't change the background colour (which stays grey):

```
text->SetDefaultStyle(wxTextAttr(*wxRED));
text->AppendText("Red text\n");
text->SetDefaultStyle(wxTextAttr(wxNullColour, *wxLIGHT_GREY));
text->AppendText("Red on grey text\n");
text->SetDefaultStyle(wxTextAttr(*wxBLUE);
text->AppendText("Blue on grey text\n");
```

wxTextCtrl and C++ streams

This class multiply-inherits from **streambuf** where compilers allow, allowing code such as the following:

```
wxTextCtrl *control = new wxTextCtrl(...);

ostream stream(control)

stream << 123.456 << " some text\n";
stream.flush();
```

If your compiler does not support derivation from **streambuf** and gives a compile error, define the symbol **NO_TEXT_WINDOW_STREAM** in the wxTextCtrl header file.

Note that independently of this setting you can always use wxTextCtrl itself in a stream-like manner:

```
wxTextCtrl *control = new wxTextCtrl(...);

*control << 123.456 << " some text\n";
```

always works. However the possibility to create an ostream associated with wxTextCtrl may be useful if you need to redirect the output of a function taking an ostream as parameter to a text control.

Another commonly requested need is to redirect **std::cout** to the text control. This could be done in the following way:

```
#include <iostream>

wxTextCtrl *control = new wxTextCtrl(...);

std::streambuf *sbOld = std::cout.rdbuf();
std::cout.rdbuf(*control);

// use cout as usual, the output appears in the text control
```

```
        ...

        std::cout.rdbuf(sbOld);
```

But wxWidgets provides a convenient class to make it even simpler so instead you may just do

```
        #include <iostream>

        wxTextCtrl *control = new wxTextCtrl(...);

        wxStreamToTextRedirector redirect(control);

        // all output to cout goes into the text control until the exit from
    current
        // scope
```

See *wxStreamToTextRedirector* (p. 1283) for more details.

Event handling

The following commands are processed by default event handlers in wxTextCtrl: wxID_CUT, wxID_COPY, wxID_PASTE, wxID_UNDO, wxID_REDO. The associated UI update events are also processed automatically, when the control has the focus.

To process input from a text control, use these event handler macros to direct input to member functions that take a *wxCommandEvent* (p. 180) argument.

EVT_TEXT(id, func)	Respond to a wxEVT_COMMAND_TEXT_UPDATED event, generated when the text changes. Notice that this event will always be sent when the text controls contents changes - whether this is due to user input or comes from the program itself (for example, if SetValue() is called)
EVT_TEXT_ENTER(id, func)	Respond to a wxEVT_COMMAND_TEXT_ENTER event, generated when enter is pressed in a text control (which must have wxTE_PROCESS_ENTER style for this event to be generated).
EVT_TEXT_URL(id, func)	A mouse event occurred over an URL in the text control (wxMSW and wxGTK2 only)
EVT_TEXT_MAXLEN(id, func)	User tried to enter more text into the control than the limit set by*SetMaxLength* (p. 1355).

wxTextCtrl::wxTextCtrl

wxTextCtrl()

Default constructor.

 wxTextCtrl(**wxWindow*** *parent*, **wxWindowID** *id*, **const wxString&** *value* = *""*, **const wxPoint&** *pos* = *wxDefaultPosition*, **const wxSize&** *size* = *wxDefaultSize*, **long** *style* = *0*, **const wxValidator&** *validator* = *wxDefaultValidator*, **const wxString&** *name* = *wxTextCtrlNameStr*)

Constructor, creating and showing a text control.

Parameters

parent

> Parent window. Should not be NULL.

id

> Control identifier. A value of -1 denotes a default value.

value

> Default text value.

pos

> Text control position.

size

> Text control size.

style

> Window style. See *wxTextCtrl* (p. 1340).

validator

> Window validator.

name

> Window name.

Remarks

The horizontal scrollbar (**wxHSCROLL** style flag) will only be created for multi-line text controls. Without a horizontal scrollbar, text lines that don't fit in the control's size will be wrapped (but no newline character is inserted). Single line controls don't have a horizontal scrollbar, the text is automatically scrolled so that the *insertion point* (p. 1347) is always visible.

See also

wxTextCtrl::Create (p. 1346), *wxValidator* (p. 1461)

wxTextCtrl::~wxTextCtrl

 ~wxTextCtrl()

Destructor, destroying the text control.

wxTextCtrl::AppendText

void AppendText(**const wxString&** *text*)

Appends the text to the end of the text control.

Parameters

text

> Text to write to the text control.

Remarks

After the text is appended, the insertion point will be at the end of the text control. If this behaviour is not desired, the programmer should use *GetInsertionPoint* (p. 1347) and *SetInsertionPoint* (p. 1355).

See also

wxTextCtrl::WriteText (p. 1357)

wxTextCtrl::CanCopy

virtual bool CanCopy()

Returns `true` if the selection can be copied to the clipboard.

wxTextCtrl::CanCut

virtual bool CanCut()

Returns `true` if the selection can be cut to the clipboard.

wxTextCtrl::CanPaste

virtual bool CanPaste()

Returns `true` if the contents of the clipboard can be pasted into the text control. On some platforms (Motif, GTK) this is an approximation and returns `true` if the control is editable, `false` otherwise.

wxTextCtrl::CanRedo

virtual bool CanRedo()

Returns `true` if there is a redo facility available and the last operation can be redone.

wxTextCtrl::CanUndo

virtual bool CanUndo()

Returns `true` if there is an undo facility available and the last operation can be undone.

wxTextCtrl::Clear

virtual void Clear()

Clears the text in the control.

Note that this function will generate a `wxEVT_COMMAND_TEXT_UPDATED`event.

wxTextCtrl::Copy

virtual void Copy()

Copies the selected text to the clipboard under Motif and MS Windows.

wxTextCtrl::Create

bool Create(**wxWindow*** *parent*, **wxWindowID** *id*, **const wxString&** *value = ""*, **const wxPoint&** *pos = wxDefaultPosition*, **const wxSize&** *size = wxDefaultSize*, **long** *style = 0*, **const wxValidator&** *validator = wxDefaultValidator*, **const wxString&** *name = wxTextCtrlNameStr*)

Creates the text control for two-step construction. Derived classes should call or replace this function. See *wxTextCtrl::wxTextCtrl* (p. 1343) for further details.

wxTextCtrl::Cut

virtual void Cut()

Copies the selected text to the clipboard and removes the selection.

wxTextCtrl::DiscardEdits

void DiscardEdits()

Resets the internal 'modified' flag as if the current edits had been saved.

wxTextCtrl::EmulateKeyPress

bool EmulateKeyPress(**const wxKeyEvent&** *event*)

This functions inserts into the control the character which would have been inserted if the given key event had occurred in the text control. The *event* object should be the same as the one passed to `EVT_KEY_DOWN` handler previously by wxWidgets.

Please note that this function doesn't currently work correctly for all keys under any platform but MSW.

Return value

`true` if the event resulted in a change to the control, `false` otherwise.

wxTextCtrl::GetDefaultStyle

const wxTextAttr& GetDefaultStyle() **const**

Returns the style currently used for the new text.

See also

SetDefaultStyle (p. 1354)

wxTextCtrl::GetInsertionPoint

virtual long GetInsertionPoint() **const**

Returns the insertion point. This is defined as the zero based index of the character position to the right of the insertion point. For example, if the insertion point is at the end of the text control, it is equal to both *GetValue()* (p. 1350).Length() and *GetLastPosition()* (p. 1347).

The following code snippet safely returns the character at the insertion point or the zero character if the point is at the end of the control.

```
char GetCurrentChar(wxTextCtrl *tc) {
  if (tc->GetInsertionPoint() == tc->GetLastPosition())
    return '\0';
  return tc->GetValue[tc->GetInsertionPoint()];
}
```

wxTextCtrl::GetLastPosition

virtual wxTextPos GetLastPosition() **const**

Returns the zero based index of the last position in the text control, which is equal to the number of characters in the control.

wxTextCtrl::GetLineLength

int GetLineLength(**long** *lineNo*) **const**

Gets the length of the specified line, not including any trailing newline character(s).

Parameters

lineNo

> Line number (starting from zero).

Return value

The length of the line, or -1 if *lineNo* was invalid.

wxTextCtrl::GetLineText

wxString GetLineText(**long** *lineNo*) **const**

Returns the contents of a given line in the text control, not including any trailing newline character(s).

Parameters

lineNo

> The line number, starting from zero.

Return value

The contents of the line.

wxTextCtrl::GetNumberOfLines

int GetNumberOfLines() **const**

Returns the number of lines in the text control buffer.

Remarks

Note that even empty text controls have one line (where the insertion point is), so GetNumberOfLines() never returns 0.

For gtk_text (multi-line) controls, the number of lines is calculated by actually counting newline characters in the buffer. You may wish to avoid using functions that work with line numbers if you are working with controls that contain large amounts of text.

wxTextCtrl::GetRange

virtual wxString GetRange(**long** *from*, **long** *to*) **const**

Returns the string containing the text starting in the positions *from* and up to *to* in the control. The positions must have been returned by another wxTextCtrl method.

Please note that the positions in a multiline wxTextCtrl do **not**correspond to the indices in the string returned by*GetValue* (p. 1350) because of the different new line representations (CR or CR LF) and so this method should be used to obtain the correct results instead of extracting parts of the entire value. It may also be more efficient, especially if the control contains a lot of data.

wxTextCtrl::GetSelection

virtual void GetSelection(**long*** *from*, **long*** *to*) **const**

Gets the current selection span. If the returned values are equal, there was no selection.

Please note that the indices returned may be used with the other wxTextctrl methods but don't necessarily represent the correct indices into the string returned by *GetValue()* (p. 1350) for multiline controls under Windows (at least,) you should use*GetStringSelection()* (p. 1349) to get the selected text.

Parameters

from

> The returned first position.

to

> The returned last position.

wxPython note: The wxPython version of this method returns a tuple consisting of the from and to values.

wxPerl note: In wxPerl this method takes no parameter and returns a 2-element list `(from, to )`.

wxTextCtrl::GetStringSelection

virtual wxString GetStringSelection()

Gets the text currently selected in the control. If there is no selection, the returned string is empty.

wxTextCtrl::GetStyle

bool GetStyle(**long** *position*, **wxTextAttr&** *style*)

Returns the style at this position in the text control. Not all platforms support this function.

Return value

`true` on success, `false` if an error occurred - it may also mean that the styles are not supported under this platform.

See also

wxTextCtrl::SetStyle (p. 1356), *wxTextAttr* (p. 1335)

wxTextCtrl::GetValue

wxString GetValue() **const**

Gets the contents of the control. Notice that for a multiline text control, the lines will be separated by (Unix-style) \n characters, even under Windows where they are separated by a \r\n sequence in the native control.

wxTextCtrl::HitTest

wxTextCtrlHitTestResult HitTest(**const wxPoint&** *pt*, **wxTextCoord** **col*, **wxTextCoord** **row*) **const**

This function finds the character at the specified position expressed in pixels. If the return code is not `wxTE_HT_UNKNOWN` the row and column of the character closest to this position are returned in the *col* and*row* parameters (unless the pointers are `NULL` which is allowed).

Please note that this function is currently only implemented in wxUniv, wxMSW and wxGTK2 ports.

See also

PositionToXY (p. 1352), *XYToPosition* (p. 1358)

wxPerl note: In wxPerl this function takes only the position argument and returns a 3-element list (result, col, row).

wxTextCtrl::IsEditable

bool IsEditable() **const**

Returns `true` if the controls contents may be edited by user (note that it always can be changed by the program), i.e. if the control hasn't been put in read-only mode by a previous call to*SetEditable* (p. 1355).

wxTextCtrl::IsModified

bool IsModified() **const**

Returns `true` if the text has been modified by user. Note that calling*SetValue* (p. 1357) doesn't make the control modified.

See also

MarkDirty (p. 1351)

wxTextCtrl::IsMultiLine

bool IsMultiLine() **const**

Returns `true` if this is a multi line edit control and `false`otherwise.

See also

IsSingleLine (p. 1351)

wxTextCtrl::IsSingleLine

bool IsSingleLine() **const**

Returns `true` if this is a single line edit control and `false`otherwise.

See also

IsMultiLine (p. 1351)

wxTextCtrl::LoadFile

bool LoadFile(**const wxString&** *filename*)

Loads and displays the named file, if it exists.

Parameters

filename

The filename of the file to load.

Return value

`true` if successful, `false` otherwise.

wxTextCtrl::MarkDirty

void MarkDirty()

Mark text as modified (dirty).

See also

IsModified (p. 1351)

wxTextCtrl::OnDropFiles

void OnDropFiles(**wxDropFilesEvent&** *event*)

This event handler function implements default drag and drop behaviour, which is to load the first dropped file into the control.

Parameters

event

The drop files event.

Remarks

This is not implemented on non-Windows platforms.

See also

wxDropFilesEvent (p. 466)

wxTextCtrl::Paste

virtual void Paste()

Pastes text from the clipboard to the text item.

wxTextCtrl::PositionToXY

bool PositionToXY(**long** *pos*, **long** *x, **long** *y) **const**

Converts given position to a zero-based column, line number pair.

Parameters

pos

Position.

x

Receives zero based column number.

y

Receives zero based line number.

Return value

`true` on success, `false` on failure (most likely due to a too large position parameter).

See also

wxTextCtrl::XYToPosition (p. 1358)

wxPython note: In Python, PositionToXY() returns a tuple containing the x and y values, so (x,y) = PositionToXY() is equivalent to the call described above.

wxPerl note: In wxPerl this method only takes the `pos` parameter, and returns a 2-element list (`x, y`).

wxTextCtrl::Redo

virtual void Redo()

If there is a redo facility and the last operation can be redone, redoes the last operation. Does nothing if there is no redo facility.

wxTextCtrl::Remove

virtual void Remove(**long** *from*, **long** *to*)

Removes the text starting at the first given position up to (but not including) the character at the last position.

Parameters

from

> The first position.

to

> The last position.

wxTextCtrl::Replace

virtual void Replace(**long** *from*, **long** *to*, **const wxString&** *value*)

Replaces the text starting at the first position up to (but not including) the character at the last position with the given text.

Parameters

from

The first position.

to

The last position.

value

The value to replace the existing text with.

wxTextCtrl::SaveFile

bool SaveFile(**const wxString&** *filename*)

Saves the contents of the control in a text file.

Parameters

filename

The name of the file in which to save the text.

Return value

`true` if the operation was successful, `false` otherwise.

wxTextCtrl::SetDefaultStyle

bool SetDefaultStyle(**const wxTextAttr&** *style*)

Changes the default style to use for the new text which is going to be added to the control using *WriteText* (p. 1357) or *AppendText* (p. 1345).

If either of the font, foreground, or background colour is not set in *style*, the values of the previous default style are used for them. If the previous default style didn't set them neither, the global font or colours of the text control itself are used as fall back.

However if the *style* parameter is the default wxTextAttr, then the default style is just reset (instead of being combined with the new style which wouldn't change it at all).

Parameters

style

The style for the new text.

Return value

`true` on success, `false` if an error occurred - may also mean that the styles are not supported under this platform.

See also

GetDefaultStyle (p. 1347)

wxTextCtrl::SetEditable

virtual void SetEditable(**const bool** *editable*)

Makes the text item editable or read-only, overriding the **wxTE_READONLY** flag.

Parameters

editable

If `true`, the control is editable. If `false`, the control is read-only.

See also

IsEditable (p. 1350)

wxTextCtrl::SetInsertionPoint

virtual void SetInsertionPoint(**long** *pos*)

Sets the insertion point at the given position.

Parameters

pos

Position to set.

wxTextCtrl::SetInsertionPointEnd

virtual void SetInsertionPointEnd()

Sets the insertion point at the end of the text control. This is equivalent to *SetInsertionPoint* (p. 1355)(*GetLastPosition* (p. 1347)()).

wxTextCtrl::SetMaxLength

virtual void SetMaxLength(**unsigned long** *len*)

This function sets the maximum number of characters the user can enter into the control. In other words, it allows to limit the text value length to *len*not counting the terminating `NUL` character.

If *len* is 0, the previously set max length limit, if any, is discarded and the user may enter as much text as the underlying native text control widget supports (typically at least 32Kb).

If the user tries to enter more characters into the text control when it already is filled up to the maximal length, a`wxEVT_COMMAND_TEXT_MAXLEN` event is sent to notify the program about it (giving it the possibility to show an explanatory message, for example) and the extra input is discarded.

Note that under GTK+, this function may only be used with single line text controls.

Compatibility

Only implemented in wxMSW/wxGTK starting with wxWidgets 2.3.2.

wxTextCtrl::SetSelection

virtual void SetSelection(**long** *from*, **long** *to*)

Selects the text starting at the first position up to (but not including) the character at the last position. If both parameters are equal to -1 all text in the control is selected.

Parameters

from

> The first position.

to

> The last position.

wxTextCtrl::SetStyle

bool SetStyle(**long** *start*, **long** *end*, **const wxTextAttr&** *style*)

Changes the style of the given range. If any attribute within *style* is not set, the corresponding attribute from *GetDefaultStyle()* (p. 1347) is used.

Parameters

start

> The start of the range to change.

end

> The end of the range to change.

style

> The new style for the range.

Return value

`true` on success, `false` if an error occurred - it may also mean that the styles are not

supported under this platform.

See also

wxTextCtrl::GetStyle (p. 1349), *wxTextAttr* (p. 1335)

wxTextCtrl::SetValue

virtual void SetValue(**const wxString&** *value*)

Sets the text value and marks the control as not-modified (which means that*IsModified* (p. 1351) would return `false` immediately after the call to SetValue).

Note that this function will generate a `wxEVT_COMMAND_TEXT_UPDATED`event.

Parameters

value

> The new value to set. It may contain newline characters if the text control is multi-line.

wxTextCtrl::ShowPosition

void ShowPosition(**long** *pos*)

Makes the line containing the given position visible.

Parameters

pos

> The position that should be visible.

wxTextCtrl::Undo

virtual void Undo()

If there is an undo facility and the last operation can be undone, undoes the last operation. Does nothing if there is no undo facility.

wxTextCtrl::WriteText

void WriteText(**const wxString&** *text*)

Writes the text into the text control at the current insertion position.

Parameters

text

> Text to write to the text control.

Remarks

Newlines in the text string are the only control characters allowed, and they will cause appropriate line breaks. See *wxTextCtrl::<<* (p. 1358) and *wxTextCtrl::AppendText* (p. 1345) for more convenient ways of writing to the window.

After the write operation, the insertion point will be at the end of the inserted text, so subsequent write operations will be appended. To append text after the user may have interacted with the control, call *wxTextCtrl::SetInsertionPointEnd* (p. 1355) before writing.

wxTextCtrl::XYToPosition

long XYToPosition(**long** *x*, **long** *y*)

Converts the given zero based column and line number to a position.

Parameters

x

> The column number.

y

> The line number.

Return value

The position value.

wxTextCtrl::operator <<

wxTextCtrl& operator <<(**const wxString&** *s*)

wxTextCtrl& operator <<(**int** *i*)

wxTextCtrl& operator <<(**long** *i*)

wxTextCtrl& operator <<(**float** *f*)

wxTextCtrl& operator <<(**double** *d*)

wxTextCtrl& operator <<(**char** *c*)

Operator definitions for appending to a text control, for example:

```
wxTextCtrl *wnd = new wxTextCtrl(my_frame);

(*wnd) << "Welcome to text control number " << 1 << ".\n";
```

wxTextDataObject

wxTextDataObject is a specialization of wxDataObject for text data. It can be used without change to paste data into the *wxClipboard* (p. 150) or a *wxDropSource* (p. 468). A user may wish to derive a new class from this class for providing text on-demand in order to minimize memory consumption when offering data in several formats, such as plain text and RTF because by default the text is stored in a string in this class, but it might as well be generated when requested. For this, *GetTextLength* (p. 1359) and *GetText* (p. 1360) will have to be overridden.

Note that if you already have the text inside a string, you will not achieve any efficiency gain by overriding these functions because copying wxStrings is already a very efficient operation (data is not actually copied because wxStrings are reference counted).

wxPython note: If you wish to create a derived wxTextDataObject class in wxPython you should derive the class from wxPyTextDataObject in order to get Python-aware capabilities for the various virtual methods.

Virtual functions to override

This class may be used as is, but all of the data transfer functions may be overridden to increase efficiency.

Derived from

wxDataObjectSimple (p. 243)
wxDataObject (p. 239)

Include files

<wx/dataobj.h>

See also

Clipboard and drag and drop overview (p. 1817), *wxDataObject* (p. 239), *wxDataObjectSimple* (p. 243), *wxFileDataObject* (p. 510), *wxBitmapDataObject* (p. 97)

wxTextDataObject::wxTextDataObject

wxTextDataObject(**const wxString&** *text* = *wxEmptyString*)

Constructor, may be used to initialise the text (otherwise *SetText* (p. 1360) should be used later).

wxTextDataObject::GetTextLength

virtual size_t GetTextLength() **const**

Returns the data size. By default, returns the size of the text data set in the constructor or using *SetText* (p. 1360). This can be overridden to provide text size data on-demand. It is recommended to return the text length plus 1 for a trailing zero, but this is not strictly required.

wxTextDataObject::GetText

virtual wxString GetText() **const**

Returns the text associated with the data object. You may wish to override this method when offering data on-demand, but this is not required by wxWidgets' internals. Use this method to get data in text form from the *wxClipboard* (p. 150).

wxTextDataObject::SetText

virtual void SetText(**const wxString&** *strText*)

Sets the text associated with the data object. This method is called when the data object receives the data and, by default, copies the text into the member variable. If you want to process the text on the fly you may wish to override this function.

wxTextDropTarget

A predefined drop target for dealing with text data.

Derived from

wxDropTarget (p. 471)

Include files

<wx/dnd.h>

See also

Drag and drop overview (p. 1817), *wxDropSource* (p. 468), *wxDropTarget* (p. 471), *wxFileDropTarget* (p. 516)

wxTextDropTarget::wxTextDropTarget

 wxTextDropTarget()

Constructor.

wxTextDropTarget::OnDrop

virtual bool OnDrop(**long** *x*, **long** *y*, **const void** **data*, **size_t** *size*)

See *wxDropTarget::OnDrop* (p. 472). This function is implemented appropriately for text, and calls *wxTextDropTarget::OnDropText* (p. 1361).

wxTextDropTarget::OnDropText

virtual bool OnDropText(**wxCoord** *x*, **wxCoord** *y*, **const wxString&** *data*)

Override this function to receive dropped text.

Parameters

x

> The x coordinate of the mouse.

y

> The y coordinate of the mouse.

data

> The data being dropped: a wxString.

Return value

Return true to accept the data, false to veto the operation.

wxTextEntryDialog

This class represents a dialog that requests a one-line text string from the user. It is implemented as a generic wxWidgets dialog.

Derived from

wxDialog (p. 406)
wxWindow (p. 1490)
wxEvtHandler (p. 486)
wxObject (p. 1012)

Include files

<wx/textdlg.h>

See also

wxTextEntryDialog overview (p. 1801)

wxTextEntryDialog::wxTextEntryDialog

 wxTextEntryDialog(**wxWindow*** *parent*, **const wxString&** *message*, **const wxString&** *caption* = *"Please enter text"*, **const wxString&** *defaultValue* = *""*, **long** *style* = *wxOK | wxCANCEL | wxCENTRE*, **const wxPoint&** *pos* = *wxDefaultPosition*)

Constructor. Use *wxTextEntryDialog::ShowModal* (p. 1363) to show the dialog.

Parameters

parent

Parent window.

message

Message to show on the dialog.

defaultValue

The default value, which may be the empty string.

style

A dialog style, specifying the buttons (wxOK, wxCANCEL) and an optional wxCENTRE style. Additionally, wxTextCtrl styles (such as**wxTE_PASSWORD**) may be specified here.

pos

Dialog position.

wxTextEntryDialog::~wxTextEntryDialog

 ~wxTextEntryDialog()

Destructor.

wxTextEntryDialog::GetValue

wxString GetValue() **const**

Returns the text that the user has entered if the user has pressed OK, or the original value if the user has pressed Cancel.

wxTextEntryDialog::SetValue

void SetValue(**const wxString&** *value*)

Sets the default text value.

wxTextEntryDialog::ShowModal

int ShowModal()

Shows the dialog, returning wxID_OK if the user pressed OK, and wxID_CANCEL otherwise.

wxTextFile

The wxTextFile is a simple class which allows to work with text files on line by line basis. It also understands the differences in line termination characters under different platforms and will not do anything bad to files with "non native" line termination sequences - in fact, it can be also used to modify the text files and change the line termination characters from one type (say DOS) to another (say Unix).

One word of warning: the class is not at all optimized for big files and thus it will load the file entirely into memory when opened. Of course, you should not work in this way with large files (as an estimation, anything over 1 Megabyte is surely too big for this class). On the other hand, it is not a serious limitation for small files like configuration files or program sources which are well handled by wxTextFile.

The typical things you may do with wxTextFile in order are:

- Create and open it: this is done with either *Create* (p. 1365) or *Open* (p. 1368) function which opens the file (name may be specified either as the argument to these functions or in the constructor), reads its contents in memory (in the case of `Open()`) and closes it.

- Work with the lines in the file: this may be done either with "direct access" functions like *GetLineCount* (p. 1365) and *GetLine* (p. 1365) (*operator[]* does exactly the same but looks more like array addressing) or with "sequential access" functions which include *GetFirstLine* (p. 1366)/*GetNextLine* (p. 1367) and also *GetLastLine* (p. 1367)/*GetPrevLine* (p. 1367). For the sequential access functions the current line number is maintained: it is returned by *GetCurrentLine* (p. 1366) and may be changed with *GoToLine* (p. 1366).

- Add/remove lines to the file: *AddLine* (p. 1364) and *InsertLine* (p. 1368) add new lines while *RemoveLine* (p. 1368) deletes the existing ones.*Clear* (p. 1368) resets the file to empty.

- Save your changes: notice that the changes you make to the file will **not** be saved automatically; calling *Close* (p. 1364) or doing nothing discards them! To save the changes you must explicitly call *Write* (p. 1368) - here, you may also change the line termination type if you wish.

Derived from

No base class

Include files

<wx/textfile.h>

Data structures

The following constants identify the line termination type:

```
enum wxTextFileType
{
    wxTextFileType_None,  // incomplete (the last line of the file only)
    wxTextFileType_Unix,  // line is terminated with 'LF' = 0xA = 10 =
'\n'
    wxTextFileType_Dos,   //                          'CR' 'LF'
    wxTextFileType_Mac    //                          'CR' = 0xD = 13 =
'\r'
};
```

See also

wxFile (p. 502)

wxTextFile::wxTextFile

wxTextFile() **const**

Default constructor, use *Create* (p. 1365) or *Open* (p. 1368) with a file name parameter to initialize the object.

wxTextFile::wxTextFile

wxTextFile(**const wxString&** *strFile*) **const**

Constructor does not load the file into memory, use Open() to do it.

wxTextFile::~wxTextFile

~wxTextFile() **const**

Destructor does nothing.

wxTextFile::AddLine

void AddLine(**const wxString&** *str*, **wxTextFileType** *type = typeDefault*) **const**

Adds a line to the end of file.

wxTextFile::Close

bool Close() **const**

Closes the file and frees memory, **losing all changes**. Use *Write()* (p. 1368) if you want to save them.

wxTextFile::Create

bool Create() **const**

bool Create(**const wxString&** *strFile*) **const**

Creates the file with the given name or the name which was given in the*constructor* (p. 1364). The array of file lines is initially empty.

It will fail if the file already exists, *Open* (p. 1368) should be used in this case.

wxTextFile::Exists

bool Exists() **const**

Return true if file exists - the name of the file should have been specified in the constructor before calling Exists().

wxTextFile::IsOpened

bool IsOpened() **const**

Returns true if the file is currently opened.

wxTextFile::GetLineCount

size_t GetLineCount() **const**

Get the number of lines in the file.

wxTextFile::GetLine

wxString& GetLine(**size_t** *n*) **const**

Retrieves the line number *n* from the file. The returned line may be modified but you shouldn't add line terminator at the end - this will be done by wxTextFile.

wxTextFile::operator[]

wxString& operator[](**size_t** *n*) **const**

The same as *GetLine* (p. 1365).

wxTextFile::GetCurrentLine

size_t GetCurrentLine() const

Returns the current line: it has meaning only when you're using GetFirstLine()/GetNextLine() functions, it doesn't get updated when you're using "direct access" functions like GetLine(). GetFirstLine() and GetLastLine() also change the value of the current line, as well as GoToLine().

wxTextFile::GoToLine

void GoToLine(size_t *n*) const

Changes the value returned by *GetCurrentLine* (p. 1366) and used by *GetFirstLine()* (p. 1366)/*GetNextLine()* (p. 1367).

wxTextFile::Eof

bool Eof() const

Returns true if the current line is the last one.

wxTextFile::GetEOL

static const char* GetEOL(wxTextFileType *type* = *typeDefault*) const

Get the line termination string corresponding to given constant. *typeDefault* is the value defined during the compilation and corresponds to the native format of the platform, i.e. it will be wxTextFileType_Dos under Windows, wxTextFileType_Unix under Unix (including Mac OS X when compiling with the Apple Developer Tools) and wxTextFileType_Mac under Mac OS (including Mac OS X when compiling with CodeWarrior).

wxTextFile::GetFirstLine

wxString& GetFirstLine() const

This method together with *GetNextLine()* (p. 1367) allows more "iterator-like" traversal of the list of lines, i.e. you may write something like:

```
wxTextFile file;
...
for ( str = file.GetFirstLine(); !file.Eof(); str = file.GetNextLine() )
{
    // do something with the current line in str
}
```

```
    // do something with the last line in str
```

wxTextFile::GetNextLine

wxString& GetNextLine()

Gets the next line (see *GetFirstLine* (p. 1366) for the example).

wxTextFile::GetPrevLine

wxString& GetPrevLine()

Gets the previous line in the file.

wxTextFile::GetLastLine

wxString& GetLastLine()

Gets the last line of the file. Together with *GetPrevLine* (p. 1367) it allows to enumerate the lines in the file from the end to the beginning like this:

```
wxTextFile file;
...
for ( str = file.GetLastLine();
      file.GetCurrentLine() > 0;
      str = file.GetPrevLine() )
{
    // do something with the current line in str
}
// do something with the first line in str
```

wxTextFile::GetLineType

wxTextFileType GetLineType(size_t *n*) const

Get the type of the line (see also *GetEOL* (p. 1366))

wxTextFile::GuessType

wxTextFileType GuessType() const

Guess the type of file (which is supposed to be opened). If sufficiently many lines of the file are in DOS/Unix/Mac format, the corresponding value will be returned. If the detection mechanism fails wxTextFileType_None is returned.

wxTextFile::GetName

const char* GetName() **const**

Get the name of the file.

wxTextFile::InsertLine

void InsertLine(**const wxString&** *str*, **size_t** *n*, **wxTextFileType** *type* = *typeDefault*) **const**

Insert a line before the line number *n*.

wxTextFile::Open

bool Open(**wxMBConv&** *conv* = *wxConvUTF8*) **const**

bool Open(**const wxString&** *strFile*, **wxMBConv&** *conv* = *wxConvUTF8*) **const**

Open() opens the file with the given name or the name which was given in the*constructor* (p. 1364) and also loads file in memory on success. It will fail if the file does not exist, *Create* (p. 1365) should be used in this case.

The *conv* argument is only meaningful in Unicode build of wxWidgets when it is used to convert the file to wide character representation.

wxTextFile::RemoveLine

void RemoveLine(**size_t** *n*) **const**

Delete line number *n* from the file.

wxTextFile::Clear

void Clear() **const**

Delete all lines from the file, set current line number to 0.

wxTextFile::Write

bool Write(**wxTextFileType** *typeNew* = *wxTextFileType_None*, **wxMBConv&** *conv* = *wxConvUTF8*) **const**

Change the file on disk. The *typeNew* parameter allows you to change the file format (default argument means "don't change type") and may be used to convert, for example, DOS files to Unix.

The *conv* argument is only meaningful in Unicode build of wxWidgets when it is used to convert all lines to multibyte representation before writing them them to physical file.

Returns true if operation succeeded, false if it failed.

wxTextInputStream

This class provides functions that read text datas using an input stream. So, you can read *text* floats, integers.

The wxTextInputStream correctly reads text files (or streams) in DOS, Macintosh and Unix formats and reports a single newline char as a line ending.

Operator >> is overloaded and you can use this class like a standard C++ iostream. Note, however, that the arguments are the fixed size types wxUint32, wxInt32 etc and on a typical 32-bit computer, none of these match to the "long" type (wxInt32 is defined as int on 32-bit architectures) so that you cannot use long. To avoid problems (here and elsewhere), make use of wxInt32, wxUint32 and similar types.

If you're scanning through a file using wxTextInputStream, you should check for EOF **before** reading the next item (word / number), because otherwise the last item may get lost. You should however be prepared to receive an empty item (empty string / zero number) at the end of file, especially on Windows systems. This is unavoidable because most (but not all) files end with whitespace (i.e. usually a newline).

For example:

```
wxFileInputStream input( "mytext.txt" );
wxTextInputStream text( input );
wxUint8 i1;
float f2;
wxString line;

text >> i1;        // read a 8 bit integer.
text >> i1 >> f2;  // read a 8 bit integer followed by float.
text >> line;      // read a text line
```

Include files

<wx/txtstrm.h>

wxTextInputStream::wxTextInputStream

 wxTextInputStream(**wxInputStream&** *stream*, **const wxString&** *sep=" \t"*, **wxMBConv&** *conv = wxConvUTF8*)

Constructs a text stream associated to the given input stream.

Parameters

stream

>   The underlying input stream.

sep

> The initial string separator characters.

conv

> *In Unicode build only:* The encoding converter used to convert the bytes in the underlying input stream to characters.

wxTextInputStream::~wxTextInputStream

~wxTextInputStream()

Destroys the wxTextInputStream object.

wxTextInputStream::Read8

wxUint8 Read8(**int** *base = 10*)

Reads a single unsigned byte from the stream, given in base *base*.

The value of *base* must be comprised between 2 and 36, inclusive, or be a special value 0 which means that the usual rules of `C` numbers are applied: if the number starts with `0x` it is considered to be in base16, if it starts with `0` - in base 8 and in base 10 otherwise. Note that you may not want to specify the base 0 if you are parsing the numbers which may have leading zeroes as they can yield unexpected (to the user not familiar with C) results.

wxTextInputStream::Read8S

wxInt8 Read8S(**int** *base = 10*)

Reads a single signed byte from the stream.

See *wxTextInputStream::Read8* (p. 1370) for the description of the *base* parameter.

wxTextInputStream::Read16

wxUint16 Read16(**int** *base = 10*)

Reads a unsigned 16 bit integer from the stream.

See *wxTextInputStream::Read8* (p. 1370) for the description of the *base* parameter.

wxTextInputStream::Read16S

wxInt16 Read16S(**int** *base = 10*)

Reads a signed 16 bit integer from the stream.

See *wxTextInputStream::Read8* (p. 1370) for the description of the *base* parameter.

wxTextInputStream::Read32

wxUint32 Read32(**int** *base = 10*)

Reads a 32 bit unsigned integer from the stream.

See *wxTextInputStream::Read8* (p. 1370) for the description of the *base* parameter.

wxTextInputStream::Read32S

wxInt32 Read32S(**int** *base = 10*)

Reads a 32 bit signed integer from the stream.

See *wxTextInputStream::Read8* (p. 1370) for the description of the *base* parameter.

wxTextInputStream::GetChar

wxChar GetChar()

Reads a character, returns 0 if there are no more characters in the stream.

wxTextInputStream::ReadDouble

double ReadDouble()

Reads a double (IEEE encoded) from the stream.

wxTextInputStream::ReadLine

wxString ReadLine()

Reads a line from the input stream and returns it (without the end of line character).

wxTextInputStream::ReadString

wxString ReadString()

NB: This method is deprecated, use *ReadLine* (p. 1371) or *ReadWord* (p. 1372) instead.

Same as *ReadLine* (p. 1371).

wxTextInputStream::ReadWord

wxString ReadWord()

Reads a word (a sequence of characters until the next separator) from the input stream.

See also

SetStringSeparators (p. 1372)

wxTextInputStream::SetStringSeparators

void SetStringSeparators(**const wxString&** *sep*)

Sets the characters which are used to define the word boundaries in *ReadWord* (p. 1372).

The default separators are the space and `TAB` characters.

wxTextOutputStream

This class provides functions that write text datas using an output stream. So, you can write *text* floats, integers.

You can also simulate the C++ cout class:

```
wxFFileOutputStream output( stderr );
wxTextOutputStream cout( output );

cout << "This is a text line" << endl;
cout << 1234;
cout << 1.23456;
```

The wxTextOutputStream writes text files (or streams) on DOS, Macintosh and Unix in their native formats (concerning the line ending).

Include files

<wx/txtstrm.h>

wxTextOutputStream::wxTextOutputStream

 wxTextOutputStream(**wxOutputStream&** *stream*, **wxEOL** *mode = wxEOL_NATIVE*, **wxMBConv&** *conv = wxConvUTF8*)

Constructs a text stream object associated to the given output stream.

Parameters

stream

> The output stream.

mode

> The end-of-line mode. One of **wxEOL_NATIVE**, **wxEOL_DOS**, **wxEOL_MAC** and **wxEOL_UNIX**.

conv

> *In Unicode build only:* The object used to convert Unicode text into ASCII characters written to the output stream.

wxTextOutputStream::~wxTextOutputStream

~wxTextOutputStream()

Destroys the wxTextOutputStream object.

wxTextOutputStream::GetMode

wxEOL GetMode()

Returns the end-of-line mode. One of **wxEOL_DOS**, **wxEOL_MAC** and **wxEOL_UNIX**.

wxTextOutputStream::PutChar

void PutChar(**wxChar** *c*)

Writes a character to the stream.

wxTextOutputStream::SetMode

void SetMode(**wxEOL** *mode = wxEOL_NATIVE*)

Set the end-of-line mode. One of **wxEOL_NATIVE**, **wxEOL_DOS**, **wxEOL_MAC** and **wxEOL_UNIX**.

wxTextOutputStream::Write8

void Write8(**wxUint8** *i8*)

Writes the single byte *i8* to the stream.

wxTextOutputStream::Write16

void Write16(**wxUint16** *i16*)

Writes the 16 bit integer *i16* to the stream.

wxTextOutputStream::Write32

void Write32(**wxUint32** *i32*)

Writes the 32 bit integer *i32* to the stream.

wxTextOutputStream::WriteDouble

virtual void WriteDouble(**double** *f*)

Writes the double *f* to the stream using the IEEE format.

wxTextOutputStream::WriteString

virtual void WriteString(**const wxString&** *string*)

Writes *string* as a line. Depending on the end-of-line mode the end of line ('\n') characters in the string are converted to the correct line ending terminator.

wxTextValidator

wxTextValidator validates text controls, providing a variety of filtering behaviours.

For more information, please see *Validator overview* (p. 1765).

Derived from

wxValidator (p. 1461)
wxEvtHandler (p. 486)
wxObject (p. 1012)

Include files

<wx/valtext.h>

See also

Validator overview (p. 1765), *wxValidator* (p. 1461),*wxGenericValidator* (p. 612)

wxTextValidator::wxTextValidator

wxTextValidator(**const wxTextValidator&** *validator*)

Copy constructor.

wxTextValidator(**long** *style* = *wxFILTER_NONE*, **wxString*** *valPtr* = *NULL*)

Constructor, taking a style and optional pointer to a wxString variable.

Parameters

style

> A bitlist of flags, which can be:

> **wxFILTER_NONE**   No filtering takes place.

> **wxFILTER_ASCII**   Non-ASCII characters are filtered out.

> **wxFILTER_ALPHA**   Non-alpha characters are filtered out.

> **wxFILTER_ALPHANUMERIC** Non-alphanumeric characters are filtered out.

> **wxFILTER_NUMERIC**   Non-numeric characters are filtered out.

> **wxFILTER_INCLUDE_LIST** Use an include list. The validator checks if the user input is on the list, complaining if not. See *wxTextValidator::SetIncludes* (p. 1376).

> **wxFILTER_EXCLUDE_LIST** Use an exclude list. The validator checks if the user input is on the list, complaining if it is. See *wxTextValidator::SetExcludes* (p. 1376).

> **wxFILTER_INCLUDE_CHAR_LIST**   Use an include list. The validator checks if each input character is in the list (one character per list element), complaining if not. See *wxTextValidator::SetIncludes* (p. 1376).

> **wxFILTER_EXCLUDE_CHAR_LIST**   Use an include list. The validator checks if each input character is in the list (one character per list element), complaining if it is. See *wxTextValidator::SetExcludes* (p. 1376).

valPtr

> A pointer to a wxString variable that contains the value. This variable should have a lifetime equal to or longer than the validator lifetime (which is usually determined by the lifetime of the window).

wxTextValidator::Clone

virtual wxValidator* Clone() **const**

Clones the text validator using the copy constructor.

wxTextValidator::GetExcludes

wxArrayString& GetExcludes() **const**

Returns a reference to the exclude list (the list of invalid values).

wxTextValidator::GetIncludes

wxArrayString& GetIncludes() **const**

Returns a reference to the include list (the list of valid values).

wxTextValidator::GetStyle

long GetStyle() **const**

Returns the validator style.

wxTextValidator::OnChar

void OnChar(**wxKeyEvent&** *event*)

Receives character input from the window and filters it according to the current validator style.

wxTextValidator::SetExcludes

void SetExcludes(**const wxArrayString&** *stringList*)

Sets the exclude list (invalid values for the user input).

wxTextValidator::SetIncludes

void SetIncludes(**const wxArrayString&** *stringList*)

Sets the include list (valid values for the user input).

wxTextValidator::SetStyle

void SetStyle(**long** *style*)

Sets the validator style.

wxTextValidator::TransferFromWindow

virtual bool TransferFromWindow()

Transfers the value in the text control to the string.

wxTextValidator::TransferToWindow

virtual bool TransferToWindow()

Transfers the string value to the text control.

wxTextValidator::Validate

virtual bool Validate(**wxWindow*** *parent*)

Validates the window contents against the include or exclude lists, depending on the validator style.

wxThread

A thread is basically a path of execution through a program. Threads are sometimes called *light-weight processes*, but the fundamental difference between threads and processes is that memory spaces of different processes are separated while all threads share the same address space. While it makes it much easier to share common data between several threads, it also makes it much easier to shoot oneself in the foot, so careful use of synchronization objects such as *mutexes* (p. 994) and/or *critical sections* (p. 223) is recommended.

There are two types of threads in wxWidgets: *detached* and *joinable*ones, just as in the POSIX thread API (but unlike Win32 threads where all threads are joinable). The difference between the two is that only joinable threads can return a return code -- this is returned by the Wait() function. Detached threads (the default type) cannot be waited for.

You shouldn't hurry to create all the threads joinable, however, because this has a disadvantage as well: you **must** Wait() for a joinable thread or the system resources used by it will never be freed, and you also must delete the corresponding wxThread object yourself. In contrast, detached threads are of the "fire-and-forget" kind: you only have to start a detached thread and it will terminate and destroy itself.

This means, of course, that all detached threads **must** be created on the heap because the thread will call `delete this;` upon termination. Joinable threads may be created on the stack although more usually they will be created on the heap as well. Don't create global thread objects because they allocate memory in their constructor, which will cause problems for the memory checking system. Finally, another consequence of the handling of the above is that you should never delete a detached thread yourself, as this will be done by the thread itself when it terminates.

Derived from

None.

Include files

<wx/thread.h>

See also

wxMutex (p. 994), *wxCondition* (p. 189), *wxCriticalSection* (p. 223)

wxThread::wxThread

wxThread(**wxThreadKind** *kind = wxTHREAD_DETACHED*)

This constructor creates a new detached (default) or joinable C++ thread object. It does not create or start execution of the real thread -- for this you should use the *Create* (p. 1378) and *Run* (p. 1382) methods.

The possible values for *kind* parameters are:

wxTHREAD_DETACHED   Create a detached thread.

wxTHREAD_JOINABLE   Create a joinable thread

wxThread::~wxThread

~wxThread()

The destructor frees the resources associated with the thread. Notice that you should never delete a detached thread -- you may only call*Delete* (p. 1379) on it or wait until it terminates (and auto destructs) itself. Because the detached threads delete themselves, they can only be allocated on the heap.

Joinable threads should be deleted explicitly. The *Delete* (p. 1379) and *Kill* (p. 1381) functions will not delete the C++ thread object. It is also safe to allocate them on stack.

wxThread::Create

wxThreadError Create(**unsigned int** *stackSize = 0*)

Creates a new thread. The thread object is created in the suspended state, and you should call *Run* (p. 1382) to start running it. You may optionally specify the stack size to be allocated to it (Ignored on platforms that don't support setting it explicitly, eg. Unix).

Return value

One of:

wxTHREAD_NO_ERROR   There was no error.

wxTHREAD_NO_RESOURCE   There were insufficient resources to create a

new thread.

wxTHREAD_RUNNING   The thread is already running.

wxThread::Delete

void Delete()

Calling *Delete* (p. 1379) is a graceful way to terminate the thread. It asks the thread to terminate and, if the thread code is well written, the thread will terminate after the next call to *TestDestroy* (p. 1383) which should happen quite soon.

However, if the thread doesn't call *TestDestroy* (p. 1383)often enough (or at all), the function will not return immediately, but wait until the thread terminates. As it may take a long time, and the message processing is not stopped during this function execution, message handlers may be called from inside it!

Delete() may be called for a thread in any state: running, paused or even not yet created. Moreover, it must be called if *Create* (p. 1378) or*Run* (p. 1382) fail in order to free the memory occupied by the thread object. However, you should not call Delete() on a detached thread which already terminated -- doing so will probably result in a crash because the thread object doesn't exist any more.

For detached threads Delete() will also delete the C++ thread object, but it will not do this for joinable ones.

This function can only be called from another thread context.

wxThread::Entry

virtual ExitCode Entry()

This is the entry point of the thread. This function is pure virtual and must be implemented by any derived class. The thread execution will start here.

The returned value is the thread exit code which is only useful for joinable threads and is the value returned by *Wait* (p. 1384).

This function is called by wxWidgets itself and should never be called directly.

wxThread::Exit

void Exit(**ExitCode** *exitcode = 0*)

This is a protected function of the wxThread class and thus can only be called from a derived class. It also can only be called in the context of this thread, i.e. a thread can only exit from itself, not from another thread.

This function will terminate the OS thread (i.e. stop the associated path of execution) and also delete the associated C++ object for detached threads.*wxThread::OnExit* (p.

1382) will be called just before exiting.

wxThread::GetCPUCount

static int GetCPUCount()

Returns the number of system CPUs or -1 if the value is unknown.

See also

SetConcurrency (p. 1383)

wxThread::GetCurrentId

static unsigned long GetCurrentId()

Returns the platform specific thread ID of the current thread as a long. This can be used to uniquely identify threads, even if they are not wxThreads.

wxThread::GetId

unsigned long GetId() **const**

Gets the thread identifier: this is a platform dependent number that uniquely identifies the thread throughout the system during its existence (i.e. the thread identifiers may be reused).

wxThread::GetPriority

int GetPriority() **const**

Gets the priority of the thread, between zero and 100.

The following priorities are defined:

WXTHREAD_MIN_PRIORITY　　　　0

WXTHREAD_DEFAULT_PRIORITY　　50

WXTHREAD_MAX_PRIORITY　　　　100

wxThread::IsAlive

bool IsAlive() **const**

Returns `true` if the thread is alive (i.e. started and not terminating).

Note that this function can only safely be used with joinable threads, not detached ones

as the latter delete themselves and so when the real thread is no longer alive, it is not possible to call this function because the wxThread object no longer exists.

wxThread::IsDetached

bool IsDetached() **const**

Returns `true` if the thread is of the detached kind, `false` if it is a joinable one.

wxThread::IsMain

static bool IsMain()

Returns `true` if the calling thread is the main application thread.

wxThread::IsPaused

bool IsPaused() **const**

Returns `true` if the thread is paused.

wxThread::IsRunning

bool IsRunning() **const**

Returns `true` if the thread is running.

This method may only be safely used for joinable threads, see the remark in *IsAlive* (p. 1380).

wxThread::Kill

wxThreadError Kill()

Immediately terminates the target thread. **This function is dangerous and should be used with extreme care (and not used at all whenever possible)!** The resources allocated to the thread will not be freed and the state of the C runtime library may become inconsistent. Use *Delete()* (p. 1379) instead.

For detached threads Kill() will also delete the associated C++ object. However this will not happen for joinable threads and this means that you will still have to delete the wxThread object yourself to avoid memory leaks. In neither case *OnExit* (p. 1382) of the dying thread will be called, so no thread-specific cleanup will be performed.

This function can only be called from another thread context, i.e. a thread cannot kill itself.

It is also an error to call this function for a thread which is not running or paused (in the latter case, the thread will be resumed first) -- if you do it, a `wxTHREAD_NOT_RUNNING` error will be returned.

wxThread::OnExit

void OnExit()

Called when the thread exits. This function is called in the context of the thread associated with the wxThread object, not in the context of the main thread. This function will not be called if the thread was *killed* (p. 1381).

This function should never be called directly.

wxThread::Pause

wxThreadError Pause()

Suspends the thread. Under some implementations (Win32), the thread is suspended immediately, under others it will only be suspended when it calls *TestDestroy* (p. 1383) for the next time (hence, if the thread doesn't call it at all, it won't be suspended).

This function can only be called from another thread context.

wxThread::Run

wxThreadError Run()

Starts the thread execution. Should be called after *Create* (p. 1378).

This function can only be called from another thread context.

wxThread::SetPriority

void SetPriority(**int** *priority*)

Sets the priority of the thread, between 0 and 100. It can only be set after calling *Create()* (p. 1378) but before calling *Run()* (p. 1382).

The following priorities are already defined:

WXTHREAD_MIN_PRIORITY   0

WXTHREAD_DEFAULT_PRIORITY   50

WXTHREAD_MAX_PRIORITY   100

wxThread::Sleep

static void Sleep(**unsigned long** *milliseconds*)

Pauses the thread execution for the given amount of time.

This function should be used instead of *wxSleep* (p. 1656) by all worker threads (i.e. all except the main one).

wxThread::Resume

wxThreadError Resume()

Resumes a thread suspended by the call to *Pause* (p. 1382).

This function can only be called from another thread context.

wxThread::SetConcurrency

static bool SetConcurrency(**size_t** *level*)

Sets the thread concurrency level for this process. This is, roughly, the number of threads that the system tries to schedule to run in parallel. The value of 0 for *level* may be used to set the default one.

Returns `true` on success or false otherwise (for example, if this function is not implemented for this platform -- currently everything except Solaris).

wxThread::TestDestroy

virtual bool TestDestroy()

This function should be called periodically by the thread to ensure that calls to *Pause* (p. 1382) and *Delete* (p. 1379) will work. If it returns `true`, the thread should exit as soon as possible.

Notice that under some platforms (POSIX), implementation of *Pause* (p. 1382) also relies on this function being called, so not calling it would prevent both stopping and suspending thread from working.

wxThread::This

static wxThread * This()

Return the thread object for the calling thread. NULL is returned if the calling thread is the main (GUI) thread, but *IsMain* (p. 1381) should be used to test whether the thread is really the main one because NULL may also be returned for the thread not created with wxThread class. Generally speaking, the return value for such a thread is undefined.

wxThread::Yield

void Yield()

Give the rest of the thread time slice to the system allowing the other threads to run. See also *Sleep()* (p. 1383).

wxThread::Wait

ExitCode Wait() **const**

Waits until the thread terminates and returns its exit code or `(ExitCode)-1` on error.

You can only Wait() for joinable (not detached) threads.

This function can only be called from another thread context.

wxThreadHelper

The wxThreadHelper class is a mix-in class that manages a single background thread. By deriving from wxThreadHelper, a class can implement the thread code in its own *wxThreadHelper::Entry* (p. 1385) method and easily share data and synchronization objects between the main thread and the worker thread. Doing this prevents the awkward passing of pointers that is needed when the original object in the main thread needs to synchronize with its worker thread in its own wxThread derived object.

For example, *wxFrame* (p. 579) may need to make some calculations in a background thread and then display the results of those calculations in the main window.

Ordinarily, a *wxThread* (p. 1377) derived object would be created with the calculation code implemented in*wxThread::Entry* (p. 1379). To access the inputs to the calculation, the frame object would often to pass a pointer to itself to the thread object. Similarly, the frame object would hold a pointer to the thread object. Shared data and synchronization objects could be stored in either object though the object without the data would have to access the data through a pointer.

However, with wxThreadHelper, the frame object and the thread object are treated as the same object. Shared data and synchronization variables are stored in the single object, eliminating a layer of indirection and the associated pointers.

Derived from

None.

Include files

<wx/thread.h>

See also

wxThread (p. 1377)

wxThreadHelper::wxThreadHelper

wxThreadHelper()

This constructor simply initializes a member variable.

wxThreadHelper::m_thread

wxThread * m_thread

the actual *wxThread* (p. 1377) object.

wxThreadHelper::~wxThreadHelper

~wxThreadHelper()

The destructor frees the resources associated with the thread.

wxThreadHelper::Create

wxThreadError Create(**unsigned int** *stackSize = 0*)

Creates a new thread. The thread object is created in the suspended state, and you should call *GetThread()->Run()* (p. 1382) to start running it. You may optionally specify the stack size to be allocated to it (Ignored on platforms that don't support setting it explicitly, eg. Unix).

Return value

One of:

wxTHREAD_NO_ERROR	There was no error.
wxTHREAD_NO_RESOURCE	There were insufficient resources to create a new thread.
wxTHREAD_RUNNING	The thread is already running.

wxThreadHelper::Entry

virtual ExitCode Entry()

This is the entry point of the thread. This function is pure virtual and must be implemented by any derived class. The thread execution will start here.

The returned value is the thread exit code which is only useful for joinable threads and is

the value returned by *GetThread()->Wait()* (p. 1384).

This function is called by wxWidgets itself and should never be called directly.

wxThreadHelper::GetThread

wxThread * GetThread()

This is a public function that returns the *wxThread* (p. 1377) object associated with the thread.

wxTimer

The wxTimer class allows you to execute code at specified intervals. Its precision is platform-dependent, but in general will not be better than 1ms nor worse than 1s.

There are two different ways to use this class:

1. You may derive a new class from wxTimer and override the *Notify* (p. 1387) member to perform the required action.

2. Or you may redirect the notifications to any *wxEvtHandler* (p. 486) derived object by using the non default constructor or *SetOwner* (p. 1387). Then use the EVT_TIMER macro to connect it to the event handler which will receive *wxTimerEvent* (p. 1388) notifications.

3. Or you may use a derived class and the EVT_TIMER macro to connect it to an event handler defined in the derived class. If the default constructor is used, the timer object will be its own owner object, since it is derived from wxEvtHandler.

In any case, you must start the timer with *Start* (p. 1388) after constructing it before it actually starts sending notifications. It can be stopped later with *Stop* (p. 1388).

NB: note that timer can only be used from the main thread currently.

Derived from

wxEvtHandler (p. 486)

Include files

<wx/timer.h>

See also

::wxStartTimer (p. 1656), *::wxGetElapsedTime* (p. 1654), *wxStopWatch* (p. 1274)

wxTimer::wxTimer

wxTimer()

Default constructor. If you use it to construct the object and don't call *SetOwner* (p. 1387) later, you must override *Notify* (p. 1387) method to process the notifications.

wxTimer(**wxEvtHandler ****owner*, **int** *id = -1*)

Creates a timer and associates it with *owner*. Please see *SetOwner* (p. 1387) for the description of parameters.

wxTimer::~wxTimer

~wxTimer()

Destructor. Stops the timer if it is running.

wxTimer::GetInterval

int GetInterval() **const**

Returns the current interval for the timer (in milliseconds).

wxTimer::IsOneShot

bool IsOneShot() **const**

Returns `true` if the timer is one shot, i.e. if it will stop after firing the first notification automatically.

wxTimer::IsRunning

bool IsRunning() **const**

Returns `true` if the timer is running, `false` if it is stopped.

wxTimer::Notify

void Notify()

This member should be overridden by the user if the default constructor was used and *SetOwner* (p. 1387) wasn't called.

Perform whatever action which is to be taken periodically here.

wxTimer::SetOwner

void SetOwner(**wxEvtHandler ****owner*, **int** *id = -1*)

Associates the timer with the given *owner* object. When the timer is running, the owner will receive *timer events* (p. 1388) with id equal to *id* specified here.

wxTimer::Start

bool Start(**int***milliseconds = -1*, **bool** *oneShot = `false`*)

(Re)starts the timer. If *milliseconds* parameter is -1 (value by default), the previous value is used. Returns `false` if the timer could not be started,`true` otherwise (in MS Windows timers are a limited resource).

If *oneShot* is `false` (the default), the *Notify* (p. 1387) function will be called repeatedly until the timer is stopped. If `true`, it will be called only once and the timer will stop automatically. To make your code more readable you may also use the following symbolic constants:

wxTIMER_CONTINUOUS   Start a normal, continuously running, timer

wxTIMER_ONE_SHOT   Start a one shot timer

If the timer was already running, it will be stopped by this method before restarting it.

wxTimer::Stop

void Stop()

Stops the timer.

wxTimerEvent

wxTimerEvent object is passed to the event handler of timer events.

For example:

```
class MyFrame : public wxFrame
{
public:
    ...
    void OnTimer(wxTimerEvent& event);

private:
    wxTimer m_timer;
};

BEGIN_EVENT_TABLE(MyFrame, wxFrame)
    EVT_TIMER(TIMER_ID, MyFrame::OnTimer)
END_EVENT_TABLE()

MyFrame::MyFrame()
        : m_timer(this, TIMER_ID)
```

```
    {
        m_timer.Start(1000);    // 1 second interval
    }

    void MyFrame::OnTimer(wxTimerEvent& event)
    {
        // do whatever you want to do every second here
    }
```

Include files

<wx/timer.h>

See also

wxTimer (p. 1386)

wxTimerEvent::GetInterval

int GetInterval() **const**

Returns the interval of the timer which generated this event.

wxTimeSpan

wxTimeSpan class represents a time interval.

Derived from

No base class

Include files

<wx/datetime.h>

See also

Date classes overview (p. 1726), *wxDateTime* (p. 257)

Static functions

Seconds (p. 1395)
Second (p. 1395)
Minutes (p. 1394)
Minute (p. 1394)
Hours (p. 1393)

Constructors

Accessors

Operations

Tests

Formatting time spans

wxTimeSpan::Abs

wxTimeSpan Abs() **const**

Returns the absolute value of the timespan: does not modify the object.

wxTimeSpan::Add

wxTimeSpan Add(**const wxTimeSpan&** *diff*) **const**

wxTimeSpan& Add(**const wxTimeSpan&** *diff*)

wxTimeSpan& operator+=(**const wxTimeSpan&***diff*)

Returns the sum of two timespans.

wxTimeSpan::Days

static wxTimespan Days(**long** *days*)

Returns the timespan for the given number of days.

wxTimeSpan::Day

static wxTimespan Day()

Returns the timespan for one day.

wxTimeSpan::Format

wxString Format(**const wxChar** * *format* = *wxDefaultTimeSpanFormat*)

Returns the string containing the formatted representation of the time span. The following format specifiers are allowed after %:

H	number of **H**ours
M	number of **M**inutes
S	number of **S**econds
l	number of mi**l**liseconds
D	number of **D**ays
E	number of w**E**eks
%	the percent character

Note that, for example, the number of hours in the description above is not well defined: it can be either the total number of hours (for example, for a time span of 50 hours this would be 50) or just the hour part of the time span, which would be 2 in this case as 50

hours is equal to 2 days and2 hours.

wxTimeSpan resolves this ambiguity in the following way: if there had been, indeed, the `%D` format specified preceding the `%H`, then it is interpreted as 2. Otherwise, it is 50.

The same applies to all other format specifiers: if they follow a specifier of larger unit, only the rest part is taken, otherwise the full value is used.

wxTimeSpan::GetDays

int GetDays() **const**

Returns the difference in number of days.

wxTimeSpan::GetHours

int GetHours() **const**

Returns the difference in number of hours.

wxTimeSpan::GetMilliseconds

wxLongLong GetMilliseconds() **const**

Returns the difference in number of milliseconds.

wxTimeSpan::GetMinutes

int GetMinutes() **const**

Returns the difference in number of minutes.

wxTimeSpan::GetSeconds

wxLongLong GetSeconds() **const**

Returns the difference in number of seconds.

wxTimeSpan::GetValue

wxLongLong GetValue() **const**

Returns the internal representation of timespan.

wxTimeSpan::GetWeeks

int GetWeeks() const

Returns the difference in number of weeks.

wxTimeSpan::Hours

static wxTimespan Hours(long *hours*)

Returns the timespan for the given number of hours.

wxTimeSpan::Hour

static wxTimespan Hour()

Returns the timespan for one hour.

wxTimeSpan::IsEqualTo

bool IsEqualTo(const wxTimeSpan& *ts*) const

Returns `true` if two timespans are equal.

wxTimeSpan::IsLongerThan

bool IsLongerThan(const wxTimeSpan& *ts*) const

Compares two timespans: works with the absolute values, i.e. -2 hours is longer than 1 hour. Also, it will return `false` if the timespans are equal in absolute value.

wxTimeSpan::IsNegative

bool IsNegative() const

Returns `true` if the timespan is negative.

wxTimeSpan::IsNull

bool IsNull() const

Returns `true` if the timespan is empty.

wxTimeSpan::IsPositive

bool IsPositive() const

Returns `true` if the timespan is positive.

wxTimeSpan::IsShorterThan

bool IsShorterThan(**const wxTimeSpan&** *ts*) **const**

Compares two timespans: works with the absolute values, i.e. 1 hour is shorter than -2 hours. Also, it will return `false` if the timespans are equal in absolute value.

wxTimeSpan::Minutes

static wxTimespan Minutes(**long** *min*)

Returns the timespan for the given number of minutes.

wxTimeSpan::Minute

static wxTimespan Minute()

Returns the timespan for one minute.

wxTimeSpan::Multiply

wxTimeSpan Multiply(**int** *n*) **const**

wxTimeSpan& Multiply(**int** *n*)

wxTimeSpan& operator*=(**int** *n*)

Multiplies timespan by a scalar.

wxTimeSpan::Negate

wxTimeSpan Negate() **const**

Returns timespan with inverted sign.

wxTimeSpan::Neg

wxTimeSpan& Neg()

wxTimeSpan& operator-()

Negate the value of the timespan.

wxTimeSpan::Seconds

static wxTimespan Seconds(**long** *sec*)

Returns the timespan for the given number of seconds.

wxTimeSpan::Second

static wxTimespan Second()

Returns the timespan for one second.

wxTimeSpan::Subtract

wxTimeSpan Subtract(**const wxTimeSpan&***diff*) **const**

wxTimeSpan& Subtract(**const wxTimeSpan&** *diff*)

wxTimeSpan& operator-=(**const wxTimeSpan&***diff*)

Returns the difference of two timespans.

wxTimeSpan::Weeks

static wxTimespan Weeks(**long** *weeks*)

Returns the timespan for the given number of weeks.

wxTimeSpan::Week

static wxTimespan Week()

Returns the timespan for one week.

wxTimeSpan::wxTimeSpan

wxTimeSpan()

Default constructor, constructs a zero timespan.

wxTimeSpan(**long** *hours*, **long** *min*, **long** *sec*, **long** *msec*)

Constructs timespan from separate values for each component, with the date set to 0. Hours are not restricted to 0..24 range, neither are minutes, seconds or milliseconds.

wxTipProvider

This is the class used together with *wxShowTip* (p. 1622) function. It must implement *GetTip* (p. 1396) function and return the current tip from it (different tip each time it is called).

You will never use this class yourself, but you need it to show startup tips with wxShowTip. Also, if you want to get the tips text from elsewhere than a simple text file, you will want to derive a new class from wxTipProvider and use it instead of the one returned by *wxCreateFileTipProvider* (p. 1614).

Derived from

None.

Include files

<wx/tipdlg.h>

See also

Startup tips overview (p. 1814), *::wxShowTip* (p. 1622)

wxTipProvider::wxTipProvider

wxTipProvider(**size_t** *currentTip*)

Constructor.

currentTip

> The starting tip index.

wxTipProvider::GetTip

wxString GetTip()

Return the text of the current tip and pass to the next one. This function is pure virtual, it should be implemented in the derived classes.

wxTipProvider::PreprocessTip

virtual wxString PreProcessTip(**const wxString&***tip*)

Returns a modified tip. This function will be called immediately after read, and before being check whether it is a comment, an empty string or a string to translate. You can optionally override this in your custom user-derived class to optionally to modify the tip as soon as it is read. You can return any modification to the string. If you return wxEmptyString, then this tip is skipped, and the next one is read.

wxCurrentTipProvider::GetCurrentTip

size_t GetCurrentTip() **const**

Returns the index of the current tip (i.e. the one which would be returned by GetTip).

The program usually remembers the value returned by this function after calling *wxShowTip* (p. 1622). Note that it is not the same as the value which was passed to wxShowTip + 1 because the user might have pressed the "Next" button in the tip dialog.

wxTipWindow

Shows simple text in a popup tip window on creation. This is used by *wxSimpleHelpProvider* (p. 1168) to show popup help. The window automatically destroys itself when the user clicks on it or it loses the focus.

You may also use this class to emulate the tooltips when you need finer control over them than what the standard tooltips provide.

Derived from

wxPopupTransientWindow
wxPopupWindow
wxWindow (p. 1490)
wxEvtHandler (p. 486)
wxObject (p. 1012)

Include files

<wx/tipwin.h>

wxTipWindow::wxTipWindow

 wxTipWindow(**wxWindow*** *parent*, **const wxString&** *text*, **wxCoord** *maxLength =
100*, **wxTipWindow**** *windowPtr*)

Constructor. The tip is shown immediately the window is constructed.

Parameters

parent

   The parent window, must be non NULL

text

   The text to show, may contain the new line characters

windowPtr

Simply passed to *SetTipWindowPtr* (p. 1398) below, please see its documentation for the description of this parameter

rectBounds

If non NULL, passed to *SetBoundingRect* (p. 1398) below, please see its documentation for the description of this parameter

wxTipWindow::SetTipWindowPtr

void SetTipWindowPtr(**wxTipWindow**** *windowPtr*)

When the tip window closes itself (which may happen at any moment and unexpectedly to the caller) it may NULL out the pointer pointed to by *windowPtr*. This is helpful to avoid dereferencing the tip window which had been already closed and deleted.

wxTipWindow::SetBoundingRect

void SetBoundingRect(**const wxRect&** *rectBound*)

By default, the tip window disappears when the user clicks the mouse or presses a keyboard key or if it loses focus in any other way - for example because the user switched to another application window.

Additionally, if a non empty *rectBound* is provided, the tip window will also automatically close if the mouse leaves this area. This is useful to dismiss the tip mouse when the mouse leaves the object it is associated with.

Parameters

rectBound

The bounding rectangle for the mouse in the screen coordinates

wxToggleButton

wxToggleButton is a button that stays pressed when clicked by the user. In other words, it is similar to *wxCheckBox* (p. 134) in functionality but looks like a *wxButton* (p. 117).

You can see wxToggleButton in action in the sixth page of the *controls* (p. 1708) sample.

NB: This class is not available under wxUniversal ports such as wxX11.

Derived from

wxControl (p. 215)
wxWindow (p. 1490)
wxEvtHandler (p. 486)
wxObject (p. 1012)

Include files

<wx/tglbtn.h>

Window styles

There are no special styles for wxToggleButton.

See also *window styles overview* (p. 1762).

Event handling

EVT_TOGGLEBUTTON(id, func)   Handles button click event.

See also

wxCheckBox (p. 134), *wxButton* (p. 117)

wxToggleButton::wxToggleButton

 wxToggleButton()

Default constructor.

 wxToggleButton(**wxWindow*** *parent*, **wxWindowID** *id*, **const wxString&** *label*, **const wxPoint&** *pos* = *wxDefaultPosition*, **const wxSize&** *size* = *wxDefaultSize*, **long** *style* = *0*, **const wxValidator&** *val*, **const wxString&** *name* = *"checkBox"*)

Constructor, creating and showing a toggle button.

Parameters

parent

> Parent window. Must not be NULL.

id

> Toggle button identifier. A value of -1 indicates a default value.

label

> Text to be displayed next to the toggle button.

pos

> Toggle button position. If the position (-1, -1) is specified then a default position is chosen.

size

> Toggle button size. If the default size (-1, -1) is specified then a default size is

chosen.

style

>   Window style. See *wxToggleButton* (p. 1398).

validator

>   Window validator.

name

>   Window name.

See also

wxToggleButton::Create (p. 1400), *wxValidator* (p. 1461)

wxToggleButton::~wxToggleButton

 ~wxToggleButton()

Destructor, destroying the toggle button.

wxToggleButton::Create

bool Create(**wxWindow*** *parent*, **wxWindowID** *id*, **const wxString&** *label*, **const wxPoint&** *pos* = *wxDefaultPosition*, **const wxSize&** *size* = *wxDefaultSize*, **long** *style* = *0*, **const wxValidator&** *val*, **const wxString&** *name* = *"checkBox"*)

Creates the toggle button for two-step construction. See *wxToggleButton::wxToggleButton* (p. 1399) for details.

wxToggleButton::GetValue

bool GetValue() **const**

Gets the state of the toggle button.

Return value

Returns `true` if it is pressed, `false` otherwise.

wxToggleButton::SetValue

void SetValue(**const bool** *state*)

Sets the toggle button to the given state. This does not cause a `EVT_TOGGLEBUTTON` event to be emitted.

Parameters

state

> If `true`, the button is pressed.

wxToolBar

The name wxToolBar is defined to be a synonym for one of the following classes:

- **wxToolBar95** The native Windows 95 toolbar. Used on Windows 95, NT 4 and above.

- **wxToolBarMSW** A Windows implementation. Used on 16-bit Windows.

- **wxToolBarGTK** The GTK toolbar.

Note that the base class **wxToolBarBase** defines automatic scrolling management functionality which is similar to *wxScrolledWindow* (p. 1150), so please refer to this class also.

Derived from

wxToolBarBase
wxControl (p. 215)
wxWindow (p. 1490)
wxEvtHandler (p. 486)
wxObject (p. 1012)

Include files

<wx/toolbar.h> (to allow wxWidgets to select an appropriate toolbar class)
<wx/tbarbase.h> (the base class)

Remarks

You may also create a toolbar that is managed by the frame, by calling *wxFrame::CreateToolBar* (p. 584). Under Pocket PC, you should *always* use this function for creating the toolbar to be managed by the frame, so that wxWidgets can use a combined menubar and toolbar. Where you manage your own toolbars, create a wxToolBar as usual.

The meaning of a "separator" is a vertical line under Windows and simple space under GTK+.

wxToolBar95: Note that this toolbar paints tools to reflect system-wide colours. If you use more than 16 colours in your tool bitmaps, you may wish to suppress this behaviour, otherwise system colours in your bitmaps will inadvertently be mapped to system colours. To do this, set the `msw.remap` system option before creating the toolbar:

```
wxSystemOptions::SetOption(wxT("msw.remap"), 0);
```

If you wish to use 32-bit images (which include an alpha channel for transparency) use:

```
wxSystemOptions::SetOption(wxT("msw.remap"), 2);
```

then colour remapping is switched off, and a transparent background used. But only use this option under Windows XP with true colour:

```
(wxTheApp->GetComCtl32Version() >= 600 && ::wxDisplayDepth() >= 32)
```

Window styles

wxTB_FLAT   Gives the toolbar a flat look (Windows and GTK only).

wxTB_DOCKABLE   Makes the toolbar floatable and dockable (GTK only).

wxTB_HORIZONTAL   Specifies horizontal layout (default).

wxTB_VERTICAL   Specifies vertical layout.

wxTB_TEXT   Shows the text in the toolbar buttons; by default only icons are shown.

wxTB_NOICONS   Specifies no icons in the toolbar buttons; by default they are shown.

wxTB_NODIVIDER   Specifies no divider (border) above the toolbar (Windows only).

wxTB_NOALIGN   Specifies no alignment with the parent window (Windows only, not very useful).

wxTB_HORZ_LAYOUT   Shows the text and the icons alongside, not vertically stacked (Windows and GTK 2 only). This style must be used with wxTB_TEXT.

wxTB_HORZ_TEXT   Combination of wxTB_HORZ_LAYOUT and wxTB_TEXT.

See also *window styles overview* (p. 1762). Note that the Win32 native toolbar ignores `wxTB_NOICONS` style. Also, toggling the `wxTB_TEXT` works only if the style was initially on.

Event handling

The toolbar class emits menu commands in the same way that a frame menubar does, so you can use one EVT_MENU macro for both a menu item and a toolbar button. The event handler functions take a wxCommandEvent argument. For most event macros, the identifier of the tool is passed, but for EVT_TOOL_ENTER the toolbar window identifier is passed and the tool identifier is retrieved from the wxCommandEvent. This is because the identifier may be -1 when the mouse moves off a tool, and -1 is not allowed as an identifier in the event system.

EVT_TOOL(id, func)   Process a
                                      wxEVT_COMMAND_TOOL_CLICKED event (a
                                      synonym for

	wxEVT_COMMAND_MENU_SELECTED). Pass the id of the tool.
EVT_MENU(id, func)	The same as EVT_TOOL.
EVT_TOOL_RANGE(id1, id2, func)	Process a wxEVT_COMMAND_TOOL_CLICKED event for a range of identifiers. Pass the ids of the tools.
EVT_MENU_RANGE(id1, id2, func)	The same as EVT_TOOL_RANGE.
EVT_TOOL_RCLICKED(id, func)	Process a wxEVT_COMMAND_TOOL_RCLICKED event. Pass the id of the tool.
EVT_TOOL_RCLICKED_RANGE(id1, id2, func)	Process a wxEVT_COMMAND_TOOL_RCLICKED event for a range of ids. Pass the ids of the tools.
EVT_TOOL_ENTER(id, func)	Process a wxEVT_COMMAND_TOOL_ENTER event. Pass the id of the toolbar itself. The value of wxCommandEvent::GetSelection is the tool id, or -1 if the mouse cursor has moved off a tool.

See also

Toolbar overview (p. 1808), *wxScrolledWindow* (p. 1150)

wxToolBar::wxToolBar

wxToolBar()

Default constructor.

wxToolBar(**wxWindow*** *parent*, **wxWindowID** *id*, **const wxPoint&** *pos* = *wxDefaultPosition*, **const wxSize&** *size* = *wxDefaultSize*, **long** *style* = *wxTB_HORIZONTAL* | *wxNO_BORDER*, **const wxString&** *name* = *wxPanelNameStr*)

Constructs a toolbar.

Parameters

parent

> Pointer to a parent window.

id

> Window identifier. If -1, will automatically create an identifier.

pos

> Window position. wxDefaultPosition is (-1, -1) which indicates that wxWidgets should generate a default position for the window. If using the wxWindow class directly, supply an actual position.

size

> Window size. wxDefaultSize is (-1, -1) which indicates that wxWidgets should generate a default size for the window.

style

> Window style. See *wxToolBar* (p. 1401) for details.

name

> Window name.

Remarks

After a toolbar is created, you use *wxToolBar::AddTool* (p. 1405) and perhaps *wxToolBar::AddSeparator* (p. 1404), and then you must call *wxToolBar::Realize* (p. 1414) to construct and display the toolbar tools.

You may also create a toolbar that is managed by the frame, by calling *wxFrame::CreateToolBar* (p. 584).

wxToolBar::~wxToolBar

void ~wxToolBar()

Toolbar destructor.

wxToolBar::AddControl

bool AddControl(**wxControl*** *control*)

Adds any control to the toolbar, typically e.g. a combobox.

control

> The control to be added.

wxToolBar::AddSeparator

void AddSeparator()

Adds a separator for spacing groups of tools.

See also

wxToolBar::AddTool (p. 1405), *wxToolBar::SetToolSeparation* (p. 1417)

wxToolBar::AddTool

wxToolBarToolBase* AddTool(**int** *toolId*, **const wxString&** *label*, **const wxBitmap&** *bitmap1*, **const wxString&** *shortHelpString* = *""*, **wxItemKind** *kind* = *wxITEM_NORMAL*)

wxToolBarToolBase* AddTool(**int** *toolId*, **const wxString&** *label*, **const wxBitmap&** *bitmap1*, **const wxBitmap&** *bitmap2* = *wxNullBitmap*, **wxItemKind** *kind* = *wxITEM_NORMAL*, **const wxString&** *shortHelpString* = *""*,**const wxString&** *longHelpString* = *""*, **wxObject*** *clientData* = *NULL*)

wxToolBarToolBase* AddTool(**wxToolBarToolBase*** *tool*)

Adds a tool to the toolbar. The first (short and most commonly used) version has fewer parameters than the full version at the price of not being able to specify some of the more rarely used button features. The last version allows you to add an existing tool.

Parameters

toolId

An integer by which the tool may be identified in subsequent operations.

kind

May be wxITEM_NORMAL for a normal button (default), wxITEM_CHECK for a checkable tool (such tool stays pressed after it had been toggled) or wxITEM_RADIO for a checkable tool which makes part of a radio group of tools each of which is automatically unchecked whenever another button in the group is checked

bitmap1

The primary tool bitmap.

bitmap2

The bitmap used when the tool is disabled. If it is equal to wxNullBitmap, the disabled bitmap is automatically generated by greing the normal one.

shortHelpString

This string is used for the tools tooltip

longHelpString

This string is shown in the statusbar (if any) of the parent frame when the mouse pointer is inside the tool

clientData

An optional pointer to client data which can be retrieved later using

wxToolBar::GetToolClientData (p. 1409).

tool

> The tool to be added.

Remarks

After you have added tools to a toolbar, you must call *wxToolBar::Realize* (p. 1414) in order to have the tools appear.

See also

wxToolBar::AddSeparator (p. 1404), *wxToolBar::AddCheckTool* (p. 1406), *wxToolBar::AddRadioTool* (p. 1406), *wxToolBar::InsertTool* (p. 1412), *wxToolBar::DeleteTool* (p. 1406), *wxToolBar::Realize* (p. 1414)

wxToolBar::AddCheckTool

wxToolBarToolBase* AddCheckTool(**int** *toolId*, **const wxString&** *label*, **const wxBitmap&** *bitmap1*, **const wxBitmap&** *bitmap2*, **const wxString&** *shortHelpString* = *""*, **const wxString&** *longHelpString* = *""*, **wxObject*** *clientData* = *NULL*)

Adds a new check (or toggle) tool to the toolbar. The parameters are the same as in *wxToolBar::AddTool* (p. 1405).

See also

wxToolBar::AddTool (p. 1405)

wxToolBar::AddRadioTool

wxToolBarToolBase* AddRadioTool(**int** *toolId*, **const wxString&** *label*, **const wxBitmap&** *bitmap1*, **const wxBitmap&** *bitmap2*, **const wxString&** *shortHelpString* = *""*, **const wxString&** *longHelpString* = *""*, **wxObject*** *clientData* = *NULL*)

Adds a new radio tool to the toolbar. Consecutive radio tools form a radio group such that exactly one button in the group is pressed at any moment, in other words whenever a button in the group is pressed the previously pressed button is automatically released. You should avoid having the radio groups of only one element as it would be impossible for the user to use such button.

By default, the first button in the radio group is initially pressed, the others are not.

See also

wxToolBar::AddTool (p. 1405)

wxToolBar::DeleteTool

bool DeleteTool(**int** *toolId*)

Removes the specified tool from the toolbar and deletes it. If you don't want to delete the tool, but just to remove it from the toolbar (to possibly add it back later), you may use *RemoveTool* (p. 1414) instead.

Note that it is unnecessary to call *Realize* (p. 1414) for the change to take place, it will happen immediately.

Returns true if the tool was deleted, false otherwise.

See also

DeleteToolByPos (p. 1407)

wxToolBar::DeleteToolByPos

bool DeleteToolByPos(**size_t** *pos*)

This function behaves like *DeleteTool* (p. 1406) but it deletes the tool at the specified position and not the one with the given id.

wxToolBar::EnableTool

void EnableTool(**int** *toolId*, **const bool** *enable*)

Enables or disables the tool.

Parameters

toolId

> Tool to enable or disable.

enable

> If true, enables the tool, otherwise disables it.

NB: This function should only be called after *Realize* (p. 1414).

Remarks

Some implementations will change the visible state of the tool to indicate that it is disabled.

See also

wxToolBar::GetToolEnabled (p. 1409), *wxToolBar::ToggleTool* (p. 1417)

wxToolBar::FindById

wxToolBarToolBase* FindById(**int** *id*)

Returns a pointer to the tool identified by *id* or NULL if no corresponding tool is found.

wxToolBar::FindControl

wxControl* FindControl(**int** *id*)

Returns a pointer to the control identified by *id* or NULL if no corresponding control is found.

wxToolBar::FindToolForPosition

wxToolBarToolBase* FindToolForPosition(**const float** *x*, **const float** *y*) **const**

Finds a tool for the given mouse position.

Parameters

x

      X position.

y

      Y position.

Return value

A pointer to a tool if a tool is found, or NULL otherwise.

Remarks

Used internally, and should not need to be used by the programmer.

wxToolBar::GetToolSize

wxSize GetToolSize()

Returns the size of a whole button, which is usually larger than a tool bitmap because of added 3D effects.

See also

wxToolBar::SetToolBitmapSize (p. 1415), *wxToolBar::GetToolBitmapSize* (p. 1408)

wxToolBar::GetToolBitmapSize

wxSize GetToolBitmapSize()

Returns the size of bitmap that the toolbar expects to have. The default bitmap size is 16 by 15 pixels.

Remarks

Note that this is the size of the bitmap you pass to *wxToolBar::AddTool* (p. 1405), and not the eventual size of the tool button.

See also

wxToolBar::SetToolBitmapSize (p. 1415), *wxToolBar::GetToolSize* (p. 1408)

wxToolBar::GetMargins

wxSize GetMargins() **const**

Returns the left/right and top/bottom margins, which are also used for inter-toolspacing.

See also

wxToolBar::SetMargins (p. 1414)

wxToolBar::GetToolClientData

wxObject* GetToolClientData(**int** *toolId*) **const**

Get any client data associated with the tool.

Parameters

toolId

> Id of the tool, as passed to *wxToolBar::AddTool* (p. 1405).

Return value

Client data, or NULL if there is none.

wxToolBar::GetToolEnabled

bool GetToolEnabled(**int** *toolId*) **const**

Called to determine whether a tool is enabled (responds to user input).

Parameters

toolId

> Id of the tool in question.

Return value

true if the tool is enabled, false otherwise.

See also

wxToolBar::EnableTool (p. 1407)

wxToolBar::GetToolLongHelp

wxString GetToolLongHelp(**int** *toolId*) **const**

Returns the long help for the given tool.

Parameters

toolId

The tool in question.

See also

wxToolBar::SetToolLongHelp (p. 1415), *wxToolBar::SetToolShortHelp* (p. 1416)

wxToolBar::GetToolPacking

int GetToolPacking() **const**

Returns the value used for packing tools.

See also

wxToolBar::SetToolPacking (p. 1416)

wxToolBar::GetToolPos

int GetToolPos(**int** *toolId*) **const**

Returns the tool position in the toolbar, or `wxNOT_FOUND` if the tool is not found.

wxToolBar::GetToolSeparation

int GetToolSeparation() **const**

Returns the default separator size.

See also

wxToolBar::SetToolSeparation (p. 1417)

wxToolBar::GetToolShortHelp

wxString GetToolShortHelp(**int** *toolId*) **const**

Returns the short help for the given tool.

Parameters

toolId

> The tool in question.

See also

wxToolBar::GetToolLongHelp (p. 1410), *wxToolBar::SetToolShortHelp* (p. 1416)

wxToolBar::GetToolState

bool GetToolState(**int** *toolId*) **const**

Gets the on/off state of a toggle tool.

Parameters

toolId

> The tool in question.

Return value

true if the tool is toggled on, false otherwise.

See also

wxToolBar::ToggleTool (p. 1417)

wxToolBar::InsertControl

wxToolBarToolBase * InsertControl(**size_t** *pos*, **wxControl ***control*)

Inserts the control into the toolbar at the given position.

You must call *Realize* (p. 1414) for the change to take place.

See also

AddControl (p. 1404),
InsertTool (p. 1412)

wxToolBar::InsertSeparator

wxToolBarToolBase * InsertSeparator(**size_t** *pos*)

Inserts the separator into the toolbar at the given position.

You must call *Realize* (p. 1414) for the change to take place.

wxToolBar::InsertTool

wxToolBarToolBase * InsertTool(**size_t** *pos*, **int** *toolId*, **const wxBitmap&** *bitmap1*,
const wxBitmap& *bitmap2 = wxNullBitmap*, **bool** *isToggle = false*, **wxObject***
clientData = NULL, **const wxString&** *shortHelpString = ""*, **const wxString&**
longHelpString = "")

wxToolBarToolBase * InsertTool(**size_t** *pos*, **wxToolBarToolBase*** *tool*)

Inserts the tool with the specified attributes into the toolbar at the given position.

You must call *Realize* (p. 1414) for the change to take place.

wxToolBar::OnLeftClick

bool OnLeftClick(**int** *toolId*, **bool** *toggleDown*)

Called when the user clicks on a tool with the left mouse button.

This is the old way of detecting tool clicks; although it will still work, you should use the EVT_MENU or EVT_TOOL macro instead.

Parameters

toolId

    The identifier passed to *wxToolBar::AddTool* (p. 1405).

toggleDown

    true if the tool is a toggle and the toggle is down, otherwise is false.

Return value

If the tool is a toggle and this function returns false, the toggle toggle state (internal and

visual) will not be changed. This provides a way of specifying that toggle operations are not permitted in some circumstances.

See also

wxToolBar::OnMouseEnter (p. 1413), *wxToolBar::OnRightClick* (p. 1413)

wxToolBar::OnMouseEnter

void OnMouseEnter(**int** *toolId*)

This is called when the mouse cursor moves into a tool or out of the toolbar.

This is the old way of detecting mouse enter events; although it will still work, you should use the EVT_TOOL_ENTER macro instead.

Parameters

toolId

> Greater than -1 if the mouse cursor has moved into the tool, or -1 if the mouse cursor has moved. The programmer can override this to provide extra information about the tool, such as a short description on the status line.

Remarks

With some derived toolbar classes, if the mouse moves quickly out of the toolbar, wxWidgets may not be able to detect it. Therefore this function may not always be called when expected.

wxToolBar::OnRightClick

void OnRightClick(**int** *toolId*, **float** *x*, **float** *y*)

Called when the user clicks on a tool with the right mouse button. The programmer should override this function to detect right tool clicks.

This is the old way of detecting tool right clicks; although it will still work, you should use the EVT_TOOL_RCLICKED macro instead.

Parameters

toolId

> The identifier passed to *wxToolBar::AddTool* (p. 1405).

x

> The x position of the mouse cursor.

y

The y position of the mouse cursor.

Remarks

A typical use of this member might be to pop up a menu.

See also

wxToolBar::OnMouseEnter (p. 1413), *wxToolBar::OnLeftClick* (p. 1412)

wxToolBar::Realize

bool Realize()

This function should be called after you have added tools.

wxToolBar::RemoveTool

wxToolBarToolBase * RemoveTool(**int** *id*)

Removes the given tool from the toolbar but doesn't delete it. This allows to insert/add this tool back to this (or another) toolbar later.

Note that it is unnecessary to call *Realize* (p. 1414) for the change to take place, it will happen immediately.

See also

DeleteTool (p. 1406)

wxToolBar::SetBitmapResource

void SetBitmapResource(**int** *resourceId*)

Sets the bitmap resource identifier for specifying tool bitmaps as indices into a custom bitmap. Windows CE only.

wxToolBar::SetMargins

void SetMargins(**const wxSize&** *size*)

void SetMargins(**int** *x*, **int** *y*)

Set the values to be used as margins for the toolbar.

Parameters

size

Margin size.

x

> Left margin, right margin and inter-tool separation value.

y

> Top margin, bottom margin and inter-tool separation value.

Remarks

This must be called before the tools are added if absolute positioning is to be used, and the default (zero-size) margins are to be overridden.

See also

wxToolBar::GetMargins (p. 1409), *wxSize* (p. 1173)

wxToolBar::SetToolBitmapSize

void SetToolBitmapSize(**const wxSize&** *size*)

Sets the default size of each tool bitmap. The default bitmap size is 16 by 15 pixels.

Parameters

size

> The size of the bitmaps in the toolbar.

Remarks

This should be called to tell the toolbar what the tool bitmap size is. Call it before you add tools.

Note that this is the size of the bitmap you pass to *wxToolBar::AddTool* (p. 1405), and not the eventual size of the tool button.

See also

wxToolBar::GetToolBitmapSize (p. 1408), *wxToolBar::GetToolSize* (p. 1408)

wxToolBar::SetToolClientData

void SetToolClientData(**int** *id*, **wxObject*** *clientData*)

Sets the client data associated with the tool.

wxToolBar::SetToolLongHelp

void SetToolLongHelp(**int** *toolId*, **const wxString&** *helpString*)

Sets the long help for the given tool.

Parameters

toolId

> The tool in question.

helpString

> A string for the long help.

Remarks

You might use the long help for displaying the tool purpose on the status line.

See also

wxToolBar::GetToolLongHelp (p. 1410), *wxToolBar::SetToolShortHelp* (p. 1416),

wxToolBar::SetToolPacking

void SetToolPacking(**int** *packing*)

Sets the value used for spacing tools. The default value is 1.

Parameters

packing

> The value for packing.

Remarks

The packing is used for spacing in the vertical direction if the toolbar is horizontal, and for spacing in the horizontal direction if the toolbar is vertical.

See also

wxToolBar::GetToolPacking (p. 1410)

wxToolBar::SetToolShortHelp

void SetToolShortHelp(**int** *toolId,* **const wxString&** *helpString*)

Sets the short help for the given tool.

Parameters

toolId

> The tool in question.

helpString

> The string for the short help.

Remarks

An application might use short help for identifying the tool purpose in a tooltip.

See also

wxToolBar::GetToolShortHelp (p. 1411), *wxToolBar::SetToolLongHelp* (p. 1415)

wxToolBar::SetToolSeparation

void SetToolSeparation(**int** *separation*)

Sets the default separator size. The default value is 5.

Parameters

separation

> The separator size.

See also

wxToolBar::AddSeparator (p. 1404)

wxToolBar::ToggleTool

void ToggleTool(**int** *toolId*, **const bool** *toggle*)

Toggles a tool on or off. This does not cause any event to get emitted.

Parameters

toolId

> Tool in question.

toggle

> If true, toggles the tool on, otherwise toggles it off.

Remarks

Only applies to a tool that has been specified as a toggle tool.

See also

wxToolBar::GetToolState (p. 1411)

wxToolTip

This class holds information about a tooltip associated with a window (see *wxWindow::SetToolTip* (p. 1542)).

The two static methods, *wxToolTip::Enable* (p. 1418) and *wxToolTip::SetDelay* (p. 1418) can be used to globally alter tooltips behaviour.

Derived from

wxObject (p. 1012)

Include files

<wx/tooltip.h>

wxToolTip::Enable

static void Enable(**bool** *flag*)

Enable or disable tooltips globally.

May not be supported on all platforms (eg. wxCocoa).

wxToolTip::SetDelay

static void SetDelay(**long** *msecs*)

Set the delay after which the tooltip appears.

May not be supported on all platforms (eg. wxCocoa).

wxToolTip::wxToolTip

 wxToolTip(**const wxString&** *tip*)

Constructor.

wxToolTip::SetTip

void SetTip(**const wxString&** *tip*)

Set the tooltip text.

wxToolTip::GetTip

wxString GetTip() **const**

Get the tooltip text.

wxToolTip::GetWindow

wxWindow* GetWindow() **const**

Get the associated window.

wxTopLevelWindow

wxTopLevelWindow is a common base class for *wxDialog* (p. 406) and *wxFrame* (p. 579). It is an abstract base class meaning that you never work with objects of this class directly, but all of its methods are also applicable for the two classes above.

Derived from

wxWindow (p. 1490)
wxEvtHandler (p. 486)
wxObject (p. 1012)

Include files

<wx/toplevel.h>

wxTopLevelWindow::GetIcon

const wxIcon& GetIcon() **const**

Returns the standard icon of the window. The icon will be invalid if it hadn't been previously set by *SetIcon* (p. 1421).

See also

GetIcons (p. 1419)

wxTopLevelWindow::GetIcons

const wxIconBundle& GetIcons() **const**

Returns all icons associated with the window, there will be none of them if neither *SetIcon* (p. 1421) nor *SetIcons* (p. 1422) had been called before.

Use *GetIcon* (p. 1419) to get the main icon of the window.

See also

wxIconBundle (p. 774)

wxTopLevelWindow::GetTitle

wxString GetTitle() **const**

Gets a string containing the window title.

See *wxTopLevelWindow::SetTitle* (p. 1423).

wxTopLevelWindow::IsActive

bool IsActive() **const**

Returns `true` if this window is currently active, i.e. if the user is currently working with it.

wxTopLevelWindow::Iconize

void Iconize(**bool** *iconize*)

Iconizes or restores the window.

Parameters

iconize

If `true`, iconizes the window; if `false`, shows and restores it.

See also

wxTopLevelWindow::IsIconized (p. 1420), *wxTopLevelWindow::Maximize* (p. 1421).

wxTopLevelWindow::IsFullScreen

bool IsFullScreen()

Returns `true` if the window is in fullscreen mode.

See also

wxTopLevelWindow::ShowFullScreen (p. 1424)

wxTopLevelWindow::IsIconized

bool IsIconized() **const**

Returns `true` if the window is iconized.

wxTopLevelWindow::IsMaximized

bool IsMaximized() **const**

Returns `true` if the window is maximized.

wxTopLevelWindow::Maximize

void Maximize(**bool** *maximize*)

Maximizes or restores the window.

Parameters

maximize

If `true`, maximizes the window, otherwise it restores it.

Remarks

This function only works under Windows.

See also

wxTopLevelWindow::Iconize (p. 1420)

wxTopLevelWindow::RequestUserAttention

void RequestUserAttention(**int** *flags = wxUSER_ATTENTION_INFO*)

Use a system-dependent way to attract users attention to the window when it is in background.

flags may have the value of either `wxUSER_ATTENTION_INFO`(default) or `wxUSER_ATTENTION_ERROR` which results in a more drastic action. When in doubt, use the default value.

Note that this function should normally be only used when the application is not already in foreground.

This function is currently only implemented for Win32 where it flashes the window icon in the taskbar.

wxTopLevelWindow::SetIcon

void SetIcon(**const wxIcon&** *icon*)

Sets the icon for this window.

Parameters

icon

> The icon to associate with this window.

Remarks

The window takes a 'copy' of *icon*, but since it uses reference counting, the copy is very quick. It is safe to delete *icon* after calling this function.

See also *wxIcon* (p. 766).

wxTopLevelWindow::SetIcons

void SetIcons(**const wxIconBundle&** *icons*)

Sets several icons of different sizes for this window: this allows to use different icons for different situations (e.g. task switching bar, taskbar, window title bar) instead of scaling, with possibly bad looking results, the only icon set by *SetIcon* (p. 1421).

Parameters

icons

> The icons to associate with this window.

See also

wxIconBundle (p. 774).

wxTopLevelWindow::SetLeftMenu

void SetLeftMenu(**int** *id* = *wxID_ANY*, **const wxString&** *label* = *wxEmptyString*, **wxMenu *** *subMenu* = *NULL*)

Sets action or menu activated by pressing left hardware button on the smart phones. Unavailable on full keyboard machines.

Parameters

id

> Identifier for this button.

label

> Text to be displayed on the screen area dedicated to this hardware button.

subMenu

> The menu to be opened after pressing this hardware button.

See also

wxTopLevelWindow::SetRightMenu (p. 1423).

wxTopLevelWindow::SetRightMenu

void SetRightMenu(**int** *id = wxID_ANY*, **const wxString&** *label = wxEmptyString*, **wxMenu *** *subMenu = NULL*)

Sets action or menu activated by pressing right hardware button on the smart phones. Unavailable on full keyboard machines.

Parameters

id

> Identifier for this button.

label

> Text to be displayed on the screen area dedicated to this hardware button.

subMenu

> The menu to be opened after pressing this hardware button.

See also

wxTopLevelWindow::SetLeftMenu (p. 1422).

wxTopLevelWindow::SetShape

bool SetShape(**const wxRegion&** *region*)

If the platform supports it, sets the shape of the window to that depicted by *region*. The system will not display or respond to any mouse event for the pixels that lie outside of the region. To reset the window to the normal rectangular shape simply call *SetShape* again with an empty region. Returns TRUE if the operation is successful.

wxTopLevelWindow::SetTitle

virtual void SetTitle(**const wxString&** *title*)

Sets the window title.

Parameters

title

> The window title.

See also

wxTopLevelWindow::ShowFullScreen

bool ShowFullScreen(**bool** *show*, **long** *style* = *wxFULLSCREEN_ALL*)

Depending on the value of *show* parameter the window is either shown full screen or restored to its normal state. *style* is a bit list containing some or all of the following values, which indicate what elements of the window to hide in full-screen mode:

- wxFULLSCREEN_NOMENUBAR

- wxFULLSCREEN_NOTOOLBAR

- wxFULLSCREEN_NOSTATUSBAR

- wxFULLSCREEN_NOBORDER

- wxFULLSCREEN_NOCAPTION

- wxFULLSCREEN_ALL (all of the above)

This function has not been tested with MDI frames.

Note that showing a window full screen also actually*Show()s* (p. 1544) if it hadn't been shown yet.

See also

wxTopLevelWindow::IsFullScreen (p. 1420)

wxTreeCtrl

A tree control presents information as a hierarchy, with items that may be expanded to show further items. Items in a tree control are referenced by wxTreeItemId handles, which may be tested for validity by calling wxTreeItemId::IsOk.

To intercept events from a tree control, use the event table macros described in *wxTreeEvent* (p. 1443).

Derived from

wxControl (p. 215)
wxWindow (p. 1490)
wxEvtHandler (p. 486)
wxObject (p. 1012)

Include files

<wx/treectrl.h>

Window styles

wxTR_EDIT_LABELS	Use this style if you wish the user to be able to edit labels in the tree control.
wxTR_NO_BUTTONS	For convenience to document that no buttons are to be drawn.
wxTR_HAS_BUTTONS	Use this style to show + and - buttons to the left of parent items.
wxTR_TWIST_BUTTONS	Use this style to show Mac-style twister buttons to the left of parent items. If both wxTR_HAS_BUTTONS and wxTR_TWIST_BUTTONS are given, twister buttons are generated. Generic only.
wxTR_NO_LINES	Use this style to hide vertical level connectors.
wxTR_FULL_ROW_HIGHLIGHT	Use this style to have the background colour and the selection highlight extend over the entire horizontal row of the tree control window. (This flag is ignored under Windows unless you specify wxTR_NO_LINES as well.)
wxTR_LINES_AT_ROOT	Use this style to show lines between root nodes. Only applicable if wxTR_HIDE_ROOT is set and wxTR_NO_LINES is not set.
wxTR_HIDE_ROOT	Use this style to suppress the display of the root node, effectively causing the first-level nodes to appear as a series of root nodes.
wxTR_ROW_LINES	Use this style to draw a contrasting border between displayed rows.
wxTR_HAS_VARIABLE_ROW_HEIGHT	Use this style to cause row heights to be just big enough to fit the content. If not set, all rows use the largest row height. The default is that this flag is unset. Generic only.
wxTR_SINGLE	For convenience to document that only one item may be selected at a time. Selecting another item causes the current selection, if any, to be deselected. This is the default.
wxTR_MULTIPLE	Use this style to allow a range of items to be selected. If a second range is selected, the current range, if any, is deselected.
wxTR_EXTENDED	Use this style to allow disjoint items to be selected. (Only partially implemented; may not work in all cases.)
wxTR_DEFAULT_STYLE	The set of flags that are closest to the defaults for the native control for a particular toolkit.

See also *window styles overview* (p. 1762).

Event handling

To process input from a tree control, use these event handler macros to direct input to member functions that take a *wxTreeEvent* (p. 1443) argument.

EVT_TREE_BEGIN_DRAG(id, func)   Begin dragging with the left mouse button.

EVT_TREE_BEGIN_RDRAG(id, func)   Begin dragging with the right mouse button.

EVT_TREE_END_DRAG(id, func)   End dragging with the left or right mouse button.

EVT_TREE_BEGIN_LABEL_EDIT(id, func)   Begin editing a label. This can be prevented by calling *Veto()* (p. 1012).

EVT_TREE_END_LABEL_EDIT(id, func)   Finish editing a label. This can be prevented by calling *Veto()* (p. 1012).

EVT_TREE_DELETE_ITEM(id, func)   Delete an item.

EVT_TREE_GET_INFO(id, func)   Request information from the application.

EVT_TREE_SET_INFO(id, func)   Information is being supplied.

EVT_TREE_ITEM_ACTIVATED(id, func)   The item has been activated, i.e. chosen by double clicking it with mouse or from keyboard

EVT_TREE_ITEM_COLLAPSED(id, func)   The item has been collapsed.

EVT_TREE_ITEM_COLLAPSING(id, func)   The item is being collapsed. This can be prevented by calling *Veto()* (p. 1012).

EVT_TREE_ITEM_EXPANDED(id, func)   The item has been expanded.

EVT_TREE_ITEM_EXPANDING(id, func)   The item is being expanded. This can be prevented by calling *Veto()* (p. 1012).

EVT_TREE_ITEM_RIGHT_CLICK(id, func)   The user has clicked the item with the right mouse button.

EVT_TREE_ITEM_MIDDLE_CLICK(id, func)   The user has clicked the item with the middle mouse button.

EVT_TREE_SEL_CHANGED(id, func)   Selection has changed.

EVT_TREE_SEL_CHANGING(id, func)   Selection is changing. This can be prevented by calling *Veto()* (p. 1012).

EVT_TREE_KEY_DOWN(id, func)   A key has been pressed.

EVT_TREE_ITEM_GETTOOLTIP(id, func)   The opportunity to set the item tooltip is being given to the application (call wxTreeEvent::SetToolTip). Windows only.

EVT_TREE_ITEM_MENU(id, func)   The menu key has been pressed, asking for a
context menu for the selected item.

See also

wxTreeItemData (p. 1446), *wxTreeCtrl overview* (p. 1795), *wxListBox* (p. 847), *wxListCtrl*
(p. 852), *wxImageList* (p. 806), *wxTreeEvent* (p. 1443)

Win32 notes

wxTreeCtrl class uses the standard common treeview control under Win32 implemented
in the system library `comctl32.dll`. Some versions of this library are known to have
bugs with handling the tree control colours: the usual symptom is that the expanded
items leave black (or otherwise incorrectly coloured) background behind them, especially
for the controls using non default background colour. The recommended solution is to
upgrade the `comctl32.dll`to a newer version: see
http://www.microsoft.com/msdownload/ieplatform/ie/comctrlx86.asp
(`http://www.microsoft.com/msdownload/ieplatform/ie/comctrlx86.asp`)
.

wxTreeCtrl::wxTreeCtrl

wxTreeCtrl()

Default constructor.

wxTreeCtrl(**wxWindow*** *parent*, **wxWindowID** *id*, **const wxPoint&** *pos* =
wxDefaultPosition, **const wxSize&** *size* = *wxDefaultSize*, **long** *style* =
wxTR_HAS_BUTTONS, **const wxValidator&** *validator* = *wxDefaultValidator*, **const
wxString&** *name* = *"listCtrl"*)

Constructor, creating and showing a tree control.

Parameters

parent

   Parent window. Must not be NULL.

id

   Window identifier. A value of -1 indicates a default value.

pos

   Window position.

size

   Window size. If the default size (-1, -1) is specified then the window is sized
   appropriately.

style

>   Window style. See *wxTreeCtrl* (p. 1424).

validator

>   Window validator.

name

>   Window name.

See also

wxTreeCtrl::Create (p. 1429), *wxValidator* (p. 1461)

wxTreeCtrl::~wxTreeCtrl

void ~wxTreeCtrl()

Destructor, destroying the list control.

wxTreeCtrl::AddRoot

wxTreeItemId AddRoot(**const wxString&** *text*, **int** *image = -1*, **int** *selImage = -1*, **wxTreeItemData*** *data =* `NULL`)

Adds the root node to the tree, returning the new item.

The *image* and *selImage* parameters are an index within the normal image list specifying the image to use for unselected and selected items, respectively. If *image* > -1 and *selImage* is -1, the same image is used for both selected and unselected items.

wxTreeCtrl::AppendItem

wxTreeItemId AppendItem(**const wxTreeItemId&** *parent*, **const wxString&** *text*, **int** *image = -1*, **int** *selImage = -1*, **wxTreeItemData*** *data =* `NULL`)

Appends an item to the end of the branch identified by *parent*, return a new item id.

The *image* and *selImage* parameters are an index within the normal image list specifying the image to use for unselected and selected items, respectively. If *image* > -1 and *selImage* is -1, the same image is used for both selected and unselected items.

wxTreeCtrl::AssignButtonsImageList

void AssignButtonsImageList(**wxImageList*** *imageList*)

Sets the buttons image list. The button images assigned with this method will be

automatically deleted by wxTreeCtrl as appropriate (i.e. it takes ownership of the list).

Setting or assigning the button image list enables the display of image buttons. Once enabled, the only way to disable the display of button images is to set the button image list to NULL.

This function is only available in the generic version.

See also *SetButtonsImageList* (p. 1439).

wxTreeCtrl::AssignImageList

void AssignImageList(**wxImageList*** *imageList*)

Sets the normal image list. Image list assigned with this method will be automatically deleted by wxTreeCtrl as appropriate (i.e. it takes ownership of the list).

See also *SetImageList* (p. 1440).

wxTreeCtrl::AssignStateImageList

void AssignStateImageList(**wxImageList*** *imageList*)

Sets the state image list. Image list assigned with this method will be automatically deleted by wxTreeCtrl as appropriate (i.e. it takes ownership of the list).

See also *SetStateImageList* (p. 1442).

wxTreeCtrl::Collapse

void Collapse(**const wxTreeItemId&** *item*)

Collapses the given item.

wxTreeCtrl::CollapseAndReset

void CollapseAndReset(**const wxTreeItemId&** *item*)

Collapses the given item and removes all children.

wxTreeCtrl::Create

bool wxTreeCtrl(**wxWindow*** *parent*, **wxWindowID** *id*, **const wxPoint&** *pos = wxDefaultPosition*, **const wxSize&** *size = wxDefaultSize*, **long** *style = wxTR_HAS_BUTTONS*, **const wxValidator&** *validator = wxDefaultValidator*, **const wxString&** *name = "listCtrl"*)

Creates the tree control. See *wxTreeCtrl::wxTreeCtrl* (p. 1427) for further details.

wxTreeCtrl::Delete

void Delete(**const wxTreeItemId&** *item*)

Deletes the specified item. A `EVT_TREE_DELETE_ITEM` event will be generated.

This function may cause a subsequent call to GetNextChild to fail.

wxTreeCtrl::DeleteAllItems

void DeleteAllItems()

Deletes all the items in the control. Note that this may not generate `EVT_TREE_DELETE_ITEM` events under some Windows versions although normally such event is generated for each removed item.

wxTreeCtrl::DeleteChildren

void DeleteChildren(**const wxTreeItemId&** *item*)

Deletes all children of the given item (but not the item itself). Note that this will **not** generate any events unlike *Delete* (p. 1430) method.

If you have called *wxTreeCtrl::SetItemHasChildren* (p. 1441), you may need to call it again since *DeleteChildren* does not automatically clear the setting.

wxTreeCtrl::EditLabel

void EditLabel(**const wxTreeItemId&** *item*)

Starts editing the label of the given item. This function generates a EVT_TREE_BEGIN_LABEL_EDIT event which can be vetoed so that no text control will appear for in-place editing.

If the user changed the label (i.e. s/he does not press ESC or leave the text control without changes, a EVT_TREE_END_LABEL_EDIT event will be sent which can be vetoed as well.

See also

wxTreeCtrl::EndEditLabel (p. 1430),*wxTreeEvent* (p. 1443)

wxTreeCtrl::EndEditLabel

void EndEditLabel(**bool** *cancelEdit*)

Ends label editing. If *cancelEdit* is `true`, the edit will be cancelled.

This function is currently supported under Windows only.

See also

wxTreeCtrl::EditLabel (p. 1430)

wxTreeCtrl::EnsureVisible

void EnsureVisible(**const wxTreeItemId&** *item*)

Scrolls and/or expands items to ensure that the given item is visible.

wxTreeCtrl::Expand

void Expand(**const wxTreeItemId&** *item*)

Expands the given item.

wxTreeCtrl::GetBoundingRect

bool GetBoundingRect(**const wxTreeItemId&** *item*, **wxRect&** *rect*, **bool** *textOnly* = *false*) **const**

Retrieves the rectangle bounding the *item*. If *textOnly* is `true`, only the rectangle around the item's label will be returned, otherwise the item's image is also taken into account.

The return value is `true` if the rectangle was successfully retrieved or `false` if it was not (in this case *rect* is not changed) - for example, if the item is currently invisible.

wxPython note: The wxPython version of this method requires only the `item` and `textOnly` parameters. The return value is either a `wxRect` object or `None`.

wxPerl note: In wxPerl this method only takes the parameters `item` and `textOnly`, and returns a Wx::Rect (or undef).

wxTreeCtrl::GetButtonsImageList

wxImageList* GetButtonsImageList() **const**

Returns the buttons image list (from which application-defined button images are taken).

This function is only available in the generic version.

wxTreeCtrl::GetChildrenCount

size_t GetChildrenCount(**const wxTreeItemId&** *item*, **bool** *recursively* = `true`) **const**

Returns the number of items in the branch. If *recursively* is `true`, returns the total number of descendants, otherwise only one level of children is counted.

wxTreeCtrl::GetCount

int GetCount() **const**

Returns the number of items in the control.

wxTreeCtrl::GetEditControl

wxTextCtrl * GetEditControl() **const**

Returns the edit control being currently used to edit a label. Returns `NULL` if no label is being edited.

NB: It is currently only implemented for wxMSW.

wxTreeCtrl::GetFirstChild

wxTreeItemId GetFirstChild(**const wxTreeItemId&** *item*, **wxTreeItemIdValue &** *cookie*) **const**

Returns the first child; call *wxTreeCtrl::GetNextChild* (p. 1434) for the next child.

For this enumeration function you must pass in a 'cookie' parameter which is opaque for the application but is necessary for the library to make these functions reentrant (i.e. allow more than one enumeration on one and the same object simultaneously). The cookie passed to GetFirstChild and GetNextChild should be the same variable.

Returns an invalid tree item (i.e. IsOk() returns `false`) if there are no further children.

See also

wxTreeCtrl::GetNextChild (p. 1434),*wxTreeCtrl::GetNextSibling* (p. 1435)

wxPython note: In wxPython the returned wxTreeItemId and the new cookie value are both returned as a tuple containing the two values.

wxPerl note: In wxPerl this method only takes the `item` parameter, and returns a 2-element list (`item,` `cookie`).

wxTreeCtrl::GetFirstVisibleItem

wxTreeItemId GetFirstVisibleItem() **const**

Returns the first visible item.

wxTreeCtrl::GetImageList

wxImageList* GetImageList() **const**

Returns the normal image list.

wxTreeCtrl::GetIndent

int GetIndent() **const**

Returns the current tree control indentation.

wxTreeCtrl::GetItemBackgroundColour

wxColour GetItemBackgroundColour(**const wxTreeItemId&** *item*) **const**

Returns the background colour of the item.

wxTreeCtrl::GetItemData

wxTreeItemData* GetItemData(**const wxTreeItemId&** *item*) **const**

Returns the tree item data associated with the item.

See also

wxTreeItemData (p. 1446)

wxPython note: wxPython provides the following shortcut method:

GetPyData(item)	Returns the Python Object associated with the wxTreeItemData for the given item Id.

wxPerl note: wxPerl provides the following shortcut method:**GetPlData(item)** Returns the Perl data associated with the Wx::TreeItemData. It is just the same as tree->GetItemData(item)->GetData().

wxTreeCtrl::GetItemFont

wxFont GetItemFont(**const wxTreeItemId&** *item*) **const**

Returns the font of the item label.

wxTreeCtrl::GetItemImage

int GetItemImage(**const wxTreeItemId&** *item*, **wxTreeItemIcon** *which* = *wxTreeItemIcon_Normal*) **const**

Gets the specified item image. The value of *which* may be:

- •_Normal to get the normal item image

- •_Selected to get the selected item image (i.e. the image which is shown when the item is currently selected)

- •_Expanded to get the expanded image (this only makes sense for items which have children - then this image is shown when the item is expanded and the normal image is shown when it is collapsed)

- •_SelectedExpanded to get the selected expanded image (which is shown when an expanded item is currently selected)

wxTreeCtrl::GetItemText

wxString GetItemText(**const wxTreeItemId&** *item*) **const**

Returns the item label.

wxTreeCtrl::GetItemTextColour

wxColour GetItemTextColour(**const wxTreeItemId&** *item*) **const**

Returns the colour of the item label.

wxTreeCtrl::GetLastChild

wxTreeItemId GetLastChild(**const wxTreeItemId&** *item*) **const**

Returns the last child of the item (or an invalid tree item if this item has no children).

See also

GetFirstChild (p. 1432),*wxTreeCtrl::GetNextSibling* (p. 1435),*GetLastChild* (p. 1434)

wxTreeCtrl::GetNextChild

wxTreeItemId GetNextChild(**const wxTreeItemId&** *item*, **wxTreeItemIdValue &** *cookie*) **const**

Returns the next child; call *wxTreeCtrl::GetFirstChild* (p. 1432) for the first child.

For this enumeration function you must pass in a 'cookie' parameter which is opaque for

the application but is necessary for the library to make these functions reentrant (i.e. allow more than one enumeration on one and the same object simultaneously). The cookie passed to GetFirstChild and GetNextChild should be the same.

Returns an invalid tree item if there are no further children.

See also

wxTreeCtrl::GetFirstChild (p. 1432)

wxPython note: In wxPython the returned wxTreeItemId and the new cookie value are both returned as a tuple containing the two values.

wxPerl note: In wxPerl this method returns a 2-element list (item, cookie), instead of modifying its parameters.

wxTreeCtrl::GetNextSibling

wxTreeItemId GetNextSibling(**const wxTreeItemId&** *item*) **const**

Returns the next sibling of the specified item; call *wxTreeCtrl::GetPrevSibling* (p. 1436) for the previous sibling.

Returns an invalid tree item if there are no further siblings.

See also

wxTreeCtrl::GetPrevSibling (p. 1436)

wxTreeCtrl::GetNextVisible

wxTreeItemId GetNextVisible(**const wxTreeItemId&** *item*) **const**

Returns the next visible item.

wxTreeCtrl::GetItemParent

wxTreeItemId GetItemParent(**const wxTreeItemId&** *item*) **const**

Returns the item's parent.

wxTreeCtrl::GetParent

wxTreeItemId GetParent(**const wxTreeItemId&** *item*) **const**

NOTE: This function is deprecated and will only work if WXWIN_COMPATIBILITY_2_2 is defined. Use *wxTreeCtrl::GetItemParent* (p. 1435) instead.

Returns the item's parent.

wxPython note: This method is named `GetItemParent` to avoid a name clash with wxWindow::GetParent.

wxTreeCtrl::GetPrevSibling

wxTreeItemId GetPrevSibling(const wxTreeItemId& *item*) **const**

Returns the previous sibling of the specified item; call *wxTreeCtrl::GetNextSibling* (p. 1435) for the next sibling.

Returns an invalid tree item if there are no further children.

See also

wxTreeCtrl::GetNextSibling (p. 1435)

wxTreeCtrl::GetPrevVisible

wxTreeItemId GetPrevVisible(const wxTreeItemId& *item*) **const**

Returns the previous visible item.

wxTreeCtrl::GetRootItem

wxTreeItemId GetRootItem() **const**

Returns the root item for the tree control.

wxTreeCtrl::GetItemSelectedImage

int GetItemSelectedImage(const wxTreeItemId& *item*) **const**

Gets the selected item image (this function is obsolete, use `GetItemImage(item, wxTreeItemIcon_Selected)` instead).

wxTreeCtrl::GetSelection

wxTreeItemId GetSelection() **const**

Returns the selection, or an invalid item if there is no selection. This function only works with the controls without wxTR_MULTIPLE style, use*GetSelections* (p. 1436) for the controls which do have this style.

wxTreeCtrl::GetSelections

size_t GetSelections(**wxArrayTreeItemIds&** *selection*) **const**

Fills the array of tree items passed in with the currently selected items. This function can be called only if the control has the wxTR_MULTIPLE style.

Returns the number of selected items.

wxPython note: The wxPython version of this method accepts no parameters and returns a Python list of `wxTreeItemId`s.

wxPerl note: In wxPerl this method takes no parameters and returns a list of `Wx::TreeItemId`s.

wxTreeCtrl::GetStateImageList

wxImageList* GetStateImageList() **const**

Returns the state image list (from which application-defined state images are taken).

wxTreeCtrl::HitTest

wxTreeItemId HitTest(**const wxPoint&** *point*, **int&** *flags*)

Calculates which (if any) item is under the given point, returning the tree item id at this point plus extra information *flags*. *flags* is a bitlist of the following:

wxTREE_HITTEST_ABOVE Above the client area.

wxTREE_HITTEST_BELOW Below the client area.

wxTREE_HITTEST_NOWHERE   In the client area but below the last item.

wxTREE_HITTEST_ONITEMBUTTON   On the button associated with an item.

wxTREE_HITTEST_ONITEMICON   On the bitmap associated with an item.

wxTREE_HITTEST_ONITEMINDENT   In the indentation associated with an item.

wxTREE_HITTEST_ONITEMLABEL On the label (string) associated with an item.

wxTREE_HITTEST_ONITEMRIGHT In the area to the right of an item.

wxTREE_HITTEST_ONITEMSTATEICON   On the state icon for a tree view item that is in a user-defined state.

wxTREE_HITTEST_TOLEFT To the right of the client area.

wxTREE_HITTEST_TORIGHT   To the left of the client area.

wxPython note: in wxPython both the wxTreeItemId and the flags are returned as a tuple.

wxPerl note: In wxPerl this method only takes the `point` parameter and returns a 2-element list (`item, flags`).

wxTreeCtrl::InsertItem

wxTreeItemId InsertItem(**const wxTreeItemId&** *parent*, **const wxTreeItemId&** *previous*, **const wxString&** *text*, **int** *image = -1*, **int** *selImage = -1*, **wxTreeItemData*** *data = NULL*)

wxTreeItemId InsertItem(**const wxTreeItemId&** *parent*, **size_t** *before*, **const wxString&** *text*, **int** *image = -1*, **int** *selImage = -1*, **wxTreeItemData*** *data = NULL*)

Inserts an item after a given one (*previous*) or before one identified by its position (*before*).*before* must be less than the number of children.

The *image* and *selImage* parameters are an index within the normal image list specifying the image to use for unselected and selected items, respectively. If *image* > -1 and *selImage* is -1, the same image is used for both selected and unselected items.

wxPython note: The second form of this method is called`InsertItemBefore` in wxPython.

wxTreeCtrl::IsBold

bool IsBold(**const wxTreeItemId&** *item*) **const**

Returns `true` if the given item is in bold state.

See also: *SetItemBold* (p. 1440)

wxTreeCtrl::IsExpanded

bool IsExpanded(**const wxTreeItemId&** *item*) **const**

Returns `true` if the item is expanded (only makes sense if it has children).

wxTreeCtrl::IsSelected

bool IsSelected(**const wxTreeItemId&** *item*) **const**

Returns `true` if the item is selected.

wxTreeCtrl::IsVisible

bool IsVisible(**const wxTreeItemId&** *item*) **const**

Returns `true` if the item is visible (it might be outside the view, or not expanded).

wxTreeCtrl::ItemHasChildren

bool ItemHasChildren(**const wxTreeItemId&** *item*) **const**

Returns `true` if the item has children.

wxTreeCtrl::OnCompareItems

int OnCompareItems(**const wxTreeItemId&** *item1*, **const wxTreeItemId&** *item2*)

Override this function in the derived class to change the sort order of the items in the tree control. The function should return a negative, zero or positive value if the first item is less than, equal to or greater than the second one.

The base class version compares items alphabetically.

See also: *SortChildren* (p. 1442)

wxTreeCtrl::PrependItem

wxTreeItemId PrependItem(**const wxTreeItemId&** *parent*, **const wxString&** *text*, **int** *image = -1*, **int** *selImage = -1*, **wxTreeItemData*** *data* = `NULL`)

Appends an item as the first child of *parent*, return a new item id.

The *image* and *selImage* parameters are an index within the normal image list specifying the image to use for unselected and selected items, respectively. If *image* > -1 and *selImage* is -1, the same image is used for both selected and unselected items.

wxTreeCtrl::ScrollTo

void ScrollTo(**const wxTreeItemId&** *item*)

Scrolls the specified item into view.

wxTreeCtrl::SelectItem

bool SelectItem(**const wxTreeItemId&** *item*, **bool** *select* = `true`)

Selects the given item. In multiple selection controls, can be also used to deselect a currently selected item if the value of *select* is false.

wxTreeCtrl::SetButtonsImageList

void SetButtonsImageList(**wxImageList*** *imageList*)

Sets the buttons image list (from which application-defined button images are taken). The button images assigned with this method will**not** be deleted by wxTreeCtrl's destructor, you must delete it yourself.

Setting or assigning the button image list enables the display of image buttons. Once enabled, the only way to disable the display of button images is to set the button image list to `NULL`.

This function is only available in the generic version.

See also *AssignButtonsImageList* (p. 1428).

wxTreeCtrl::SetIndent

void SetIndent(**int** *indent*)

Sets the indentation for the tree control.

wxTreeCtrl::SetImageList

void SetImageList(**wxImageList*** *imageList*)

Sets the normal image list. Image list assigned with this method will**not** be deleted by wxTreeCtrl's destructor, you must delete it yourself.

See also *AssignImageList* (p. 1429).

wxTreeCtrl::SetItemBackgroundColour

void SetItemBackgroundColour(**const wxTreeItemId&** *item*, **const wxColour&** *col*)

Sets the colour of the item's background.

wxTreeCtrl::SetItemBold

void SetItemBold(**const wxTreeItemId&** *item*, **bool** *bold* = `true`)

Makes item appear in bold font if *bold* parameter is `true` or resets it to the normal state.

See also: *IsBold* (p. 1438)

wxTreeCtrl::SetItemData

void SetItemData(**const wxTreeItemId&** *item*, **wxTreeItemData*** *data*)

Sets the item client data.

wxPython note: wxPython provides the following shortcut method:

> **SetPyData(item, obj)**   Associate the given Python Object with the wxTreeItemData for the given item Id.

wxPerl note: wxPerl provides the following shortcut method:**SetPlData(item, data)**   Sets the Perl data associated with the Wx::TreeItemData. It is just the same as tree->GetItemData(item)->SetData(data).

wxTreeCtrl::SetItemDropHighlight

void SetItemDropHighlight(**const wxTreeItemId&** *item*, **bool***highlight* = *true*)

Gives the item the visual feedback for Drag'n'Drop actions, which is useful if something is dragged from the outside onto the tree control (as opposed to a DnD operation within the tree control, which already is implemented internally).

wxTreeCtrl::SetItemFont

void SetItemFont(**const wxTreeItemId&** *item*, **const wxFont&** *font*)

Sets the item's font. All items in the tree should have the same height to avoid text clipping, so the fonts height should be the same for all of them, although font attributes may vary.

See also

SetItemBold (p. 1440)

wxTreeCtrl::SetItemHasChildren

void SetItemHasChildren(**const wxTreeItemId&** *item*, **bool** *hasChildren* = *true*)

Force appearance of the button next to the item. This is useful to allow the user to expand the items which don't have any children now, but instead adding them only when needed, thus minimizing memory usage and loading time.

wxTreeCtrl::SetItemImage

void SetItemImage(**const wxTreeItemId&** *item*, **int** *image*, **wxTreeItemIcon** *which* = *wxTreeItemIcon_Normal*)

Sets the specified item image. See *GetItemImage* (p. 1434)for the description of the *which* parameter.

wxTreeCtrl::SetItemSelectedImage

void SetItemSelectedImage(**const wxTreeItemId&** *item*, **int** *selImage*)

Sets the selected item image (this function is obsolete, use `SetItemImage(item, wxTreeItemIcon_Selected)` instead).

wxTreeCtrl::SetItemText

void SetItemText(**const wxTreeItemId&** *item*, **const wxString&** *text*)

Sets the item label.

wxTreeCtrl::SetItemTextColour

void SetItemTextColour(**const wxTreeItemId&** *item*, **const wxColour&** *col*)

Sets the colour of the item's text.

wxTreeCtrl::SetStateImageList

void SetStateImageList(**wxImageList*** *imageList*)

Sets the state image list (from which application-defined state images are taken). Image list assigned with this method will**not** be deleted by wxTreeCtrl's destructor, you must delete it yourself.

See also *AssignStateImageList* (p. 1429).

wxTreeCtrl::SetWindowStyle

void SetWindowStyle(**long***styles*)

Sets the mode flags associated with the display of the tree control. The new mode takes effect immediately. (Generic only; MSW ignores changes.)

wxTreeCtrl::SortChildren

void SortChildren(**const wxTreeItemId&** *item*)

Sorts the children of the given item using*OnCompareItems* (p. 1439) method of wxTreeCtrl. You should override that method to change the sort order (the default is ascending case-sensitive alphabetical order).

See also

wxTreeItemData (p. 1446), *OnCompareItems* (p. 1439)

wxTreeCtrl::Toggle

void Toggle(**const wxTreeItemId&** *item*)

Toggles the given item between collapsed and expanded states.

wxTreeCtrl::ToggleItemSelection

void ToggleItemSelection(**const wxTreeItemId&** *item*)

Toggles the given item between selected and unselected states. For multiselection controls only.

wxTreeCtrl::Unselect

void Unselect()

Removes the selection from the currently selected item (if any).

wxTreeCtrl::UnselectAll

void UnselectAll()

This function either behaves the same as *Unselect* (p. 1443)if the control doesn't have wxTR_MULTIPLE style, or removes the selection from all items if it does have this style.

wxTreeCtrl::UnselectItem

void UnselectItem(**const wxTreeItemId&** *item*)

Unselects the given item. This works in multiselection controls only.

wxTreeEvent

A tree event holds information about events associated with wxTreeCtrl objects.

Derived from

wxNotifyEvent (p. 1011)
wxCommandEvent (p. 180)
wxEvent (p. 483)
wxObject (p. 1012)

Include files

<wx/treectrl.h>

Event table macros

To process input from a tree control, use these event handler macros to direct input to member functions that take a wxTreeEvent argument.

EVT_TREE_BEGIN_DRAG(id, func)	The user has started dragging an item with the left mouse button. The event handler must call **wxTreeEvent::Allow()** for the drag operation to continue.
EVT_TREE_BEGIN_RDRAG(id, func)	The user has started dragging an item with the right mouse button. The event handler must call **wxTreeEvent::Allow()** for the drag operation to continue.
EVT_TREE_BEGIN_LABEL_EDIT(id, func)	Begin editing a label. This can be prevented by calling *Veto()* (p. 1012).
EVT_TREE_END_DRAG(id, func)	The user has released the mouse after dragging an item.
EVT_TREE_END_LABEL_EDIT(id, func)	The user has finished editing a label. This can be prevented by calling *Veto()* (p. 1012).
EVT_TREE_DELETE_ITEM(id, func)	A tree item has been deleted.
EVT_TREE_ITEM_ACTIVATED(id, func)	An item has been activated (e.g. double clicked).
EVT_TREE_ITEM_COLLAPSED(id, func)	The item has been collapsed.
EVT_TREE_ITEM_COLLAPSING(id, func)	The item is being collapsed. This can be prevented by calling *Veto()* (p. 1012).
EVT_TREE_ITEM_EXPANDED(id, func)	The item has been expanded.
EVT_TREE_ITEM_EXPANDING(id, func)	The item is being expanded. This can be prevented by calling *Veto()* (p. 1012).
EVT_TREE_ITEM_RIGHT_CLICK(id, func)	The user has clicked the item with the right mouse button.
EVT_TREE_ITEM_MIDDLE_CLICK(id, func)	The user has clicked the item with the middle mouse button.
EVT_TREE_KEY_DOWN(id, func)	A key has been pressed.
EVT_TREE_SEL_CHANGED(id, func)	Selection has changed.
EVT_TREE_SEL_CHANGING(id, func)	Selection is changing. This can be

	prevented by calling *Veto()* (p. 1012).
EVT_TREE_KEY_DOWN(id, func)	A key has been pressed.
EVT_TREE_ITEM_GETTOOLTIP(id, func)	The opportunity to set the item tooltip is being given to the application (call wxTreeEvent::SetToolTip). Windows only.

See also

wxTreeCtrl (p. 1424)

wxTreeEvent::wxTreeEvent

 wxTreeEvent(**WXTYPE** *commandType = 0*, **int** *id = 0*)

Constructor.

wxTreeEvent::GetKeyCode

int GetKeyCode() **const**

Returns the key code if the event was is a key event. Use *GetKeyEvent* (p. 1445) to get the values of the modifier keys for this event (i.e. Shift or Ctrl).

wxTreeEvent::GetItem

wxTreeItemId GetItem() **const**

Returns the item (valid for all events).

wxTreeEvent::GetKeyEvent

const wxKeyEvent& GetKeyEvent() **const**

Returns the key event for `EVT_TREE_KEY_DOWN` events.

wxTreeEvent::GetLabel

const wxString& GetLabel() **const**

Returns the label if the event was a begin or end edit label event.

wxTreeEvent::GetOldItem

wxTreeItemId GetOldItem() **const**

Returns the old item index (valid for EVT_TREE_ITEM_CHANGING and CHANGED events)

wxTreeEvent::GetPoint()

wxPoint GetPoint() **const**

Returns the position of the mouse pointer if the event is a drag event.

wxTreeEvent::IsEditCancelled()

bool IsEditCancelled() **const**

Returns true if the label edit was cancelled. This should be called from within an EVT_TREE_END_LABEL_EDIT handler.

wxTreeEvent::SetToolTip

void SetToolTip(**const wxString&** *tooltip*)

Set the tooltip for the item (valid for EVT_TREE_ITEM_GETTOOLTIP events). Windows only.

wxTreeItemData

wxTreeItemData is some (arbitrary) user class associated with some item. The main advantage of having this class is that wxTreeItemData objects are destroyed automatically by the tree and, as this class has virtual destructor, it means that the memory and any other resources associated with a tree item will be automatically freed when it is deleted. Note that we don't use wxObject as the base class for wxTreeItemData because the size of this class is critical: in many applications, each tree leaf will have wxTreeItemData associated with it and the number of leaves may be quite big.

Also please note that because the objects of this class are deleted by the tree using the operator `delete`, they must always be allocated on the heap using `new`.

Derived from

wxClientData (p. 148)

Include files

<wx/treectrl.h>

wxTreeItemData::wxTreeItemData

wxTreeItemData()

Default constructor.

wxPython note: The wxPython version of this constructor optionally accepts any Python object as a parameter. This object is then associated with the tree item using the wxTreeItemData as a container.

In addition, the following methods are added in wxPython for accessing the object:

GetData()	Returns a reference to the Python Object
SetData(obj)	Associates a new Python Object with the wxTreeItemData

wxPerl note: In wxPerl the constructor accepts as parameter an optional scalar, and stores it as client data. You may retrieve this data by calling **GetData()**, and set it by calling **SetData(data).**

wxTreeItemData::~wxTreeItemData

void ~wxTreeItemData()

Virtual destructor.

wxTreeItemData::GetId

const wxTreeItemId& GetId()

Returns the item associated with this node.

wxTreeItemData::SetId

void SetId(**const wxTreeItemId&** *id*)

Sets the item associated with this node.

wxUpdateUIEvent

This class is used for pseudo-events which are called by wxWidgets to give an application the chance to update various user interface elements.

Derived from

wxCommandEvent (p. 180)
wxEvent (p. 483)
wxObject (p. 1012)

Include files

<wx/event.h>

Event table macros

To process an update event, use these event handler macros to direct input to member functions that take a wxUpdateUIEvent argument.

EVT_UPDATE_UI(id, func)   Process a wxEVT_UPDATE_UI event for the command with the given id.

EVT_UPDATE_UI_RANGE(id1, id2, func) Process a wxEVT_UPDATE_UI event for any command with id included in the given range.

Remarks

Without update UI events, an application has to work hard to check/uncheck, enable/disable, and set the text for elements such as menu items and toolbar buttons. The code for doing this has to be mixed up with the code that is invoked when an action is invoked for a menu item or button.

With update UI events, you define an event handler to look at the state of the application and change UI elements accordingly. wxWidgets will call your member functions in idle time, so you don't have to worry where to call this code. In addition to being a clearer and more declarative method, it also means you don't have to worry whether you're updating a toolbar or menubar identifier. The same handler can update a menu item and toolbar button, if the identifier is the same.

Instead of directly manipulating the menu or button, you call functions in the event object, such as *wxUpdateUIEvent::Check* (p. 1449). wxWidgets will determine whether such a call has been made, and which UI element to update.

These events will work for popup menus as well as menubars. Just before a menu is popped up, *wxMenu::UpdateUI* (p. 953) is called to process any UI events for the window that owns the menu.

If you find that the overhead of UI update processing is affecting your application, you can do one or both of the following:

1.   Call *wxUpdateUIEvent::SetMode* (p. 1451) with a value of

wxUPDATE_UI_PROCESS_SPECIFIED, and set the extra style wxWS_EX_PROCESS_UPDATE_EVENTS for every window that should receive update events. No other windows will receive update events.

2. Call *wxUpdateUIEvent::SetUpdateInterval* (p. 1452) with a millisecond value to set the delay between updates. You may need to call *wxWindow::UpdateWindowUI* (p. 1546) at critical points, for example when a dialog is about to be shown, in case the user sees a slight delay before windows are updated.

Note that although events are sent in idle time, defining a wxIdleEvent handler for a window does not affect this because the events are sent from *wxWindow::OnInternalIdle* (p. 1520) which is **always** called in idle time.

wxWidgets tries to optimize update events on some platforms. On Windows and GTK+, events for menubar items are only sent when the menu is about to be shown, and not in idle time.

See also

Event handling overview (p. 1751)

wxUpdateUIEvent::wxUpdateUIEvent

 wxUpdateUIEvent(**wxWindowID** *commandId = 0*)

Constructor.

wxUpdateUIEvent::CanUpdate

static bool CanUpdate(**wxWindow*** *window*)

Returns `true` if it is appropriate to update (send UI update events to) this window.

This function looks at the mode used (see *wxUpdateUIEvent::SetMode* (p. 1451)), the wxWS_EX_PROCESS_UPDATE_EVENTS flag in *window*, the time update events were last sent in idle time, and the update interval, to determine whether events should be sent to this window now. By default this will always return `true` because the update mode is initially wxUPDATE_UI_PROCESS_ALL and the interval is set to 0; so update events will be sent as often as possible. You can reduce the frequency that events are sent by changing the mode and/or setting an update interval.

See also

wxUpdateUIEvent::ResetUpdateTime (p. 1451), *wxUpdateUIEvent::SetUpdateInterval* (p. 1452), *wxUpdateUIEvent::SetMode* (p. 1451)

wxUpdateUIEvent::Check

void Check(**bool** *check*)

Check or uncheck the UI element.

wxUpdateUIEvent::Enable

void Enable(**bool** *enable*)

Enable or disable the UI element.

wxUpdateUIEvent::GetChecked

bool GetChecked() **const**

Returns true if the UI element should be checked.

wxUpdateUIEvent::GetEnabled

bool GetEnabled() **const**

Returns true if the UI element should be enabled.

wxUpdateUIEvent::GetSetChecked

bool GetSetChecked() **const**

Returns true if the application has called *wxUpdateUIEvent::Check* (p. 1449). For wxWidgets internal use only.

wxUpdateUIEvent::GetSetEnabled

bool GetSetEnabled() **const**

Returns true if the application has called *wxUpdateUIEvent::Enable* (p. 1450). For wxWidgets internal use only.

wxUpdateUIEvent::GetSetText

bool GetSetText() **const**

Returns true if the application has called *wxUpdateUIEvent::SetText* (p. 1452). For wxWidgets internal use only.

wxUpdateUIEvent::GetText

wxString GetText() **const**

Returns the text that should be set for the UI element.

wxUpdateUIEvent::GetMode

static wxUpdateUIMode GetMode()

Static function returning a value specifying how wxWidgets will send update events: to all windows, or only to those which specify that they will process the events.

See *wxUpdateUIEvent::SetMode* (p. 1451).

wxUpdateUIEvent::GetUpdateInterval

static long GetUpdateInterval()

Returns the current interval between updates in milliseconds. -1 disables updates, 0 updates as frequently as possible.

See *wxUpdateUIEvent::SetUpdateInterval* (p. 1452).

wxUpdateUIEvent::ResetUpdateTime

static void ResetUpdateTime()

Used internally to reset the last-updated time to the current time. It is assumed that update events are normally sent in idle time, so this is called at the end of idle processing.

See also

wxUpdateUIEvent::CanUpdate (p. 1449), *wxUpdateUIEvent::SetUpdateInterval* (p. 1452), *wxUpdateUIEvent::SetMode* (p. 1451)

wxUpdateUIEvent::SetMode

static void SetMode(**wxUpdateUIMode** *mode*)

Specify how wxWidgets will send update events: to all windows, or only to those which specify that they will process the events.

mode may be one of the following values. The default is wxUPDATE_UI_PROCESS_ALL.

```
enum wxUpdateUIMode
{
        // Send UI update events to all windows
    wxUPDATE_UI_PROCESS_ALL,
```

```
        // Send UI update events to windows that have
        // the wxWS_EX_PROCESS_UI_UPDATES flag specified
    wxUPDATE_UI_PROCESS_SPECIFIED
};
```

wxUpdateUIEvent::SetText

void SetText(**const wxString&** *text*)

Sets the text for this UI element.

wxUpdateUIEvent::SetUpdateInterval

static void SetUpdateInterval(**long** *updateInterval*)

Sets the interval between updates in milliseconds. Set to -1 to disable updates, or to 0 to update as frequently as possible. The default is 0.

Use this to reduce the overhead of UI update events if your application has a lot of windows. If you set the value to -1 or greater than 0, you may also need to call *wxWindow::UpdateWindowUI* (p. 1546) at appropriate points in your application, such as when a dialog is about to be shown.

wxURI

wxURI is used to extract information from a URI (Uniform Resource Identifier).

For information about URIs, see RFC 3986 (http://www.ietf.org/rfc/rfc3986.txt).

In short, a URL *is* a URI. In other words, URL is a subset of a URI - all acceptable URLs are also acceptable URIs.

wxURI automatically escapes invalid characters in a string, so there is no chance of wxURI "failing" on construction/creation.

wxURI supports copy construction and standard assignment operators. wxURI can also be inherited from to provide furthur functionality.

Derived from

wxObject (p. 1012)

Include files

Obtaining individual components

To obtain individual components you can use one of the following methods

However, you should check HasXXX before calling a get method, which determines whether or not the component referred to by the method is defined according to RFC 2396.

Consider an undefined component equivalent to a NULL C string.

```
Example://protocol will hold the http protocol (i.e. "http")
wxString protocol;
wxURI myuri(wxT("http://mysite.com"));
if(myuri.HasScheme())
   protocol = myuri.GetScheme();
```

wxURI::wxURI

wxURI()

Creates an empty URI.

wxURI(**const wxChar*** *uri*)

Constructor for quick creation.

uri

> string to initialize with

wxURI(**const wxURI&** *uri*)

Copies this URI from another URI.

uri

> URI (Uniform Resource Identifier) to initialize with

wxURI::BuildURI

wxString BuildURI() **const**

Builds the URI from its individual components and adds proper seperators.

If the URI is not a reference or is not resolved, the URI that is returned from Get is the same one passed to Create.

wxURI::BuildUnescapedURI

wxString BuildUnescapedURI() **const**

Builds the URI from its individual components, adds proper seperators, and returns escape sequences to normal characters.

Note that it is preferred to call this over Unescape(BuildURI()) since*BuildUnescapedURI* (p. 1454) performs some optimizations over the plain method.

wxURI::Create

const wxChar* Create(**const wxString&***uri*)

Creates this URI from the string *uri*.

Returns the position at which parsing stopped (there is no such thing as an "invalid" wxURI).

uri

> string to initialize from

wxURI::GetFragment

const wxString& GetFragment() **const**

Obtains the fragment of this URI.

The fragment of a URI is the last value of the URI, and is the value after a " character after the path of the URI.

```
http://mysite.com/mypath#<fragment>
```

wxURI::GetHostType

const HostType& GetHostType() **const**

Obtains the host type of this URI, which is of type wxURI::HostType:

wxURI_REGNAME	Server is a host name, or the Server component itself is undefined.
wxURI_IPV4ADDRESS	Server is a IP version 4 address (XXX.XXX.XXX.XXX)
wxURI_IPV6ADDRESS	Server is a IP version 6 address ((XXX:)XXX::(XXX)XXX:XXX
wxURI_IPVFUTURE	Server is an IP address, but not versions 4 or 6

wxURI::GetPassword

const wxString& GetPassword() const

Returns the password part of the userinfo component of this URI. Note that this is explicitly depreciated by RFC 1396 and should generally be avoided if possible.

```
http://<user>:<password>@mysite.com/mypath
```

wxURI::GetPath

const wxString& GetPath() const

Returns the (normalized) path of the URI.

The path component of a URI comes directly after the scheme component if followed by zero or one slashes ('/'), or after the server/port component.

Absolute paths include the leading '/' character.

```
http://mysite.com<path>
```

wxURI::GetPort

const wxString& GetPort() const

Returns a string representation of the URI's port.

The Port of a URI is a value after the server, and must come after a colon (:).

```
http://mysite.com:<port>
```

Note that you can easily get the numeric value of the port by using wxAtoi or wxString::Format.

wxURI::GetQuery

const wxString& GetQuery() const

Returns the Query component of the URI.

The query component is what is commonly passed to a cgi application, and must come after the path component, and after a '?' character.

```
http://mysite.com/mypath?<query>
```

wxURI::GetScheme

const wxString& GetScheme() **const**

Returns the Scheme component of the URI.

The first part of the uri.

```
<scheme>://mysite.com
```

wxURI::GetServer

const wxString& GetServer() **const**

Returns the Server component of the URI.

The server of the uri can be a server name or a type of ip address. See*GetHostType* (p. 1454) for the possible values for the host type of the server component.

```
http://<server>/mypath
```

wxURI::GetUser

const wxString& GetUser() **const**

Returns the username part of the userinfo component of this URI. Note that this is explicitly depreciated by RFC 1396 and should generally be avoided if possible.

```
http://<user>:<password>@mysite.com/mypath
```

wxURI::GetUserInfo

const wxString& GetUserInfo() **const**

Returns the UserInfo component of the URI.

The component of a URI before the server component that is postfixed by a '@' character.

```
http://<userinfo>@mysite.com/mypath
```

wxURI::HasFragment

bool HasFragment() **const**

Returns `true` if the Fragment component of the URI exists.

wxURI::HasPath

bool HasPath() **const**

Returns `true` if the Path component of the URI exists.

wxURI::HasPort

bool HasPort() **const**

Returns `true` if the Port component of the URI exists.

wxURI::HasQuery

bool HasQuery() **const**

Returns `true` if the Query component of the URI exists.

wxURI::HasScheme

bool HasScheme() **const**

Returns `true` if the Scheme component of the URI exists.

wxURI::HasServer

bool HasServer() **const**

Returns `true` if the Server component of the URI exists.

wxURI::HasUser

bool HasUser() **const**

Returns `true` if the User component of the URI exists.

wxURI::IsReference

bool IsReference() **const**

Returns `true` if a valid [absolute] URI, otherwise this URI is a URI reference and not a full URI, and IsReference returns `false`.

wxURI::operator ==

void operator ==(**const wxURI&** *uricomp*)

Compares this URI to another URI, and returns `true` if this URI equals *uricomp*, otherwise it returns `false`.

uricomp

> URI to compare to

wxURI::Resolve

void Resolve(**const wxURI&** *base*, **int** *flags* = `wxURI_STRICT`)

Inherits this URI from a base URI - components that do not exist in this URI are copied from the base, and if this URI's path is not an absolute path (prefixed by a '/'), then this URI's path is merged with the base's path.

For instance, resolving "../mydir" from "http://mysite.com/john/doe" results in the scheme (http) and server (mysite.com) being copied into this URI, since they do not exist. In addition, since the path of this URI is not absolute (does not begin with '/'), the path of the base's is merged with this URI's path, resulting in the URI "http://mysite.com/john/mydir".

base

> Base URI to inherit from. Must be a full URI and not a reference

flags

> Currently either `wxURI_STRICT` or 0, in non strict mode some compatibility layers are enabled to allow loopholes from RFCs prior to 2396

wxURI::Unescape

wxString Unescape(**const wxString&** *uri*)

Translates all escape sequences (normal characters and returns the result.

This is the preferred over deprecated wxURL::ConvertFromURI.

If you want to unescape an entire wxURI, use *BuildUnescapedURI* (p. 1454) instead, as it performs some optimizations over this method.

uri

>   string with escaped characters to convert

wxURL

Parses URLs.

Supports standard assignment operators, copy constructors, and comparison operators.

Derived from

wxURI (p. 1452)

Include files

<wx/url.h>

See also

wxSocketBase (p. 1203), *wxProtocol* (p. 1083)

wxURL::wxURL

wxURL(**const wxString&** *url*)

Constructs a URL object from the string. The URL must be valid according to RFC 1738. In particular, file URLs must be of the format 'file://hostname/path/to/file'. It is valid to leave out the hostname but slashes must remain in place-- i.e. a file URL without a hostname must contain three consecutive slashes.

Parameters

url

>   Url string to parse.

wxURL::~wxURL

~wxURL()

Destroys the URL object.

wxURL::GetProtocol

wxProtocol& GetProtocol()

Returns a reference to the protocol which will be used to get the URL.

wxURL::GetError

wxURLError GetError() **const**

Returns the last error. This error refers to the URL parsing or to the protocol. It can be one of these errors:

wxURL_NOERR   No error.

wxURL_SNTXERR   Syntax error in the URL string.

wxURL_NOPROTO   Found no protocol which can get this URL.

wxURL_NOHOST   An host name is required for this protocol.

wxURL_NOPATH   A path is required for this protocol.

wxURL_CONNERR   Connection error.

wxURL_PROTOERR   An error occurred during negotiation.

wxURL::GetInputStream

wxInputStream * GetInputStream()

Creates a new input stream on the specified URL. You can use all but seek functionality of wxStream. Seek isn't available on all stream. For example, http or ftp streams doesn't deal with it.

Note that this method is somewhat depreciated, all future wxWidgets applications should really use *wxFileSystem* (p. 540) instead.

Example:

```
wxURL url("http://a.host/a.dir/a.file");
if (url.GetError() == wxURL_NOERR)
{
    wxInputStream *in_stream;

    in_stream = url.GetInputStream();
    // Then, you can use all IO calls of in_stream (See wxStream)
}
```

Return value

Returns the initialized stream. You will have to delete it yourself.

See also

wxInputStream (p. 814)

wxURL::SetDefaultProxy

static void SetDefaultProxy(**const wxString&** *url_proxy*)

Sets the default proxy server to use to get the URL. The string specifies the proxy like this: <hostname>:<port number>.

Parameters

url_proxy

> Specifies the proxy to use

See also

wxURL::SetProxy (p. 1461)

wxURL::SetProxy

void SetProxy(**const wxString&** *url_proxy*)

Sets the proxy to use for this URL.

See also

wxURL::SetDefaultProxy (p. 1461)

wxValidator

wxValidator is the base class for a family of validator classes that mediate between a class of control, and application data.

A validator has three major roles:

1. to transfer data from a C++ variable or own storage to and from a control;

2. to validate data in a control, and show an appropriate error message;

3. to filter events (such as keystrokes), thereby changing the behaviour of the associated control.

Validators can be plugged into controls dynamically.

To specify a default, 'null' validator, use the symbol **wxDefaultValidator**.

For more information, please see *Validator overview* (p. 1765).

wxPython note: If you wish to create a validator class in wxPython you should derive the class from `wxPyValidator` in order to get Python-aware capabilities for the various virtual methods.

Derived from

wxEvtHandler (p. 486)
wxObject (p. 1012)

Include files

<wx/validate.h>

See also

Validator overview (p. 1765), *wxTextValidator* (p. 1374), *wxGenericValidator* (p. 612),

wxValidator::wxValidator

wxValidator()

Constructor.

wxValidator::~wxValidator

~wxValidator()

Destructor.

wxValidator::Clone

virtual wxObject* Clone() **const**

All validator classes must implement the **Clone** function, which returns an identical copy of itself. This is because validators are passed to control constructors as references which must be copied. Unlike objects such as pens and brushes, it does not make sense to have a reference counting scheme to do this cloning, because all validators should have separate data.

This base function returns NULL.

wxValidator::GetWindow

wxWindow* GetWindow() **const**

Returns the window associated with the validator.

wxValidator::SetBellOnError

void SetBellOnError(**bool** *doIt = true*)

This functions switches on or turns off the error sound produced by the validators if an

invalid key is pressed.

wxValidator::SetWindow

void SetWindow(**wxWindow*** *window*)

Associates a window with the validator.

wxValidator::TransferFromWindow

virtual bool TransferToWindow()

This overridable function is called when the value in the window must be transferred to the validator. Return false if there is a problem.

wxValidator::TransferToWindow

virtual bool TransferToWindow()

This overridable function is called when the value associated with the validator must be transferred to the window. Return false if there is a problem.

wxValidator::Validate

virtual bool Validate(**wxWindow*** *parent*)

This overridable function is called when the value in the associated window must be validated. Return false if the value in the window is not valid; you may pop up an error dialog.

wxVariant

The **wxVariant** class represents a container for any type. A variant's value can be changed at run time, possibly to a different type of value.

As standard, wxVariant can store values of type bool, char, double, long, string, string list, time, date, void pointer, list of strings, and list of variants. However, an application can extend wxVariant's capabilities by deriving from the class *wxVariantData* (p. 1472) and using the wxVariantData form of the wxVariant constructor or assignment operator to assign this data to a variant. Actual values for user-defined types will need to be accessed via the wxVariantData object, unlike the case for basic data types where convenience functions such as GetLong can be used.

This class is useful for reducing the programming for certain tasks, such as an editor for different data types, or a remote procedure call protocol.

An optional name member is associated with a wxVariant. This might be used, for

example, in CORBA or OLE automation classes, where named parameters are required.

Derived from

wxObject (p. 1012)

Include files

<wx/variant.h>

See also

wxVariantData (p. 1472)

wxVariant::wxVariant

wxVariant()

Default constructor.

wxVariant(**const wxVariant&** *variant*)

Copy constructor.

wxVariant(**const char*** *value*, **const wxString&** *name* = *""*)

wxVariant(**const wxString&** *value*, **const wxString&** *name* = *""*)

Construction from a string value.

wxVariant(**char** *value*, **const wxString&** *name* = *""*)

Construction from a character value.

wxVariant(**long** *value*, **const wxString&** *name* = *""*)

Construction from an integer value. You may need to cast to (long) to avoid confusion with other constructors (such as the bool constructor).

wxVariant(**bool** *value*, **const wxString&** *name* = *""*)

Construction from a boolean value.

wxVariant(**double** *value*, **const wxString&** *name* = *""*)

Construction from a double-precision floating point value.

wxVariant(**const wxList&** *value*, **const wxString&** *name* = *""*)

Construction from a list of wxVariant objects. This constructor copies *value*, the application is still responsible for deleting *value* and its contents.

wxVariant(**void*** *value*, **const wxString&** *name* = *""*)

Construction from a void pointer.

   wxVariant(**wxObject*** *value*, **const wxString&** *name = ""*)

Construction from a wxObject pointer.

   wxVariant(**wxVariantData*** *data*, **const wxString&** *name = ""*)

Construction from user-defined data. The variant holds onto the *data* pointer.

   wxVariant(**wxDateTime&** *val*, **const wxString&** *name = ""*)

Construction from a *wxDateTime* (p. 257).

   wxVariant(**wxArrayString&** *val*, **const wxString&** *name = ""*)

Construction from an array of strings. This constructor copies *value* and its contents.

   wxVariant(**DATE_STRUCT*** *val*, **const wxString&** *name = ""*)

Construction from a odbc date value. Represented internally by a *wxDateTime* (p. 257) value.

   wxVariant(**TIME_STRUCT*** *val*, **const wxString&** *name = ""*)

Construction from a odbc time value. Represented internally by a *wxDateTime* (p. 257) value.

   wxVariant(**TIMESTAMP_STRUCT*** *val*, **const wxString&** *name = ""*)

Construction from a odbc timestamp value. Represented internally by a *wxDateTime* (p. 257) value.

wxVariant::~wxVariant

   ~wxVariant()

Destructor.

wxVariant::Append

   void Append(**const wxVariant&** *value*)

Appends a value to the list.

wxVariant::ClearList

   void ClearList()

Deletes the contents of the list.

wxVariant::Convert

bool Convert(**long*** *value*) **const**

bool Convert(**bool*** *value*) **const**

bool Convert(**double*** *value*) **const**

bool Convert(**wxString*** *value*) **const**

bool Convert(**char*** *value*) **const**

bool Convert(**wxDateTime*** *value*) **const**

Retrieves and converts the value of this variant to the type that *value* is.

wxVariant::GetCount

int GetCount() **const**

Returns the number of elements in the list.

wxVariant::Delete

bool Delete(**int** *item*)

Deletes the zero-based *item* from the list.

wxVariant::GetArrayString

wxArrayString GetArrayString() **const**

Returns the string array value.

wxVariant::GetBool

bool GetBool() **const**

Returns the boolean value.

wxVariant::GetChar

char GetChar() **const**

Returns the character value.

wxVariant::GetData

wxVariantData* GetData() **const**

Returns a pointer to the internal variant data.

wxVariant::GetDateTime

wxDateTime GetDateTime() **const**

Returns the date value.

wxVariant::GetDouble

double GetDouble() **const**

Returns the floating point value.

wxVariant::GetLong

long GetLong() **const**

Returns the integer value.

wxVariant::GetName

const wxString& GetName() **const**

Returns a constant reference to the variant name.

wxVariant::GetString

wxString GetString() **const**

Gets the string value.

wxVariant::GetType

wxString GetType() **const**

Returns the value type as a string. The built-in types are: bool, char, date, double, list, long, string, stringlist, time, void*.

If the variant is null, the value type returned is the string "null" (not the empty string).

wxVariant::GetVoidPtr

void* GetVoidPtr() **const**

Gets the void pointer value.

wxVariant::GetWxObjectPtr

void* GetWxObjectPtr() **const**

Gets the wxObject pointer value.

wxVariant::Insert

void Insert(**const wxVariant&** *value*)

Inserts a value at the front of the list.

wxVariant::IsNull

bool IsNull() **const**

Returns true if there is no data associated with this variant, false if there is data.

wxVariant::IsType

bool IsType(**const wxString&** *type*) **const**

Returns true if *type* matches the type of the variant, false otherwise.

wxVariant::IsValueKindOf

bool IsValueKindOf(**const wxClassInfo* type** *type*) **const**

Returns true if the data is derived from the class described by *type*, false otherwise.

wxVariant::MakeNull

void MakeNull()

Makes the variant null by deleting the internal data.

wxVariant::MakeString

wxString MakeString() **const**

Makes a string representation of the variant value (for any type).

wxVariant::Member

bool Member(**const wxVariant&** *value*) **const**

Returns true if *value* matches an element in the list.

wxVariant::NullList

void NullList()

Makes an empty list. This differs from a null variant which has no data; a null list is of type list, but the number of elements in the list is zero.

wxVariant::SetData

void SetData(**wxVariantData*** *data*)

Sets the internal variant data, deleting the existing data if there is any.

wxVariant::operator =

void operator =(**const wxVariant&** *value*)

void operator =(**wxVariantData*** *value*)

void operator =(**const wxString&** *value*)

void operator =(**const char*** *value*)

void operator =(**char** *value*)

void operator =(**const long** *value*)

void operator =(**const bool** *value*)

void operator =(**const double** *value*)

void operator =(**void*** *value*)

void operator =(**const wxList&** *value*)

void operator =(**const wxDateTime&** *value*)

void operator =(**const wxArrayString&** *value*)

void operator =(**const DATE_STRUCT*** *value*)

void operator =(**const TIME_STRUCT*** *value*)

void operator =(const TIMESTAMP_STRUCT* *value*)

Assignment operators.

wxVariant::operator ==

bool operator ==(const wxVariant& *value*) **const**

bool operator ==(const wxString& *value*) **const**

bool operator ==(const char* *value*) **const**

bool operator ==(char *value*) **const**

bool operator ==(const long *value*) **const**

bool operator ==(const bool *value*) **const**

bool operator ==(const double *value*) **const**

bool operator ==(void* *value*) **const**

bool operator ==(const wxList& *value*) **const**

bool operator ==(const wxArrayString& *value*) **const**

bool operator ==(const wxDateTime& *value*) **const**

Equality test operators.

wxVariant::operator !=

bool operator !=(const wxVariant& *value*) **const**

bool operator !=(const wxString& *value*) **const**

bool operator !=(const char* *value*) **const**

bool operator !=(char *value*) **const**

bool operator !=(const long *value*) **const**

bool operator !=(const bool *value*) **const**

bool operator !=(const double *value*) **const**

bool operator !=(void* *value*) **const**

bool operator !=(const wxList& *value*) **const**

bool operator !=(const wxArrayString& *value*) **const**

bool operator !=(const wxDateTime& *value*) **const**

Inequality test operators.

wxVariant::operator []

wxVariant operator [](**size_t** *idx*) **const**

Returns the value at *idx* (zero-based).

wxVariant& operator [](**size_t** *idx*)

Returns a reference to the value at *idx* (zero-based). This can be used to change the value at this index.

wxVariant::operator char

char operator char() **const**

Operator for implicit conversion to a char, using *wxVariant::GetChar* (p. 1466).

wxVariant::operator double

double operator double() **const**

Operator for implicit conversion to a double, using *wxVariant::GetDouble* (p. 1467).

long operator long() **const**

Operator for implicit conversion to a long, using *wxVariant::GetLong* (p. 1467).

wxVariant::operator wxString

wxString operator wxString() **const**

Operator for implicit conversion to a string, using *wxVariant::MakeString* (p. 1468).

wxVariant::operator void*

void* operator void*() **const**

Operator for implicit conversion to a pointer to a void, using *wxVariant::GetVoidPtr* (p. 1468).

wxVariant::operator wxDateTime

void* operator wxDateTime() **const**

Operator for implicit conversion to a pointer to a *wxDateTime* (p. 257), using *wxVariant::GetDateTime* (p. 1467).

wxVariantData

The **wxVariantData** is used to implement a new type for wxVariant. Derive from wxVariantData, and override the pure virtual functions.

Derived from

wxObject (p. 1012)

Include files

<wx/variant.h>

See also

wxVariant (p. 1463)

wxVariantData::wxVariantData

 wxVariantData()

Default constructor.

wxVariantData::Copy

void Copy(**wxVariantData&** *data*)

Copy the data from 'this' object to *data*.

wxVariantData::Eq

bool Eq(**wxVariantData&** *data*) **const**

Returns true if this object is equal to *data*.

wxVariantData::GetType

wxString GetType() **const**

Returns the string type of the data.

wxVariantData::GetValueClassInfo

wxClassInfo* GetValueClassInfo() const

If the data is a wxObject returns a pointer to the objects wxClassInfo structure, if the data isn't a wxObject the method returns NULL.

wxVariantData::Read

bool Read(ostream& *stream*)

bool Read(wxString& *string*)

Reads the data from *stream* or *string*.

wxVariantData::Write

bool Write(ostream& *stream*) const

bool Write(wxString& *string*) const

Writes the data to *stream* or *string*.

wxGetVariantCast

classname * wxGetVariantCast(wxVariant&, classname)

This macro returns the data stored in *variant* cast to the type *classname* * if the data is of this type (the check is done during the run-time) or NULL otherwise.

See also

RTTI overview (p. 1719)
wxDynamicCast (p. 1646)

wxView

The view class can be used to model the viewing and editing component of an application's file-based data. It is part of the document/view framework supported by wxWidgets, and cooperates with the *wxDocument* (p. 454), *wxDocTemplate* (p. 448) and *wxDocManager* (p. 435) classes.

Derived from

wxEvtHandler (p. 486)
wxObject (p. 1012)

Include files

<wx/docview.h>

See also

wxView overview (p. 1804), *wxDocument* (p. 454), *wxDocTemplate* (p. 448), *wxDocManager* (p. 435)

wxView::m_viewDocument

wxDocument* m_viewDocument

The document associated with this view. There may be more than one view per document, but there can never be more than one document for one view.

wxView::m_viewFrame

wxFrame* m_viewFrame

Frame associated with the view, if any.

wxView::m_viewTypeName

wxString m_viewTypeName

The view type name given to the wxDocTemplate constructor, copied to this variable when the view is created. Not currently used by the framework.

wxView::wxView

wxView()

Constructor. Define your own default constructor to initialize application-specific data.

wxView::~wxView

~wxView()

Destructor. Removes itself from the document's list of views.

wxView::Activate

virtual void Activate(bool *activate*)

Call this from your view frame's OnActivate member to tell the framework which view is

currently active. If your windowing system doesn't call OnActivate, you may need to call this function from OnMenuCommand or any place where you know the view must be active, and the framework will need to get the current view.

The prepackaged view frame wxDocChildFrame calls wxView::Activate from its OnActivate member and from its OnMenuCommand member.

This function calls wxView::OnActivateView.

wxView::Close

virtual bool Close(**bool** *deleteWindow = true*)

Closes the view by calling OnClose. If *deleteWindow* is true, this function should delete the window associated with the view.

wxView::GetDocument

wxDocument* GetDocument() **const**

Gets a pointer to the document associated with the view.

wxView::GetDocumentManager

wxDocManager* GetDocumentManager() **const**

Returns a pointer to the document manager instance associated with this view.

wxView::GetFrame

wxWindow * GetFrame()

Gets the frame associated with the view (if any). Note that this "frame" is not a wxFrame at all in the generic MDI implementation which uses the notebook pages instead of the frames and this is why this method returns a wxWindow and not a wxFrame.

wxView::GetViewName

wxString GetViewName() **const**

Gets the name associated with the view (passed to the wxDocTemplate constructor). Not currently used by the framework.

wxView::OnActivateView

virtual void OnActivateView(**bool** *activate*, **wxView ****activeView*, **wxView**

*deactiveView)

Called when a view is activated by means of wxView::Activate. The default implementation does nothing.

wxView::OnChangeFilename

virtual void OnChangeFilename()

Called when the filename has changed. The default implementation constructs a suitable title and sets the title of the view frame (if any).

wxView::OnClose

virtual bool OnClose(**bool** *deleteWindow*)

Implements closing behaviour. The default implementation calls wxDocument::Close to close the associated document. Does not delete the view. The application may wish to do some cleaning up operations in this function, *if* a call to wxDocument::Close succeeded. For example, if your views all share the same window, you need to disassociate the window from the view and perhaps clear the window. If *deleteWindow* is true, delete the frame associated with the view.

wxView::OnClosingDocument

virtual void OnClosingDoocument()

Override this to clean up the view when the document is being closed.

wxView::OnCreate

virtual bool OnCreate(**wxDocument*** *doc*, **long** *flags*)

wxDocManager or wxDocument creates a wxView via a wxDocTemplate. Just after the wxDocTemplate creates the wxView, it calls wxView::OnCreate. In its OnCreate member function, the wxView can create a wxDocChildFrame or a derived class. This wxDocChildFrame provides user interface elements to view and/or edit the contents of the wxDocument.

By default, simply returns true. If the function returns false, the view will be deleted.

wxView::OnCreatePrintout

virtual wxPrintout* OnCreatePrintout()

If the printing framework is enabled in the library, this function returns a *wxPrintout* (p. 1065) object for the purposes of printing. It should create a new object every time it is

called; the framework will delete objects it creates.

By default, this function returns an instance of wxDocPrintout, which prints and previews one page by calling wxView::OnDraw.

Override to return an instance of a class other than wxDocPrintout.

wxView::OnDraw

virtual void OnDraw(**wxDC&** *dc*)

Override this function to render the view on the given device context.

wxView::OnUpdate

virtual void OnUpdate(**wxView*** *sender*, **wxObject*** *hint*)

Called when the view should be updated. *sender* is a pointer to the view that sent the update request, or NULL if no single view requested the update (for instance, when the document is opened). *hint* is as yet unused but may in future contain application-specific information for making updating more efficient.

wxView::SetDocument

void SetDocument(**wxDocument*** *doc*)

Associates the given document with the view. Normally called by the framework.

wxView::SetFrame

void SetFrame(**wxWindow*** *frame*)

Sets the frame associated with this view. The application should call this if possible, to tell the view about the frame.

See *GetFrame* (p. 1475) for the explanation about the mismatch between the "Frame" in the method name and the type of its parameter.

wxView::SetViewName

void SetViewName(**const wxString&** *name*)

Sets the view type name. Should only be called by the framework.

wxVListBox

wxVListBox is a listbox-like control with the following two main differences from a regular listbox: it can have an arbitrarily huge number of items because it doesn't store them itself but uses *OnDrawItem()* (p. 1481) callback to draw them (so it is a Virtual listbox) and its items can have variable height as determined by *OnMeasureItem()* (p. 1482) (so it is also a listbox with the lines of Variable height).

Also, as a consequence of its virtual nature, it doesn't have any methods to append or insert items in it as it isn't necessary to do it: you just have to call *SetItemCount()* (p. 1483) to tell the control how many items it should display. Of course, this also means that you will never use this class directly because it has pure virtual functions, but will need to derive your own class, such as *wxHtmlListBox* (p. 734), from it.

However it emits the same events as *wxListBox* (p. 847) and the same event macros may be used with it. **Derived from**

wxVScrolledWindow (p. 1484)

Include files

<wx/vlbox.h>

wxVListBox::wxVListBox

 wxVListBox(**wxWindow*** *parent*, **wxWindowID** *id = wxID_ANY*, **const wxPoint&** *pos = wxDefaultPosition*, **const wxSize&** *size = wxDefaultSize*, **size_t** *countItems = 0*, **long** *style = 0*, **const wxString&** *name = wxVListBoxNameStr*)

Normal constructor which calls *Create()* (p. 1478) internally.

 wxVListBox()

Default constructor, you must call *Create()* (p. 1478) later.

wxVListBox::Clear

 void Clear()

Deletes all items from the control.

wxVListBox::Create

bool Create(**wxWindow*** *parent*, **wxWindowID** *id = wxID_ANY*, **const wxPoint&** *pos = wxDefaultPosition*, **const wxSize&** *size = wxDefaultSize*, **long** *style = 0*, **const wxString&** *name = wxVListBoxNameStr*)

Creates the control. To finish creating it you also should call *SetItemCount()* (p. 1483) to let it know about the number of items it contains.

The only special style which may be used with wxVListBox is `wxLB_MULTIPLE` which indicates that the listbox should support multiple selection.

Returns `true` on success or `false` if the control couldn't be created

wxVListBox::DeselectAll

bool DeselectAll()

Deselects all the items in the listbox.

Returns `true` if any items were changed, i.e. if there had been any selected items before, or `false` if all the items were already deselected.

This method is only valid for multi selection listboxes.

See also

SelectAll (p. 1483), *Select* (p. 1482)

wxVListBox::GetFirstSelected

int GetFirstSelected(**unsigned long&** *cookie*) **const**

Returns the index of the first selected item in the listbox or `wxNOT_FOUND` if no items are currently selected.

cookie is an opaque parameter which should be passed to the subsequent calls to *GetNextSelected* (p. 1480). It is needed in order to allow parallel iterations over the selected items.

```
Here is a typical example of using these functions:unsigned long cookie;
int item = hlbox->GetFirstSelected(cookie);
while ( item != wxNOT_FOUND )
{
    ... process item ...
    item = hlbox->GetNextSelected(cookie);
}
```

This method is only valid for multi selection listboxes.

wxVListBox::GetItemCount

size_t GetItemCount() **const**

Get the number of items in the control.

See also

SetItemCount() (p. 1483)

wxVListBox::GetMargins

wxPoint GetMargins() **const**

Returns the margins used by the control. The x field of the returned point is the horizontal margin and the y field is the vertical one.

See also

SetMargins (p. 1483)

wxVListBox::GetNextSelected

int GetNextSelected(**unsigned long&** *cookie*) **const**

Returns the index of the next selected item or wxNOT_FOUND if there are no more.

This method is only valid for multi selection listboxes.

See also

GetFirstSelected (p. 1479)

wxVListBox::GetSelectedCount

size_t GetSelectedCount() **const**

Returns the number of the items currently selected.

It is valid for both single and multi selection controls. In the former case it may only return 0 or 1 however.

See also

IsSelected (p. 1481),
GetFirstSelected (p. 1479),
GetNextSelected (p. 1480)

wxVListBox::GetSelection

int GetSelection() **const**

Get the currently selected item or -1 if there is no selection.

wxVListBox::GetSelectionBackground

const wxColour& GetSelectionBackground() **const**

Returns the background colour used for the selected cells. By default the standard

system colour is used.

See also

wxSystemSettings::GetColour (p. 1319),
SetSelectionBackground (p. 1484)

wxVListBox::HasMultipleSelection

bool HasMultipleSelection() **const**

Returns `true` if the listbox was created with `wxLB_MULTIPLE` style and so supports multiple selection or `false` if it is a single selection listbox.

wxVListBox::IsCurrent

bool IsCurrent(**size_t** *item*) **const**

Returns `true` if this item is the current one, `false` otherwise.

Current item is always the same as selected one for the single selection listbox and in this case this method is equivalent to *IsSelected* (p. 1481) but they are different for multi selection listboxes where many items may be selected but only one (at most) is current.

wxVListBox::IsSelected

bool IsSelected(**size_t** *item*) **const**

Returns `true` if this item is selected, `false` otherwise.

wxVListBox::OnDrawBackground

void OnDrawBackground(**wxDC&** *dc*, **const wxRect&** *rect*, **size_t** *n*) **const**

This method is used to draw the items background and, maybe, a border around it.

The base class version implements a reasonable default behaviour which consists in drawing the selected item with the standard background colour and drawing a border around the item if it is either selected or current.

wxVListBox::OnDrawItem

void OnDrawItem(**wxDC&** *dc*, **const wxRect&** *rect*, **size_t** *n*) **const**

The derived class must implement this function to actually draw the item with the given index on the provided DC.

Parameters

dc

The device context to use for drawing

rect

The bounding rectangle for the item being drawn (DC clipping region is set to this rectangle before calling this function)

n

The index of the item to be drawn

wxVListBox::OnDrawSeparator

void OnDrawSeparator(**wxDC&** *dc*, **wxRect&** *rect*, **size_t** *n*) **const**

This method may be used to draw separators between the lines. The rectangle passed to it may be modified, typically to deflate it a bit before passing to *OnDrawItem()* (p. 1481).

The base class version of this method doesn't do anything.

Parameters

dc

The device context to use for drawing

rect

The bounding rectangle for the item

n

The index of the item

wxVListBox::OnMeasureItem

wxCoord OnMeasureItem(**size_t** *n*) **const**

The derived class must implement this method to return the height of the specified item (in pixels).

wxVListBox::Select

bool Select(**size_t** *item*, **bool** *select* = *true*)

Selects or deselects the specified item which must be valid (i.e. not equal to

`wxNOT_FOUND`).

Return `true` if the items selection status has changed or `false` otherwise.

This function is only valid for the multiple selection listboxes, use *SetSelection* (p. 1484) for the single selection ones.

wxVListBox::SelectAll

bool SelectAll()

Selects all the items in the listbox.

Returns `true` if any items were changed, i.e. if there had been any unselected items before, or `false` if all the items were already selected.

This method is only valid for multi selection listboxes.

See also

DeselectAll (p. 1479), *Select* (p. 1482)

wxVListBox::SelectRange

bool SelectRange(**size_t** *from*, **size_t** *to*)

Selects all items in the specified range which may be given in any order.

Return `true` if the items selection status has changed or `false` otherwise.

This method is only valid for multi selection listboxes.

See also

SelectAll (p. 1483), *Select* (p. 1482)

wxVListBox::SetItemCount

void SetItemCount(**size_t** *count*)

Set the number of items to be shown in the control.

This is just a synonym for *wxVScrolledWindow::SetLineCount()* (p. 1490).

wxVListBox::SetMargins

void SetMargins(**const wxPoint&** *pt*)

void SetMargins(**wxCoord** *x*, **wxCoord** *y*)

Set the margins: horizontal margin is the distance between the window border and the item contents while vertical margin is half of the distance between items.

By default both margins are 0.

wxVListBox::SetSelection

void SetSelection(**int** *selection*)

Set the selection to the specified item, if it is -1 the selection is unset. The selected item will be automatically scrolled into view if it isn't currently visible.

This method may be used both with single and multiple selection listboxes.

wxVListBox::SetSelectionBackground

void SetSelectionBackground(**const wxColour&** *col*)

Sets the colour to be used for the selected cells background. The background of the standard cells may be changed by simply calling *SetBackgroundColour* (p. 1528).

See also

GetSelectionBackground (p. 1480)

wxVListBox::Toggle

void Toggle(**size_t** *item*)

Toggles the state of the specified *item*, i.e. selects it if it was unselected and deselects it if it was selected.

This method is only valid for multi selection listboxes.

See also

Select (p. 1482)

wxVScrolledWindow

In the name of this class, "V" may stand for "variable" because it can be used for scrolling lines of variable heights; "virtual" because it is not necessary to know the heights of all lines in advance -- only those which are shown on the screen need to be measured; or, even, "vertical" because this class only supports scrolling in one direction currently (this could and probably will change in the future however).

In any case, this is a generalization of the *wxScrolledWindow* (p. 1150) class which can be only used when all lines have the same height. It lacks some other

wxScrolledWindow features however, notably there is currently no support for horizontal scrolling; it can't scroll another window nor only a rectangle of the window and not its entire client area. To use this class, you need to derive from it and implement *OnGetLineHeight()* (p. 1488) pure virtual method. You also must call *SetLineCount* (p. 1490) to let the base class know how many lines it should display but from that moment on the scrolling is handled entirely by wxVScrolledWindow, you only need to draw the visible part of contents in your `OnPaint()` method as usual. You should use *GetFirstVisibleLine()* (p. 1486) and *GetLastVisibleLine()* (p. 1486) to select the lines to display. Note that the device context origin is not shifted so the first visible line always appears at the point (0, 0) in physical as well as logical coordinates.

Derived from

wxPanel (p. 1030)

Include files

<wx/vscroll.h>

wxVScrolledWindow::wxVScrolledWindow

 wxVScrolledWindow(**wxWindow*** *parent*, **wxWindowID** *id* = *wxID_ANY*, **const wxPoint&** *pos* = *wxDefaultPosition*, **const wxSize&** *size* = *wxDefaultSize*, **long** *style* = *0*, **const wxString&** *name* = *wxPanelNameStr*)

This is the normal constructor, no need to call Create() after using this one.

Note that `wxVSCROLL` is always automatically added to our style, there is no need to specify it explicitly.

 wxVScrolledWindow()

Default constructor, you must call *Create()* (p. 1486) later.

Parameters

parent

> The parent window, must not be `NULL`

id

> The identifier of this window, `wxID_ANY` by default

pos

> The initial window position

size

> The initial window size

style

>   The window style. There are no special style bits defined for this class.

name

>   The name for this window; usually not used

wxVScrolledWindow::Create

bool Create(**wxWindow*** *parent*, **wxWindowID** *id = wxID_ANY*, **const wxPoint&** *pos = wxDefaultPosition*, **const wxSize&** *size = wxDefaultSize*, **long** *style = 0*, **const wxString&** *name = wxPanelNameStr*)

Same as the *non default ctor* (p. 1485) but returns status code: `true` if ok, `false` if the window couldn't have been created.

Just as with the ctor above, `wxVSCROLL` style is always used, there is no need to specify it explicitly.

wxVScrolledWindow::EstimateTotalHeight

virtual wxCoord EstimateTotalHeight() **const**

This protected function is used internally by wxVScrolledWindow to estimate the total height of the window when *SetLineCount* (p. 1490) is called. The default implementation uses the brute force approach if the number of the items in the control is small enough. Otherwise, it tries to find the average line height using some lines in the beginning, middle and the end.

If it is undesirable to access all these lines (some of which might be never shown) just for the total height calculation, you may override the function and provide your own guess better and/or faster.

Note that although returning a totally wrong value would still work, it risks to result in very strange scrollbar behaviour so this function should really try to make the best guess possible.

wxVScrolledWindow::GetFirstVisibleLine

size_t GetFirstVisibleLine() **const**

Returns the index of the first currently visible line.

This is same as *GetVisibleBegin* (p. 1487) and exists only for symmetry with *GetLastVisibleLine* (p. 1486).

wxVScrolledWindow::GetLastVisibleLine

size_t GetLastVisibleLine() const

Returns the index of the last currently visible line. Note that this method returns `(size_t)-1` (i.e. a huge positive number) if the control is empty so if this is possible you should use *GetVisibleEnd* (p. 1487) instead.

See also

GetFirstVisibleLine (p. 1486)

wxVScrolledWindow::GetLineCount

size_t GetLineCount() const

Get the number of lines this window contains (previously set by *SetLineCount()* (p. 1490))

wxVScrolledWindow::GetVisibleBegin

size_t GetVisibleBegin() const

Returns the index of the first currently visible line.

See also

GetVisibleEnd (p. 1487)

wxVScrolledWindow::GetVisibleEnd

size_t GetVisibleEnd() const

Returns the index of the first line after the currently visible one. If the return value is 0 it means that no lines are currently shown (which only happens if the control is empty). Note that the index returned by this method is not always a valid index as it may be equal to *GetLineCount* (p. 1490).

See also

GetVisibleBegin (p. 1487)

wxVScrolledWindow::HitTest

int HitTest(wxCoord *x*, wxCoord *y*) const

int HitTest(const wxPoint& *pt*) const

Return the item at the specified (in physical coordinates) position or `wxNOT_FOUND` if none, i.e. if it is below the last item.

wxVScrolledWindow::IsVisible

bool IsVisible(**size_t** *line*) **const**

Returns `true` if the given line is (at least partially) visible or `false` otherwise.

wxVScrolledWindow::OnGetLineHeight

wxCoord OnGetLineHeight(**size_t** *n*) **const**

This protected virtual function must be overridden in the derived class and it should return the height of the given line in pixels.

See also

OnGetLinesHint (p. 1488)

wxVScrolledWindow::OnGetLinesHint

void OnGetLinesHint(**size_t** *lineMin*, **size_t** *lineMax*) **const**

This function doesn't have to be overridden but it may be useful to do it if calculating the lines heights is a relatively expensive operation as it gives the user code a possibility to calculate several of them at once.

`OnGetLinesHint()` is normally called just before *OnGetLineHeight()* (p. 1488) but you shouldn't rely on the latter being called for all lines in the interval specified here. It is also possible that OnGetLineHeight() will be called for the lines outside of this interval, so this is really just a hint, not a promise.

Finally note that *lineMin* is inclusive, while *lineMax* is exclusive, as usual.

wxVScrolledWindow::RefreshLine

void RefreshLine(**size_t** *line*)

Refreshes the specified line -- it will be redrawn during the next main loop iteration.

See also

RefreshLines (p. 1488)

wxVScrolledWindow::RefreshLines

void RefreshLines(**size_t** *from*, **size_t** *to*)

Refreshes all lines between *from* and *to*, inclusive. *from*should be less than or equal to *to*.

RefreshLine (p. 1488)

wxVScrolledWindow::RefreshAll

void RefreshAll()

This function completely refreshes the control, recalculating the number of items shown on screen and repainting them. It should be called when the values returned by *OnGetLineHeight* (p. 1488) change for some reason and the window must be updated to reflect this.

wxVScrolledWindow::ScrollLines

bool ScrollLines(**int** *lines*)

Scroll by the specified number of lines which may be positive (to scroll down) or negative (to scroll up).

Returns `true` if the window was scrolled, `false` otherwise (for example if we're trying to scroll down but we are already showing the last line).

See also

LineUp (p. 1518), *LineDown* (p. 1518)

wxVScrolledWindow::ScrollPages

bool ScrollPages(**int** *pages*)

Scroll by the specified number of pages which may be positive (to scroll down) or negative (to scroll up).

See also

ScrollLines (p. 1489),
PageUp (p. 1520), *PageDown* (p. 1520)

wxVScrolledWindow::ScrollToLine

bool ScrollToLine(**size_t** *line*)

Scroll to the specified line: it will become the first visible line in the window.

Return `true` if we scrolled the window, `false` if nothing was done.

wxVScrolledWindow::SetLineCount

void SetLineCount(**size_t** *count*)

Set the number of lines the window contains: the derived class must provide the heights for all lines with indices up to the one given here in its *OnGetLineHeight()* (p. 1488).

wxWindow

wxWindow is the base class for all windows and represents any visible object on screen. All controls, top level windows and so on are windows. Sizers and device contexts are not, however, as they don't appear on screen themselves.

Please note that all children of the window will be deleted automatically by the destructor before the window itself is deleted which means that you don't have to worry about deleting them manually. Please see the *window deletion overview* (p. 1763) for more information.

Also note that in this, and many others, wxWidgets classes some `GetXXX()` methods may be overloaded (as, for example, *GetSize* (p. 1512) or *GetClientSize* (p. 1506)). In this case, the overloads are non-virtual because having multiple virtual functions with the same name results in a virtual function name hiding at the derived class level (in English, this means that the derived class has to override all overloaded variants if it overrides any of them). To allow overriding them in the derived class, wxWidgets uses a unique protected virtual `DoGetXXX()` method and all `GetXXX()` ones are forwarded to it, so overriding the former changes the behaviour of the latter.

Derived from

wxEvtHandler (p. 486)
wxObject (p. 1012)

Include files

<wx/window.h>

Window styles

The following styles can apply to all windows, although they will not always make sense for a particular window class or on all platforms.

wxSIMPLE_BORDER	Displays a thin border around the window. wxBORDER is the old name for this style.
wxDOUBLE_BORDER	Displays a double border. Windows and Mac only.
wxSUNKEN_BORDER	Displays a sunken border.
wxRAISED_BORDER	Displays a raised border.
wxSTATIC_BORDER	Displays a border suitable for a static control. Windows only.

wxNO_BORDER   Displays no border, overriding the default border style for the window.

wxTRANSPARENT_WINDOW   The window is transparent, that is, it will not receive paint events. Windows only.

wxTAB_TRAVERSAL   Use this to enable tab traversal for non-dialog windows.

wxWANTS_CHARS   Use this to indicate that the window wants to get all char/key events for all keys - even for keys like TAB or ENTER which are usually used for dialog navigation and which wouldn't be generated without this style. If you need to use this style in order to get the arrows or etc., but would still like to have normal keyboard navigation take place, you should create and send a wxNavigationKeyEvent in response to the key events for Tab and Shift-Tab.

wxNO_FULL_REPAINT_ON_RESIZE   Disables repainting the window completely when its size is changed - you will have to repaint the new window area manually if you use this style. Currently only has an effect for Windows.

wxVSCROLL   Use this style to enable a vertical scrollbar.

wxHSCROLL   Use this style to enable a horizontal scrollbar.

wxALWAYS_SHOW_SB   If a window has scrollbars, disable them instead of hiding them when they are not needed (i.e. when the size of the window is big enough to not require the scrollbars to navigate it). This style is currently only implemented for wxMSW and wxUniversal and does nothing on the other platforms.

wxCLIP_CHILDREN   Use this style to eliminate flicker caused by the background being repainted, then children being painted over them. Windows only.

wxFULL_REPAINT_ON_RESIZE   Use this style to force a complete redraw of the window whenever it is resized instead of redrawing just the part of the window affected by resizing. Note that this was the behaviour by default before 2.5.1 release and that if you experience redraw problems with the code which previously used to work you may want to try this.

See also *window styles overview* (p. 1762).

Extra window styles

The following are extra styles, set using *wxWindow::SetExtraStyle* (p. 1533).

wxWS_EX_VALIDATE_RECURSIVELY   By default, Validate/TransferDataTo/FromWindow() only work on direct children of the window (compatible behaviour). Set

this flag to make them recursively descend into all subwindows.

wxWS_EX_BLOCK_EVENTS   wxCommandEvents and the objects of the derived classes are forwarded to the parent window and so on recursively by default. Using this flag for the given window allows to block this propagation at this window, i.e. prevent the events from being propagated further upwards. Dialogs have this flag on by default.

wxWS_EX_TRANSIENT   Don't use this window as an implicit parent for the other windows: this must be used with transient windows as otherwise there is the risk of creating a dialog/frame with this window as a parent which would lead to a crash if the parent is destroyed before the child.

wxWS_EX_PROCESS_IDLE   This window should always process idle events, even if the mode set by *wxIdleEvent::SetMode* (p. 778) is wxIDLE_PROCESS_SPECIFIED.

wxWS_EX_PROCESS_UI_UPDATES   This window should always process UI update events, even if the mode set by *wxUpdateUIEvent::SetMode* (p. 1451) is wxUPDATE_UI_PROCESS_SPECIFIED.

See also

Event handling overview (p. 1751)

wxWindow::wxWindow

wxWindow()

Default constructor.

wxWindow(**wxWindow*** *parent*, **wxWindowID** *id*, **const wxPoint&** *pos* = *wxDefaultPosition*, **const wxSize&** *size = wxDefaultSize*, **long** *style = 0*, **const wxString&** *name = wxPanelNameStr*)

Constructs a window, which can be a child of a frame, dialog or any other non-control window.

Parameters

parent

Pointer to a parent window.

id

Window identifier. If -1, will automatically create an identifier.

pos

> Window position. wxDefaultPosition is (-1, -1) which indicates that wxWidgets should generate a default position for the window. If using the wxWindow class directly, supply an actual position.

size

> Window size. wxDefaultSize is (-1, -1) which indicates that wxWidgets should generate a default size for the window. If no suitable size can be found, the window will be sized to 20x20 pixels so that the window is visible but obviously not correctly sized.

style

> Window style. For generic window styles, please see *wxWindow* (p. 1490).

name

> Window name.

wxWindow::~wxWindow

~wxWindow()

Destructor. Deletes all subwindows, then deletes itself. Instead of using the **delete** operator explicitly, you should normally use *wxWindow::Destroy* (p. 1499) so that wxWidgets can delete a window only when it is safe to do so, in idle time.

See also

Window deletion overview (p. 1763), *wxWindow::Destroy* (p. 1499), *wxCloseEvent* (p. 153)

wxWindow::AddChild

virtual void AddChild(**wxWindow*** *child*)

Adds a child window. This is called automatically by window creation functions so should not be required by the application programmer.

Notice that this function is mostly internal to wxWidgets and shouldn't be called by the user code.

Parameters

child

> Child window to add.

wxWindow::CacheBestSize

void CacheBestSize(**const wxSize&** *size*) **const**

Sets the cached best size value.

wxWindow::CaptureMouse

virtual void CaptureMouse()

Directs all mouse input to this window. Call *wxWindow::ReleaseMouse* (p. 1524) to release the capture.

Note that wxWidgets maintains the stack of windows having captured the mouse and when the mouse is released the capture returns to the window which had had captured it previously and it is only really released if there were no previous window. In particular, this means that you must release the mouse as many times as you capture it.

See also

wxWindow::ReleaseMouse (p. 1524)

wxWindow::Center

void Center(**int** *direction*)

A synonym for *Centre* (p. 1494).

wxWindow::CenterOnParent

void CenterOnParent(**int** *direction*)

A synonym for *CentreOnParent* (p. 1495).

wxWindow::CenterOnScreen

void CenterOnScreen(**int** *direction*)

A synonym for *CentreOnScreen* (p. 1495).

wxWindow::Centre

void Centre(**int** *direction = wxBOTH*)

Centres the window.

Parameters

direction

> Specifies the direction for the centering. May be `wxHORIZONTAL`, `wxVERTICAL` or `wxBOTH`. It may also include `wxCENTRE_ON_SCREEN` flag if you want to center the window on the entire screen and not on its parent window.

The flag `wxCENTRE_FRAME` is obsolete and should not be used any longer (it has no effect).

Remarks

If the window is a top level one (i.e. doesn't have a parent), it will be centered relative to the screen anyhow.

See also

wxWindow::Center (p. 1494)

wxWindow::CentreOnParent

void CentreOnParent(**int** *direction* = *wxBOTH*)

Centres the window on its parent. This is a more readable synonym for*Centre* (p. 1494).

Parameters

direction

> Specifies the direction for the centering. May be `wxHORIZONTAL`, `wxVERTICAL` or `wxBOTH`.

Remarks

This methods provides for a way to center top level windows over their parents instead of the entire screen. If there is no parent or if the window is not a top level window, then behaviour is the same as*wxWindow::Centre* (p. 1494).

See also

wxWindow::CentreOnScreen (p. 1494)

wxWindow::CentreOnScreen

void CentreOnScreen(**int** *direction* = *wxBOTH*)

Centres the window on screen. This only works for top level windows - otherwise, the window will still be centered on its parent.

Parameters

direction

Specifies the direction for the centering. May be `wxHORIZONTAL`, `wxVERTICAL` or `wxBOTH`.

See also

wxWindow::CentreOnParent (p. 1494)

wxWindow::ClearBackground

void ClearBackground()

Clears the window by filling it with the current background colour. Does not cause an erase background event to be generated.

wxWindow::ClientToScreen

virtual void ClientToScreen(**int*** *x*, **int*** *y*) **const**

wxPerl note: In wxPerl this method returns a 2-element list instead of modifying its parameters.

virtual wxPoint ClientToScreen(**const wxPoint&** *pt*) **const**

Converts to screen coordinates from coordinates relative to this window.

x

A pointer to a integer value for the x coordinate. Pass the client coordinate in, and a screen coordinate will be passed out.

y

A pointer to a integer value for the y coordinate. Pass the client coordinate in, and a screen coordinate will be passed out.

pt

The client position for the second form of the function.

wxPython note: In place of a single overloaded method name, wxPython implements the following methods:

ClientToScreen(point)	Accepts and returns a wxPoint
ClientToScreenXY(x, y)	Returns a 2-tuple, (x, y)

wxWindow::Close

bool Close(**bool** *force* = `false`)

This function simply generates a *wxCloseEvent* (p. 153) whose handler usually tries to close the window. It doesn't close the window itself, however.

Parameters

force

> `false` if the window's close handler should be able to veto the destruction of this window, `true` if it cannot.

Remarks

Close calls the *close handler* (p. 153) for the window, providing an opportunity for the window to choose whether to destroy the window. Usually it is only used with the top level windows (wxFrame and wxDialog classes) as the others are not supposed to have any special OnClose() logic.

The close handler should check whether the window is being deleted forcibly, using *wxCloseEvent::GetForce* (p. 154), in which case it should destroy the window using *wxWindow::Destroy* (p. 1499).

Note that calling Close does not guarantee that the window will be destroyed; but it provides a way to simulate a manual close of a window, which may or may not be implemented by destroying the window. The default implementation of wxDialog::OnCloseWindow does not necessarily delete the dialog, since it will simply simulate an wxID_CANCEL event which is handled by the appropriate button event handler and may do anything at all.

To guarantee that the window will be destroyed, call*wxWindow::Destroy* (p. 1499) instead

See also

Window deletion overview (p. 1763), *wxWindow::Destroy* (p. 1499), *wxCloseEvent* (p. 153)

wxWindow::ConvertDialogToPixels

wxPoint ConvertDialogToPixels(**const wxPoint&** *pt*)

wxSize ConvertDialogToPixels(**const wxSize&** *sz*)

Converts a point or size from dialog units to pixels.

For the x dimension, the dialog units are multiplied by the average character width and then divided by 4.

For the y dimension, the dialog units are multiplied by the average character height and then divided by 8.

Remarks

Dialog units are used for maintaining a dialog's proportions even if the font changes.

You can also use these functions programmatically. A convenience macro is defined:

```
#define wxDLG_UNIT(parent, pt) parent->ConvertDialogToPixels(pt)
```

See also

wxWindow::ConvertPixelsToDialog (p. 1498)

wxPython note: In place of a single overloaded method name, wxPython implements the following methods:

      ConvertDialogPointToPixels(point)   Accepts and returns a wxPoint

      ConvertDialogSizeToPixels(size)   Accepts and returns a wxSize

Additionally, the following helper functions are defined:

      wxDLG_PNT(win, point)   Converts a wxPoint from dialog units to pixels

      wxDLG_SZE(win, size)   Converts a wxSize from dialog units to pixels

wxWindow::ConvertPixelsToDialog

wxPoint ConvertPixelsToDialog(**const wxPoint&** *pt*)

wxSize ConvertPixelsToDialog(**const wxSize&** *sz*)

Converts a point or size from pixels to dialog units.

For the x dimension, the pixels are multiplied by 4 and then divided by the average character width.

For the y dimension, the pixels are multiplied by 8 and then divided by the average character height.

Remarks

Dialog units are used for maintaining a dialog's proportions even if the font changes.

See also

wxWindow::ConvertDialogToPixels (p. 1497)

wxPython note: In place of a single overloaded method name, wxPython implements the following methods:

      ConvertDialogPointToPixels(point)   Accepts and returns a wxPoint

      ConvertDialogSizeToPixels(size)   Accepts and returns a wxSize

wxWindow::Destroy

virtual bool Destroy()

Destroys the window safely. Use this function instead of the delete operator, since different window classes can be destroyed differently. Frames and dialogs are not destroyed immediately when this function is called -- they are added to a list of windows to be deleted on idle time, when all the window's events have been processed. This prevents problems with events being sent to non-existent windows.

Return value

`true` if the window has either been successfully deleted, or it has been added to the list of windows pending real deletion.

wxWindow::DestroyChildren

virtual void DestroyChildren()

Destroys all children of a window. Called automatically by the destructor.

wxWindow::Disable

bool Disable()

Disables the window, same as *Enable(`false`)* (p. 1500).

Return value

Returns `true` if the window has been disabled, `false` if it had been already disabled before the call to this function.

wxWindow::DoGetBestSize

virtual wxSize DoGetBestSize() **const**

Gets the size which best suits the window: for a control, it would be the minimal size which doesn't truncate the control, for a panel - the same size as it would have after a call to *Fit()* (p. 1502).

wxWindow::DoUpdateWindowUI

virtual void DoUpdateWindowUI(**wxUpdateUIEvent&** *event*)

Does the window-specific updating after processing the update event. This function is called by *wxWindow::UpdateWindowUI* (p. 1546)in order to check return values in the

wxUpdateUIEvent (p. 1448) and act appropriately. For example, to allow frame and dialog title updating, wxWidgets implements this function as follows:

```
// do the window-specific processing after processing the update event
void wxTopLevelWindowBase::DoUpdateWindowUI(wxUpdateUIEvent& event)
{
    if ( event.GetSetEnabled() )
        Enable(event.GetEnabled());

    if ( event.GetSetText() )
    {
        if ( event.GetText() != GetTitle() )
            SetTitle(event.GetText());
    }
}
```

wxWindow::DragAcceptFiles

virtual void DragAcceptFiles(**bool** *accept*)

Enables or disables eligibility for drop file events (OnDropFiles).

Parameters

accept

If `true`, the window is eligible for drop file events. If `false`, the window will not accept drop file events.

Remarks

Windows only.

wxWindow::Enable

virtual bool Enable(**bool** *enable* = `true`)

Enable or disable the window for user input. Note that when a parent window is disabled, all of its children are disabled as well and they are reenabled again when the parent is.

Parameters

enable

If `true`, enables the window for input. If `false`, disables the window.

Return value

Returns `true` if the window has been enabled or disabled, `false` if nothing was done, i.e. if the window had already been in the specified state.

See also

wxWindow::IsEnabled (p. 1516), *wxWindow::Disable* (p. 1499), *wxRadioBox::Enable* (p. 1092)

wxWindow::FindFocus

static wxWindow* FindFocus()

Finds the window or control which currently has the keyboard focus.

Remarks

Note that this is a static function, so it can be called without needing a wxWindow pointer.

See also

wxWindow::SetFocus (p. 1533)

wxWindow::FindWindow

wxWindow* FindWindow(**long** *id*) **const**

Find a child of this window, by identifier.

wxWindow* FindWindow(**const wxString&** *name*) **const**

Find a child of this window, by name.

wxPython note: In place of a single overloaded method name, wxPython implements the following methods:

FindWindowById(id)	Accepts an integer
FindWindowByName(name)	Accepts a string

wxWindow::FindWindowById

static wxWindow* FindWindowById(**long** *id*, **wxWindow*** *parent = NULL*)

Find the first window with the given *id*.

If *parent* is NULL, the search will start from all top-level frames and dialog boxes; if non-NULL, the search will be limited to the given window hierarchy. The search is recursive in both cases.

See also

FindWindow (p. 1501)

wxWindow::FindWindowByName

static wxWindow* FindWindowByName(**const wxString&** *name*, **wxWindow*** *parent = NULL*)

Find a window by its name (as given in a window constructor or **Create** function call). If *parent* is NULL, the search will start from all top-level frames and dialog boxes; if non-NULL, the search will be limited to the given window hierarchy. The search is recursive in both cases.

If no window with such name is found,*FindWindowByLabel* (p. 1502) is called.

See also

FindWindow (p. 1501)

wxWindow::FindWindowByLabel

static wxWindow* FindWindowByLabel(**const wxString&** *label*, **wxWindow*** *parent = NULL*)

Find a window by its label. Depending on the type of window, the label may be a window title or panel item label. If *parent* is NULL, the search will start from all top-level frames and dialog boxes; if non-NULL, the search will be limited to the given window hierarchy. The search is recursive in both cases.

See also

FindWindow (p. 1501)

wxWindow::Fit

virtual void Fit()

Sizes the window so that it fits around its subwindows. This function won't do anything if there are no subwindows and will only really work correctly if the sizers are used for the subwindows layout. Also, if the window has exactly one subwindow it is better (faster and the result is more precise as Fit adds some margin to account for fuzziness of its calculations) to call

```
window->SetClientSize(child->GetSize());
```

instead of calling Fit.

wxWindow::FitInside

virtual void FitInside()

Similar to *Fit* (p. 1502), but sizes the interior (virtual) size of a window. Mainly useful with scrolled windows to reset scrollbars after sizing changes that do not trigger a size

event, and/or scrolled windows without an interior sizer. This function similarly won't do anything if there are no subwindows.

wxWindow::Freeze

virtual void Freeze()

Freezes the window or, in other words, prevents any updates from taking place on screen, the window is not redrawn at all. *Thaw* (p. 1544) must be called to reenable window redrawing. Calls to these two functions may be nested.

This method is useful for visual appearance optimization (for example, it is a good idea to use it before doing many large text insertions in a row into a wxTextCtrl under wxGTK) but is not implemented on all platforms nor for all controls so it is mostly just a hint to wxWidgets and not a mandatory directive.

wxWindow::GetAcceleratorTable

wxAcceleratorTable* GetAcceleratorTable() **const**

Gets the accelerator table for this window. See *wxAcceleratorTable* (p. 19).

wxWindow::GetAccessible

wxAccessible* GetAccessible()

Returns the accessible object for this window, if any.

See also *wxAccessible* (p. 22).

wxWindow::GetAdjustedBestSize

wxSize GetAdjustedBestSize() **const**

This method is similar to *GetBestSize* (p. 1504), except in one thing. GetBestSize should return the minimum untruncated size of the window, while this method will return the largest of BestSize and any user specified minimum size. ie. it is the minimum size the window should currently be drawn at, not the minimal size it can possibly tolerate.

wxWindow::GetBackgroundColour

virtual wxColour GetBackgroundColour() **const**

Returns the background colour of the window.

See also

wxWindow::SetBackgroundColour (p. 1528), *wxWindow::SetForegroundColour* (p. 1534), *wxWindow::GetForegroundColour* (p. 1508)

wxWindow::GetBackgroundStyle

virtual wxBackgroundStyle GetBackgroundStyle() **const**

Returns the background style of the window. The background style indicates whether background colour should be determined by the system (wxBG_STYLE_SYSTEM), be set to a specific colour (wxBG_STYLE_COLOUR), or should be left to the application to implement (wxBG_STYLE_CUSTOM).

On GTK+, use of wxBG_STYLE_CUSTOM allows the flicker-free drawing of a custom background, such as a tiled bitmap. Currently the style has no effect on other platforms.

See also

wxWindow::SetBackgroundColour (p. 1528), *wxWindow::GetForegroundColour* (p. 1508), *wxWindow::SetBackgroundStyle* (p. 1528)

wxWindow::GetBestFittingSize

wxSize GetBestFittingSize() **const**

Merges the window's best size into the min size and returns the result.

See also

wxWindow::GetBestSize (p. 1504), *wxWindow::SetBestFittingSize* (p. 1529), *wxWindow::SetSizeHints* (p. 1539)

wxWindow::GetBestSize

wxSize GetBestSize() **const**

This functions returns the best acceptable minimal size for the window. For example, for a static control, it will be the minimal size such that the control label is not truncated. For windows containing subwindows (typically*wxPanel* (p. 1030)), the size returned by this function will be the same as the size the window would have had after calling*Fit* (p. 1502).

wxWindow::GetCaret

wxCaret * GetCaret() **const**

Returns the *caret* (p. 131) associated with the window.

wxWindow::GetCapture

static wxWindow * GetCapture()

Returns the currently captured window.

See also

wxWindow::HasCapture (p. 1515),*wxWindow::CaptureMouse* (p. 1494),*wxWindow::ReleaseMouse* (p. 1524),*wxMouseCaptureChangedEvent* (p. 981)

wxWindow::GetCharHeight

virtual int GetCharHeight() **const**

Returns the character height for this window.

wxWindow::GetCharWidth

virtual int GetCharWidth() **const**

Returns the average character width for this window.

wxWindow::GetChildren

wxList& GetChildren()

Returns a reference to the list of the window's children.

wxWindow::GetClassDefaultAttributes

static wxVisualAttributes GetClassDefaultAttributes(**wxWindowVariant** *variant* = *wxWINDOW_VARIANT_NORMAL*)

Returns the default font and colours which are used by the control. This is useful if you want to use the same font or colour in your own control as in a standard control -- which is a much better idea than hard coding specific colours or fonts which might look completely out of place on the users system, especially if it uses themes.

The *variant* parameter is only relevant under Mac currently and is ignore under other platforms. Under Mac, it will change the size of the returned font. See *wxWindow::SetWindowVariant* (p. 1543)for more about this.

This static method is "overridden" in many derived classes and so calling, for example, *wxButton* (p. 117)::GetClassDefaultAttributes() will typically return the values appropriate for a button which will be normally different from those returned by, say, *wxListCtrl* (p. 852)::GetClassDefaultAttributes().

The `wxVisualAttributes` structure has at least the fields`font`, `colFg` and `colBg`. All of them may be invalid if it was not possible to determine the default control appearance or, especially for the background colour, if the field doesn't make sense as is the case for `colBg` for the controls with themed background.

See also

InheritAttributes (p. 1516)

wxWindow::GetClientSize

void GetClientSize(**int*** *width*, **int*** *height*) **const**

wxPerl note: In wxPerl this method takes no parameter and returns a 2-element list (`width, height`).

wxSize GetClientSize() **const**

This gets the size of the window 'client area' in pixels. The client area is the area which may be drawn on by the programmer, excluding title bar, border, scrollbars, etc.

Parameters

width

> Receives the client width in pixels.

height

> Receives the client height in pixels.

wxPython note: In place of a single overloaded method name, wxPython implements the following methods:

GetClientSizeTuple()	Returns a 2-tuple of (width, height)
GetClientSize()	Returns a wxSize object

See also

GetSize (p. 1512)*, GetVirtualSize* (p. 1514)

wxWindow::GetConstraints

wxLayoutConstraints* GetConstraints() **const**

Returns a pointer to the window's layout constraints, or NULL if there are none.

wxWindow::GetContainingSizer

const wxSizer * GetContainingSizer() **const**

Return the sizer that this window is a member of, if any, otherwise `NULL`.

wxWindow::GetCursor

const wxCursor& GetCursor() **const**

Return the cursor associated with this window.

See also

wxWindow::SetCursor (p. 1530)

wxWindow::GetDefaultAttributes

virtual wxVisualAttributes GetDefaultAttributes() **const**

Currently this is the same as calling *GetClassDefaultAttributes* (p. 1505)(*GetWindowVariant* (p. 1515)()).

One advantage of using this function compared to the static version is that the call is automatically dispatched to the correct class (as usual with virtual functions) and you don't have to specify the class name explicitly.

The other one is that in the future this function could return different results, for example it might return a different font for an "Ok" button than for a generic button if the users GUI is configured to show such buttons in bold font. Of course, the down side is that it is impossible to call this function without actually having an object to apply it to whereas the static version can be used without having to create an object first.

wxWindow::GetDropTarget

wxDropTarget* GetDropTarget() **const**

Returns the associated drop target, which may be NULL.

See also

wxWindow::SetDropTarget (p. 1532),*Drag and drop overview* (p. 1817)

wxWindow::GetEventHandler

wxEvtHandler* GetEventHandler() **const**

Returns the event handler for this window. By default, the window is its own event handler.

See also

wxWindow::SetEventHandler (p. 1532), *wxWindow::PushEventHandler* (p. 1522), *wxWindow::PopEventHandler* (p. 1522), *wxEvtHandler::ProcessEvent* (p. 491), *wxEvtHandler* (p. 486)

wxWindow::GetExtraStyle

long GetExtraStyle() **const**

Returns the extra style bits for the window.

wxWindow::GetFont

wxFont GetFont() **const**

Returns the font for this window.

See also

wxWindow::SetFont (p. 1534)

wxWindow::GetForegroundColour

virtual wxColour GetForegroundColour()

Returns the foreground colour of the window.

Remarks

The interpretation of foreground colour is open to interpretation according to the window class; it may be the text colour or other colour, or it may not be used at all.

See also

wxWindow::SetForegroundColour (p. 1534), *wxWindow::SetBackgroundColour* (p. 1528), *wxWindow::GetBackgroundColour* (p. 1503)

wxWindow::GetGrandParent

wxWindow* GetGrandParent() **const**

Returns the grandparent of a window, or NULL if there isn't one.

wxWindow::GetHandle

void* GetHandle() **const**

Returns the platform-specific handle of the physical window. Cast it to an appropriate handle, such as **HWND** for Windows, **Widget** for Motif, **GtkWidget** for GTK or

WinHandle for PalmOS.

wxPython note: This method will return an integer in wxPython.

wxPerl note: This method will return an integer in wxPerl.

wxWindow::GetHelpText

virtual wxString GetHelpText() **const**

Gets the help text to be used as context-sensitive help for this window.

Note that the text is actually stored by the current *wxHelpProvider* (p. 703) implementation, and not in the window object itself.

See also

SetHelpText (p. 1535), *wxHelpProvider* (p. 703)

wxWindow::GetId

int GetId() **const**

Returns the identifier of the window.

Remarks

Each window has an integer identifier. If the application has not provided one (or the default Id -1) an unique identifier with a negative value will be generated.

See also

wxWindow::SetId (p. 1535), *Window identifiers* (p. 1755)

wxWindow::GetLabel

virtual wxString GetLabel() **const**

Generic way of getting a label from any window, for identification purposes.

Remarks

The interpretation of this function differs from class to class. For frames and dialogs, the value returned is the title. For buttons or static text controls, it is the button text. This function can be useful for meta-programs (such as testing tools or special-needs access programs) which need to identify windows by name.

wxWindow::GetMaxSize

wxSize GetMaxSize() **const**

Returns the maximum size of the window, an indication to the sizer layout mechanism that this is the maximum possible size.

wxWindow::GetMinSize

wxSize GetMinSize() **const**

Returns the minimum size of the window, an indication to the sizer layout mechanism that this is the minimum required size.

wxWindow::GetName

virtual wxString GetName() **const**

Returns the window's name.

Remarks

This name is not guaranteed to be unique; it is up to the programmer to supply an appropriate name in the window constructor or via *wxWindow::SetName* (p. 1536).

See also

wxWindow::SetName (p. 1536)

wxWindow::GetParent

virtual wxWindow* GetParent() **const**

Returns the parent of the window, or NULL if there is no parent.

wxWindow::GetPosition

virtual void GetPosition(**int*** *x*, **int*** *y*) **const**

wxPoint GetPosition() **const**

This gets the position of the window in pixels, relative to the parent window for the child windows or relative to the display origin for the top level windows.

Parameters

x

    Receives the x position of the window.

y

Receives the y position of the window.

wxPython note: In place of a single overloaded method name, wxPython implements the following methods:

GetPosition()	Returns a wxPoint
GetPositionTuple()	Returns a tuple (x, y)

wxPerl note: In wxPerl there are two methods instead of a single overloaded method:

GetPosition()	Returns a Wx::Point
GetPositionXY()	Returns a 2-element list (x, y)

wxWindow::GetRect

virtual wxRect GetRect() **const**

Returns the size and position of the window as a *wxRect* (p. 1099) object.

wxWindow::GetScrollPos

virtual int GetScrollPos(**int** *orientation*)

Returns the built-in scrollbar position.

See also

See *wxWindow::SetScrollbar* (p. 1536)

wxWindow::GetScrollRange

virtual int GetScrollRange(**int** *orientation*)

Returns the built-in scrollbar range.

See also

wxWindow::SetScrollbar (p. 1536)

wxWindow::GetScrollThumb

virtual int GetScrollThumb(**int** *orientation*)

Returns the built-in scrollbar thumb size.

See also

wxWindow::SetScrollbar (p. 1536)

wxWindow::GetSize

void GetSize(**int*** *width*, **int*** *height*) **const**

wxSize GetSize() **const**

This gets the size of the entire window in pixels, including title bar, border, scrollbars, etc.

Parameters

width

> Receives the window width.

height

> Receives the window height.

wxPython note: In place of a single overloaded method name, wxPython implements the following methods:

> **GetSize()**   Returns a wxSize
>
> **GetSizeTuple()**   Returns a 2-tuple (width, height)

wxPerl note: In wxPerl there are two methods instead of a single overloaded method:

> **GetSize()**   Returns a Wx::Size
>
> **GetSizeWH()**   Returns a 2-element list (`width, height`)

See also

GetClientSize (p. 1506), *GetVirtualSize* (p. 1514)

wxWindow::GetSizer

wxSizer * GetSizer() **const**

Return the sizer associated with the window by a previous call to*SetSizer()* (p. 1540) or `NULL`.

wxWindow::GetTextExtent

virtual void GetTextExtent(**const wxString&** *string*, **int*** *x*, **int*** *y*, **int*** *descent = NULL*, **int*** *externalLeading = NULL*, **const wxFont*** *font = NULL*, **bool** *use16 =* `false`) **const**

Gets the dimensions of the string as it would be drawn on the window with the currently selected font.

Parameters

string

String whose extent is to be measured.

x

Return value for width.

y

Return value for height.

descent

Return value for descent (optional).

externalLeading

Return value for external leading (optional).

font

Font to use instead of the current window font (optional).

use16

If `true`, *string* contains 16-bit characters. The default is `false`.

wxPython note: In place of a single overloaded method name, wxPython implements the following methods:

> **GetTextExtent(string)** Returns a 2-tuple, (width, height)
>
> **GetFullTextExtent(string, font=NULL)**Returns a 4-tuple, (width, height, descent, externalLeading)

wxPerl note: In wxPerl this method takes only the `string` and optionally `font` parameters, and returns a 4-element list (`x, y, descent, externalLeading`).

wxWindow::GetTitle

virtual wxString GetTitle()

Gets the window's title. Applicable only to frames and dialogs.

See also

wxWindow::SetTitle (p. 1541)

wxWindow::GetToolTip

wxToolTip* GetToolTip() **const**

Get the associated tooltip or NULL if none.

wxWindow::GetUpdateRegion

virtual wxRegion GetUpdateRegion() **const**

Returns the region specifying which parts of the window have been damaged. Should only be called within an *wxPaintEvent* (p. 1025) handler.

See also

wxRegion (p. 1111), *wxRegionIterator* (p. 1115)

wxWindow::GetValidator

wxValidator* GetValidator() **const**

Returns a pointer to the current validator for the window, or NULL if there is none.

wxWindow::GetVirtualSize

void GetVirtualSize(**int*** *width*, **int*** *height*) **const**

wxSize GetVirtualSize() **const**

This gets the virtual size of the window in pixels. By default it returns the client size of the window, but after a call to *SetVirtualSize* (p. 1542) it will return that size.

Parameters

width

Receives the window virtual width.

height

Receives the window virtual height.

GetSize (p. 1512), *GetClientSize* (p. 1506)

wxWindow::GetWindowStyleFlag

long GetWindowStyleFlag() **const**

Gets the window style that was passed to the constructor or **Create** method.

GetWindowStyle() is another name for the same function.

wxWindow::GetWindowVariant

wxWindowVariant GetWindowVariant() **const**

Returns the value previously passed to *wxWindow::SetWindowVariant* (p. 1543).

wxWindow::HasCapture

virtual bool HasCapture() **const**

Returns `true` if this window has the current mouse capture.

See also

wxWindow::CaptureMouse (p. 1494),*wxWindow::ReleaseMouse* (p. 1524),*wxMouseCaptureChangedEvent* (p. 981)

wxWindow::HasScrollbar

virtual bool HasScrollbar(**int** *orient*) **const**

Returns `true` if this window has a scroll bar for this orientation.

Parameters

orient

> Orientation to check, either `wxHORIZONTAL` or `wxVERTICAL`.

wxWindow::HasTransparentBackground

virtual bool HasTransparentBackground() **const**

Returns `true` if this window background is transparent (as, for example, for wxStaticText) and should show the parent window background.

This method is mostly used internally by the library itself and you normally shouldn't have to call it. You may, however, have to override it in your wxWindow-derived class to ensure that background is painted correctly.

wxWindow::Hide

bool Hide()

Equivalent to calling *Show* (p. 1544)(`false`).

wxWindow::InheritAttributes

void InheritAttributes()

This function is (or should be, in case of custom controls) called during window creation to intelligently set up the window visual attributes, that is the font and the foreground and background colours.

By "intelligently" the following is meant: by default, all windows use their own *default* (p. 1505) attributes. However if some of the parents attributes are explicitly (that is, using *SetFont* (p. 1534) and not *SetOwnFont* (p. 1531)) changed *and* if the corresponding attribute hadn't been explicitly set for this window itself, then this window takes the same value as used by the parent. In addition, if the window overrides *ShouldInheritColours* (p. 1544) to return `false`, the colours will not be changed no matter what and only the font might.

This rather complicated logic is necessary in order to accommodate the different usage scenarios. The most common one is when all default attributes are used and in this case, nothing should be inherited as in modern GUIs different controls use different fonts (and colours) than their siblings so they can't inherit the same value from the parent. However it was also deemed desirable to allow to simply change the attributes of all children at once by just changing the font or colour of their common parent, hence in this case we do inherit the parents attributes.

wxWindow::InitDialog

void InitDialog()

Sends an `wxEVT_INIT_DIALOG` event, whose handler usually transfers data to the dialog via validators.

wxWindow::InvalidateBestSize

void InvalidateBestSize()

Resets the cached best size value so it will be recalculated the next time it is needed.

wxWindow::IsEnabled

virtual bool IsEnabled() **const**

Returns `true` if the window is enabled for input, `false` otherwise.

See also

wxWindow::Enable (p. 1500)

wxWindow::IsExposed

bool IsExposed(**int** *x,* **int** *y*) **const**

bool IsExposed(**wxPoint** *&pt*) **const**

bool IsExposed(**int** *x,* **int** *y,* **int** *w,* **int** *h*) **const**

bool IsExposed(**wxRect** *&rect*) **const**

Returns `true` if the given point or rectangle area has been exposed since the last repaint. Call this in an paint event handler to optimize redrawing by only redrawing those areas, which have been exposed.

wxPython note: In place of a single overloaded method name, wxPython implements the following methods:

> **IsExposed(x,y, w=0,h=0)**

> **IsExposedPoint(pt)**

> **IsExposedRect(rect)**

wxWindow::IsRetained

virtual bool IsRetained() **const**

Returns `true` if the window is retained, `false` otherwise.

Remarks

Retained windows are only available on X platforms.

wxWindow::IsShown

virtual bool IsShown() **const**

Returns `true` if the window is shown, `false` if it has been hidden.

wxWindow::IsTopLevel

bool IsTopLevel() **const**

Returns `true` if the given window is a top-level one. Currently all frames and dialogs are considered to be top-level windows (even if they have a parent window).

wxWindow::Layout

void Layout()

Invokes the constraint-based layout algorithm or the sizer-based algorithm for this window.

See *wxWindow::SetAutoLayout* (p. 1527): when auto layout is on, this function gets called automatically when the window is resized.

wxWindow::LineDown

This is just a wrapper for *ScrollLines* (p. 1526)(1).

wxWindow::LineUp

This is just a wrapper for *ScrollLines* (p. 1526)(-1).

wxWindow::Lower

void Lower()

Lowers the window to the bottom of the window hierarchy if it is a managed window (dialog or frame).

wxWindow::MakeModal

virtual void MakeModal(**bool** *flag*)

Disables all other windows in the application so that the user can only interact with this window.

Parameters

flag

> If `true`, this call disables all other windows in the application so that the user can only interact with this window. If `false`, the effect is reversed.

wxWindow::Move

void Move(**int** *x*, **int** *y*)

void Move(**const wxPoint&** *pt*)

Moves the window to the given position.

Parameters

x

> Required x position.

y

>Required y position.

pt

>*wxPoint* (p. 1045) object representing the position.

Remarks

Implementations of SetSize can also implicitly implement the wxWindow::Move function, which is defined in the base wxWindow class as the call:

```
SetSize(x, y, -1, -1, wxSIZE_USE_EXISTING);
```

See also

wxWindow::SetSize (p. 1538)

wxPython note: In place of a single overloaded method name, wxPython implements the following methods:

Move(point)	Accepts a wxPoint
MoveXY(x, y)	Accepts a pair of integers

wxWindow::MoveAfterInTabOrder

void MoveAfterInTabOrder(**wxWindow ****win*)

Moves this window in the tab navigation order after the specified *win*. This means that when the user presses TAB key on that other window, the focus switches to this window.

Default tab order is the same as creation order, this function and*MoveBeforeInTabOrder()* (p. 1519) allow to change it after creating all the windows.

Parameters

win

>A sibling of this window which should precede it in tab order, must not be NULL

wxWindow::MoveBeforeInTabOrder

void MoveBeforeInTabOrder(**wxWindow ****win*)

Same as *MoveAfterInTabOrder* (p. 1519) except that it inserts this window just before *win* instead of putting it right after it.

wxWindow::Navigate

bool Navigate(**int** *flags* = *wxNavigationKeyEvent::IsForward*)

Does keyboard navigation from this window to another, by sending a wxNavigationKeyEvent.

Parameters

flags

> A combination of wxNavigationKeyEvent::IsForward and wxNavigationKeyEvent::WinChange.

Remarks

You may wish to call this from a text control custom keypress handler to do the default navigation behaviour for the tab key, since the standard default behaviour for a multiline text control with the wxTE_PROCESS_TAB style is to insert a tab and not navigate to the next control.

wxWindow::OnInternalIdle

virtual void OnInternalIdle()

This virtual function is normally only used internally, but sometimes an application may need it to implement functionality that should not be disabled by an application defining an OnIdle handler in a derived class.

This function may be used to do delayed painting, for example, and most implementations call *wxWindow::UpdateWindowUI* (p. 1546)in order to send update events to the window in idle time.

wxWindow::PageDown

This is just a wrapper for *ScrollPages()* (p. 1526)(1).

wxWindow::PageUp

This is just a wrapper for *ScrollPages()* (p. 1526)(-1).

wxWindow::PopEventHandler

wxEvtHandler* PopEventHandler(**bool** *deleteHandler* = `false`) **const**

Removes and returns the top-most event handler on the event handler stack.

Parameters

deleteHandler

If this is `true`, the handler will be deleted after it is removed. The default value is `false`.

See also

wxWindow::SetEventHandler (p. 1532), *wxWindow::GetEventHandler* (p. 1507), *wxWindow::PushEventHandler* (p. 1522), *wxEvtHandler::ProcessEvent* (p. 491), *wxEvtHandler* (p. 486)

wxWindow::PopupMenu

bool PopupMenu(**wxMenu*** *menu*, **const wxPoint&** *pos* = *wxDefaultPosition*)

bool PopupMenu(**wxMenu*** *menu*, **int** *x*, **int** *y*)

Pops up the given menu at the specified coordinates, relative to this window, and returns control when the user has dismissed the menu. If a menu item is selected, the corresponding menu event is generated and will be processed as usually. If the coordinates are not specified, current mouse cursor position is used.

Parameters

menu

Menu to pop up.

pos

The position where the menu will appear.

x

Required x position for the menu to appear.

y

Required y position for the menu to appear.

See also

wxMenu (p. 940)

Remarks

Just before the menu is popped up, *wxMenu::UpdateUI* (p. 953)is called to ensure that the menu items are in the correct state. The menu does not get deleted by the window.

It is recommended to not explicitly specify coordinates when calling PopupMenu in response to mouse click, because some of the ports (namely, wxGTK) can do a better job of positioning the menu in that case.

wxPython note: In place of a single overloaded method name, wxPython implements the following methods:

PopupMenu(menu, point)   Specifies position with a wxPoint

PopupMenuXY(menu, x, y) Specifies position with two integers (x, y)

wxWindow::PushEventHandler

void PushEventHandler(**wxEvtHandler*** *handler*)

Pushes this event handler onto the event stack for the window.

Parameters

handler

Specifies the handler to be pushed.

Remarks

An event handler is an object that is capable of processing the events sent to a window. By default, the window is its own event handler, but an application may wish to substitute another, for example to allow central implementation of event-handling for a variety of different window classes.

wxWindow::PushEventHandler (p. 1522) allows an application to set up a chain of event handlers, where an event not handled by one event handler is handed to the next one in the chain. Use *wxWindow::PopEventHandler* (p. 1520) to remove the event handler.

See also

wxWindow::SetEventHandler (p. 1532), *wxWindow::GetEventHandler* (p. 1507), *wxWindow::PopEventHandler* (p. 1522), *wxEvtHandler::ProcessEvent* (p. 491), *wxEvtHandler* (p. 486)

wxWindow::Raise

void Raise()

Raises the window to the top of the window hierarchy if it is a managed window (dialog or frame).

wxWindow::Refresh

virtual void Refresh(**bool** *eraseBackground* = `true`, **const wxRect*** *rect* = *NULL*)

Causes this window, and all of its children recursively (except under wxGTK1 where this is not implemented), to be repainted. Note that repainting doesn't happen immediately but only during the next event loop iteration, if you need to update the window immediately you should use *Update* (p. 1546) instead.

Parameters

eraseBackground

If `true`, the background will be erased.

rect

If non-NULL, only the given rectangle will be treated as damaged.

See also

wxWindow::RefreshRect (p. 1523)

wxWindow::RefreshRect

void RefreshRect(**const wxRect&** *rect*, **bool** *eraseBackground* = `true`)

Redraws the contents of the given rectangle: only the area inside it will be repainted.

This is the same as *Refresh* (p. 1522) but has a nicer syntax as it can be called with a temporary wxRect object as argument like this `RefreshRect(wxRect(x, y, w, h))`.

wxWindow::RegisterHotKey

bool RegisterHotKey(**int** *hotkeyId*, **int** *modifiers*, **int** *virtualKeyCode*)

Registers a system wide hotkey. Every time the user presses the hotkey registered here, this window will receive a hotkey event. It will receive the event even if the application is in the background and does not have the input focus because the user is working with some other application.

Parameters

hotkeyId

Numeric identifier of the hotkey. For applications this must be between 0 and 0xBFFF. If this function is called from a shared DLL, it must be a system wide unique identifier between 0xC000 and 0xFFFF. This is a MSW specific detail.

modifiers

A bitwise combination of `wxMOD_SHIFT`, `wxMOD_CONTROL`, `wxMOD_ALT`or `wxMOD_WIN` specifying the modifier keys that have to be pressed along with the key.

virtualKeyCode

The virtual key code of the hotkey.

Return value

`true` if the hotkey was registered successfully. `false` if some other application already registered a hotkey with this modifier/virtualKeyCode combination.

Remarks

Use EVT_HOTKEY(hotkeyId, fnc) in the event table to capture the event. This function is currently only implemented under Windows. It is used in the *Windows CE port* (p. 1887) for detecting hardware button presses.

See also

wxWindow::UnregisterHotKey (p. 1545)

wxWindow::ReleaseMouse

virtual void ReleaseMouse()

Releases mouse input captured with *wxWindow::CaptureMouse* (p. 1494).

See also

wxWindow::CaptureMouse (p. 1494),*wxWindow::HasCapture* (p. 1515),*wxWindow::ReleaseMouse* (p. 1524),*wxMouseCaptureChangedEvent* (p. 981)

wxWindow::RemoveChild

virtual void RemoveChild(**wxWindow*** *child*)

Removes a child window. This is called automatically by window deletion functions so should not be required by the application programmer.

Notice that this function is mostly internal to wxWidgets and shouldn't be called by the user code.

Parameters

child

    Child window to remove.

wxWindow::RemoveEventHandler

bool RemoveEventHandler(**wxEvtHandler ****handler*)

Find the given *handler* in the windows event handler chain and remove (but not delete) it from it.

Parameters

handler

The event handler to remove, must be non `NULL` and must be present in this windows event handlers chain

Return value

Returns `true` if it was found and `false` otherwise (this also results in an assert failure so this function should only be called when the handler is supposed to be there).

See also

PushEventHandler (p. 1522), *PopEventHandler* (p. 1520)

wxWindow::Reparent

virtual bool Reparent(**wxWindow*** *newParent*)

Reparents the window, i.e the window will be removed from its current parent window (e.g. a non-standard toolbar in a wxFrame) and then re-inserted into another. Available on Windows and GTK.

Parameters

newParent

New parent.

wxWindow::ScreenToClient

virtual void ScreenToClient(**int*** *x*, **int*** *y*) **const**

virtual wxPoint ScreenToClient(**const wxPoint&** *pt*) **const**

Converts from screen to client window coordinates.

Parameters

x

Stores the screen x coordinate and receives the client x coordinate.

y

Stores the screen x coordinate and receives the client x coordinate.

pt

The screen position for the second form of the function.

wxPython note: In place of a single overloaded method name, wxPython implements the following methods:

ScreenToClient(point)   Accepts and returns a wxPoint

ScreenToClientXY(x, y)   Returns a 2-tuple, (x, y)

wxWindow::ScrollLines

virtual bool ScrollLines(**int** *lines*)

Scrolls the window by the given number of lines down (if *lines* is positive) or up.

Return value

Returns `true` if the window was scrolled, `false` if it was already on top/bottom and nothing was done.

Remarks

This function is currently only implemented under MSW and wxTextCtrl under wxGTK (it also works for wxScrolledWindow derived classes under all platforms).

See also

ScrollPages (p. 1526)

wxWindow::ScrollPages

virtual bool ScrollPages(**int** *pages*)

Scrolls the window by the given number of pages down (if *pages* is positive) or up.

Return value

Returns `true` if the window was scrolled, `false` if it was already on top/bottom and nothing was done.

Remarks

This function is currently only implemented under MSW and wxTextCtrl under wxGTK (it also works for wxScrolledWindow derived classes under all platforms).

See also

ScrollLines (p. 1526)

wxWindow::ScrollWindow

virtual void ScrollWindow(**int** *dx*, **int** *dy*, **const wxRect*** *rect = NULL*)

Physically scrolls the pixels in the window and move child windows accordingly.

Parameters

dx

> Amount to scroll horizontally.

dy

> Amount to scroll vertically.

rect

> Rectangle to invalidate. If this is NULL, the whole window is invalidated. If you pass a rectangle corresponding to the area of the window exposed by the scroll, your painting handler can optimize painting by checking for the invalidated region. This parameter is ignored under GTK.

Remarks

Use this function to optimise your scrolling implementations, to minimise the area that must be redrawn. Note that it is rarely required to call this function from a user program.

wxWindow::SetAcceleratorTable

virtual void SetAcceleratorTable(**const wxAcceleratorTable&** *accel*)

Sets the accelerator table for this window. See *wxAcceleratorTable* (p. 19).

wxWindow::SetAccessible

void SetAccessible(**wxAccessible*** *accessible*)

Sets the accessible for this window. Any existing accessible for this window will be deleted first, if not identical to *accessible*.

See also *wxAccessible* (p. 22).

wxWindow::SetAutoLayout

void SetAutoLayout(**bool** *autoLayout*)

Determines whether the *wxWindow::Layout* (p. 1517) function will be called automatically when the window is resized. It is called implicitly by *wxWindow::SetSizer* (p. 1540) but if you use *wxWindow::SetConstraints* (p. 1530) you should call it manually or otherwise the window layout won't be correctly updated when its size changes.

Parameters

autoLayout

> Set this to `true` if you wish the Layout function to be called from within wxWindow::OnSize functions.

See also

wxWindow::SetConstraints (p. 1530)

wxWindow::SetBackgroundColour

virtual bool SetBackgroundColour(**const wxColour&** *colour*)

Sets the background colour of the window.

Please see *InheritAttributes* (p. 1516) for explanation of the difference between this method and*SetOwnBackgroundColour* (p. 1531).

Parameters

colour

> The colour to be used as the background colour, pass `wxNullColour` to reset to the default colour.

Remarks

The background colour is usually painted by the default *wxEraseEvent* (p. 482) event handler function under Windows and automatically under GTK.

Note that setting the background colour does not cause an immediate refresh, so you may wish to call *wxWindow::ClearBackground* (p. 1496) or *wxWindow::Refresh* (p. 1522) after calling this function.

Using this function will disable attempts to use themes for this window, if the system supports them. Use with care since usually the themes represent the appearance chosen by the user to be used for all applications on the system.

See also

wxWindow::GetBackgroundColour (p. 1503), *wxWindow::SetForegroundColour* (p. 1534), *wxWindow::GetForegroundColour* (p. 1508), *wxWindow::ClearBackground* (p. 1496), *wxWindow::Refresh* (p. 1522), *wxEraseEvent* (p. 482)

wxWindow::SetBackgroundStyle

virtual void SetBackgroundStyle(**wxBackgroundStyle** *style*)

Sets the background style of the window. The background style indicates whether background colour should be determined by the system (wxBG_STYLE_SYSTEM), be set to a specific colour (wxBG_STYLE_COLOUR), or should be left to the application to implement (wxBG_STYLE_CUSTOM).

On GTK+, use of wxBG_STYLE_CUSTOM allows the flicker-free drawing of a custom background, such as a tiled bitmap. Currently the style has no effect on other platforms.

See also

wxWindow::SetBackgroundColour (p. 1528), *wxWindow::GetForegroundColour* (p. 1508), *wxWindow::GetBackgroundStyle* (p. 1504)

wxWindow::SetBestFittingSize

void SetBestFittingSize(**const wxSize&** *size = wxDefaultSize*)

A *smart* SetSize that will fill in default size components with the window's *best* size values. Also sets the window's minsize to the value passed in for use with sizers. This means that if a full or partial size is passed to this function then the sizers will use that size instead of the results of GetBestSize to determine the minimum needs of the window for layout.

See also

wxWindow::SetSize (p. 1538), *wxWindow::GetBestSize* (p. 1504), *wxWindow::GetBestFittingSize* (p. 1504), *wxWindow::SetSizeHints* (p. 1539)

wxWindow::SetCaret

void SetCaret(**wxCaret ****caret*) **const**

Sets the *caret* (p. 131) associated with the window.

wxWindow::SetClientSize

virtual void SetClientSize(**int** *width*, **int** *height*)

virtual void SetClientSize(**const wxSize&** *size*)

This sets the size of the window client area in pixels. Using this function to size a window tends to be more device-independent than *wxWindow::SetSize* (p. 1538), since the application need not worry about what dimensions the border or title bar have when trying to fit the window around panel items, for example.

Parameters

width

> The required client area width.

height

> The required client area height.

size

> The required client size.

wxPython note: In place of a single overloaded method name, wxPython implements

the following methods:

SetClientSize(size)   Accepts a wxSize

SetClientSizeWH(width, height)

wxWindow::SetContainingSizer

void SetContainingSizer(**wxSizer*** *sizer*)

This normally does not need to be called by user code. It is called when a window is added to a sizer, and is used so the window can remove itself from the sizer when it is destroyed.

wxWindow::SetCursor

virtual void SetCursor(**const wxCursor&***cursor*)

Sets the window's cursor. Notice that the window cursor also sets it for the children of the window implicitly.

The *cursor* may be wxNullCursor in which case the window cursor will be reset back to default.

Parameters

cursor

> Specifies the cursor that the window should normally display.

See also

::wxSetCursor (p. 1625), *wxCursor* (p. 226)

wxWindow::SetConstraints

void SetConstraints(**wxLayoutConstraints*** *constraints*)

Sets the window to have the given layout constraints. The window will then own the object, and will take care of its deletion. If an existing layout constraints object is already owned by the window, it will be deleted.

Parameters

constraints

> The constraints to set. Pass NULL to disassociate and delete the window's constraints.

Remarks

You must call *wxWindow::SetAutoLayout* (p. 1527) to tell a window to use the constraints automatically in OnSize; otherwise, you must override OnSize and call Layout() explicitly. When setting both a wxLayoutConstraints and a *wxSizer* (p. 1176), only the sizer will have effect.

wxWindow::SetInitialBestSize

virtual void SetInitialBestSize(**const wxSize&** *size*)

Sets the initial window size if none is given (i.e. at least one of the components of the size passed to ctor/Create() is wxDefaultCoord).

wxWindow::SetMaxSize

void SetMaxSize(**const wxSize&** *size*)

Sets the maximum size of the window, to indicate to the sizer layout mechanism that this is the maximum possible size.

wxWindow::SetMinSize

void SetMinSize(**const wxSize&** *size*)

Sets the minimum size of the window, to indicate to the sizer layout mechanism that this is the minimum required size. You may need to call this if you change the window size after construction and before adding to its parent sizer.

wxWindow::SetOwnBackgroundColour

void SetOwnBackgroundColour(**const wxColour&** *colour*)

Sets the background colour of the window but prevents it from being inherited by the children of this window.

See also

SetBackgroundColour (p. 1528), *InheritAttributes* (p. 1516)

wxWindow::SetOwnFont

void SetOwnFont(**const wxFont&** *font*)

Sets the font of the window but prevents it from being inherited by the children of this window.

See also

SetFont (p. 1534), *InheritAttributes* (p. 1516)

wxWindow::SetOwnForegroundColour

void SetOwnForegroundColour(**const wxColour&** *colour*)

Sets the foreground colour of the window but prevents it from being inherited by the children of this window.

See also

SetForegroundColour (p. 1534), *InheritAttributes* (p. 1516)

wxWindow::SetDropTarget

void SetDropTarget(**wxDropTarget*** *target*)

Associates a drop target with this window.

If the window already has a drop target, it is deleted.

See also

wxWindow::GetDropTarget (p. 1507),*Drag and drop overview* (p. 1817)

wxWindow::SetEventHandler

void SetEventHandler(**wxEvtHandler*** *handler*)

Sets the event handler for this window.

Parameters

handler

> Specifies the handler to be set.

Remarks

An event handler is an object that is capable of processing the events sent to a window. By default, the window is its own event handler, but an application may wish to substitute another, for example to allow central implementation of event-handling for a variety of different window classes.

It is usually better to use *wxWindow::PushEventHandler* (p. 1522) since this sets up a chain of event handlers, where an event not handled by one event handler is handed to the next one in the chain.

See also

wxWindow::GetEventHandler (p. 1507), *wxWindow::PushEventHandler* (p. 1522),

wxWindow::PopEventHandler (p. 1522), *wxEvtHandler::ProcessEvent* (p. 491), *wxEvtHandler* (p. 486)

wxWindow::SetExtraStyle

void SetExtraStyle(**long** *exStyle*)

Sets the extra style bits for the window. The currently defined extra style bits are:

wxWS_EX_VALIDATE_RECURSIVELY   TransferDataTo/FromWindow() and Validate() methods will recursively descend into all children of the window if it has this style flag set.

wxWS_EX_BLOCK_EVENTS   Normally, the command events are propagated upwards to the window parent recursively until a handler for them is found. Using this style allows to prevent them from being propagated beyond this window. Notice that wxDialog has this style on by default for the reasons explained in the*event processing overview* (p. 1752).

wxWS_EX_TRANSIENT   This can be used to prevent a window from being used as an implicit parent for the dialogs which were created without a parent. It is useful for the windows which can disappear at any moment as creating children of such windows results in fatal problems.

wxFRAME_EX_CONTEXTHELP   Under Windows, puts a query button on the caption. When pressed, Windows will go into a context-sensitive help mode and wxWidgets will send a wxEVT_HELP event if the user clicked on an application window. This style cannot be used together with wxMAXIMIZE_BOX or wxMINIMIZE_BOX, so you should use the style of`wxDEFAULT_FRAME_STYLE & (wxMINIMIZE_BOX | wxMAXIMIZE_BOX)` for the frames having this style (the dialogs don't have minimize nor maximize box by default)

wxWS_EX_PROCESS_IDLE   This window should always process idle events, even if the mode set by *wxIdleEvent::SetMode* (p. 778) is wxIDLE_PROCESS_SPECIFIED.

wxWS_EX_PROCESS_UI_UPDATES   This window should always process UI update events, even if the mode set by *wxUpdateUIEvent::SetMode* (p. 1451) is wxUPDATE_UI_PROCESS_SPECIFIED.

wxWindow::SetFocus

virtual void SetFocus()

This sets the window to receive keyboard input.

See also

wxFocusEvent (p. 557)*wxPanel::SetFocus* (p. 1033)*wxPanel::SetFocusIgnoringChildren* (p. 1034)

wxWindow::SetFocusFromKbd

virtual void SetFocusFromKbd()

This function is called by wxWidgets keyboard navigation code when the user gives the focus to this window from keyboard (e.g. using `TAB` key). By default this method simply calls *SetFocus* (p. 1533) but can be overridden to do something in addition to this in the derived classes.

wxWindow::SetFont

void SetFont(**const wxFont&** *font*)

Sets the font for this window. This function should not be called for the parent window if you don't want its font to be inherited by its children, use *SetOwnFont* (p. 1531) instead in this case and see *InheritAttributes* (p. 1516) for more explanations.

Parameters

font

> Font to associate with this window, pass`wxNullFont` to reset to the default font.

See also

wxWindow::GetFont (p. 1508),
InheritAttributes (p. 1516)

wxWindow::SetForegroundColour

virtual void SetForegroundColour(**const wxColour&** *colour*)

Sets the foreground colour of the window.

Please see *InheritAttributes* (p. 1516) for explanation of the difference between this method and*SetOwnForegroundColour* (p. 1532).

Parameters

colour

> The colour to be used as the foreground colour, pass `wxNullColour` to reset to the default colour.

Remarks

The interpretation of foreground colour is open to interpretation according to the window class; it may be the text colour or other colour, or it may not be used at all.

Using this function will disable attempts to use themes for this window, if the system supports them. Use with care since usually the themes represent the appearance chosen by the user to be used for all applications on the system.

See also

wxWindow::GetForegroundColour (p. 1508), *wxWindow::SetBackgroundColour* (p. 1528), *wxWindow::GetBackgroundColour* (p. 1503), *wxWindow::ShouldInheritColours* (p. 1544)

wxWindow::SetHelpText

virtual void SetHelpText(**const wxString&** *helpText*)

Sets the help text to be used as context-sensitive help for this window.

Note that the text is actually stored by the current *wxHelpProvider* (p. 703) implementation, and not in the window object itself.

See also

GetHelpText (p. 1509), *wxHelpProvider* (p. 703)

wxWindow::SetId

void SetId(**int** *id*)

Sets the identifier of the window.

Remarks

Each window has an integer identifier. If the application has not provided one, an identifier will be generated. Normally, the identifier should be provided on creation and should not be modified subsequently.

See also

wxWindow::GetId (p. 1509), *Window identifiers* (p. 1755)

wxWindow::SetLabel

virtual void SetLabel(**const wxString&** *label*)

Sets the window's label.

Parameters

label

> The window label.

See also

wxWindow::GetLabel (p. 1509)

wxWindow::SetName

virtual void SetName(**const wxString&** *name*)

Sets the window's name.

Parameters

name

> A name to set for the window.

See also

wxWindow::GetName (p. 1510)

wxWindow::SetPalette

virtual void SetPalette(**wxPalette*** *palette*)

Obsolete - use *wxDC::SetPalette* (p. 384) instead.

wxWindow::SetScrollbar

virtual void SetScrollbar(**int** *orientation*, **int** *position*, **int** *thumbSize*, **int** *range*, **bool** *refresh* = `true`)

Sets the scrollbar properties of a built-in scrollbar.

Parameters

orientation

> Determines the scrollbar whose page size is to be set. May be wxHORIZONTAL or wxVERTICAL.

position

> The position of the scrollbar in scroll units.

thumbSize

> The size of the thumb, or visible portion of the scrollbar, in scroll units.

range

The maximum position of the scrollbar.

refresh

`true` to redraw the scrollbar, `false` otherwise.

Remarks

Let's say you wish to display 50 lines of text, using the same font. The window is sized so that you can only see 16 lines at a time.

You would use:

```
SetScrollbar(wxVERTICAL, 0, 16, 50);
```

Note that with the window at this size, the thumb position can never go above 50 minus 16, or 34.

You can determine how many lines are currently visible by dividing the current view size by the character height in pixels.

When defining your own scrollbar behaviour, you will always need to recalculate the scrollbar settings when the window size changes. You could therefore put your scrollbar calculations and SetScrollbar call into a function named AdjustScrollbars, which can be called initially and also from your *wxSizeEvent* (p. 1175) handler function.

See also

Scrolling overview (p. 1786), *wxScrollBar* (p. 1143), *wxScrolledWindow* (p. 1150)

wxWindow::SetScrollPos

virtual void SetScrollPos(**int** *orientation*, **int** *pos*, **bool** *refresh* = `true`)

Sets the position of one of the built-in scrollbars.

Parameters

orientation

Determines the scrollbar whose position is to be set. May be wxHORIZONTAL or wxVERTICAL.

pos

Position in scroll units.

refresh

`true` to redraw the scrollbar, `false` otherwise.

Remarks

This function does not directly affect the contents of the window: it is up to the application to take note of scrollbar attributes and redraw contents accordingly.

See also

wxWindow::SetScrollbar (p. 1536), *wxWindow::GetScrollPos* (p. 1511), *wxWindow::GetScrollThumb* (p. 1511), *wxScrollBar* (p. 1143), *wxScrolledWindow* (p. 1150)

wxWindow::SetSize

virtual void SetSize(**int** *x*, **int** *y*, **int** *width*, **int** *height*, **int** *sizeFlags* = *wxSIZE_AUTO*)

virtual void SetSize(**const wxRect&** *rect*)

Sets the size and position of the window in pixels.

virtual void SetSize(**int** *width*, **int** *height*)

virtual void SetSize(**const wxSize&** *size*)

Sets the size of the window in pixels.

Parameters

x

Required x position in pixels, or -1 to indicate that the existing value should be used.

y

Required y position in pixels, or -1 to indicate that the existing value should be used.

width

Required width in pixels, or -1 to indicate that the existing value should be used.

height

Required height position in pixels, or -1 to indicate that the existing value should be used.

size

wxSize (p. 1173) object for setting the size.

rect

wxRect (p. 1099) object for setting the position and size.

sizeFlags

Indicates the interpretation of other parameters. It is a bit list of the following:

wxSIZE_AUTO_WIDTH: a -1 width value is taken to indicate a wxWidgets-supplied default width.

wxSIZE_AUTO_HEIGHT: a -1 height value is taken to indicate a wxWidgets-supplied default width.

wxSIZE_AUTO: -1 size values are taken to indicate a wxWidgets-supplied default size.

wxSIZE_USE_EXISTING: existing dimensions should be used if -1 values are supplied.

wxSIZE_ALLOW_MINUS_ONE: allow dimensions of -1 and less to be interpreted as real dimensions, not default values.

Remarks

The second form is a convenience for calling the first form with default x and y parameters, and must be used with non-default width and height values.

The first form sets the position and optionally size, of the window. Parameters may be -1 to indicate either that a default should be supplied by wxWidgets, or that the current value of the dimension should be used.

See also

wxWindow::Move (p. 1518)

wxPython note: In place of a single overloaded method name, wxPython implements the following methods:

SetDimensions(x, y, width, height, sizeFlags=wxSIZE_AUTO)

SetSize(size)

SetPosition(point)

wxWindow::SetSizeHints

virtual void SetSizeHints(**int** *minW=-1*, **int** *minH=-1*, **int** *maxW=-1*, **int** *maxH=-1*, **int** *incW=-1*, **int** *incH=-1*)

void SetSizeHints(**const wxSize&** *minSize*,**const wxSize&** *maxSize=wxDefaultSize*, **const wxSize&** *incSize=wxDefaultSize*)

Allows specification of minimum and maximum window sizes, and window size increments. If a pair of values is not set (or set to -1), the default values will be used.

Parameters

minW

Specifies the minimum width allowable.

minH

Specifies the minimum height allowable.

maxW

Specifies the maximum width allowable.

maxH

Specifies the maximum height allowable.

incW

Specifies the increment for sizing the width (Motif/Xt only).

incH

Specifies the increment for sizing the height (Motif/Xt only).

minSize

Minimum size.

maxSize

Maximum size.

incSize

Increment size (Motif/Xt only).

Remarks

If this function is called, the user will not be able to size the window outside the given bounds.

The resizing increments are only significant under Motif or Xt.

wxWindow::SetSizer

void SetSizer(**wxSizer*** *sizer*, **bool** *deleteOld=true*)

Sets the window to have the given layout sizer. The window will then own the object, and will take care of its deletion. If an existing layout constraints object is already owned by the window, it will be deleted if the deleteOld parameter is true.

Note that this function will also call*SetAutoLayout* (p. 1527) implicitly with `true`parameter if the *sizer* is non-NULL and `false` otherwise.

Parameters

sizer

> The sizer to set. Pass NULL to disassociate and conditionally delete the window's sizer. See below.

deleteOld

> If true (the default), this will delete any prexisting sizer. Pass false if you wish to handle deleting the old sizer yourself.

Remarks

SetSizer now enables and disables Layout automatically, but prior to wxWidgets 2.3.3 the following applied:

You must call *wxWindow::SetAutoLayout* (p. 1527) to tell a window to use the sizer automatically in OnSize; otherwise, you must override OnSize and call Layout() explicitly. When setting both a wxSizer and a *wxLayoutConstraints* (p. 837), only the sizer will have effect.

wxWindow::SetSizerAndFit

void SetSizerAndFit(**wxSizer*** *sizer*, **bool** *deleteOld=true*)

The same as *SetSizer* (p. 1540), except it also sets the size hints for the window based on the sizer's minimum size.

wxWindow::SetTitle

virtual void SetTitle(**const wxString&** *title*)

Sets the window's title. Applicable only to frames and dialogs.

Parameters

title

> The window's title.

See also

wxWindow::GetTitle (p. 1513)

wxWindow::SetThemeEnabled

virtual void SetThemeEnabled(**bool** *enable*)

This function tells a window if it should use the system's "theme" code to draw the windows' background instead if its own background drawing code. This does not always have any effect since the underlying platform obviously needs to support the notion of

themes in user defined windows. One such platform is GTK+ where windows can have (very colourful) backgrounds defined by a user's selected theme.

Dialogs, notebook pages and the status bar have this flag set to true by default so that the default look and feel is simulated best.

wxWindow::SetToolTip

void SetToolTip(**const wxString&** *tip*)

void SetToolTip(**wxToolTip*** *tip*)

Attach a tooltip to the window.

See also: *GetToolTip* (p. 1514), *wxToolTip* (p. 1418)

wxWindow::SetValidator

virtual void SetValidator(**const wxValidator&** *validator*)

Deletes the current validator (if any) and sets the window validator, having called wxValidator::Clone to create a new validator of this type.

wxWindow::SetVirtualSize

void SetVirtualSize(**int** *width*, **int** *height*)

void SetVirtualSize(**const wxSize&** *size*)

Sets the virtual size of the window in pixels.

wxWindow::SetVirtualSizeHints

virtual void SetVirtualSizeHints(**int** *minW*,**int** *minH*, **int** *maxW=-1*, **int** *maxH=-1*)

void SetVirtualSizeHints(**const wxSize&** *minSize=wxDefaultSize*,**const wxSize&** *maxSize=wxDefaultSize*)

Allows specification of minimum and maximum virtual window sizes. If a pair of values is not set (or set to -1), the default values will be used.

Parameters

minW

Specifies the minimum width allowable.

minH

Specifies the minimum height allowable.

maxW

Specifies the maximum width allowable.

maxH

Specifies the maximum height allowable.

minSize

Minimum size.

maxSize

Maximum size.

Remarks

If this function is called, the user will not be able to size the virtual area of the window outside the given bounds.

wxWindow::SetWindowStyle

void SetWindowStyle(**long** *style*)

Identical to *SetWindowStyleFlag* (p. 1543).

wxWindow::SetWindowStyleFlag

virtual void SetWindowStyleFlag(**long** *style*)

Sets the style of the window. Please note that some styles cannot be changed after the window creation and that *Refresh()* (p. 1522) might be called after changing the others for the change to take place immediately.

See *Window styles* (p. 1762) for more information about flags.

See also

GetWindowStyleFlag (p. 1514)

wxWindow::SetWindowVariant

void SetWindowVariant(**wxWindowVariant***variant*)

```
This function can be called under all platforms but only does anything
under Mac OS X 10.3+ currently. Under this system, each of the standard
control can exist in several sizes which correspond to the elements of
wxWindowVariant enum:enum wxWindowVariant
{
```

```
    wxWINDOW_VARIANT_NORMAL,          // Normal size
    wxWINDOW_VARIANT_SMALL,           // Smaller size (about 25 % smaller
than normal )
    wxWINDOW_VARIANT_MINI,            // Mini size (about 33 % smaller than
normal )
    wxWINDOW_VARIANT_LARGE,           // Large size (about 25 % larger than
normal )
};
```

By default the controls use the normal size, of course, but this function can be used to change this.

wxWindow::ShouldInheritColours

virtual bool ShouldInheritColours()

Return `true` from here to allow the colours of this window to be changed by*InheritAttributes* (p. 1516), returning `false`forbids inheriting them from the parent window.

The base class version returns `false`, but this method is overridden in*wxControl* (p. 215) where it returns `true`.

wxWindow::Show

virtual bool Show(bool *show* = *true*)

Shows or hides the window. You may need to call *Raise* (p. 1522)for a top level window if you want to bring it to top, although this is not needed if Show() is called immediately after the frame creation.

Parameters

show

　　If `true` displays the window. Otherwise, hides it.

Return value

`true` if the window has been shown or hidden or `false` if nothing was done because it already was in the requested state.

See also

wxWindow::IsShown (p. 1517), *wxWindow::Hide* (p. 1515), *wxRadioBox::Show* (p. 1095)

wxWindow::Thaw

virtual void Thaw()

Reenables window updating after a previous call to*Freeze* (p. 1503). To really thaw the

control, it must be called exactly the same number of times as *Freeze* (p. 1503).

wxWindow::TransferDataFromWindow

virtual bool TransferDataFromWindow()

Transfers values from child controls to data areas specified by their validators. Returns`false` if a transfer failed.

If the window has `wxWS_EX_VALIDATE_RECURSIVELY` extra style flag set, the method will also call TransferDataFromWindow() of all child windows.

See also

wxWindow::TransferDataToWindow (p. 1545), *wxValidator* (p. 1461), *wxWindow::Validate* (p. 1547)

wxWindow::TransferDataToWindow

virtual bool TransferDataToWindow()

Transfers values to child controls from data areas specified by their validators.

If the window has `wxWS_EX_VALIDATE_RECURSIVELY` extra style flag set, the method will also call TransferDataToWindow() of all child windows.

Return value

Returns `false` if a transfer failed.

See also

wxWindow::TransferDataFromWindow (p. 1545), *wxValidator* (p. 1461), *wxWindow::Validate* (p. 1547)

wxWindow::UnregisterHotKey

bool UnregisterHotKey(**int** *hotkeyId*)

Unregisters a system wide hotkey.

Parameters

hotkeyId

> Numeric identifier of the hotkey. Must be the same id that was passed to RegisterHotKey.

Return value

`true` if the hotkey was unregistered successfully, `false` if the id was invalid.

Remarks

This function is currently only implemented under MSW.

See also

wxWindow::RegisterHotKey (p. 1523)

wxWindow::Update

virtual void Update()

Calling this method immediately repaints the invalidated area of the window and all of its children recursively while this would usually only happen when the flow of control returns to the event loop. Notice that this function doesn't refresh the window and does nothing if the window hadn't been already repainted. Use *Refresh* (p. 1522) first if you want to immediately redraw the window unconditionally.

wxWindow::UpdateWindowUI

virtual void UpdateWindowUI(**long** *flags* = *wxUPDATE_UI_NONE*)

This function sends *wxUpdateUIEvents* (p. 1448) to the window. The particular implementation depends on the window; for example a wxToolBar will send an update UI event for each toolbar button, and a wxFrame will send an update UI event for each menubar menu item. You can call this function from your application to ensure that your UI is up-to-date at this point (as far as your wxUpdateUIEvent handlers are concerned). This may be necessary if you have called*wxUpdateUIEvent::SetMode* (p. 1451) or*wxUpdateUIEvent::SetUpdateInterval* (p. 1452) to limit the overhead that wxWidgets incurs by sending update UI events in idle time.

flags should be a bitlist of one or more of the following values.

```
enum wxUpdateUI
{
    wxUPDATE_UI_NONE          = 0x0000, // No particular value
    wxUPDATE_UI_RECURSE       = 0x0001, // Call the function for
descendants
    wxUPDATE_UI_FROMIDLE      = 0x0002  // Invoked from On(Internal)Idle
};
```

If you are calling this function from an OnInternalIdle or OnIdle function, make sure you pass the wxUPDATE_UI_FROMIDLE flag, since this tells the window to only update the UI elements that need to be updated in idle time. Some windows update their elements only when necessary, for example when a menu is about to be shown. The following is an example of how to call UpdateWindowUI from an idle function.

```
void MyWindow::OnInternalIdle()
{
    if (wxUpdateUIEvent::CanUpdate(this))
        UpdateWindowUI(wxUPDATE_UI_FROMIDLE);
}
```

wxWindow::Validate

virtual bool Validate()

Validates the current values of the child controls using their validators.

If the window has `wxWS_EX_VALIDATE_RECURSIVELY` extra style flag set, the method will also call Validate() of all child windows.

Return value

Returns `false` if any of the validations failed.

See also

wxWindow::WarpPointer

void WarpPointer(**int** *x*, **int** *y*)

Moves the pointer to the given position on the window.

NB: This function is not supported under Mac because Apple Human Interface Guidelines forbid moving the mouse cursor programmatically.

Parameters

x

> The new x position for the cursor.

y

> The new y position for the cursor.

wxWindowDC

A wxWindowDC must be constructed if an application wishes to paint on the whole area of a window (client and decorations). This should normally be constructed as a temporary stack object; don't store a wxWindowDC object.

To draw on a window from inside **OnPaint**, construct a *wxPaintDC* (p. 1024) object.

To draw on the client area of a window from outside **OnPaint**, construct a *wxClientDC* (p. 147) object.

To draw on the whole window including decorations, construct a *wxWindowDC* (p. 1547) object (Windows only).

Derived from

wxDC (p. 365)

Include files

<wx/dcclient.h>

See also

wxDC (p. 365), *wxMemoryDC* (p. 935), *wxPaintDC* (p. 1024), *wxClientDC* (p. 147), *wxScreenDC* (p. 1142)

wxWindowDC::wxWindowDC

 wxWindowDC(**wxWindow*** *window*)

Constructor. Pass a pointer to the window on which you wish to paint.

wxWindowDisabler

This class disables all windows of the application (may be with the exception of one of them) in its constructor and enables them back in its destructor. This comes in handy when you want to indicate to the user that the application is currently busy and cannot respond to user input.

Derived from

None

Include files

<wx/utils.h>

See also

wxBusyCursor (p. 115)

wxWindowDisabler::wxWindowDisabler

 wxWindowDisabler(**wxWindow ****winToSkip = NULL*)

Disables all top level windows of the applications with the exception of *winToSkip* if it is not `NULL`.

wxWindowDisabler::~wxWindowDisabler

~wxWindowDisabler()

Reenables back the windows disabled by the constructor.

wxWizard

wxWizard is the central class for implementing 'wizard-like' dialogs. These dialogs are mostly familiar to Windows users and are nothing other than a sequence of 'pages', each displayed inside a dialog which has the buttons to navigate to the next (and previous) pages.

The wizards are typically used to decompose a complex dialog into several simple steps and are mainly useful to the novice users, hence it is important to keep them as simple as possible.

To show a wizard dialog, you must first create an instance of the wxWizard class using either the non-default constructor or a default one followed by call to the *Create* (p. 1551) function. Then you should add all pages you want the wizard to show and call *RunWizard* (p. 1553). Finally, don't forget to call `wizard->Destroy()`.

Derived from

wxDialog (p. 406)
wxPanel (p. 1030)
wxWindow (p. 1490)
wxEvtHandler (p. 486)
wxObject (p. 1012)

Include files

<wx/wizard.h>

Event table macros

To process input from a wizard dialog, use these event handler macros to direct input to member functions that take a *wxWizardEvent* (p. 1554) argument. For some events, *Veto()* (p. 1012) can be called to prevent the event from happening.

EVT_WIZARD_PAGE_CHANGED(id, func) The page has just been changed (this event cannot be vetoed).

EVT_WIZARD_PAGE_CHANGING(id, func) The page is being changed (this event can be vetoed).

EVT_WIZARD_CANCEL(id, func) The user attempted to cancel the wizard (this event may also be vetoed).

EVT_WIZARD_HELP(id, func)   The wizard help button was pressed.

EVT_WIZARD_FINISHED(id, func)   The wizard finished button was pressed.

Extended styles

Use the *wxWindow::SetExtraStyle* (p. 1533) function to set the following style. You will need to use two-step construction (use the default constructor, call **SetExtraStyle**, then call **Create**).

wxWIZARD_EX_HELPBUTTON   Shows a Help button using wxID_HELP.

See also *wxDialog* (p. 406) for other extended styles.

See also

wxWizardEvent (p. 1554), *wxWizardPage* (p. 1555), *wxWizard sample* (p. 1717)

wxWizard::wxWizard

 wxWizard()

Default constructor. Use this if you wish to derive from wxWizard and then call *Create* (p. 1551), for example if you wish to set an extra style with *wxWindow::SetExtraStyle* (p. 1533) between the two calls.

 wxWizard(**wxWindow*** *parent*, **int** *id = -1*, **const wxString&** *title = wxEmptyString*, **const wxBitmap&** *bitmap = wxNullBitmap*, **const wxPoint&** *pos = wxDefaultPosition*, **long** *style = wxDEFAULT_DIALOG_STYLE*)

Constructor which really creates the wizard -- if you use this constructor, you shouldn't call *Create* (p. 1551).

Notice that unlike almost all other wxWidgets classes, there is no *size* parameter in the wxWizard constructor because the wizard will have a predefined default size by default. If you want to change this, you should use the *GetPageAreaSizer* (p. 1552) function.

Parameters

parent

The parent window, may be NULL.

id

The id of the dialog, will usually be just -1.

title

The title of the dialog.

bitmap

The default bitmap used in the left side of the wizard. See also *GetBitmap* (p. 1557).

pos

The position of the dialog, it will be centered on the screen by default.

style

Window style is passed to wxDialog.

wxWizard::Create

bool Create(**wxWindow*** *parent*, **int** *id = -1*, **const wxString&** *title = wxEmptyString*, **const wxBitmap&** *bitmap = wxNullBitmap*, **const wxPoint&** *pos = wxDefaultPosition*, **long** *style = wxDEFAULT_DIALOG_STYLE*)

Creates the wizard dialog. Must be called if the default constructor had been used to create the object.

Notice that unlike almost all other wxWidgets classes, there is no *size* parameter in the wxWizard constructor because the wizard will have a predefined default size by default. If you want to change this, you should use the *GetPageAreaSizer* (p. 1552) function.

Parameters

parent

The parent window, may be NULL.

id

The id of the dialog, will usually be just -1.

title

The title of the dialog.

bitmap

The default bitmap used in the left side of the wizard. See also *GetBitmap* (p. 1557).

pos

The position of the dialog, it will be centered on the screen by default.

style

Window style is passed to wxDialog.

wxWizard::FitToPage

void FitToPage(**const wxWizardPage*** *firstPage*)

This method is obsolete, use*GetPageAreaSizer* (p. 1552) instead.

Sets the page size to be big enough for all the pages accessible via the given *firstPage*, i.e. this page, its next page and so on.

This method may be called more than once and it will only change the page size if the size required by the new page is bigger than the previously set one. This is useful if the decision about which pages to show is taken during run-time, as in this case, the wizard won't be able to get to all pages starting from a single one and you should call *Fit* separately for the others.

wxWizard::GetCurrentPage

wxWizardPage* GetCurrentPage() **const**

Get the current page while the wizard is running. NULL is returned if *RunWizard()* (p. 1553) is not being executed now.

wxWizard::GetPageAreaSizer

virtual wxSizer* GetPageAreaSizer() **const**

Returns pointer to page area sizer. The wizard is laid out using sizers and the page area sizer is the place-holder for the pages. All pages are resized before being shown to match the wizard page area.

Page area sizer has a minimal size that is the maximum of several values. First, all pages (or other objects) added to the sizer. Second, all pages reachable by repeatedly applying *wxWizardPage::GetNext* (p. 1556) to any page inserted into the sizer. Third, the minimal size specified using *SetPageSize* (p. 1553) and *FitToPage* (p. 1551). Fourth, the total wizard height may be increased to accommodate the bitmap height. Fifth and finally, wizards are never smaller than some built-in minimal size to avoid wizards that are too small.

The caller can use *wxSizer::SetMinSize* (p. 1184) to enlarge it beyond the minimal size. If wxRESIZE_BORDER was passed to constructor, user can resize wizard and consequently the page area (but not make it smaller than the minimal size).

It is recommended to add the first page to the page area sizer. For simple wizards, this will enlarge the wizard to fit the biggest page. For non-linear wizards, the first page of every separate chain should be added. Caller-specified size can be accomplished using *wxSizer::SetMinSize* (p. 1184).

Adding pages to the page area sizer affects the default border width around page area that can be altered with *SetBorder* (p. 1554).

wxWizard::GetPageSize

wxSize GetPageSize() **const**

Returns the size available for the pages.

wxWizard::HasNextPage

virtual bool HasNextPage(**wxWizardPage ****page*)

Return `true` if this page is not the last one in the wizard. The base class version implements this by calling *page->GetNext* (p. 1556) but this could be undesirable if, for example, the pages are created on demand only.

See also

HasPrevPage (p. 1553)

wxWizard::HasPrevPage

virtual bool HasPrevPage(**wxWizardPage ****page*)

Returns `true` if this page is not the last one in the wizard. The base class version implements this by calling *page->GetPrev* (p. 1556) but this could be undesirable if, for example, the pages are created on demand only.

See also

HasNextPage (p. 1553)

wxWizard::RunWizard

bool RunWizard(**wxWizardPage*** *firstPage*)

Executes the wizard starting from the given page, returning `true` if it was successfully finished or `false` if user cancelled it. The *firstPage* can not be `NULL`.

wxWizard::SetPageSize

void SetPageSize(**const wxSize&** *sizePage*)

This method is obsolete, use*GetPageAreaSizer* (p. 1552) instead.

Sets the minimal size to be made available for the wizard pages. The wizard will take into account the size of the bitmap (if any) itself. Also, the wizard will never be smaller than the default size.

The recommended way to use this function is to lay out all wizard pages using the sizers (even though the wizard is not resizeable) and then use *wxSizer::CalcMin* (p. 1180) in a loop to calculate the maximum of minimal sizes of the pages and pass it to SetPageSize().

wxWizard::SetBorder

void SetBorder(**int** *border*)

Sets width of border around page area. Default is zero. For backward compatibility, if there are no pages in*GetPageAreaSizer* (p. 1552), the default is 5 pixels.

If there is a five point border around all controls in a page and the border around page area is left as zero, a five point white space along all dialog borders will be added to the control border in order to space page controls ten points from the dialog border and non-page controls.

wxWizardEvent

wxWizardEvent class represents an event generated by the*wizard* (p. 1549): this event is first sent to the page itself and, if not processed there, goes up the window hierarchy as usual.

Derived from

wxNotifyEvent (p. 1011)
wxCommandEvent (p. 180)
wxEvent (p. 483)
wxObject (p. 1012)

Include files

<wx/wizard.h>

Event table macros

To process input from a wizard dialog, use these event handler macros to direct input to member functions that take a wxWizardEvent argument.

EVT_WIZARD_PAGE_CHANGED(id, func)	The page has been just changed (this event can not be vetoed).
EVT_WIZARD_PAGE_CHANGING(id, func)	The page is being changed (this event can be vetoed).
EVT_WIZARD_CANCEL(id, func)	The user attempted to cancel the wizard (this event may also be vetoed).
EVT_WIZARD_HELP(id, func)	The wizard help button was pressed.
EVT_WIZARD_FINISHED(id, func)	The wizard finished button was pressed.

See also

wxWizard (p. 1549), *wxWizard sample* (p. 1717)

wxWizardEvent::wxWizardEvent

 wxWizardEvent(**wxEventType** *type = wxEVT_NULL*, **int** *id = -1*, **bool** *direction = true*)

Constructor. It is not normally used by the user code as the objects of this type are constructed by wxWizard.

wxWizardEvent::GetDirection

bool GetDirection() **const**

Return the direction in which the page is changing: for `EVT_WIZARD_PAGE_CHANGING`, return `true` if we're going forward or false otherwise and for `EVT_WIZARD_PAGE_CHANGED` return `true` if we came from the previous page and `false` if we returned from the next one.

wxWizardEvent::GetPage

wxWizardPage* GetPage() **const**

Returns the *wxWizardPage* (p. 1555) which was active when this event was generated.

wxWizardPage

wxWizardPage is one of the screens in *wxWizard* (p. 1549): it must know what are the following and preceding pages (which may be `NULL` for the first/last page). Except for this extra knowledge, wxWizardPage is just a panel, so the controls may be placed directly on it in the usual way.

This class allows the programmer to decide the order of pages in the wizard dynamically (during run-time) and so provides maximal flexibility. Usually, however, the order of pages is known in advance in which case *wxWizardPageSimple* (p. 1557) class is enough and it is simpler to use.

Virtual functions to override

To use this class, you must override *GetPrev* (p. 1556) and *GetNext* (p. 1556) pure virtual functions (or you may use *wxWizardPageSimple* (p. 1557) instead).

GetBitmap (p. 1557) can also be overridden, but this should be very rarely needed.

Derived from

wxPanel (p. 1030)
wxWindow (p. 1490)
wxEvtHandler (p. 486)
wxObject (p. 1012)

Include files

<wx/wizard.h>

See also

wxWizard (p. 1549), *wxWizard sample* (p. 1717)

wxWizardPage::wxWizardPage

 wxWizardPage(**wxWizard*** *parent*, **const wxBitmap&** *bitmap* = *wxNullBitmap*, **const wxChar** **resource = NULL*)

Constructor accepts an optional bitmap which will be used for this page instead of the default one for this wizard (note that all bitmaps used should be of the same size). Notice that no other parameters are needed because the wizard will resize and reposition the page anyhow.

Parameters

parent

> The parent wizard

bitmap

> The page-specific bitmap if different from the global one

resource

> Load the page from the specified resource if non NULL

wxWizardPage::GetPrev

wxWizardPage* GetPrev() **const**

Get the page which should be shown when the user chooses the `"Back"`button: if `NULL` is returned, this button will be disabled. The first page of the wizard will usually return `NULL` from here, but the others will not.

See also

GetNext (p. 1556)

wxWizardPage::GetNext

wxWizardPage* GetNext() **const**

Get the page which should be shown when the user chooses the `"Next"`button: if `NULL` is returned, this button will be disabled. The last page of the wizard will usually return

`NULL` from here, but the others will not.

---

wxWizardPage::GetBitmap

wxBitmap GetBitmap() **const**

This method is called by wxWizard to get the bitmap to display alongside the page. By default, `m_bitmap` member variable which was set in the *constructor* (p. 1556).

If the bitmap was not explicitly set (i.e. if `wxNullBitmap` is returned), the default bitmap for the wizard should be used.

The only cases when you would want to override this function is if the page bitmap depends dynamically on the user choices, i.e. almost never.

wxWizardPageSimple

wxWizardPageSimple is the simplest possible *wxWizardPage* (p. 1555) implementation: it just returns the pointers given to its constructor from GetNext() and GetPrev() functions.

This makes it very easy to use the objects of this class in the wizards where the pages order is known statically - on the other hand, if this is not the case you must derive your own class from *wxWizardPage* (p. 1555) instead.

Derived from

wxWizardPage (p. 1555)
wxPanel (p. 1030)
wxWindow (p. 1490)
wxEvtHandler (p. 486)
wxObject (p. 1012)

Include files

<wx/wizard.h>

---

wxWizardPageSimple::wxWizardPageSimple

wxWizardPageSimple(**wxWizard*** *parent = NULL*, **wxWizardPage*** *prev = NULL*,

---

wxWizardPage* *next* = *NULL*, **const wxBitmap&** *bitmap* = *wxNullBitmap*)

Constructor takes the previous and next pages. They may be modified later by*SetPrev()* (p. 1558) or *SetNext()* (p. 1558).

wxWizardPageSimple::SetPrev

void SetPrev(**wxWizardPage*** *prev*)

Sets the previous page.

wxWizardPageSimple::SetNext

void SetNext(**wxWizardPage*** *next*)

Sets the next page.

wxWizardPageSimple::Chain

static void Chain(**wxWizardPageSimple*** *first*, **wxWizardPageSimple*** *second*)

A convenience function to make the pages follow each other.

Example:

```
wxRadioboxPage *page3 = new wxRadioboxPage(wizard);
wxValidationPage *page4 = new wxValidationPage(wizard);

wxWizardPageSimple::Chain(page3, page4);
```

wxXmlResource

This is the main class for interacting with the XML-based resource system.

The class holds XML resources from one or more .xml files, binary files or zip archive files.

See *XML-based resource system overview* (p. 1777) for details.

Derived from

wxObject (p. 1012)

Include files

<wx/xrc/xmlres.h>

Data structures

```
enum wxXmlResourceFlags
```

```
    {
        wxXRC_USE_LOCALE     = 1,
        wxXRC_NO_SUBCLASSING = 2
    };
```

wxXmlResource::wxXmlResource

wxXmlResource(**const wxString&** *filemask*, **int** *flags* = *wxXRC_USE_LOCALE*)

Constructor.

filemask

> The XRC file, archive file, or wildcard specification that will be used to load all resource files inside a zip archive.

flags

> wxXRC_USE_LOCALE: translatable strings will be translated via _().
> wxXRC_NO_SUBCLASSING: subclass property of object nodes will be ignored (useful for previews in XRC editors).

wxXmlResource(**int** *flags* = *wxXRC_USE_LOCALE*)

Constructor.

flags

> wxXRC_USE_LOCALE: translatable strings will be translated via _().
> wxXRC_NO_SUBCLASSING: subclass property of object nodes will be ignored (useful for previews in XRC editors).

wxXmlResource::~wxXmlResource

~wxXmlResource()

Destructor.

wxXmlResource::AddHandler

void AddHandler(**wxXmlResourceHandler*** *handler*)

Initializes only a specific handler (or custom handler). Convention says that the handler name is equal to the control's name plus 'XmlHandler', for example wxTextCtrlXmlHandler, wxHtmlWindowXmlHandler. The XML resource compiler (wxxrc) can create include file that contains initialization code for all controls used within the resource.

wxXmlResource::AttachUnknownControl

bool AttachUnknownControl(**const wxString&** *name*, **wxWindow*** *control*, **wxWindow*** *parent* = *NULL*)

Attaches an unknown control to the given panel/window/dialog. Unknown controls are used in conjunction with <object class="unknown">.

wxXmlResource::ClearHandlers

void ClearHandlers()

Removes all handlers.

wxXmlResource::CompareVersion

int CompareVersion(**int** *major*, **int** *minor*, **int** *release*, **int** *revision*) **const**

Compares the XRC version to the argument. Returns -1 if the XRC version is less than the argument, +1 if greater, and 0 if they equal.

wxXmlResource::Get

wxXmlResource* Get()

Gets the global resources object or creates one if none exists.

wxXmlResource::GetFlags

int GetFlags()

Returns flags, which may be a bitlist of wxXRC_USE_LOCALE and wxXRC_NO_SUBCLASSING.

wxXmlResource::GetVersion

long GetVersion() **const**

Returns version information (a.b.c.d = d+ 256*c + 2562*b + 2563*a).

wxXmlResource::GetXRCID

int GetXRCID(**const wxChar*** *str_id*)

Returns a numeric ID that is equivalent to the string ID used in an XML resource. To be used in event tables. The macro XRCID(name) is provided for convenience.

wxXmlResource::InitAllHandlers

void InitAllHandlers()

Initializes handlers for all supported controls/windows. This will make the executable quite big because it forces linking against most of the wxWidgets library.

wxXmlResource::Load

bool Load(**const wxString&** *filemask*)

Loads resources from XML files that match given filemask. This method understands VFS (see filesys.h).

wxXmlResource::LoadBitmap

wxBitmap LoadBitmap(**const wxString&** *name*)

Loads a bitmap resource from a file.

wxXmlResource::LoadDialog

wxDialog* LoadDialog(**wxWindow*** *parent*, **const wxString&** *name*)

Loads a dialog. *dlg* points to a parent window (if any).

bool LoadDialog(**wxDialog*** *dlg*, **wxWindow*** *parent*, **const wxString&** *name*)

Loads a dialog. *dlg* points to parent window (if any).

This form is used to finish creation of an already existing instance (the main reason for this is that you may want to use derived class with a new event table).

Example:

```
MyDialog dlg;
wxTheXmlResource->LoadDialog(&dlg, mainFrame, "my_dialog");
dlg->ShowModal();
```

wxXmlResource::LoadFrame

bool LoadFrame(**wxFrame*** *frame*, **wxWindow*** *parent*, **const wxString&** *name*)

Loads a frame.

wxXmlResource::LoadIcon

wxIcon LoadIcon(**const wxString&** *name*)

Loads an icon resource from a file.

wxXmlResource::LoadMenu

wxMenu* LoadMenu(**const wxString&** *name*)

Loads menu from resource. Returns NULL on failure.

wxXmlResource::LoadMenuBar

wxMenuBar* LoadMenuBar(**wxWindow*** *parent*, **const wxString&** *name*)

Loads a menubar from resource. Returns NULL on failure.

wxMenuBar* LoadMenuBar(**const wxString&** *name*)

Loads a menubar from resource. Returns NULL on failure.

wxXmlResource::LoadPanel

wxPanel* LoadPanel(**wxWindow*** *parent*, **const wxString&** *name*)

Loads a panel. *panel* points to parent window (if any).

bool LoadPanel(**wxPanel*** *panel*, **wxWindow*** *parent*, **const wxString&** *name*)

Loads a panel. *panel* points to parent window (if any). This form is used to finish creation of an already existing instance.

wxXmlResource::LoadToolBar

wxToolBar* LoadToolBar(**wxWindow*** *parent*, **const wxString&** *name*)

Loads a toolbar.

wxXmlResource::Set

wxXmlResource* Set(**wxXmlResource*** *res*)

Sets the global resources object and returns a pointer to the previous one (may be NULL).

wxXmlResource::SetFlags

void SetFlags(**int** *flags*)

Sets flags (bitlist of wxXRC_USE_LOCALE and wxXRC_NO_SUBCLASSING).

wxXmlResourceHandler

wxXmlResourceHandler is an abstract base class for resource handlers capable of creating a control from an XML node.

See *XML-based resource system overview* (p. 1777) for details.

Derived from

wxObject (p. 1012)

Include files

<wx/xrc/xmlres.h>

wxXmlResourceHandler::wxXmlResourceHandler

wxXmlResourceHandler()

Default constructor.

wxXmlResourceHandler::~wxXmlResourceHandler

~wxXmlResourceHandler()

Destructor.

wxXmlResourceHandler::AddStyle

void AddStyle(**const wxString&** *name*, **int** *value*)

Add a style flag (e.g. wxMB_DOCKABLE) to the list of flags understood by this handler.

wxXmlResourceHandler::AddWindowStyles

void AddWindowStyles()

Add styles common to all wxWindow-derived classes.

wxXmlResourceHandler::CanHandle

bool CanHandle(**wxXmlNode*** *node*)

Returns true if it understands this node and can create a resource from it, false otherwise.

Note

You must **not** call any wxXmlResourceHandler methods except *IsOfClass* (p. 1567) from this method! The instance is not yet initialized with node data at the time CanHandle is called and it is only safe to operate on *node* directly or to call IsOfClass.

wxXmlResourceHandler::CreateChildren

void CreateChildren(**wxObject*** *parent*, **bool** *this_hnd_only = false*)

Creates children.

wxXmlResourceHandler::CreateChildrenPrivately

void CreateChildrenPrivately(**wxObject*** *parent*, **wxXmlNode*** *rootnode = NULL*)

Helper function.

wxXmlResourceHandler::CreateResFromNode

wxObject* CreateResFromNode(**wxXmlNode*** *node*, **wxObject*** *parent*, **wxObject*** *instance = NULL*)

Creates a resource from a node.

wxXmlResourceHandler::CreateResource

wxObject* CreateResource(**wxXmlNode*** *node*, **wxObject*** *parent*, **wxObject*** *instance*)

Creates an object (menu, dialog, control, ...) from an XML node. Should check for validity. *parent* is a higher-level object (usually window, dialog or panel) that is often necessary to create the resource. If **instance** is non-NULL it should not create a new instance via 'new' but should rather use this one, and call its Create method.

wxXmlResourceHandler::DoCreateResource

wxObject* DoCreateResource()

Called from CreateResource after variables were filled.

wxXmlResourceHandler::GetBitmap

wxBitmap GetBitmap(**const wxString&** *param = wxT("bitmap")*, **wxSize** *size = wxDefaultSize*)

Gets a bitmap.

wxXmlResourceHandler::GetBool

bool GetBool(**const wxString&** *param*, **bool** *defaultv = false*)

Gets a bool flag (1, t, yes, on, true are true, everything else is false).

wxXmlResourceHandler::GetColour

wxColour GetColour(**const wxString&** *param*)

Gets colour in HTML syntax (#RRGGBB).

wxXmlResourceHandler::GetCurFileSystem

wxFileSystem& GetCurFileSystem()

Returns the current file system.

wxXmlResourceHandler::GetDimension

wxCoord GetDimension(**const wxString&** *param*, **wxCoord** *defaultv = 0*)

Gets a dimension (may be in dialog units).

wxXmlResourceHandler::GetFont

wxFont GetFont(**const wxString&** *param = wxT("font")*)

Gets a font.

wxXmlResourceHandler::GetID

int GetID()

Returns the XRCID.

wxXmlResourceHandler::GetIcon

wxIcon GetIcon(**const wxString&** *param = wxT("icon")*, **wxSize** *size = wxDefaultSize*)

Returns an icon.

wxXmlResourceHandler::GetLong

long GetLong(**const wxString&** *param*, **long** *defaultv = 0*)

Gets the integer value from the parameter.

wxXmlResourceHandler::GetName

wxString GetName()

Returns the resource name.

wxXmlResourceHandler::GetNodeContent

wxString GetNodeContent(**wxXmlNode*** *node*)

Gets node content from wxXML_ENTITY_NODE.

wxXmlResourceHandler::GetParamNode

wxXmlNode* GetParamNode(**const wxString&** *param*)

Finds the node or returns NULL.

wxXmlResourceHandler::GetParamValue

wxString GetParamValue(**const wxString&** *param*)

Finds the parameter value or returns the empty string.

wxXmlResourceHandler::GetPosition

wxPoint GetPosition(**const wxString&** *param = wxT("pos")*)

Gets the position (may be in dialog units).

wxXmlResourceHandler::GetSize

wxSize GetSize(**const wxString&** *param = wxT("size")*)

Gets the size (may be in dialog units).

wxXmlResourceHandler::GetStyle

int GetStyle(**const wxString&** *param = wxT("style")*, **int** *defaults = 0*)

Gets style flags from text in form "flag | flag2| flag3 |..." Only understands flags added with AddStyle.

wxXmlResourceHandler::GetText

wxString GetText(**const wxString&** *param*)

Gets text from param and does some conversions:

- replaces \n, \r, \t by respective characters (according to C syntax)

- replaces $ by & and $$ by $ (needed for _File to &Filetranslation because of XML syntax)

- calls wxGetTranslations (unless disabled in wxXmlResource)

wxXmlResourceHandler::HasParam

bool HasParam(**const wxString&** *param*)

Check to see if a parameter exists.

wxXmlResourceHandler::IsOfClass

bool IsOfClass(**wxXmlNode*** *node*, **const wxString&** *classname*)

Convenience function. Returns true if the node has a property class equal to classname, e.g. <object class="wxDialog">.

wxXmlResourceHandler::SetParentResource

void SetParentResource(**wxXmlResource*** *res*)

Sets the parent resource.

wxXmlResourceHandler::SetupWindow

void SetupWindow(**wxWindow*** *wnd*)

Sets common window options.

wxZipClassFactory

Class factory for the zip archive format. See the base class for details.

Derived from

wxArchiveClassFactory (p. 42)

Include files

<wx/zipstrm.h>

See also

Archive formats such as zip (p. 1879)
Generic archive programming (p. 1883)*wxZipEntry* (p. 1568)
wxZipInputStream (p. 1574)
wxZipOutputStream (p. 1577)

wxZipEntry

Holds the meta-data for an entry in a zip.

Derived from

wxArchiveEntry (p. 44)

Include files

<wx/zipstrm.h>

Data structures

Constants for *Get/SetMethod* (p. 1573):

```
// Compression Method, only 0 (store) and 8 (deflate) are supported here
//
enum wxZipMethod
{
    wxZIP_METHOD_STORE,
    wxZIP_METHOD_SHRINK,
    wxZIP_METHOD_REDUCE1,
    wxZIP_METHOD_REDUCE2,
    wxZIP_METHOD_REDUCE3,
    wxZIP_METHOD_REDUCE4,
    wxZIP_METHOD_IMPLODE,
    wxZIP_METHOD_TOKENIZE,
    wxZIP_METHOD_DEFLATE,
    wxZIP_METHOD_DEFLATE64,
    wxZIP_METHOD_BZIP2 = 12,
    wxZIP_METHOD_DEFAULT = 0xffff
};
```

Constants for *Get/SetSystemMadeBy* (p. 1574):

```
// Originating File-System.
//
```

```
// These are Pkware's values. Note that Info-zip disagree on some of them,
// most notably NTFS.
//
enum wxZipSystem
{
    wxZIP_SYSTEM_MSDOS,
    wxZIP_SYSTEM_AMIGA,
    wxZIP_SYSTEM_OPENVMS,
    wxZIP_SYSTEM_UNIX,
    wxZIP_SYSTEM_VM_CMS,
    wxZIP_SYSTEM_ATARI_ST,
    wxZIP_SYSTEM_OS2_HPFS,
    wxZIP_SYSTEM_MACINTOSH,
    wxZIP_SYSTEM_Z_SYSTEM,
    wxZIP_SYSTEM_CPM,
    wxZIP_SYSTEM_WINDOWS_NTFS,
    wxZIP_SYSTEM_MVS,
    wxZIP_SYSTEM_VSE,
    wxZIP_SYSTEM_ACORN_RISC,
    wxZIP_SYSTEM_VFAT,
    wxZIP_SYSTEM_ALTERNATE_MVS,
    wxZIP_SYSTEM_BEOS,
    wxZIP_SYSTEM_TANDEM,
    wxZIP_SYSTEM_OS_400
};
```

Constants for *Get/SetExternalAttributes* (p. 1571):

```
// Dos/Win file attributes
//
enum wxZipAttributes
{
    wxZIP_A_RDONLY = 0x01,
    wxZIP_A_HIDDEN = 0x02,
    wxZIP_A_SYSTEM = 0x04,
    wxZIP_A_SUBDIR = 0x10,
    wxZIP_A_ARCH   = 0x20,

    wxZIP_A_MASK   = 0x37
};
```

Constants for *Get/SetFlags* (p. 1572):

```
// Values for the flags field in the zip headers
//
enum wxZipFlags
{
    wxZIP_ENCRYPTED         = 0x0001,
    wxZIP_DEFLATE_NORMAL    = 0x0000,   // normal compression
    wxZIP_DEFLATE_EXTRA     = 0x0002,   // extra compression
    wxZIP_DEFLATE_FAST      = 0x0004,   // fast compression
    wxZIP_DEFLATE_SUPERFAST = 0x0006,   // superfast compression
    wxZIP_DEFLATE_MASK      = 0x0006,
    wxZIP_SUMS_FOLLOW       = 0x0008,   // crc and sizes come after the
data
    wxZIP_ENHANCED          = 0x0010,
    wxZIP_PATCH             = 0x0020,
    wxZIP_STRONG_ENC        = 0x0040,
    wxZIP_UNUSED            = 0x0F80,
    wxZIP_RESERVED          = 0xF000
```

```
      };
```

See also

Archive formats such as zip (p. 1879)
wxZipInputStream (p. 1574)
wxZipOutputStream (p. 1577)
wxZipNotifier (p. 1576)

Field availability

When reading a zip from a stream that is seekable, *GetNextEntry()* (p. 1576) returns a fully populated wxZipEntry object except for *wxZipEntry::GetLocalExtra()* (p. 1573). GetLocalExtra() becomes available when the entry is opened, either by calling *wxZipInputStream::OpenEntry* (p. 1576) or by making an attempt to read the entry's data.

For zips on *non-seekable* (p. 1884) streams, the following fields are always available when GetNextEntry() returns:

GetDateTime (p. 45)
GetInternalFormat (p. 45)
GetInternalName (p. 1572)
GetFlags (p. 1572)
GetLocalExtra (p. 1573)
GetMethod (p. 1573)
GetName (p. 45)
GetOffset (p. 45)
IsDir (p. 46)

The following fields are also usually available when GetNextEntry() returns, however, if the zip was also written to a non-seekable stream the zipper is permitted to store them after the entry's data. In that case they become available when the entry's data has been read to Eof(), or *CloseEntry()* (p. 47) has been called. `(GetFlags() & wxZIP_SUMS_FOLLOW) != 0` indicates that one or more of these come after the data:

GetCompressedSize (p. 1571)
GetCrc (p. 1571)
GetSize (p. 46)

The following are stored at the end of the zip, and become available when the end of the zip has been reached, i.e. after GetNextEntry() returns NULL and Eof() is true:

GetComment (p. 1571)
GetExternalAttributes (p. 1571)
GetExtra (p. 1572)
GetMode (p. 1573)
GetSystemMadeBy (p. 1574)
IsReadOnly (p. 46)
IsMadeByUnix (p. 1574)
IsText (p. 1574)

wxZipEntry::wxZipEntry

 wxZipEntry(**const wxString&** *name = wxEmptyString*, **const wxDateTime&** *dt = wxDateTime::Now()*, **off_t** *size = wxInvalidOffset*)

Constructor.

 wxZipEntry(**const wxZipEntry&** *entry*)

Copy constructor.

wxZipEntry::Clone

wxZipEntry* Clone() **const**

Make a copy of this entry.

wxZipEntry::Get/SetComment

wxString GetComment() **const**

void SetComment(**const wxString&** *comment*)

A short comment for this entry.

wxZipEntry::GetCompressedSize

off_t GetCompressedSize() **const**

The compressed size of this entry in bytes.

wxZipEntry::GetCrc

wxUint32 GetCrc() **const**

CRC32 for this entry's data.

wxZipEntry::Get/SetExternalAttributes

wxUint32 GetExternalAttributes() **const**

void SetExternalAttributes(**wxUint32** *attr*)

The low 8 bits are always the DOS/Windows file attributes for this entry. The values of

these attributes are given in the enumeration `wxZipAttributes`.

The remaining bits can store platform specific permission bits or attributes, and their meaning depends on the value of *SetSystemMadeBy()* (p. 1574). If *IsMadeByUnix()* (p. 1574) is true then the high 16 bits are unix mode bits.

The following other accessors access these bits:

IsReadOnly/SetIsReadOnly (p. 46)
IsDir/SetIsDir (p. 46)
Get/SetMode (p. 1573)

wxZipEntry::Get/SetExtra

const char* GetExtra() **const**

size_t GetExtraLen() **const**

void SetExtra(**const char*** *extra*, **size_t** *len*)

The extra field from the entry's central directory record.

The extra field is used to store platform or application specific data. See Pkware's document 'appnote.txt' for information on its format.

wxZipEntry::GetFlags

int GetFlags() **const**

Returns a combination of the bits flags in the enumeration `wxZipFlags`.

wxZipEntry::GetInternalName

wxString GetInternalName() **const**

Returns the entry's filename in the internal format used within the archive. The name can include directory components, i.e. it can be a full path.

The names of directory entries are returned without any trailing path separator. This gives a canonical name that can be used in comparisons.

wxString GetInternalName(**const wxString&** *name*, **wxPathFormat** *format* = *wxPATH_NATIVE*, **bool*** *pIsDir = NULL*)

A static member that translates a filename into the internal format used within the archive. If the third parameter is provided, the bool pointed to is set to indicate whether the name looks like a directory name (i.e. has a trailing path separator).

See also

Looking up an archive entry by name (p. 1881)

wxZipEntry::Get/SetLocalExtra

const char* GetLocalExtra() **const**

size_t GetLocalExtraLen() **const**

void SetLocalExtra(**const char*** *extra*, **size_t** *len*)

The extra field from the entry's local record.

The extra field is used to store platform or application specific data. See Pkware's document 'appnote.txt' for information on its format.

wxZipEntry::Get/SetMethod

int GetMethod() **const**

void SetMethod(**int** *method*)

The compression method. The enumeration `wxZipMethod` lists the possible values.

The default constructor sets this to wxZIP_METHOD_DEFAULT, which allows *wxZipOutputStream* (p. 1577) to choose the method when writing the entry.

wxZipEntry::Get/SetMode

int GetMode() **const**

If *IsMadeByUnix()* (p. 1574) is true then returns the unix permission bits stored in *GetExternalAttributes()* (p. 1571). Otherwise synthesises them from the DOS attributes.

void SetMode(**int** *mode*)

Sets the DOS attributes in *GetExternalAttributes()* (p. 1571)to be consistent with the `mode` given.

If *IsMadeByUnix()* (p. 1574) is true then also stores `mode` in GetExternalAttributes().

Note that the default constructor sets *GetSystemMadeBy()* (p. 1574) to wxZIP_SYSTEM_MSDOS by default. So to be able to store unix permissions when creating zips, call SetSystemMadeBy(wxZIP_SYSTEM_UNIX).

wxZipEntry::SetNotifier

void SetNotifier(**wxZipNotifier&** *notifier*)

void UnsetNotifier()

Sets the *notifier* (p. 1576) for this entry. Whenever the *wxZipInputStream* (p. 1574) updates this entry, it will then invoke the associated notifier's *OnEntryUpdated* (p. 1577)method.

Setting a notifier is not usually necessary. It is used to handle certain cases when modifying an zip in a pipeline (i.e. between non-seekable streams).

See also

Archives on non-seekable streams (p. 1884)
wxZipNotifier (p. 1576)

wxZipEntry::Get/SetSystemMadeBy

int GetSystemMadeBy() **const**

void SetSystemMadeBy(**int** *system*)

The originating file-system. The default constructor sets this to wxZIP_SYSTEM_MSDOS. Set it to wxZIP_SYSTEM_UNIX in order to be able to store unix permissions using *SetMode()* (p. 1573).

wxZipEntry::IsMadeByUnix

bool IsMadeByUnix() **const**

Returns true if *GetSystemMadeBy()* (p. 1574)is a flavour of unix.

wxZipEntry::IsText/SetIsText

bool IsText() **const**

void SetIsText(**bool** *isText = true*)

Indicates that this entry's data is text in an 8-bit encoding.

wxZipEntry::operator=

wxZipEntry& operator operator=(**const wxZipEntry&** *entry*)

Assignment operator.

wxZipInputStream

Input stream for reading zip files.

GetNextEntry() (p. 1576) returns an *wxZipEntry* (p. 1568) object containing the meta-

data for the next entry in the zip (and gives away ownership). Reading from the wxZipInputStream then returns the entry's data. Eof() becomes true after an attempt has been made to read past the end of the entry's data. When there are no more entries, GetNextEntry() returns NULL and sets Eof().

Note that in general zip entries are not seekable, and wxZipInputStream::SeekI() always returns wxInvalidOffset.

Derived from

wxArchiveInputStream (p. 47)

Include files

<wx/zipstrm.h>

> **Data structures**typedef wxZipEntry entry_type

See also

Archive formats such as zip (p. 1879)
wxZipEntry (p. 1568)
wxZipOutputStream (p. 1577)

wxZipInputStream::wxZipInputStream

wxZipInputStream(**wxInputStream&** *stream*, **wxMBConv&** *conv* = *wxConvLocal*)

Constructor. In a Unicode build the second parameter `conv` is used to translate the filename and comment fields into Unicode. It has no effect on the stream's data.

wxZipInputStream(**const wxString&** *archive*, **const wxString&** *file*)

Compatibility constructor.

When this constructor is used, an emulation of seeking is switched on for compatibility with previous versions. Note however, that it is deprecated.

wxZipInputStream::CloseEntry

bool CloseEntry()

Closes the current entry. On a non-seekable stream reads to the end of the current entry first.

wxZipInputStream::GetComment

wxString GetComment()

Returns the zip comment.

This is stored at the end of the zip, therefore when reading a zip from a non-seekable stream, it returns the empty string until the end of the zip has been reached, i.e. when GetNextEntry() returns NULL.

wxZipInputStream::GetNextEntry

wxZipEntry* GetNextEntry()

Closes the current entry if one is open, then reads the meta-data for the next entry and returns it in a *wxZipEntry* (p. 1568)object, giving away ownership. The stream is then open and can be read.

wxZipInputStream::GetTotalEntries

int GetTotalEntries()

For a zip on a seekable stream returns the total number of entries in the zip. For zips on non-seekable streams returns the number of entries returned so far by *GetNextEntry()* (p. 1576).

wxZipInputStream::OpenEntry

bool OpenEntry(**wxZipEntry&** *entry*)

Closes the current entry if one is open, then opens the entry specified by the *entry* object.

entry should be from the same zip file, and the zip should be on a seekable stream.

See also

Looking up an archive entry by name (p. 1881)

wxZipNotifier

If you need to know when a *wxZipInputStream* (p. 1574)updates a *wxZipEntry* (p. 1568), you can create a notifier by deriving from this abstract base class, overriding *OnEntryUpdated()* (p. 1577). An instance of your notifier class can then be assigned to wxZipEntry objects, using *wxZipEntry::SetNotifier()* (p. 1573).

Setting a notifier is not usually necessary. It is used to handle certain cases when modifying an zip in a pipeline (i.e. between non-seekable streams). See '*Archives on non-seekable streams* (p. 1884)'.

Derived from

No base class

Include files

<wx/zipstrm.h>

See also

wxZipNotifier::OnEntryUpdated

void OnEntryUpdated(**wxZipEntry&** *entry*)

Override this to receive notifications when an *wxZipEntry* (p. 1568) object changes.

wxZipOutputStream

Output stream for writing zip files.

PutNextEntry() (p. 1579) is used to create a new entry in the output zip, then the entry's data is written to the wxZipOutputStream. Another call to PutNextEntry() closes the current entry and begins the next.

Derived from

wxArchiveOutputStream (p. 51)

Include files

<wx/zipstrm.h>

See also

wxZipOutputStream::wxZipOutputStream

 wxZipOutputStream(**wxOutputStream&** *stream*, **int** *level = -1*, **wxMBConv&** *conv = wxConvLocal*)

Constructor. `level` is the compression level to use. It can be a value between 0 and 9 or -1 to use the default value which currently is equivalent to 6.

In a Unicode build the third parameter `conv` is used to translate the filename and comment fields to Unicode. It has no effect on the stream's data.

wxZipOutputStream::~wxZipOutputStream

~wxZipOutputStream()

The destructor calls *Close()* (p. 1578) to finish writing the zip if it has not been called already.

wxZipOutputStream::Close

bool Close()

Finishes writing the zip, returning true if successfully. Called by the destructor if not called explicitly.

wxZipOutputStream::CloseEntry

bool CloseEntry()

Close the current entry. It is called implicitly whenever another new entry is created with *CopyEntry()* (p. 1578)or *PutNextEntry()* (p. 1579), or when the zip is closed.

wxZipOutputStream::CopyArchiveMetaData

bool CopyArchiveMetaData(**wxZipInputStream&** *inputStream*)

Transfers the zip comment from the *wxZipInputStream* (p. 1574)to this output stream.

wxZipOutputStream::CopyEntry

bool CopyEntry(**wxZipEntry*** *entry*, **wxZipInputStream&** *inputStream*)

Takes ownership of `entry` and uses it to create a new entry in the zip. `entry` is then opened in `inputStream` and its contents copied to this stream.

CopyEntry() is much more efficient than transferring the data using Read() and Write() since it will copy them without decompressing and recompressing them.

For zips on seekable streams, `entry` must be from the same zip file as `stream`. For non-seekable streams, `entry` must also be the last thing read from `inputStream`.

wxZipOutputStream::Get/SetLevel

int GetLevel() **const**

void SetLevel(**int** *level*)

Set the compression level that will be used the next time an entry is created. It can be a value between 0 and 9 or -1 to use the default value which currently is equivalent to 6.

wxZipOutputStream::PutNextDirEntry

bool PutNextDirEntry(**const wxString&** *name*, **const wxDateTime&** *dt* = *wxDateTime::Now()*)

Create a new directory entry (see *wxArchiveEntry::IsDir()* (p. 46)) with the given name and timestamp.

PutNextEntry() (p. 1579) can also be used to create directory entries, by supplying a name with a trailing path separator.

wxZipOutputStream::PutNextEntry

bool PutNextEntry(**wxZipEntry*** *entry*)

Takes ownership of `entry` and uses it to create a new entry in the zip.

bool PutNextEntry(**const wxString&** *name*, **const wxDateTime&** *dt* = *wxDateTime::Now()*, **off_t** *size* = *wxInvalidOffset*)

Create a new entry with the given name, timestamp and size.

wxZipOutputStream::SetComment

void SetComment(**const wxString&** *comment*)

Sets a comment for the zip as a whole. It is written at the end of the zip.

wxZlibInputStream

This filter stream decompresses a stream that is in zlib or gzip format. Note that reading the gzip format requires zlib version 1.2.1 or greater, (the builtin version does support gzip format).

The stream is not seekable, *SeekI()* (p. 816) returns *wxInvalidOffset*. Also *GetSize()* (p. 1276) is not supported, it always returns 0.

Derived from

wxFilterInputStream (p. 549)

Include files

<wx/zstream.h>

See also

wxInputStream (p. 814)*, wxZlibOutputStream* (p. 1580).

wxZlibInputStream::wxZlibInputStream

 wxZlibInputStream(**wxInputStream&** *stream*, **int** *flags* = *wxZLIB_AUTO*)

The *flags* wxZLIB_ZLIB and wxZLIB_GZIP specify whether the input data is in zlib or gzip format. If wxZLIB_AUTO is used, then zlib will autodetect the stream type, this is the default.

If *flags* is wxZLIB_NO_HEADER, then the data is assumed to be a raw deflate stream without either zlib or gzip headers. This is a lower level mode, which is not usually used directly. It can be used to read a raw deflate stream embedded in a higher level protocol.

This version is not by default compatible with the output produced by the version of *wxZlibOutputStream* in wxWidgets 2.4.x. However, there is a compatibility mode, which is switched on by passing wxZLIB_24COMPATIBLE for flags. Note that in when operating in compatibility mode error checking is very much reduced. The following symbols can be use for the flags:

```
// Flags
enum {
#if WXWIN_COMPATIBILITY_2_4
    wxZLIB_24COMPATIBLE = 4, // read v2.4.x data without error
#endif
    wxZLIB_NO_HEADER = 0,    // raw deflate stream, no header or checksum
    wxZLIB_ZLIB = 1,         // zlib header and checksum
    wxZLIB_GZIP = 2,         // gzip header and checksum, requires zlib
1.2.1+
    wxZLIB_AUTO = 3          // autodetect header zlib or gzip
};
```

wxZlibInputStream::CanHandleGZip

static bool CanHandleGZip()

Returns true if zlib library in use can handle gzip compressed data.

wxZlibOutputStream

This stream compresses all data written to it. The compressed output can be in zlib or

gzip format. Note that writing the gzip format requires zlib version 1.2.1 or greater (the builtin version does support gzip format).

The stream is not seekable, *SeekO()* (p. 1017) returns *wxInvalidOffset*.

Derived from

wxFilterOutputStream (p. 549)

Include files

<wx/zstream.h>

See also

wxOutputStream (p. 1016), *wxZlibInputStream* (p. 1579)

wxZlibOutputStream::wxZlibOutputStream

 wxZlibOutputStream(**wxOutputStream&** *stream*, **int** *level = -1*, **int** *flags = wxZLIB_ZLIB*)

Creates a new write-only compressed stream. *level* means level of compression. It is number between 0 and 9 (including these values) where 0 means no compression and 9 best but slowest compression. -1 is default value (currently equivalent to 6).

The *flags* wxZLIB_ZLIB and wxZLIB_GZIP specify whether the output data will be in zlib or gzip format. wxZLIB_ZLIB is the default.

If *flags* is wxZLIB_NO_HEADER, then a raw deflate stream is output without either zlib or gzip headers. This is a lower level mode, which is not usually used directly. It can be used to embed a raw deflate stream in a higher level protocol.

The following symbols can be use for the compression level and flags:

```
// Compression level
enum {
    wxZ_DEFAULT_COMPRESSION = -1,
    wxZ_NO_COMPRESSION = 0,
    wxZ_BEST_SPEED = 1,
    wxZ_BEST_COMPRESSION = 9
};

// Flags
enum {
    wxZLIB_NO_HEADER = 0,   // raw deflate stream, no header or checksum
    wxZLIB_ZLIB = 1,        // zlib header and checksum
    wxZLIB_GZIP = 2         // gzip header and checksum, requires zlib
1.2.1+
};
```

wxZlibOutputStream::CanHandleGZip

static bool CanHandleGZip()

Returns true if zlib library in use can handle gzip compressed data.

Chapter 8 Functions

The functions and macros defined in wxWidgets are described here: you can either look up a function using the alphabetical listing of them or find it in the corresponding topic.

Alphabetical functions and macros list

_ (p. 1613)
_*T* (p. 1613)

Version macros

The following constants are defined in wxWidgets:

- `wxMAJOR_VERSION` is the major version of wxWidgets

- `wxMINOR_VERSION` is the minor version of wxWidgets

- `wxRELEASE_NUMBER` is the release number

- `wxSUBRELEASE_NUMBER` is the subrelease number which is 0 for all official releases

For example, the values or these constants for wxWidgets 2.1.15 are 2, 1 and 15.

Additionally, `wxVERSION_STRING` is a user-readable string containing the full wxWidgets version and `wxVERSION_NUMBER` is a combination of the three version numbers above: for 2.1.15, it is 2115 and it is 2200 for wxWidgets 2.2.

The subrelease number is only used for the sources in between official releases and so normally is not useful.

Include files

<wx/version.h> or <wx/defs.h>

wxCHECK_GCC_VERSION

bool wxCHECK_GCC_VERSION(*major, minor, release*)

Returns 1 if the compiler being used to compile the code is GNU C++ compiler (g++) version major.minor.release or greater. Otherwise, and also if the compiler is not GNU C++ at all, returns 0.

wxCHECK_VERSION

bool wxCHECK_VERSION(*major, minor, release*)

This is a macro which evaluates to true if the current wxWidgets version is at least major.minor.release.

For example, to test if the program is compiled with wxWidgets 2.2 or higher, the following can be done:

```
    wxString s;
#if wxCHECK_VERSION(2, 2, 0)
    if ( s.StartsWith("foo") )
```

```
#else // replacement code for old version
    if ( strncmp(s, "foo", 3) == 0 )
#endif
    {
        ...
    }
```

wxCHECK_VERSION_FULL

bool wxCHECK_VERSION_FULL(*major, minor, release, subrel*)

Same as *wxCHECK_VERSION* (p. 1588) but also checks that `wxSUBRELEASE_NUMBER` is at least *subrel*.

wxCHECK_W32API_VERSION

bool wxCHECK_GCC_VERSION(*major, minor, release*)

Returns 1 if the version of w32api headers used is major.minor.release or greater. Otherwise, and also if we are not compiling with mingw32/cygwin under Win32 at all, returns 0.

Application initialization and termination

The functions in this section are used on application startup/shutdown and also to control the behaviour of the main event loop of the GUI programs.

::wxEntry

This initializes wxWidgets in a platform-dependent way. Use this if you are not using the default wxWidgets entry code (e.g. main or WinMain). For example, you can initialize wxWidgets from an Microsoft Foundation Classes application using this function.

void wxEntry(**HANDLE** *hInstance*, **HANDLE** *hPrevInstance*, **const wxString&** *commandLine*, **int** *cmdShow*, **bool** *enterLoop* = *true*)

wxWidgets initialization under Windows (non-DLL). If *enterLoop* is false, the function will return immediately after calling wxApp::OnInit. Otherwise, the wxWidgets message loop will be entered.

void wxEntry(**HANDLE** *hInstance*, **HANDLE** *hPrevInstance*, **WORD** *wDataSegment*, **WORD** *wHeapSize*, **const wxString&** *commandLine*)

wxWidgets initialization under Windows (for applications constructed as a DLL).

int wxEntry(**int** *argc*, **const wxString&** *argv)

wxWidgets initialization under Unix.

Remarks

To clean up wxWidgets, call wxApp::OnExit followed by the static function wxApp::CleanUp. For example, if exiting from an MFC application that also uses wxWidgets:

```
int CTheApp::ExitInstance()
{
  // OnExit isn't called by CleanUp so must be called explicitly.
  wxTheApp->OnExit();
  wxApp::CleanUp();

  return CWinApp::ExitInstance();
}
```

Include files

<wx/app.h>

::wxGetApp

wxAppDerivedClass& wxGetApp()

This function doesn't exist in wxWidgets but it is created by using the *IMPLEMENT_APP* (p. 1644) macro. Thus, before using it anywhere but in the same module where this macro is used, you must make it available using *DECLARE_APP* (p. 1642).

The advantage of using this function compared to directly using the global wxTheApp pointer is that the latter is of type `wxApp *` and so wouldn't allow you to access the functions specific to your application class but not present in wxApp while wxGetApp() returns the object of the right type.

::wxHandleFatalExceptions

bool wxHandleFatalExceptions(bool *doIt* = *true*)

If *doIt* is true, the fatal exceptions (also known as general protection faults under Windows or segmentation violations in the Unix world) will be caught and passed to *wxApp::OnFatalException* (p. 37). By default, i.e. before this function is called, they will be handled in the normal way which usually just means that the application will be terminated. Calling wxHandleFatalExceptions() with *doIt* equal to false will restore this default behaviour.

::wxInitAllImageHandlers

void wxInitAllImageHandlers()

Initializes all available image handlers. For a list of available handlers, see *wxImage* (p. 779).

See also

wxImage (p. 779), *wxImageHandler* (p. 802)

Include files

<wx/image.h>

::wxInitialize

bool wxInitialize()

This function is used in wxBase only and only if you don't create*wxApp* (p. 31) object at all. In this case you must call it from your`main()` function before calling any other wxWidgets functions.

If the function returns `false` the initialization could not be performed, in this case the library cannot be used and*wxUninitialize* (p. 1591) shouldn't be called neither.

This function may be called several times but*wxUninitialize* (p. 1591) must be called for each successful call to this function.

Include files

<wx/app.h>

::wxSafeYield

bool wxSafeYield(**wxWindow*** *win* = *NULL*, **bool** *onlyIfNeeded* = *false*)

This function is similar to wxYield, except that it disables the user input to all program windows before calling wxYield and re-enables it again afterwards. If *win* is not NULL, this window will remain enabled, allowing the implementation of some limited user interaction.

Returns the result of the call to *::wxYield* (p. 1592).

Include files

<wx/utils.h>

::wxUninitialize

void wxUninitialize()

This function is for use in console (wxBase) programs only. It must be called once for each previous successful call to *wxInitialize* (p. 1591).

Include files

<wx/app.h>

::wxYield

bool wxYield()

Calls *wxApp::Yield* (p. 42).

This function is kept only for backwards compatibility. Please use the *wxApp::Yield* (p. 42) method instead in any new code.

Include files

<wx/app.h> or <wx/utils.h>

::wxWakeUpIdle

void wxWakeUpIdle()

This functions wakes up the (internal and platform dependent) idle system, i.e. it will force the system to send an idle event even if the system currently *is* idle and thus would not send any idle event until after some other event would get sent. This is also useful for sending events between two threads and is used by the corresponding functions *::wxPostEvent* (p. 1638) and*wxEvtHandler::AddPendingEvent* (p. 487).

Include files

<wx/event.h>

Process control functions

The functions in this section are used to launch or terminate the other processes.

::wxExecute

long wxExecute(**const wxString&** *command*, **int** *sync = wxEXEC_ASYNC*, **wxProcess *****callback = NULL*)

wxPerl note: In wxPerl this function is called `Wx::ExecuteCommand`

long wxExecute(**char *****argv*, **int** *flags = wxEXEC_ASYNC*, **wxProcess *****callback = NULL*)

wxPerl note: In wxPerl this function is called `Wx::ExecuteArgs`

long wxExecute(**const wxString&** *command*, **wxArrayString&** *output*, **int** *flags = 0*)

wxPerl note: In wxPerl this function is called `Wx::ExecuteStdout` and it only takes the `command` argument, and returns a 2-element list (`status, output`), where `output` is an array reference.

long wxExecute(**const wxString&** *command*, **wxArrayString&** *output*,

wxArrayString& *errors*, **int** *flags = 0*)

wxPerl note: In wxPerl this function is called `Wx::ExecuteStdoutStderr`and it only takes the `command` argument, and returns a 3-element list (`status, output, errors`), where`output` and `errors` are array references.

Executes another program in Unix or Windows.

The first form takes a command string, such as `"emacs file.txt"`.

The second form takes an array of values: a command, any number of arguments, terminated by NULL.

The semantics of the third and fourth versions is different from the first two and is described in more details below.

If *flags* parameter contains `wxEXEC_ASYNC` flag (the default), flow of control immediately returns. If it contains `wxEXEC_SYNC`, the current application waits until the other program has terminated.

In the case of synchronous execution, the return value is the exit code of the process (which terminates by the moment the function returns) and will be-1 if the process couldn't be started and typically 0 if the process terminated successfully. Also, while waiting for the process to terminate, wxExecute will call *wxYield* (p. 1592). Because of this, by default this function disables all application windows to avoid unexpected reentrancies which could result from the users interaction with the program while the child process is running. If you are sure that it is safe to not disable the program windows, you may pass `wxEXEC_NODISABLE` flag to prevent this automatic disabling from happening.

For asynchronous execution, however, the return value is the process id and zero value indicates that the command could not be executed. As an added complication, the return value of -1 in this case indicates that we didn't launch a new process, but connected to the running one (this can only happen in case of using DDE under Windows for command execution). In particular, in this, and only this, case the calling code will not get the notification about process termination.

If callback isn't NULL and if execution is asynchronous,*wxProcess::OnTerminate* (p. 1077) will be called when the process finishes. Specifying this parameter also allows you to redirect the standard input and/or output of the process being launched by calling*Redirect* (p. 1078). If the child process IO is redirected, under Windows the process window is not shown by default (this avoids having to flush an unnecessary console for the processes which don't create any windows anyhow) but a `wxEXEC_NOHIDE` flag can be used to prevent this from happening, i.e. with this flag the child process window will be shown normally.

Under Unix the flag `wxEXEC_MAKE_GROUP_LEADER` may be used to ensure that the new process is a group leader (this will create a new session if needed). Calling *wxKill* (p. 1594) passing wxKILL_CHILDREN will will kill this process as well as all of its children (except those which have started their own session).

Finally, you may use the third overloaded version of this function to execute a process

(always synchronously, the contents of *flags* is or'd with wxEXEC_SYNC) and capture its output in the array *output*. The fourth version adds the possibility to additionally capture the messages from standard error output in the *errors* array.

NB: Currently wxExecute() can only be used from the main thread, calling this function from another thread will result in an assert failure in debug build and won't work.

See also

wxShell (p. 1596), *wxProcess* (p. 1073), *Exec sample* (p. 1711).

Parameters

command

> The command to execute and any parameters to pass to it as a single string.

argv

> The command to execute should be the first element of this array, any additional ones are the command parameters and the array must be terminated with a NULL pointer.

flags

> Combination of bit masks wxEXEC_ASYNC, wxEXEC_SYNC and wxEXEC_NOHIDE

callback

> An optional pointer to *wxProcess* (p. 1073)

Include files

<wx/utils.h>

::wxExit

void wxExit()

Exits application after calling *wxApp::OnExit* (p. 37). Should only be used in an emergency: normally the top-level frame should be deleted (after deleting all other frames) to terminate the application. See *wxCloseEvent* (p. 153) and *wxApp* (p. 31).

Include files

<wx/app.h>

::wxKill

int wxKill(**long** *pid*, **int** *sig = wxSIGTERM*, **wxKillError** **rc = NULL*, **int** *flags = 0*)

Equivalent to the Unix kill function: send the given signal *sig* to the process with PID *pid*.

The valid signal values are

```
enum wxSignal
{
    wxSIGNONE = 0,  // verify if the process exists under Unix
    wxSIGHUP,
    wxSIGINT,
    wxSIGQUIT,
    wxSIGILL,
    wxSIGTRAP,
    wxSIGABRT,
    wxSIGEMT,
    wxSIGFPE,
    wxSIGKILL,      // forcefully kill, dangerous!
    wxSIGBUS,
    wxSIGSEGV,
    wxSIGSYS,
    wxSIGPIPE,
    wxSIGALRM,
    wxSIGTERM       // terminate the process gently
};
```

`wxSIGNONE`, `wxSIGKILL` and `wxSIGTERM` have the same meaning under both Unix and Windows but all the other signals are equivalent to `wxSIGTERM` under Windows.

Returns 0 on success, -1 on failure. If *rc* parameter is not NULL, it will be filled with an element of `wxKillError` enum:

```
enum wxKillError
{
    wxKILL_OK,              // no error
    wxKILL_BAD_SIGNAL,      // no such signal
    wxKILL_ACCESS_DENIED,   // permission denied
    wxKILL_NO_PROCESS,      // no such process
    wxKILL_ERROR            // another, unspecified error
};
```

The *flags* parameter can be wxKILL_NOCHILDREN (the default), or wxKILL_CHILDREN, in which case the child processes of this process will be killed too. Note that under Unix, for wxKILL_CHILDREN to work you should have created the process by passing wxEXEC_MAKE_GROUP_LEADER to wxExecute.

See also

wxProcess::Kill (p. 1076), *wxProcess::Exists* (p. 1076), *Exec sample* (p. 1711)

Include files

<wx/utils.h>

::wxGetProcessId

unsigned long wxGetProcessId()

Returns the number uniquely identifying the current process in the system.

If an error occurs, 0 is returned.

Include files

<wx/utils.h>

::wxShell

bool wxShell(**const wxString&** *command = NULL*)

Executes a command in an interactive shell window. If no command is specified, then just the shell is spawned.

See also *wxExecute* (p. 1592), *Exec sample* (p. 1711).

Include files

<wx/utils.h>

::wxShutdown

bool wxShutdown(**wxShutdownFlags***flags*)

This function shuts down or reboots the computer depending on the value of the*flags*. Please notice that doing this requires the corresponding access rights (superuser under Unix, SE_SHUTDOWN privelege under Windows NT) and that this function is only implemented under Unix and Win32.

Parameters

flags

> Either wxSHUTDOWN_POWEROFF or wxSHUTDOWN_REBOOT

Returns

true on success, false if an error occurred.

Include files

<wx/utils.h>

Thread functions

The functions and macros here mainly exist to make it writing the code which may be compiled in multi thread build (wxUSE_THREADS = 1) as well as in single thread configuration (wxUSE_THREADS = 0).

For example, a static variable must be protected against simultaneous access by multiple threads in the former configuration but in the latter the extra overhead of using the critical section is not needed. To solve this problem, the *wxCRITICAL_SECTION* (p. 1597) macro may be used to create and use the critical section only when needed.

Include files

<wx/thread.h>

See also

wxThread (p. 1377), *wxMutex* (p. 994), *Multithreading overview* (p. 1816)

wxCRIT_SECT_DECLARE

wxCRIT_SECT_DECLARE(*cs*)

This macro declares a (static) critical section object named *cs* if wxUSE_THREADS is 1 and does nothing if it is 0.

wxCRIT_SECT_DECLARE_MEMBER

wxCRIT_SECT_DECLARE(*cs*)

This macro declares a critical section object named *cs* if wxUSE_THREADS is 1 and does nothing if it is 0. As it doesn't include the static keyword (unlike *wxCRIT_SECT_DECLARE* (p. 1597)), it can be used to declare a class or struct member which explains its name.

wxCRIT_SECT_LOCKER

wxCRIT_SECT_LOCKER(*name*, *cs*)

This macro creates a *critical section lock* (p. 224) object named *name* and associated with the critical section *cs* if wxUSE_THREADS is 1 and does nothing if it is 0.

wxCRITICAL_SECTION

wxCRITICAL_SECTION(*name*)

This macro combines *wxCRIT_SECT_DECLARE* (p. 1597) and *wxCRIT_SECT_LOCKER* (p. 1597): it creates a static critical section object and also the lock object associated with it. Because of this, it can be only used inside a function, not at global scope. For example:

```
int IncCount()
{
    static int s_counter = 0;

    wxCRITICAL_SECTION(counter);

    return ++s_counter;
}
```

(note that we suppose that the function is called the first time from the main thread so that the critical section object is initialized correctly by the time other threads start calling it, if this is not the case this approach can**not** be used and the critical section must be made a global instead).

wxENTER_CRIT_SECT

wxENTER_CRIT_SECT(**wxCriticalSection&** *cs*)

This macro is equivalent to *cs.Enter()* (p. 224) if`wxUSE_THREADS` is 1 and does nothing if it is 0.

::wxIsMainThread

bool wxIsMainThread()

Returns `true` if this thread is the main one. Always returns `true` if`wxUSE_THREADS` is 0.

wxLEAVE_CRIT_SECT

wxLEAVE_CRIT_SECT(**wxCriticalSection&** *cs*)

This macro is equivalent to *cs.Leave()* (p. 224) if`wxUSE_THREADS` is 1 and does nothing if it is 0.

::wxMutexGuiEnter

void wxMutexGuiEnter()

This function must be called when any thread other than the main GUI thread wants to get access to the GUI library. This function will block the execution of the calling thread until the main thread (or any other thread holding the main GUI lock) leaves the GUI library and no other thread will enter the GUI library until the calling thread calls *::wxMutexGuiLeave()* (p. 1599).

Typically, these functions are used like this:

```
void MyThread::Foo(void)
{
    // before doing any GUI calls we must ensure that this thread is the
only
    // one doing it!

    wxMutexGuiEnter();

    // Call GUI here:
    my_window->DrawSomething();

    wxMutexGuiLeave();
```

```
    }
```

Note that under GTK, no creation of top-level windows is allowed in any thread but the main one.

This function is only defined on platforms which support preemptive threads.

::wxMutexGuiLeave

void wxMutexGuiLeave()

See *::wxMutexGuiEnter()* (p. 1598).

This function is only defined on platforms which support preemptive threads.

File functions

Include files

<wx/filefn.h>

See also

wxPathList (p. 1035)
wxDir (p. 420)
wxFile (p. 502)
wxFileName (p. 521)

::wxDos2UnixFilename

void wxDos2UnixFilename(**wxChar ****s*)

Converts a DOS to a Unix filename by replacing backslashes with forward slashes.

::wxFileExists

bool wxFileExists(**const wxString&** *filename*)

Returns true if the file exists and is a plain file.

::wxFileModificationTime

time_t wxFileModificationTime(**const wxString&** *filename*)

Returns time of last modification of given file.

::wxFileNameFromPath

wxString wxFileNameFromPath(**const wxString&** *path*)

char * wxFileNameFromPath(**char ***path*)

NB: This function is obsolete, please use*wxFileName::SplitPath* (p. 537) instead.

Returns the filename for a full path. The second form returns a pointer to temporary storage that should not be deallocated.

::wxFindFirstFile

wxString wxFindFirstFile(**const char ***spec*, **int** *flags = 0*)

This function does directory searching; returns the first file that matches the path *spec*, or the empty string. Use *wxFindNextFile* (p. 1600) to get the next matching file. Neither will report the current directory "." or the parent directory "..".

Warning

As of wx 2.5.2, these functions are not thread-safe! (they use static variables). You probably want to use *wxDir::GetFirst* (p. 422) or *wxDirTraverser* (p. 426) instead.

spec may contain wildcards.

flags may be wxDIR for restricting the query to directories, wxFILE for files or zero for either.

For example:

```
wxString f = wxFindFirstFile("/home/project/*.*");
while ( !f.empty() )
{
  ...
  f = wxFindNextFile();
}
```

::wxFindNextFile

wxString wxFindNextFile()

Returns the next file that matches the path passed to *wxFindFirstFile* (p. 1600).

See *wxFindFirstFile* (p. 1600) for an example.

::wxGetDiskSpace

bool wxGetDiskSpace(**const wxString&** *path*, **wxLongLong** **total = NULL*, **wxLongLong** **free = NULL*)

This function returns the total number of bytes and number of free bytes on the disk containing the directory *path* (it should exist). Both*total* and *free* parameters may be NULL if the corresponding information is not needed.

Returns

true on success, false if an error occurred (for example, the directory doesn't exist).

Portability

This function is implemented for Win32, Mac OS and generic Unix provided the system has statfs() function.

This function first appeared in wxWidgets 2.3.2.

::wxGetFileKind

wxFileKind wxGetFileKind(**int** *fd*)

wxFileKind wxGetFileKind(**FILE ****fp*)

Returns the type of an open file. Possible return values are:

```
enum wxFileKind
{
  wxFILE_KIND_UNKNOWN,
  wxFILE_KIND_DISK,     // a file supporting seeking to arbitrary offsets
  wxFILE_KIND_TERMINAL, // a tty
  wxFILE_KIND_PIPE      // a pipe
};
```

Include files

<wx/filefn.h>

::wxGetOSDirectory

wxString wxGetOSDirectory()

Returns the Windows directory under Windows; on other platforms returns the empty string.

::wxIsAbsolutePath

bool wxIsAbsolutePath(**const wxString&** *filename*)

Returns true if the argument is an absolute filename, i.e. with a slash or drive name at the beginning.

::wxDirExists

bool wxDirExists(**const wxString&** *dirname*)

Returns true if the path exists.

::wxPathOnly

wxString wxPathOnly(**const wxString&** *path*)

Returns the directory part of the filename.

::wxUnix2DosFilename

void wxUnix2DosFilename(**const wxString&** *s*)

Converts a Unix to a DOS filename by replacing forward slashes with backslashes.

wxCHANGE_UMASK

wxCHANGE_UMASK(**int** *mask*)

Under Unix this macro changes the current process umask to the given value, unless it is equal to -1 in which case nothing is done, and restores it to the original value on scope exit. It works by declaring a variable which sets umask to *mask* in its constructor and restores it in its destructor.

Under other platforms this macro expands to nothing.

::wxConcatFiles

bool wxConcatFiles(**const wxString&** *file1*, **const wxString&** *file2*,**const wxString&** *file3*)

Concatenates *file1* and *file2* to *file3*, returning true if successful.

::wxCopyFile

bool wxCopyFile(**const wxString&** *file1*, **const wxString&** *file2*, **bool** *overwrite = true*)

Copies *file1* to *file2*, returning true if successful. If*overwrite* parameter is true (default), the destination file is overwritten if it exists, but if *overwrite* is false, the functions fails in this case.

::wxGetCwd

wxString wxGetCwd()

Returns a string containing the current (or working) directory.

::wxGetWorkingDirectory

wxString wxGetWorkingDirectory(**char** **buf=NULL*, **int** *sz=1000*)

NB: This function is obsolete: use *wxGetCwd* (p. 1602) instead.

Copies the current working directory into the buffer if supplied, or copies the working directory into new storage (which you *must* delete yourself) if the buffer is NULL.

sz is the size of the buffer if supplied.

::wxGetTempFileName

char * wxGetTempFileName(**const wxString&** *prefix*, **char** **buf=NULL*)

bool wxGetTempFileName(**const wxString&** *prefix*, **wxString&** *buf*)

NB: These functions are obsolete, please use *wxFileName::CreateTempFileName* (p. 526) instead.

::wxIsWild

bool wxIsWild(**const wxString&** *pattern*)

Returns true if the pattern contains wildcards. See *wxMatchWild* (p. 1603).

::wxMatchWild

bool wxMatchWild(**const wxString&** *pattern*, **const wxString&** *text*, **bool** *dot_special*)

Returns true if the *pattern* matches the *text*; if *dot_special* is true, filenames beginning with a dot are not matched with wildcard characters. See *wxIsWild* (p. 1603).

::wxMkdir

bool wxMkdir(**const wxString&** *dir*, **int** *perm = 0777*)

Makes the directory *dir*, returning true if successful.

perm is the access mask for the directory for the systems on which it is supported (Unix) and doesn't have effect for the other ones.

::wxParseCommonDialogsFilter

int wxParseCommonDialogsFilter(**const wxString&** *wildCard*, **wxArrayString&** *descriptions*, **wxArrayString&** *filters*)

```
Parses the wildCard, returning the number of filters. Returns 0 if none or
if there's a problem. The arrays will contain an equal number of items
found before the error. On platforms where native dialogs handle only one
filter per entry, entries in arrays are automatically adjusted.wildCard is
in the form: "All files (*)|*|Image Files (*.jpeg *.png)|*.jpg;*.png"
```

::wxRemoveFile

bool wxRemoveFile(**const wxString&** *file*)

Removes *file*, returning true if successful.

::wxRenameFile

bool wxRenameFile(**const wxString&** *file1*, **const wxString&** *file2*)

Renames *file1* to *file2*, returning true if successful.

::wxRmdir

bool wxRmdir(**const wxString&** *dir*, **int** *flags=0*)

Removes the directory *dir*, returning true if successful. Does not work under VMS.

The *flags* parameter is reserved for future use.

::wxSetWorkingDirectory

bool wxSetWorkingDirectory(**const wxString&** *dir*)

Sets the current working directory, returning true if the operation succeeded. Under MS Windows, the current drive is also changed if *dir* contains a drive specification.

::wxSplitPath

void wxSplitPath(**const char *** *fullname*, **wxString *** *path*, **wxString *** *name*, **wxString *** *ext*)

NB: This function is obsolete, please use*wxFileName::SplitPath* (p. 537) instead.

This function splits a full file name into components: the path (including possible disk/drive specification under Windows), the base name and the extension. Any of the output parameters (*path*, *name* or *ext*) may be NULL if you are not interested in the

value of a particular component.

wxSplitPath() will correctly handle filenames with both DOS and Unix path separators under Windows, however it will not consider backslashes as path separators under Unix (where backslash is a valid character in a filename).

On entry, *fullname* should be non-NULL (it may be empty though).

On return, *path* contains the file path (without the trailing separator), *name*contains the file name and *ext* contains the file extension without leading dot. All three of them may be empty if the corresponding component is. The old contents of the strings pointed to by these parameters will be overwritten in any case (if the pointers are not NULL).

::wxTransferFileToStream

bool wxTransferFileToStream(**const wxString&** *filename*, **ostream&** *stream*)

Copies the given file to *stream*. Useful when converting an old application to use streams (within the document/view framework, for example).

Include files

<wx/docview.h>

::wxTransferStreamToFile

bool wxTransferStreamToFile(**istream&** *stream* **const wxString&** *filename*)

Copies the given stream to the file *filename*. Useful when converting an old application to use streams (within the document/view framework, for example).

Include files

<wx/docview.h>

Network, user and OS functions

The functions in this section are used to retrieve information about the current computer and/or user characteristics.

::wxGetEmailAddress

wxString wxGetEmailAddress()

bool wxGetEmailAddress(**char *** *buf,* **int** *sz*)

Copies the user's email address into the supplied buffer, by concatenating the values returned by *wxGetFullHostName* (p. 1606) and *wxGetUserId* (p. 1609).

Returns true if successful, false otherwise.

Include files

<wx/utils.h>

::wxGetFreeMemory

wxMemorySize wxGetFreeMemory()

Returns the amount of free memory in bytes under environments which support it, and -1 if not supported or failed to perform measurement.

Include files

<wx/utils.h>

::wxGetFullHostName

wxString wxGetFullHostName()

Returns the FQDN (fully qualified domain host name) or an empty string on error.

See also

wxGetHostName (p. 1606)

Include files

<wx/utils.h>

::wxGetHomeDir

wxString wxGetHomeDir()

Return the (current) user's home directory.

See also

wxGetUserHome (p. 1608)

Include files

<wx/utils.h>

::wxGetHostName

wxString wxGetHostName()

bool wxGetHostName(**char *** *buf*, **int** *sz*)

Copies the current host machine's name into the supplied buffer. Please note that the returned name is *not* fully qualified, i.e. it does not include the domain name.

Under Windows or NT, this function first looks in the environment variable SYSTEM_NAME; if this is not found, the entry **HostName** in the **wxWidgets** section of the WIN.INI file is tried.

The first variant of this function returns the hostname if successful or an empty string otherwise. The second (deprecated) function returns true if successful, false otherwise.

See also

wxGetFullHostName (p. 1606)

Include files

<wx/utils.h>

::wxGetOsDescription

wxString wxGetOsDescription()

Returns the string containing the description of the current platform in a user-readable form. For example, this function may return strings like `Windows NT Version 4.0` or `Linux 2.2.2 i386`.

See also

::wxGetOsVersion (p. 1607)

Include files

<wx/utils.h>

::wxGetOsVersion

int wxGetOsVersion(**int ****major* = *NULL*, **int ****minor* = *NULL*)

Gets operating system version information.

Platform	Return types
Mac OS	Return value is wxMAC when compiled with CodeWarrior under Mac OS 8.x/9.x and Mac OS X, wxMAC_DARWIN when compiled with the Apple Developer Tools under Mac OS X.
	Both *major* and *minor* have to be looked at as hexadecimal numbers. So System 10.2.4 returns 0x10, resp 16 for *major* and 0x24, resp 36 for *minor*.

GTK	Return value is wxGTK, For GTK 1.0, *major* is 1, *minor* is 0.
Motif	Return value is wxMOTIF_X, *major* is X version, *minor* is X revision.
OS/2	Return value is wxOS2_PM.
Windows 3.1	Return value is wxWINDOWS, *major* is 3, *minor* is 1.
Windows NT/2000	Return value is wxWINDOWS_NT, version is returned in *major* and *minor*.
Windows 98	Return value is wxWIN95, *major* is 4, *minor* is 1 or greater.
Windows 95	Return value is wxWIN95, *major* is 4, *minor* is 0.
Win32s (Windows 3.1)	Return value is wxWIN32S, *major* is 3, *minor* is 1.
Windows PocketPC	Return value is wxWINDOWS_POCKETPC, version is returned in *major* and *minor*.
Windows Smartphone	Return value is wxWINDOWS_SMARTPHONE, version is returned in *major* and *minor*.
Windows CE (non-specific)	Return value is wxWINDOWS_CE, version is returned in *major* and *minor*.
Watcom C++ 386 supervisor mode (Windows 3.1)	Return value is wxWIN386, *major* is 3, *minor* is 1.

See also

::wxGetOsDescription (p. 1607)

Include files

<wx/utils.h>

::wxGetUserHome

const wxChar * wxGetUserHome(**const wxString&** *user = ""*)

Returns the home directory for the given user. If the username is empty (default value), this function behaves like*wxGetHomeDir* (p. 1606).

Include files

<wx/utils.h>

::wxGetUserId

wxString wxGetUserId()

bool wxGetUserId(**char *** *buf*, **int** *sz*)

This function returns the "user id" also known as "login name" under Unix i.e. something like "jsmith". It uniquely identifies the current user (on this system).

Under Windows or NT, this function first looks in the environment variables USER and LOGNAME; if neither of these is found, the entry **UserId** in the **wxWidgets** section of the WIN.INI file is tried.

The first variant of this function returns the login name if successful or an empty string otherwise. The second (deprecated) function returns true if successful, false otherwise.

See also

wxGetUserName (p. 1609)

Include files

<wx/utils.h>

::wxGetUserName

wxString wxGetUserName()

bool wxGetUserName(**char *** *buf*, **int** *sz*)

This function returns the full user name (something like "Mr. John Smith").

Under Windows or NT, this function looks for the entry **UserName** in the **wxWidgets** section of the WIN.INI file. If PenWindows is running, the entry **Current** in the section **User** of the PENWIN.INI file is used.

The first variant of this function returns the user name if successful or an empty string otherwise. The second (deprecated) function returns `true`if successful, `false` otherwise.

See also

wxGetUserId (p. 1609)

Include files

<wx/utils.h>

String functions

::copystring

char * copystring(**const char ***s)

Makes a copy of the string *s* using the C++ new operator, so it can be deleted with the *delete* operator.

This function is deprecated, use *wxString* (p. 1284) class instead.

::wxGetTranslation

const char * wxGetTranslation(**const char *** *str*)

const char * wxGetTranslation(**const char *** *str*, **const char *** *strPlural*, **size_t** *n*)

This function returns the translation of string *str* in the current*locale* (p. 882). If the string is not found in any of the loaded message catalogs (see *internationalization overview* (p. 1736)), the original string is returned. In debug build, an error message is logged -- this should help to find the strings which were not yet translated. As this function is used very often, an alternative (and also common in Unix world) syntax is provided: the *_()* (p. 1613) macro is defined to do the same thing as wxGetTranslation.

The second form is used when retrieving translation of string that has different singular and plural form in English or different plural forms in some other language. It takes two extra arguments: *str*parameter must contain the singular form of the string to be converted. It is also used as the key for the search in the catalog. The *strPlural* parameter is the plural form (in English). The parameter *n* is used to determine the plural form. If no message catalog is found *str* is returned if 'n == 1', otherwise *strPlural*. See GNU gettext manual (`http://www.gnu.org/manual/gettext/html_chapter/gettext_10.html#S EC150`) for additional information on plural forms handling.

Both versions call *wxLocale::GetString* (p. 886).

::wxIsEmpty

bool wxIsEmpty(**const char *** *p*)

Returns `true` if the pointer is either `NULL` or points to an empty string, `false` otherwise.

::wxStrcmp

int wxStrcmp(**const char ***p1, **const char ***p2)

Returns a negative value, 0, or positive value if *p1* is less than, equal to or greater than *p2*. The comparison is case-sensitive.

This function complements the standard C function *stricmp()* which performs case-insensitive comparison.

::wxStricmp

int wxStricmp(**const char** **p1*, **const char** **p2*)

Returns a negative value, 0, or positive value if *p1* is less than, equal to or greater than *p2*. The comparison is case-insensitive.

This function complements the standard C function *strcmp()* which performs case-sensitive comparison.

::wxStringMatch

bool wxStringMatch(**const wxString&** *s1*, **const wxString&** *s2*,
  bool *subString = true*, **bool** *exact = false*)

NB: This function is obsolete, use *wxString::Find* (p. 1295) instead.

Returns `true` if the substring *s1* is found within *s2*, ignoring case if *exact* is false. If *subString* is `false`, no substring matching is done.

::wxStringEq

bool wxStringEq(**const wxString&** *s1*, **const wxString&** *s2*)

NB: This function is obsolete, use *wxString* (p. 1284) instead.

A macro defined as:

```
#define wxStringEq(s1, s2) (s1 && s2 && (strcmp(s1, s2) == 0))
```

::wxStrlen

size_t wxStrlen(**const char** * *p*)

This is a safe version of standard function *strlen()*: it does exactly the same thing (i.e. returns the length of the string) except that it returns 0 if*p* is the `NULL` pointer.

::wxSnprintf

int wxSnprintf(**wxChar** **buf*, **size_t** *len*, **const wxChar** **format*, *...*)

This function replaces the dangerous standard function `sprintf()` and is like `snprintf()` available on some platforms. The only difference with sprintf() is that an additional argument - buffer size - is taken and the buffer is never overflowed.

Returns the number of characters copied to the buffer or -1 if there is not enough space.

See also

wxVsnprintf (p. 1613), *wxString::Printf* (p. 1301)

wxT

wxChar wxT(**char** *ch*)

const wxChar * wxT(**const char ***s*)

wxT() is a macro which can be used with character and string literals (in other words, `'x'` or `"foo"`) to automatically convert them to Unicode in Unicode build configuration. Please see the*Unicode overview* (p. 1730) for more information.

```
This macro is simply returns the value passed to it without changes in
ASCII build. In fact, its definition is:#ifdef UNICODE
#define wxT(x) L ## x
#else // !Unicode
#define wxT(x) x
#endif
```

wxTRANSLATE

const wxChar * wxTRANSLATE(**const char ***s*)

This macro doesn't do anything in the program code -- it simply expands to the value of its argument (except in Unicode build where it is equivalent to*wxT* (p. 1612) which makes it unnecessary to use both wxTRANSLATE and wxT with the same string which would be really unreadable).

However it does have a purpose and it is to mark the literal strings for the extraction into the message catalog created by `xgettext` program. Usually this is achieved using *_()* (p. 1613) but that macro not only marks the string for extraction but also expands into a*wxGetTranslation* (p. 1610) function call which means that it cannot be used in some situations, notably for static array initialization.

Here is an example which should make it more clear: suppose that you have a static array of strings containing the weekday names and which have to be translated (note that it is a bad example, really, as*wxDateTime* (p. 257) already can be used to get the localized week day names already). If you write

```
static const wxChar * const weekdays[] = { _("Mon"), ..., _("Sun") };
...
// use weekdays[n] as usual
```

the code wouldn't compile because the function calls are forbidden in the array initializer. So instead you should do

```
static const wxChar * const weekdays[] = { wxTRANSLATE("Mon"), ...,
wxTRANSLATE("Sun") };
...
// use wxGetTranslation(weekdays[n])
```

here.

Note that although the code **would** compile if you simply omit wxTRANSLATE() in the above, it wouldn't work as expected because there would be no translations for the weekday names in the program message catalog and wxGetTranslation wouldn't find them.

::wxVsnprintf

int wxVsnprintf(**wxChar ****buf*, **size_t** *len*, **const wxChar ****format*, **va_list** *argPtr*)

The same as *wxSnprintf* (p. 1611) but takes a `va_list` argument instead of arbitrary number of parameters.

See also

wxSnprintf (p. 1611), *wxString::PrintfV* (p. 1301)

_

const wxChar * _(**const char ****s*)

This macro expands into a call to *wxGetTranslation* (p. 1610)function, so it marks the message for the extraction by `xgettext` just as*wxTRANSLATE* (p. 1612) does, but also returns the translation of the string for the current locale during execution.

Don't confuse this macro with *_T()* (p. 1613)!

_T

wxChar _T(**char** *ch*)

const wxChar * _T(**const wxChar** *ch*)

This macro is exactly the same as *wxT* (p. 1612) and is defined in wxWidgets simply because it may be more intuitive for Windows programmers as the standard Win32 headers also define it (as well as yet another name for the same macro which is `_TEXT()`).

Don't confuse this macro with *_()* (p. 1613)!

Dialog functions

Below are a number of convenience functions for getting input from the user or displaying messages. Note that in these functions the last three parameters are optional. However, it is recommended to pass a parent frame parameter, or (in MS Windows or Motif) the wrong window frame may be brought to the front when the dialog box is popped up.

::wxBeginBusyCursor

void wxBeginBusyCursor(**wxCursor ****cursor* = *wxHOURGLASS_CURSOR*)

Changes the cursor to the given cursor for all windows in the application. Use *wxEndBusyCursor* (p. 1616) to revert the cursor back to its previous state. These two calls can be nested, and a counter ensures that only the outer calls take effect.

See also *wxIsBusy* (p. 1621), *wxBusyCursor* (p. 115).

Include files

<wx/utils.h>

::wxBell

void wxBell()

Ring the system bell.

Include files

<wx/utils.h>

::wxCreateFileTipProvider

wxTipProvider * wxCreateFileTipProvider(**const wxString&** *filename*, **size_t** *currentTip*)

This function creates a *wxTipProvider* (p. 1395) which may be used with *wxShowTip* (p. 1622).

filename

> The name of the file containing the tips, one per line

currentTip

> The index of the first tip to show - normally this index is remembered between the 2 program runs.

See also

Tips overview (p. 1814)

Include files

<wx/tipdlg.h>

::wxDirSelector

wxString wxDirSelector(**const wxString&** *message = wxDirSelectorPromptStr*,
 const wxString& *default_path = ""*,
 long *style = 0*, **const wxPoint&** *pos = wxDefaultPosition*,
 wxWindow **parent = NULL*)

Pops up a directory selector dialog. The arguments have the same meaning as those of wxDirDialog::wxDirDialog(). The message is displayed at the top, and the default_path, if specified, is set as the initial selection.

The application must check for an empty return value (if the user pressed Cancel). For example:

```
const wxString& dir = wxDirSelector("Choose a folder");
if ( !dir.empty() )
{
   ...
}
```

Include files

<wx/dirdlg.h>

::wxFileSelector

wxString wxFileSelector(**const wxString&** *message*, **const wxString&** *default_path = ""*,
 const wxString& *default_filename = ""*, **const wxString&** *default_extension = ""*,
 const wxString& *wildcard = "*.*"*, **int** *flags = 0*, **wxWindow ****parent = NULL*,
 int *x = -1*, **int** *y = -1*)

Pops up a file selector box. In Windows, this is the common file selector dialog. In X, this is a file selector box with the same functionality. The path and filename are distinct elements of a full file pathname. If path is empty, the current directory will be used. If filename is empty, no default filename will be supplied. The wildcard determines what files are displayed in the file selector, and file extension supplies a type extension for the required filename. Flags may be a combination of wxOPEN, wxSAVE, wxOVERWRITE_PROMPT, wxFILE_MUST_EXIST, wxMULTIPLE or 0.

Both the Unix and Windows versions implement a wildcard filter. Typing a filename containing wildcards (*, ?) in the filename text item, and clicking on Ok, will result in only those files matching the pattern being displayed.

The wildcard may be a specification for multiple types of file with a description for each, such as:

```
"BMP files (*.bmp)|*.bmp|GIF files (*.gif)|*.gif"
```

The application must check for an empty return value (the user pressed Cancel). For example:

```
wxString filename = wxFileSelector("Choose a file to open");
if ( !filename.empty() )
{
```

```
        // work with the file
        ...
    }
    //else: cancelled by user
```

Include files

<wx/filedlg.h>

::wxEndBusyCursor

void wxEndBusyCursor()

Changes the cursor back to the original cursor, for all windows in the application. Use with *wxBeginBusyCursor* (p. 1614).

See also *wxIsBusy* (p. 1621), *wxBusyCursor* (p. 115).

Include files

<wx/utils.h>

::wxGetColourFromUser

wxColour wxGetColourFromUser(wxWindow *parent, const wxColour& *colInit*)

Shows the colour selection dialog and returns the colour selected by user or invalid colour (use *wxColour::Ok* (p. 166) to test whether a colour is valid) if the dialog was cancelled.

Parameters

parent

> The parent window for the colour selection dialog

colInit

> If given, this will be the colour initially selected in the dialog.

Include files

<wx/colordlg.h>

::wxGetFontFromUser

wxFont wxGetFontFromUser(wxWindow *parent, const wxFont& *fontInit*)

Shows the font selection dialog and returns the font selected by user or invalid font (use *wxFont::Ok* (p. 565) to test whether a font is valid) if the dialog was cancelled.

Parameters

parent

> The parent window for the font selection dialog

fontInit

> If given, this will be the font initially selected in the dialog.

Include files

<wx/fontdlg.h>

::wxGetMultipleChoices

size_t wxGetMultipleChoices(
 wxArrayInt& *selections*,
 const wxString& *message*,
 const wxString& *caption*,
 const wxArrayString& *aChoices*,
 wxWindow **parent = NULL*,
 int *x = -1*, **int** *y = -1*,
 bool *centre = true*,
 int *width=150*, **int** *height=200*)

size_t wxGetMultipleChoices(
 wxArrayInt& *selections*,
 const wxString& *message*,
 const wxString& *caption*,
 int *n*, **const wxString&** *choices[]*,
 wxWindow **parent = NULL*,
 int *x = -1*, **int** *y = -1*,
 bool *centre = true*,
 int *width=150*, **int** *height=200*)

Pops up a dialog box containing a message, OK/Cancel buttons and a multiple-selection listbox. The user may choose an arbitrary (including 0) number of items in the listbox whose indices will be returned in*selection* array. The initial contents of this array will be used to select the items when the dialog is shown.

You may pass the list of strings to choose from either using *choices*which is an array of *n* strings for the listbox or by using a single*aChoices* parameter of type *wxArrayString* (p. 65).

If *centre* is true, the message text (which may include new line characters) is centred; if false, the message is left-justified.

Include files

<wx/choicdlg.h>

wxPerl note: In wxPerl there is just an array reference in place of `n`and `choices`, and no `selections` parameter; the function returns an array containing the user selections.

::wxGetNumberFromUser

long wxGetNumberFromUser(**const wxString&** *message*, **const wxString&** *prompt*, **const wxString&** *caption*, **long** *value*, **long** *min = 0*, **long** *max = 100*, **wxWindow** **parent = NULL*, **const wxPoint&** *pos = wxDefaultPosition*)

Shows a dialog asking the user for numeric input. The dialogs title is set to*caption*, it contains a (possibly) multiline *message* above the single line *prompt* and the zone for entering the number.

The number entered must be in the range *min*..*max* (both of which should be positive) and *value* is the initial value of it. If the user enters an invalid value or cancels the dialog, the function will return -1.

Dialog is centered on its *parent* unless an explicit position is given in*pos*.

Include files

<wx/numdlg.h>

::wxGetPasswordFromUser

wxString wxGetPasswordFromUser(**const wxString&** *message*, **const wxString&** *caption = "Input text"*,
 const wxString& *default_value = ""*, **wxWindow** **parent = NULL*,
 int *x = wxDefaultCoord*, **int** *y = wxDefaultCoord*, **bool** *centre = true*)

Similar to *wxGetTextFromUser* (p. 1618) but the text entered in the dialog is not shown on screen but replaced with stars. This is intended to be used for entering passwords as the function name implies.

Include files

<wx/textdlg.h>

::wxGetTextFromUser

wxString wxGetTextFromUser(**const wxString&** *message*, **const wxString&** *caption = "Input text"*,
 const wxString& *default_value = ""*, **wxWindow** **parent = NULL*,
 int *x = wxDefaultCoord*, **int** *y = wxDefaultCoord*, **bool** *centre = true*)

Pop up a dialog box with title set to *caption*, *message*, and a *default_value*. The user may type in text and press OK to return this text, or press Cancel to return the empty string.

If *centre* is true, the message text (which may include new line characters) is centred; if

false, the message is left-justified.

Include files

<wx/textdlg.h>

::wxGetMultipleChoice

int wxGetMultipleChoice(**const wxString&** *message*, **const wxString&** *caption*, **int** *n*, **const wxString&** *choices[]*,
 int *nsel*, **int ****selection*, **wxWindow ****parent = NULL*, **int** *x = -1*, **int** *y = -1*,
 bool *centre = true*, **int** *width=150*, **int** *height=200*)

Pops up a dialog box containing a message, OK/Cancel buttons and a multiple-selection listbox. The user may choose one or more item(s) and press OK or Cancel.

The number of initially selected choices, and array of the selected indices, are passed in; this array will contain the user selections on exit, with the function returning the number of selections. *selection* must be as big as the number of choices, in case all are selected.

If Cancel is pressed, -1 is returned.

choices is an array of *n* strings for the listbox.

If *centre* is true, the message text (which may include new line characters) is centred; if false, the message is left-justified.

Include files

<wx/choicdlg.h>

::wxGetSingleChoice

wxString wxGetSingleChoice(**const wxString&** *message*,
 const wxString& *caption*,
 const wxArrayString& *aChoices*,
 wxWindow **parent = NULL*,
 int *x = -1*, **int** *y = -1*,
 bool *centre = true*,
 int *width=150*, **int** *height=200*)

wxString wxGetSingleChoice(**const wxString&** *message*,
 const wxString& *caption*,
 int *n*, **const wxString&** *choices[]*,
 wxWindow **parent = NULL*,
 int *x = -1*, **int** *y = -1*,
 bool *centre = true*,
 int *width=150*, **int** *height=200*)

Pops up a dialog box containing a message, OK/Cancel buttons and a single-selection

listbox. The user may choose an item and press OK to return a string or Cancel to return the empty string. Use *wxGetSingleChoiceIndex* (p. 1620) if empty string is a valid choice and if you want to be able to detect pressing Cancel reliably.

You may pass the list of strings to choose from either using *choices* which is an array of *n* strings for the listbox or by using a single *aChoices* parameter of type *wxArrayString* (p. 65).

If *centre* is true, the message text (which may include new line characters) is centred; if false, the message is left-justified.

Include files

<wx/choicdlg.h>

wxPerl note: In wxPerl there is just an array reference in place of `n` and `choices`.

::wxGetSingleChoiceIndex

int wxGetSingleChoiceIndex(**const wxString&** *message*,
 const wxString& *caption*,
 const wxArrayString& *aChoices*,
 wxWindow *****parent* = *NULL*, **int *x* = *-1*, **int** *y* = *-1*,
 bool *centre* = *true*, **int** *width=150*, **int** *height=200*)

int wxGetSingleChoiceIndex(**const wxString&** *message*,
 const wxString& *caption*,
 int *n*, **const wxString&** *choices[]*,
 wxWindow *****parent* = *NULL*, **int *x* = *-1*, **int** *y* = *-1*,
 bool *centre* = *true*, **int** *width=150*, **int** *height=200*)

As **wxGetSingleChoice** but returns the index representing the selected string. If the user pressed cancel, -1 is returned.

Include files

<wx/choicdlg.h>

wxPerl note: In wxPerl there is just an array reference in place of `n` and `choices`.

::wxGetSingleChoiceData

wxString wxGetSingleChoiceData(**const wxString&** *message*,
 const wxString& *caption*,
 const wxArrayString& *aChoices*,
 const wxString& *client_data[]*,
 **wxWindow *****parent* = *NULL*,
 int *x* = *-1*, **int** *y* = *-1*,
 bool *centre* = *true*, **int** *width=150*, **int** *height=200*)

wxString wxGetSingleChoiceData(**const wxString&** *message*,

const wxString& *caption*,
int *n*, **const wxString&** *choices[]*,
const wxString& *client_data[]*,
wxWindow **parent = NULL*,
int *x = -1*, **int** *y = -1*,
bool *centre = true*, **int** *width=150*, **int** *height=200*)

As **wxGetSingleChoice** but takes an array of client data pointers corresponding to the strings, and returns one of these pointers or NULL if Cancel was pressed. The *client_data* array must have the same number of elements as *choices* or *aChoices*!

Include files

<wx/choicdlg.h>

wxPerl note: In wxPerl there is just an array reference in place of `n` and `choices`, and the client data array must have the same length as the choices array.

::wxIsBusy

bool wxIsBusy()

Returns true if between two *wxBeginBusyCursor* (p. 1614) and *wxEndBusyCursor* (p. 1616) calls.

See also *wxBusyCursor* (p. 115).

Include files

<wx/utils.h>

::wxMessageBox

int wxMessageBox(**const wxString&** *message*, **const wxString&** *caption =*
"Message", **int** *style = wxOK*,
 wxWindow **parent = NULL*, **int** *x = -1*, **int** *y = -1*)

General purpose message dialog. *style* may be a bit list of the following identifiers:

wxYES_NO	Puts Yes and No buttons on the message box. May be combined with wxCANCEL.
wxCANCEL	Puts a Cancel button on the message box. May be combined with wxYES_NO or wxOK.
wxOK	Puts an Ok button on the message box. May be combined with wxCANCEL.
wxICON_EXCLAMATION	Displays an exclamation mark symbol.
wxICON_HAND	Displays an error symbol.

wxICON_ERROR   Displays an error symbol - the same as wxICON_HAND.

wxICON_QUESTION   Displays a question mark symbol.

wxICON_INFORMATION   Displays an information symbol.

The return value is one of: wxYES, wxNO, wxCANCEL, wxOK.

For example:

```
...
int answer = wxMessageBox("Quit program?", "Confirm",
                          wxYES_NO | wxCANCEL, main_frame);
if (answer == wxYES)
  main_frame->Close();
...
```

message may contain newline characters, in which case the message will be split into separate lines, to cater for large messages.

Include files

<wx/msgdlg.h>

::wxShowTip

bool wxShowTip(**wxWindow** **parent*, **wxTipProvider** **tipProvider*, **bool** *showAtStartup = true*)

This function shows a "startup tip" to the user. The return value is the state of the 'Show tips at startup' checkbox.

parent

> The parent window for the modal dialog

tipProvider

> An object which is used to get the text of the tips. It may be created with the *wxCreateFileTipProvider* (p. 1614) function.

showAtStartup

> Should be true if startup tips are shown, false otherwise. This is used as the initial value for "Show tips at startup" checkbox which is shown in the tips dialog.

See also

Tips overview (p. 1814)

Include files

<wx/tipdlg.h>

Math functions

Include files

<wx/math.h>

wxFinite

int wxFinite(**double** *x*)

Returns a non-zero value if *x* is neither infinite or NaN (not a number), returns 0 otherwise.

wxIsNaN

bool wxIsNaN(**double** *x*)

Returns a non-zero value if *x* is NaN (not a number), returns 0 otherwise.

GDI functions

The following are relevant to the GDI (Graphics Device Interface).

Include files

<wx/gdicmn.h>

wxBITMAP

wxBITMAP(bitmapName)

This macro loads a bitmap from either application resources (on the platforms for which they exist, i.e. Windows and OS2) or from an XPM file. It allows to avoid using `#ifdef`s when creating bitmaps.

See also

Bitmaps and icons overview (p. 1788),*wxICON* (p. 1625)

Include files

<wx/gdicmn.h>

::wxClientDisplayRect

void wxClientDisplayRect(**int ****x*, **int ****y*,**int ****width*, **int ****height*)

wxRect wxGetClientDisplayRect()

Returns the dimensions of the work area on the display. On Windows this means the area not covered by the taskbar, etc. Other platforms are currently defaulting to the whole display until a way is found to provide this info for all window managers, etc.

::wxColourDisplay

bool wxColourDisplay()

Returns true if the display is colour, false otherwise.

::wxDisplayDepth

int wxDisplayDepth()

Returns the depth of the display (a value of 1 denotes a monochrome display).

::wxDisplaySize

void wxDisplaySize(**int ***_width_, **int ***_height_)

wxSize wxGetDisplaySize()

Returns the display size in pixels.

::wxDisplaySizeMM

void wxDisplaySizeMM(**int ***_width_, **int ***_height_)

wxSize wxGetDisplaySizeMM()

Returns the display size in millimeters.

::wxDROP_ICON

wxIconOrCursor wxDROP_ICON(**const char ***_name_)

This macro creates either a cursor (MSW) or an icon (elsewhere) with the given name. Under MSW, the cursor is loaded from the resource file and the icon is loaded from XPM file under other platforms.

This macro should be used with*wxDropSource constructor* (p. 469).

Include files

<wx/dnd.h>

wxICON

wxICON(iconName)

This macro loads an icon from either application resources (on the platforms for which they exist, i.e. Windows and OS2) or from an XPM file. It allows to avoid using `#ifdef`s when creating icons.

See also

Bitmaps and icons overview (p. 1788),*wxBITMAP* (p. 1623)

Include files

<wx/gdicmn.h>

::wxMakeMetafilePlaceable

bool wxMakeMetafilePlaceable(**const wxString&** *filename*, **int** *minX*, **int** *minY*, **int** *maxX*, **int** *maxY*, **float** *scale=1.0*)

Given a filename for an existing, valid metafile (as constructed using *wxMetafileDC* (p. 973)) makes it into a placeable metafile by prepending a header containing the given bounding box. The bounding box may be obtained from a device context after drawing into it, using the functions wxDC::MinX, wxDC::MinY, wxDC::MaxX and wxDC::MaxY.

In addition to adding the placeable metafile header, this function adds the equivalent of the following code to the start of the metafile data:

```
SetMapMode(dc, MM_ANISOTROPIC);
SetWindowOrg(dc, minX, minY);
SetWindowExt(dc, maxX - minX, maxY - minY);
```

This simulates the wxMM_TEXT mapping mode, which wxWidgets assumes.

Placeable metafiles may be imported by many Windows applications, and can be used in RTF (Rich Text Format) files.

scale allows the specification of scale for the metafile.

This function is only available under Windows.

::wxSetCursor

void wxSetCursor(**wxCursor ***cursor*)

Globally sets the cursor; only has an effect in Windows and GTK. See also *wxCursor* (p. 226), *wxWindow::SetCursor* (p. 1530).

Printer settings

NB: These routines are obsolete and should no longer be used!

The following functions are used to control PostScript printing. Under Windows, PostScript output can only be sent to a file.

Include files

<wx/dcps.h>

::wxGetPrinterCommand

wxString wxGetPrinterCommand()

Gets the printer command used to print a file. The default is `lpr`.

::wxGetPrinterFile

wxString wxGetPrinterFile()

Gets the PostScript output filename.

::wxGetPrinterMode

int wxGetPrinterMode()

Gets the printing mode controlling where output is sent (PS_PREVIEW, PS_FILE or PS_PRINTER). The default is PS_PREVIEW.

::wxGetPrinterOptions

wxString wxGetPrinterOptions()

Gets the additional options for the print command (e.g. specific printer). The default is nothing.

::wxGetPrinterOrientation

int wxGetPrinterOrientation()

Gets the orientation (PS_PORTRAIT or PS_LANDSCAPE). The default is PS_PORTRAIT.

::wxGetPrinterPreviewCommand

wxString wxGetPrinterPreviewCommand()

Gets the command used to view a PostScript file. The default depends on the platform.

::wxGetPrinterScaling

void wxGetPrinterScaling(**float ****x*, **float ****y*)

Gets the scaling factor for PostScript output. The default is 1.0, 1.0.

::wxGetPrinterTranslation

void wxGetPrinterTranslation(**float ****x*, **float ****y*)

Gets the translation (from the top left corner) for PostScript output. The default is 0.0, 0.0.

::wxSetPrinterCommand

void wxSetPrinterCommand(**const wxString&** *command*)

Sets the printer command used to print a file. The default is `lpr`.

::wxSetPrinterFile

void wxSetPrinterFile(**const wxString&** *filename*)

Sets the PostScript output filename.

::wxSetPrinterMode

void wxSetPrinterMode(**int** *mode*)

Sets the printing mode controlling where output is sent (PS_PREVIEW, PS_FILE or PS_PRINTER). The default is PS_PREVIEW.

::wxSetPrinterOptions

void wxSetPrinterOptions(**const wxString&** *options*)

Sets the additional options for the print command (e.g. specific printer). The default is nothing.

::wxSetPrinterOrientation

void wxSetPrinterOrientation(**int** *orientation*)

Sets the orientation (PS_PORTRAIT or PS_LANDSCAPE). The default is
PS_PORTRAIT.

::wxSetPrinterPreviewCommand

void wxSetPrinterPreviewCommand(**const wxString&** *command*)

Sets the command used to view a PostScript file. The default depends on the platform.

::wxSetPrinterScaling

void wxSetPrinterScaling(**float** *x*, **float** *y*)

Sets the scaling factor for PostScript output. The default is 1.0, 1.0.

::wxSetPrinterTranslation

void wxSetPrinterTranslation(**float** *x*, **float** *y*)

Sets the translation (from the top left corner) for PostScript output. The default is 0.0,
0.0.

Clipboard functions

These clipboard functions are implemented for Windows only. The use of these
functions is deprecated and the code is no longer maintained. Use the *wxClipboard* (p.
150)class instead.

Include files

<wx/clipbrd.h>

::wxClipboardOpen

bool wxClipboardOpen()

Returns true if this application has already opened the clipboard.

::wxCloseClipboard

bool wxCloseClipboard()

Closes the clipboard to allow other applications to use it.

::wxEmptyClipboard

bool wxEmptyClipboard()

Empties the clipboard.

::wxEnumClipboardFormats

int wxEnumClipboardFormats(**int***dataFormat*)

Enumerates the formats found in a list of available formats that belong to the clipboard. Each call to this function specifies a known available format; the function returns the format that appears next in the list.

dataFormat specifies a known format. If this parameter is zero, the function returns the first format in the list.

The return value specifies the next known clipboard data format if the function is successful. It is zero if the *dataFormat* parameter specifies the last format in the list of available formats, or if the clipboard is not open.

Before it enumerates the formats function, an application must open the clipboard by using the wxOpenClipboard function.

::wxGetClipboardData

wxObject * wxGetClipboardData(**int***dataFormat*)

Gets data from the clipboard.

dataFormat may be one of:

- wxCF_TEXT or wxCF_OEMTEXT: returns a pointer to new memory containing a null-terminated text string.

- wxCF_BITMAP: returns a new wxBitmap.

The clipboard must have previously been opened for this call to succeed.

::wxGetClipboardFormatName

bool wxGetClipboardFormatName(**int***dataFormat*, **const wxString&** *formatName*, **int***maxCount*)

Gets the name of a registered clipboard format, and puts it into the buffer *formatName* which is of maximum length *maxCount*. *dataFormat* must not specify a predefined clipboard format.

::wxIsClipboardFormatAvailable

bool wxIsClipboardFormatAvailable(**int***dataFormat*)

Returns true if the given data format is available on the clipboard.

::wxOpenClipboard

bool wxOpenClipboard()

Opens the clipboard for passing data to it or getting data from it.

::wxRegisterClipboardFormat

int wxRegisterClipboardFormat(**const wxString&** *formatName*)

Registers the clipboard data format name and returns an identifier.

::wxSetClipboardData

bool wxSetClipboardData(**int***dataFormat*, **wxObject ****data*, **int***width*, **int***height*)

Passes data to the clipboard.

dataFormat may be one of:

- wxCF_TEXT or wxCF_OEMTEXT: *data* is a null-terminated text string.

- wxCF_BITMAP: *data* is a wxBitmap.

- wxCF_DIB: *data* is a wxBitmap. The bitmap is converted to a DIB (device independent bitmap).

- wxCF_METAFILE: *data* is a wxMetafile. *width* and *height* are used to give recommended dimensions.

The clipboard must have previously been opened for this call to succeed.

Miscellaneous functions

wxCONCAT

wxCONCAT(*x, y*)

This macro returns the concatenation of two tokens *x* and *y*.

wxDYNLIB_FUNCTION

wxDYNLIB_FUNCTION(*type*, *name*, *dynlib*)

When loading a function from a DLL you always have to cast the returned `void *` pointer to the correct type and, even more annoyingly, you have to repeat this type twice if you want to declare and define a function pointer all in one line

This macro makes this slightly less painful by allowing you to specify the type only once, as the first parameter, and creating a variable of this type named after the function but with `pfn` prefix and initialized with the function *name* from the *wxDynamicLibrary* (p. 474)*dynlib*.

Parameters

type

> the type of the function

name

> the name of the function to load, not a string (without quotes, it is quoted automatically by the macro)

dynlib

> the library to load the function from

wxEXPLICIT

`wxEXPLICIT` is a macro which expands to the C++ `explicit` keyword if the compiler supports it or nothing otherwise. Thus, it can be used even in the code which might have to be compiled with an old compiler without support for this language feature but still take advantage of it when it is available.

::wxGetKeyState

bool wxGetKeyState(**wxKeyCode** *key*)

For normal keys, returns `true` if the specified key is currently down.

For togglable keys (Caps Lock, Num Lock and Scroll Lock), returns `true` if the key is toggled such that its LED indicator is lit. There is currently no way to test whether togglable keys are up or down.

Even though there are virtual key codes defined for mouse buttons, they cannot be used with this function currently.

Include files

<wx/utils.h>

wxLL

wxLongLong_t wxLL(*number*)

This macro is defined for the platforms with a native 64 bit integer type and allows to define 64 bit compile time constants:

```
#ifdef wxLongLong_t
    wxLongLong_t ll = wxLL(0x1234567890abcdef);
#endif
```

Include files

<wx/longlong.h>

See also

wxULL (p. 1639), *wxLongLong* (p. 903)

wxLongLongFmtSpec

This macro is defined to contain the `printf()` format specifier using which 64 bit integer numbers (i.e. those of type `wxLongLong_t`) can be printed. Example of using it:

```
#ifdef wxLongLong_t
    wxLongLong_t ll = wxLL(0x1234567890abcdef);
    printf("Long long = %" wxLongLongFmtSpec "x\n", ll);
#endif
```

See also

wxLL (p. 1632)

Include files

<wx/longlong.h>

::wxNewId

long wxNewId()

Generates an integer identifier unique to this run of the program.

Include files

<wx/utils.h>

::wxRegisterId

void wxRegisterId(**long** *id*)

Ensures that ids subsequently generated by **NewId** do not clash with the given **id**.

Include files

<wx/utils.h>

::wxDDECleanUp

void wxDDECleanUp()

Called when wxWidgets exits, to clean up the DDE system. This no longer needs to be called by the application.

See also *wxDDEInitialize* (p. 1633).

Include files

<wx/dde.h>

::wxDDEInitialize

void wxDDEInitialize()

Initializes the DDE system. May be called multiple times without harm.

This no longer needs to be called by the application: it will be called by wxWidgets if necessary.

See also *wxDDEServer* (p. 391), *wxDDEClient* (p. 386), *wxDDEConnection* (p. 387), *wxDDECleanUp* (p. 1633).

Include files

<wx/dde.h>

::wxEnableTopLevelWindows

void wxEnableTopLevelWindow(**bool** *enable* = *true*)

This function enables or disables all top level windows. It is used by*::wxSafeYield* (p. 1591).

Include files

<wx/utils.h>

::wxFindMenuItemId

int wxFindMenuItemId(**wxFrame** **frame*, **const wxString&** *menuString*, **const**

wxString& *itemString*)

Find a menu item identifier associated with the given frame's menu bar.

Include files

<wx/utils.h>

::wxFindWindowByLabel

wxWindow * wxFindWindowByLabel(**const wxString&** *label*, **wxWindow**
parent=NULL)

NB: This function is obsolete, please use*wxWindow::FindWindowByLabel* (p. 1502)
instead.

Find a window by its label. Depending on the type of window, the label may be a window
title or panel item label. If *parent* is NULL, the search will start from all top-level frames
and dialog boxes; if non-NULL, the search will be limited to the given window hierarchy.
The search is recursive in both cases.

Include files

<wx/utils.h>

::wxFindWindowByName

wxWindow * wxFindWindowByName(**const wxString&** *name*, **wxWindow**
parent=NULL)

NB: This function is obsolete, please use*wxWindow::FindWindowByName* (p. 1502)
instead.

Find a window by its name (as given in a window constructor or **Create** function call). If
parent is NULL, the search will start from all top-level frames and dialog boxes; if non-
NULL, the search will be limited to the given window hierarchy. The search is recursive
in both cases.

If no such named window is found, **wxFindWindowByLabel** is called.

Include files

<wx/utils.h>

::wxFindWindowAtPoint

wxWindow * wxFindWindowAtPoint(**const wxPoint&** *pt*)

Find the deepest window at the given mouse position in screen coordinates, returning
the window if found, or NULL if not.

::wxFindWindowAtPointer

wxWindow * wxFindWindowAtPointer(**wxPoint&** *pt*)

Find the deepest window at the mouse pointer position, returning the window and current pointer position in screen coordinates.

::wxGetActiveWindow

wxWindow * wxGetActiveWindow()

Gets the currently active window (Windows only).

Include files

<wx/windows.h>

::wxGetBatteryState

wxBatteryState wxGetBatteryState()

Returns battery state as one of `wxBATTERY_NORMAL_STATE`,`wxBATTERY_LOW_STATE`, `wxBATTERY_CRITICAL_STATE`,`wxBATTERY_SHUTDOWN_STATE` or `wxBATTERY_UNKNOWN_STATE`.`wxBATTERY_UNKNOWN_STATE` is also the default on platforms where this feature is not implemented.

Include files

<wx/utils.h>

::wxGetDisplayName

wxString wxGetDisplayName()

Under X only, returns the current display name. See also *wxSetDisplayName* (p. 1638).

Include files

<wx/utils.h>

::wxGetPowerType

wxPowerType wxGetPowerType()

Returns the type of power source as one of `wxPOWER_SOCKET`,`wxPOWER_BATTERY` or `wxPOWER_UNKNOWN`.`wxPOWER_UNKNOWN` is also the default on platforms where this feature is not implemented.

Include files

<wx/utils.h>

::wxGetMousePosition

wxPoint wxGetMousePosition()

Returns the mouse position in screen coordinates.

<wx/utils.h>

::wxGetResource

bool wxGetResource(**const wxString&** *section*, **const wxString&** *entry*, **const wxString& ****value*, **const wxString&** *file* = *NULL*)

bool wxGetResource(**const wxString&** *section*, **const wxString&** *entry*, **float ****value*, **const wxString&** *file* = *NULL*)

bool wxGetResource(**const wxString&** *section*, **const wxString&** *entry*, **long ****value*, **const wxString&** *file* = *NULL*)

bool wxGetResource(**const wxString&** *section*, **const wxString&** *entry*, **int ****value*, **const wxString&** *file* = *NULL*)

Gets a resource value from the resource database (for example, WIN.INI, or .Xdefaults). If *file* is NULL, WIN.INI or .Xdefaults is used, otherwise the specified file is used.

Under X, if an application class (wxApp::GetClassName) has been defined, it is appended to the string /usr/lib/X11/app-defaults/ to try to find an applications default file when merging all resource databases.

The reason for passing the result in an argument is that it can be convenient to define a default value, which gets overridden if the value exists in the resource file. It saves a separate test for that resource's existence, and it also allows the overloading of the function for different types.

See also *wxWriteResource* (p. 1639), *wxConfigBase* (p. 193).

Include files

<wx/utils.h>

::wxGetStockLabel

wxString wxGetStockLabel(**wxWindowID** *id*, **bool** *withCodes* = *true*, **wxString** *accelerator* = *wxEmptyString*)

Returns label that should be used for given *id* element.

Parameters

id

> given id of the *wxMenuItem* (p. 964), *wxButton* (p. 117), *wxToolBar* (p. 1401) tool, etc.

withCodes

> if false then strip accelerator code from the label; usefull for getting labels without accelerator char code like for toolbar tooltip or under platforms without traditional keyboard like smartphones

accelerator

> optional accelerator string automatically added to label; useful for building labels for *wxMenuItem* (p. 964)

Include files

<wx/stockitem.h>

::wxGetTopLevelParent

wxWindow * wxGetTopLevelParent(**wxWindow** **win*)

Returns the first top level parent of the given window, or in other words, the frame or dialog containing it, or `NULL`.

Include files

<wx/window.h>

::wxLoadUserResource

wxString wxLoadUserResource(**const wxString&** *resourceName*, **const wxString&** *resourceType="TEXT"*)

Loads a user-defined Windows resource as a string. If the resource is found, the function creates a new character array and copies the data into it. A pointer to this data is returned. If unsuccessful, NULL is returned.

The resource must be defined in the `.rc` file using the following syntax:

```
myResource TEXT file.ext
```

where `file.ext` is a file that the resource compiler can find.

This function is available under Windows only.

Include files

<wx/utils.h>

::wxPostDelete

void wxPostDelete(**wxObject ****object*)

Tells the system to delete the specified object when all other events have been processed. In some environments, it is necessary to use this instead of deleting a frame directly with the delete operator, because some GUIs will still send events to a deleted window.

Now obsolete: use *wxWindow::Close* (p. 1496) instead.

Include files

<wx/utils.h>

::wxPostEvent

void wxPostEvent(**wxEvtHandler ****dest*, **wxEvent&** *event*)

In a GUI application, this function posts *event* to the specified *dest*object using *wxEvtHandler::AddPendingEvent* (p. 487). Otherwise, it dispatches *event* immediately using*wxEvtHandler::ProcessEvent* (p. 491). See the respective documentation for details (and caveats).

Include files

<wx/app.h>

::wxSetDisplayName

void wxSetDisplayName(**const wxString&** *displayName*)

Under X only, sets the current display name. This is the X host and display name such as "colonsay:0.0", and the function indicates which display should be used for creating windows from this point on. Setting the display within an application allows multiple displays to be used.

See also *wxGetDisplayName* (p. 1635).

Include files

<wx/utils.h>

::wxStripMenuCodes

wxString wxStripMenuCodes(**const wxString&** *in*)

void wxStripMenuCodes(**char ****in*, **char ****out*)

NB: This function is obsolete, please use*wxMenuItem::GetLabelFromText* (p. 966) instead.

Strips any menu codes from *in* and places the result in *out* (or returns the new string, in the first form).

Menu codes include & (mark the next character with an underline as a keyboard shortkey in Windows and Motif) and \t (tab in Windows).

Include files

<wx/utils.h>

wxULL

wxLongLong_t wxULL(*number*)

This macro is defined for the platforms with a native 64 bit integer type and allows to define unsigned 64 bit compile time constants:

```
#ifdef wxLongLong_t
    unsigned wxLongLong_t ll = wxULL(0x1234567890abcdef);
#endif
```

Include files

<wx/longlong.h>

See also

wxLL (p. 1632), *wxLongLong* (p. 903)

wxVaCopy

void wxVaCopy(**va_list** *argptrDst*, **va_list***argptrSrc*)

This macro is the same as the standard C99 `va_copy` for the compilers which support it or its replacement for those that don't. It must be used to preserve the value of a `va_list` object if you need to use it after passing it to another function because it can be modified by the latter.

As with `va_start`, each call to `wxVaCopy` must have a matching`va_end`.

::wxWriteResource

bool wxWriteResource(**const wxString&** *section*, **const wxString&** *entry*, **const wxString&** *value*, **const wxString&** *file* = *NULL*)

bool wxWriteResource(**const wxString&** *section*, **const wxString&** *entry*, **float** *value*,

const wxString& *file* = *NULL*)

bool wxWriteResource(**const wxString&** *section*, **const wxString&** *entry*, **long** *value*, **const wxString&** *file* = *NULL*)

bool wxWriteResource(**const wxString&** *section*, **const wxString&** *entry*, **int** *value*, **const wxString&** *file* = *NULL*)

Writes a resource value into the resource database (for example, WIN.INI, or .Xdefaults). If *file* is NULL, WIN.INI or .Xdefaults is used, otherwise the specified file is used.

Under X, the resource databases are cached until the internal function **wxFlushResources** is called automatically on exit, when all updated resource databases are written to their files.

Note that it is considered bad manners to write to the .Xdefaults file under Unix, although the WIN.INI file is fair game under Windows.

See also *wxGetResource* (p. 1636), *wxConfigBase* (p. 193).

Include files

<wx/utils.h>

Byte order macros

The endian-ness issues (that is the difference between big-endian and little-endian architectures) are important for the portable programs working with the external binary data (for example, data files or data coming from network) which is usually in some fixed, platform-independent format. The macros are helpful for transforming the data to the correct format.

wxINTXX_SWAP_ALWAYS

wxInt32 wxINT32_SWAP_ALWAYS(**wxInt32** *value*)

wxUint32 wxUINT32_SWAP_ALWAYS(**wxUint32** *value*)

wxInt16 wxINT16_SWAP_ALWAYS(**wxInt16** *value*)

wxUint16 wxUINT16_SWAP_ALWAYS(**wxUint16** *value*)

These macros will swap the bytes of the *value* variable from little endian to big endian or vice versa unconditionally, i.e. independently of the current platform.

wxINTXX_SWAP_ON_BE

wxInt32 wxINT32_SWAP_ON_BE(**wxInt32** *value*)

wxUint32 wxUINT32_SWAP_ON_BE(**wxUint32** *value*)

wxInt16 wxINT16_SWAP_ON_BE(**wxInt16** *value*)

wxUint16 wxUINT16_SWAP_ON_BE(**wxUint16** *value*)

This macro will swap the bytes of the *value* variable from little endian to big endian or vice versa if the program is compiled on a big-endian architecture (such as Sun work stations). If the program has been compiled on a little-endian architecture, the value will be unchanged.

Use these macros to read data from and write data to a file that stores data in little-endian (for example Intel i386) format.

wxINTXX_SWAP_ON_LE

wxInt32 wxINT32_SWAP_ON_LE(**wxInt32** *value*)

wxUint32 wxUINT32_SWAP_ON_LE(**wxUint32** *value*)

wxInt16 wxINT16_SWAP_ON_LE(**wxInt16** *value*)

wxUint16 wxUINT16_SWAP_ON_LE(**wxUint16** *value*)

This macro will swap the bytes of the *value* variable from little endian to big endian or vice versa if the program is compiled on a little-endian architecture (such as Intel PCs). If the program has been compiled on a big-endian architecture, the value will be unchanged.

Use these macros to read data from and write data to a file that stores data in big-endian format.

RTTI functions

wxWidgets uses its own RTTI ("run-time type identification") system which predates the current standard C++ RTTI and so is kept for backwards compatibility reasons but also because it allows some things which the standard RTTI doesn't directly support (such as creating a class from its name).

The standard C++ RTTI can be used in the user code without any problems and in general you shouldn't need to use the functions and the macros in this section unless you are thinking of modifying or adding any wxWidgets classes.

See also

RTTI overview

CLASSINFO

wxClassInfo * CLASSINFO(className)

Returns a pointer to the wxClassInfo object associated with this class.

Include files

<wx/object.h>

DECLARE_ABSTRACT_CLASS

DECLARE_ABSTRACT_CLASS(className)

Used inside a class declaration to declare that the class should be made known to the class hierarchy, but objects of this class cannot be created dynamically. The same as DECLARE_CLASS.

Example:

```
class wxCommand: public wxObject
{
  DECLARE_ABSTRACT_CLASS(wxCommand)

 private:
  ...
 public:
  ...
};
```

Include files

<wx/object.h>

DECLARE_APP

DECLARE_APP(className)

This is used in headers to create a forward declaration of the*wxGetApp* (p. 1590) function implemented by*IMPLEMENT_APP* (p. 1644). It creates the declaration`className& wxGetApp(void).`

Example:

```
DECLARE_APP(MyApp)
```

Include files

<wx/app.h>

DECLARE_CLASS

DECLARE_CLASS(className)

Used inside a class declaration to declare that the class should be made known to the class hierarchy, but objects of this class cannot be created dynamically. The same as

DECLARE_ABSTRACT_CLASS.

<wx/object.h>

DECLARE_DYNAMIC_CLASS

DECLARE_DYNAMIC_CLASS(className)

Used inside a class declaration to declare that the objects of this class should be dynamically creatable from run-time type information.

Example:

```
class wxFrame: public wxWindow
{
  DECLARE_DYNAMIC_CLASS(wxFrame)

 private:
  const wxString& frameTitle;
 public:
  ...
};
```

Include files

<wx/object.h>

IMPLEMENT_ABSTRACT_CLASS

IMPLEMENT_ABSTRACT_CLASS(className, baseClassName)

Used in a C++ implementation file to complete the declaration of a class that has run-time type information. The same as IMPLEMENT_CLASS.

Example:

```
IMPLEMENT_ABSTRACT_CLASS(wxCommand, wxObject)

wxCommand::wxCommand(void)
{
...
}
```

Include files

<wx/object.h>

IMPLEMENT_ABSTRACT_CLASS2

IMPLEMENT_ABSTRACT_CLASS2(className, baseClassName1, baseClassName2)

Used in a C++ implementation file to complete the declaration of a class that has run-time type information and two base classes. The same as IMPLEMENT_CLASS2.

Include files

<wx/object.h>

IMPLEMENT_APP

IMPLEMENT_APP(className)

This is used in the application class implementation file to make the application class known to wxWidgets for dynamic construction. You use this instead of

Old form:

```
MyApp myApp;
```

New form:

```
IMPLEMENT_APP(MyApp)
```

See also *DECLARE_APP* (p. 1642).

Include files

<wx/app.h>

IMPLEMENT_CLASS

IMPLEMENT_CLASS(className, baseClassName)

Used in a C++ implementation file to complete the declaration of a class that has run-time type information. The same as IMPLEMENT_ABSTRACT_CLASS.

Include files

<wx/object.h>

IMPLEMENT_CLASS2

IMPLEMENT_CLASS2(className, baseClassName1, baseClassName2)

Used in a C++ implementation file to complete the declaration of a class that has run-time type information and two base classes. The same as IMPLEMENT_ABSTRACT_CLASS2.

Include files

<wx/object.h>

IMPLEMENT_DYNAMIC_CLASS

IMPLEMENT_DYNAMIC_CLASS(className, baseClassName)

Used in a C++ implementation file to complete the declaration of a class that has run-time type information, and whose instances can be created dynamically.

Example:

```
IMPLEMENT_DYNAMIC_CLASS(wxFrame, wxWindow)

wxFrame::wxFrame(void)
{
...
}
```

Include files

<wx/object.h>

IMPLEMENT_DYNAMIC_CLASS2

IMPLEMENT_DYNAMIC_CLASS2(className, baseClassName1, baseClassName2)

Used in a C++ implementation file to complete the declaration of a class that has run-time type information, and whose instances can be created dynamically. Use this for classes derived from two base classes.

Include files

<wx/object.h>

wxConstCast

classname * wxConstCast(ptr, classname)

This macro expands into `const_cast<classname *>(ptr)` if the compiler supports *const_cast* or into an old, C-style cast, otherwise.

See also

wx_const_cast (p. 1647)
wxDynamicCast (p. 1646)
wxStaticCast (p. 1647)

::wxCreateDynamicObject

wxObject * wxCreateDynamicObject(**const wxString&** *className*)

Creates and returns an object of the given class, if the class has been registered with the dynamic class system using DECLARE... and IMPLEMENT... macros.

WXDEBUG_NEW

WXDEBUG_NEW(arg)

This is defined in debug mode to be call the redefined new operator with filename and line number arguments. The definition is:

```
#define WXDEBUG_NEW new(__FILE__,__LINE__)
```

In non-debug mode, this is defined as the normal new operator.

Include files

<wx/object.h>

wxDynamicCast

classname * wxDynamicCast(ptr, classname)

This macro returns the pointer *ptr* cast to the type *classname ** if the pointer is of this type (the check is done during the run-time) orNULL otherwise. Usage of this macro is preferred over obsoleted wxObject::IsKindOf() function.

The *ptr* argument may be NULL, in which case NULL will be returned.

Example:

```
wxWindow *win = wxWindow::FindFocus();
wxTextCtrl *text = wxDynamicCast(win, wxTextCtrl);
if ( text )
{
    // a text control has the focus...
}
else
{
    // no window has the focus or it is not a text control
}
```

See also

RTTI overview (p. 1719)
wxDynamicCastThis (p. 1646)
wxConstCast (p. 1645)
wxStaticCast (p. 1647)

wxDynamicCastThis

classname * wxDynamicCastThis(classname)

This macro is equivalent to wxDynamicCast(this, classname) but the latter provokes spurious compilation warnings from some compilers (because it tests whether this pointer is non NULL which is always true), so this macro should be used to avoid them.

wxDynamicCast (p. 1646)

wxStaticCast

classname * wxStaticCast(ptr, classname)

This macro checks that the cast is valid in debug mode (an assert failure will result if `wxDynamicCast(ptr, classname) == NULL`) and then returns the result of executing an equivalent of `static_cast<classname *>(ptr)`.

See also

wx_static_cast (p. 1647)
wxDynamicCast (p. 1646)
wxConstCast (p. 1645)

wx_const_cast

T wx_const_cast(T, x)

Same as `const_cast<T>(x)` if the compiler supports const cast or `(T)x` for old compilers. Unlike *wxConstCast* (p. 1645), the cast it to the type *T* and not to `T *` and also the order of arguments is the same as for the standard cast.

See also

wx_reinterpret_cast (p. 1647),
wx_static_cast (p. 1647)

wx_reinterpret_cast

T wx_reinterpret_cast(T, x)

Same as `reinterpret_cast<T>(x)` if the compiler supports reinterpret cast or `(T)x` for old compilers.

See also

wx_const_cast (p. 1647),
wx_static_cast (p. 1647)

wx_static_cast

T wx_static_cast(T, x)

Same as `static_cast<T>(x)` if the compiler supports static cast or `(T)x` for old

compilers. Unlike *wxStaticCast* (p. 1647), there are no checks being done and the meaning of the macro arguments is exactly the same as for the standard static cast, i.e. *T* is the full type name and star is not appended to it.

See also

wx_const_cast (p. 1647),
wx_reinterpret_cast (p. 1647)

Log functions

These functions provide a variety of logging functions: see *Log classes overview* (p. 1743) for further information. The functions use (implicitly) the currently active log target, so their descriptions here may not apply if the log target is not the standard one (installed by wxWidgets in the beginning of the program).

Include files

<wx/log.h>

::wxDebugMsg

void wxDebugMsg(**const wxString&** *fmt*, **...**)

NB: This function is now obsolete, replaced by *Log functions* (p. 1648) and *wxLogDebug* (p. 1651) in particular.

Display a debugging message; under Windows, this will appear on the debugger command window, and under Unix, it will be written to standard error.

The syntax is identical to **printf**: pass a format string and a variable list of arguments.

Tip: under Windows, if your application crashes before the message appears in the debugging window, put a wxYield call after each wxDebugMsg call. wxDebugMsg seems to be broken under WIN32s (at least for Watcom C++): preformat your messages and use OutputDebugString instead.

Include files

<wx/utils.h>

::wxError

void wxError(**const wxString&** *msg*, **const wxString&** *title* = *"wxWidgets Internal Error"*)

NB: This function is now obsolete, please use *wxLogError* (p. 1649)instead.

Displays *msg* and continues. This writes to standard error under Unix, and pops up a message box under Windows. Used for internal wxWidgets errors. See also

wxFatalError (p. 1649).

Include files

<wx/utils.h>

::wxFatalError

void wxFatalError(**const wxString&** *msg*, **const wxString&** *title* = *"wxWidgets Fatal Error"*)

NB: This function is now obsolete, please use*wxLogFatalError* (p. 1649) instead.

Displays *msg* and exits. This writes to standard error under Unix, and pops up a message box under Windows. Used for fatal internal wxWidgets errors. See also *wxError* (p. 1648).

Include files

<wx/utils.h>

::wxLogError

void wxLogError(**const char** **formatString*, **...**)

void wxVLogError(**const char** **formatString*, **va_list** *argPtr*)

The functions to use for error messages, i.e. the messages that must be shown to the user. The default processing is to pop up a message box to inform the user about it.

::wxLogFatalError

void wxLogFatalError(**const char** **formatString*, **...**)

void wxVLogFatalError(**const char** **formatString*, **va_list** *argPtr*)

Like *wxLogError* (p. 1649), but also terminates the program with the exit code 3. Using *abort()* standard function also terminates the program with this exit code.

::wxLogWarning

void wxLogWarning(**const char** **formatString*, **...**)

void wxVLogWarning(**const char** **formatString*, **va_list** *argPtr*)

For warnings - they are also normally shown to the user, but don't interrupt the program work.

::wxLogMessage

void wxLogMessage(**const char ***formatString*, **...**)

void wxVLogMessage(**const char ***formatString*, **va_list** *argPtr*)

For all normal, informational messages. They also appear in a message box by default (but it can be changed).

::wxLogVerbose

void wxLogVerbose(**const char ***formatString*, **...**)

void wxVLogVerbose(**const char ***formatString*, **va_list** *argPtr*)

For verbose output. Normally, it is suppressed, but might be activated if the user wishes to know more details about the program progress (another, but possibly confusing name for the same function is **wxLogInfo**).

::wxLogStatus

void wxLogStatus(**wxFrame ***frame*, **const char ***formatString*, **...**)

void wxVLogStatus(**wxFrame ***frame*, **const char ***formatString*, **va_list** *argPtr*)

void wxLogStatus(**const char ***formatString*, **...**)

void wxVLogStatus(**const char ***formatString*, **va_list** *argPtr*)

Messages logged by these functions will appear in the statusbar of the *frame* or of the top level application window by default (i.e. when using the second version of the functions).

If the target frame doesn't have a statusbar, the message will be lost.

::wxLogSysError

void wxLogSysError(**const char ***formatString*, **...**)

void wxVLogSysError(**const char ***formatString*, **va_list** *argPtr*)

Mostly used by wxWidgets itself, but might be handy for logging errors after system call (API function) failure. It logs the specified message text as well as the last system error code (*errno* or *::GetLastError()* depending on the platform) and the corresponding error message. The second form of this function takes the error code explicitly as the first argument.

See also

wxSysErrorCode (p. 1652),*wxSysErrorMsg* (p. 1653)

::wxLogDebug

void wxLogDebug(**const char** **formatString*, **...**)

void wxVLogDebug(**const char** **formatString*, **va_list** *argPtr*)

The right functions for debug output. They only do something in debug mode (when the preprocessor symbol __WXDEBUG__ is defined) and expand to nothing in release mode (otherwise).

::wxLogTrace

void wxLogTrace(**const char** **formatString*, **...**)

void wxVLogTrace(**const char** **formatString*, **va_list** *argPtr*)

void wxLogTrace(**const char** **mask*, **const char** **formatString*, **...**)

void wxVLogTrace(**const char** **mask*, **const char** **formatString*, **va_list** *argPtr*)

void wxLogTrace(**wxTraceMask** *mask*, **const char** **formatString*, **...**)

void wxVLogTrace(**wxTraceMask** *mask*, **const char** **formatString*, **va_list** *argPtr*)

As **wxLogDebug**, trace functions only do something in debug build and expand to nothing in the release one. The reason for making it a separate function from it is that usually there are a lot of trace messages, so it might make sense to separate them from other debug messages.

The trace messages also usually can be separated into different categories and the second and third versions of this function only log the message if the *mask* which it has is currently enabled in *wxLog* (p. 889). This allows to selectively trace only some operations and not others by changing the value of the trace mask (possible during the run-time).

For the second function (taking a string mask), the message is logged only if the mask has been previously enabled by the call to *AddTraceMask* (p. 892) or by setting *WXTRACE environment variable* (p. 1854). The predefined string trace masks used by wxWidgets are:

- wxTRACE_MemAlloc: trace memory allocation (new/delete)

- wxTRACE_Messages: trace window messages/X callbacks

- wxTRACE_ResAlloc: trace GDI resource allocation

- wxTRACE_RefCount: trace various ref counting operations

- wxTRACE_OleCalls: trace OLE method calls (Win32 only)

Caveats: since both the mask and the format string are strings, this might lead to function signature confusion in some cases: if you intend to call the format string only version of wxLogTrace, then add a %s format string parameter and then supply a

second string parameter for that %s, the string mask version of wxLogTrace will erroneously get called instead, since you are supplying two string parameters to the function. In this case you'll unfortunately have to avoid having two leading string parameters, e.g. by adding a bogus integer (with its %d format string).

The third version of the function only logs the message if all the bits corresponding to the *mask* are set in the wxLog trace mask which can be set by *SetTraceMask* (p. 895). This version is less flexible than the previous one because it doesn't allow defining the user trace masks easily - this is why it is deprecated in favour of using string trace masks.

- wxTraceMemAlloc: trace memory allocation (new/delete)

- wxTraceMessages: trace window messages/X callbacks

- wxTraceResAlloc: trace GDI resource allocation

- wxTraceRefCount: trace various ref counting operations

- wxTraceOleCalls: trace OLE method calls (Win32 only)

::wxSafeShowMessage

void wxSafeShowMessage(**const wxString&** *title*, **const wxString&** *text*)

This function shows a message to the user in a safe way and should be safe to call even before the application has been initialized or if it is currently in some other strange state (for example, about to crash). Under Windows this function shows a message box using a native dialog instead of*wxMessageBox* (p. 1621) (which might be unsafe to call), elsewhere it simply prints the message to the standard output using the title as prefix.

Parameters

title

The title of the message box shown to the user or the prefix of the message string

text

The text to show to the user

See also

wxLogFatalError (p. 1649)

Include files

<wx/log.h>

::wxSysErrorCode

unsigned long wxSysErrorCode()

Returns the error code from the last system call. This function uses `errno` on Unix platforms and `GetLastError` under Win32.

See also

wxSysErrorMsg (p. 1653), *wxLogSysError* (p. 1650)

::wxSysErrorMsg

const wxChar * wxSysErrorMsg(**unsigned long** *errCode = 0*)

Returns the error message corresponding to the given system error code. If *errCode* is 0 (default), the last error code (as returned by *wxSysErrorCode* (p. 1652)) is used.

See also

wxSysErrorCode (p. 1652), *wxLogSysError* (p. 1650)

WXTRACE

Include files

<wx/object.h>

WXTRACE(formatString, ...)

NB: This macro is now obsolete, replaced by *Log functions* (p. 1648).

Calls wxTrace with printf-style variable argument syntax. Output is directed to the current output stream (see *wxDebugContext* (p. 1747)).

Include files

<wx/memory.h>

WXTRACELEVEL

WXTRACELEVEL(level, formatString, ...)

NB: This function is now obsolete, replaced by *Log functions* (p. 1648).

Calls wxTraceLevel with printf-style variable argument syntax. Output is directed to the current output stream (see *wxDebugContext* (p. 1747)). The first argument should be the level at which this information is appropriate. It will only be output if the level returned by wxDebugContext::GetLevel is equal to or greater than this value.

Include files

<wx/memory.h>

::wxTrace

void wxTrace(**const wxString&** *fmt*, **...**)

NB: This function is now obsolete, replaced by *Log functions* (p. 1648).

Takes printf-style variable argument syntax. Output is directed to the current output stream (see *wxDebugContext* (p. 1747)).

Include files

<wx/memory.h>

::wxTraceLevel

void wxTraceLevel(**int** *level*, **const wxString&** *fmt*, **...**)

NB: This function is now obsolete, replaced by *Log functions* (p. 1648).

Takes printf-style variable argument syntax. Output is directed to the current output stream (see *wxDebugContext* (p. 1747)). The first argument should be the level at which this information is appropriate. It will only be output if the level returned by wxDebugContext::GetLevel is equal to or greater than this value.

Include files

<wx/memory.h>

Time functions

The functions in this section deal with getting the current time and starting/stopping the global timers. Please note that the timer functions are deprecated because they work with one global timer only and*wxTimer* (p. 1386) and/or *wxStopWatch* (p. 1274) classes should be used instead. For retrieving the current time, you may also use*wxDateTime::Now* (p. 267) or*wxDateTime::UNow* (p. 268) methods.

::wxGetElapsedTime

long wxGetElapsedTime(**bool** *resetTimer = true*)

Gets the time in milliseconds since the last *::wxStartTimer* (p. 1656).

If *resetTimer* is true (the default), the timer is reset to zero by this call.

See also *wxTimer* (p. 1386).

Include files

<wx/timer.h>

::wxGetLocalTime

long wxGetLocalTime()

Returns the number of seconds since local time 00:00:00 Jan 1st 1970.

See also

wxDateTime::Now (p. 267)

Include files

<wx/timer.h>

::wxGetLocalTimeMillis

wxLongLong wxGetLocalTimeMillis()

Returns the number of milliseconds since local time 00:00:00 Jan 1st 1970.

See also

wxDateTime::Now (p. 267),
wxLongLong (p. 903)

Include files

<wx/timer.h>

::wxGetUTCTime

long wxGetUTCTime()

Returns the number of seconds since GMT 00:00:00 Jan 1st 1970.

See also

wxDateTime::Now (p. 267)

Include files

<wx/timer.h>

::wxMicroSleep

void wxMicroSleep(**unsigned long** *microseconds*)

Sleeps for the specified number of microseconds. The microsecond resolution may not, in fact, be available on all platforms (currently only Unix platforms with nanosleep(2) may provide it) in which case this is the same as*wxMilliSleep* (p. 1656)(*microseconds*/1000).

Include files

<wx/utils.h>

::wxMilliSleep

void wxMilliSleep(**unsigned long** *milliseconds*)

Sleeps for the specified number of milliseconds. Notice that usage of this function is encouraged instead of calling usleep(3) directly because the standard usleep() function is not MT safe.

Include files

<wx/utils.h>

::wxNow

wxString wxNow()

Returns a string representing the current date and time.

Include files

<wx/utils.h>

::wxSleep

void wxSleep(**int** *secs*)

Sleeps for the specified number of seconds.

Include files

<wx/utils.h>

::wxStartTimer

void wxStartTimer()

Starts a stopwatch; use *::wxGetElapsedTime* (p. 1654) to get the elapsed time.

See also *wxTimer* (p. 1386).

Include files

<wx/timer.h>

::wxUsleep

void wxUsleep(**unsigned long** *milliseconds*)

This function is deprecated because its name is misleading: notice that the argument is in milliseconds, not microseconds. Please use either*wxMilliSleep* (p. 1656) or *wxMicroSleep* (p. 1655)depending on the resolution you need.

Debugging macros and functions

Useful macros and functions for error checking and defensive programming. wxWidgets defines three families of the assert-like macros: the wxASSERT and wxFAIL macros only do anything if __WXDEBUG__ is defined (in other words, in the debug build) but disappear completely in the release build. On the other hand, the wxCHECK macros stay event in release builds but a check failure doesn't generate any user-visible effects then. Finally, the compile time assertions don't happen during the run-time but result in the compilation error messages if the condition they check fail.

Include files

<wx/debug.h>

::wxOnAssert

void wxOnAssert(**const char** **fileName*, **int** *lineNumber*, **const char** **cond*, **const char** **msg = NULL*)

This function is called whenever one of debugging macros fails (i.e. condition is false in an assertion). It is only defined in the debug mode, in release builds the *wxCHECK* (p. 1659) failures don't result in anything.

To override the default behaviour in the debug builds which is to show the user a dialog asking whether he wants to abort the program, continue or continue ignoring any subsequent assert failures, you may override*wxApp::OnAssert* (p. 35) which is called by this function if the global application object exists.

wxASSERT

 wxASSERT(*condition*)

Assert macro. An error message will be generated if the condition is false in debug mode, but nothing will be done in the release build.

Please note that the condition in wxASSERT() should have no side effects because it will not be executed in release mode at all.

See also

wxASSERT_MSG (p. 1658),

wxASSERT_MIN_BITSIZE

wxASSERT_MIN_BITSIZE(*type*, *size*)

This macro results in a *compile time assertion failure* (p. 1658) if the size of the given type *type* is less than *size* bits.

You may use it like this, for example:

```
// we rely on the int being able to hold values up to 2^32
wxASSERT_MIN_BITSIZE(int, 32);

// can't work with the platforms using UTF-8 for wchar_t
wxASSERT_MIN_BITSIZE(wchar_t, 16);
```

wxASSERT_MSG

wxASSERT_MSG(*condition*, *msg*)

Assert macro with message. An error message will be generated if the condition is false.

See also

wxASSERT (p. 1657),
wxCOMPILE_TIME_ASSERT (p. 1658)

wxCOMPILE_TIME_ASSERT

wxCOMPILE_TIME_ASSERT(*condition*, *msg*)

Using `wxCOMPILE_TIME_ASSERT` results in a compilation error if the specified *condition* is false. The compiler error message should include the *msg* identifier - please note that it must be a valid C++ identifier and not a string unlike in the other cases.

This macro is mostly useful for testing the expressions involving the `sizeof` operator as they can't be tested by the preprocessor but it is sometimes desirable to test them at the compile time.

Note that this macro internally declares a struct whose name it tries to make unique by using the `__LINE__` in it but it may still not work if you use it on the same line in two different source files. In this case you may either change the line in which either of them appears on or use the *wxCOMPILE_TIME_ASSERT2* (p. 1659) macro.

Also note that Microsoft Visual C++ has a bug which results in compiler errors if you use this macro with 'Program Database For Edit And Continue' (`/ZI`) option, so you shouldn't use it ('Program Database' (`/Zi`) is ok though) for the code making use of this macro.

See also

wxASSERT_MSG (p. 1658),
wxASSERT_MIN_BITSIZE (p. 1658)

wxCOMPILE_TIME_ASSERT2

wxCOMPILE_TIME_ASSERT(*condition*, *msg*, *name*)

This macro is identical to *wxCOMPILE_TIME_ASSERT2* (p. 1659)except that it allows you to specify a unique *name* for the struct internally defined by this macro to avoid getting the compilation errors described *above* (p. 1658).

wxFAIL

wxFAIL()

Will always generate an assert error if this code is reached (in debug mode).

See also: *wxFAIL_MSG* (p. 1659)

wxFAIL_MSG

wxFAIL_MSG(*msg*)

Will always generate an assert error with specified message if this code is reached (in debug mode).

This macro is useful for marking unreachable" code areas, for example it may be used in the "default:" branch of a switch statement if all possible cases are processed above.

See also

wxFAIL (p. 1659)

wxCHECK

wxCHECK(*condition*, *retValue*)

Checks that the condition is true, returns with the given return value if not (FAILs in debug mode). This check is done even in release mode.

wxCHECK_MSG

wxCHECK_MSG(*condition*, *retValue*, *msg*)

Checks that the condition is true, returns with the given return value if not (FAILs in debug mode). This check is done even in release mode.

This macro may be only used in non void functions, see also*wxCHECK_RET* (p. 1660).

wxCHECK_RET

wxCHECK_RET(*condition*, *msg*)

Checks that the condition is true, and returns if not (FAILs with given error message in debug mode). This check is done even in release mode.

This macro should be used in void functions instead of*wxCHECK_MSG* (p. 1659).

wxCHECK2

wxCHECK2(*condition*, *operation*)

Checks that the condition is true and *wxFAIL* (p. 1659) and execute*operation* if it is not. This is a generalisation of*wxCHECK* (p. 1659) and may be used when something else than just returning from the function must be done when the *condition* is false.

This check is done even in release mode.

wxCHECK2_MSG

wxCHECK2(*condition*, *operation*, *msg*)

This is the same as *wxCHECK2* (p. 1660), but*wxFAIL_MSG* (p. 1659) with the specified *msg* is called instead of wxFAIL() if the *condition* is false.

::wxTrap

void wxTrap()

In debug mode (when __WXDEBUG__ is defined) this function generates a debugger exception meaning that the control is passed to the debugger if one is attached to the process. Otherwise the program just terminates abnormally.

In release mode this function does nothing.

Include files

<wx/debug.h>

::wxIsDebuggerRunning

bool wxIsDebuggerRunning()

Returns `true` if the program is running under debugger, `false` otherwise.

Please note that this function is currently only implemented for Win32 and Mac builds using CodeWarrior and always returns `false` elsewhere.

Environment access functions

The functions in this section allow to access (get) or change value of environment variables in a portable way. They are currently implemented under Win32 and POSIX-like systems (Unix).

Include files

<wx/utils.h>

wxGetenv

wxChar * wxGetEnv(**const wxString&** *var*)

This is a macro defined as `getenv()` or its wide char version in Unicode mode.

Note that under Win32 it may not return correct value for the variables set with *wxSetEnv* (p. 1661), use *wxGetEnv* (p. 1661) function instead.

wxGetEnv

bool wxGetEnv(**const wxString&** *var*, **wxString ****value*)

Returns the current value of the environment variable *var* in *value*.*value* may be `NULL` if you just want to know if the variable exists and are not interested in its value.

Returns `true` if the variable exists, `false` otherwise.

wxSetEnv

bool wxSetEnv(**const wxString&** *var*, **const wxChar ****value*)

Sets the value of the environment variable *var* (adding it if necessary) to *value*.

Returns `true` on success.

wxUnsetEnv

bool wxUnsetEnv(**const wxString&** *var*)

Removes the variable *var* from the environment.*wxGetEnv* (p. 1661) will return `NULL` after the call to this function.

Returns `true` on success.

Chapter 9 Constants

This chapter describes the constants defined by wxWidgets.

Preprocesser symbols defined by wxWidgets

These are preprocessor symbols used in the wxWidgets source, grouped by category (and sorted by alphabetical order inside each category). All of these macros except for the `wxUSE_XXX` variety is defined if the corresponding condition is true and undefined if it isn't, so they should be always tested using `ifdef` and not `if`.

GUI system

__WINDOWS__	any Windows, yom may also use __WXMSW__
__WIN16__	Win16 API (not supported since wxWidgets 2.6)
__WIN32__	Win32 API
__WIN95__	Windows 95 or NT 4.0 and above system (not NT 3.5x)
__WXBASE__	Only wxBase, no GUI features (same as `wxUSE_GUI` == 0)
__WXCOCOA__	OS X using Cocoa API
__WXWINCE__	Windows CE
__WXGTK__	GTK+
__WXGTK12__	GTK+ 1.2 or higher
__WXGTK20__	GTK+ 2.0 or higher
__WXMOTIF__	Motif
__WXMOTIF20__	Motif 2.0 or higher
__WXMAC__	Mac OS all targets
__WXMAC_CLASSIC__	MacOS for Classic

__WXMAC_CARBON__	MacOS for Carbon CFM (running under Classic or OSX) or true OS X Mach-O Builds
__WXMAC_OSX__	MacOS X Carbon Mach-O Builds
__WXMGL__	SciTech Soft MGL (__WXUNIVERSAL__ will be also defined)
__WXMSW__	Any Windows
__WXOS2__	Identical to __WXPM__
__WXOSX__	Any Mac OS X port (either Carbon or Cocoa)
__WXPALMOS__	PalmOS
__WXPM__	OS/2 native Presentation Manager
__WXSTUBS__	Stubbed version ('template' wxWin implementation)
__WXXT__	Xt; mutually exclusive with WX_MOTIF, not implemented in wxWidgets 2.x
__WXX11__	wxX11 (__WXUNIVERSAL__ will be also defined)
__WXWINE__	WINE (i.e. WIN32 on Unix)
__WXUNIVERSAL__	wxUniversal port, always defined in addition to one of the symbols above so this should be tested first.
__X__	any X11-based GUI toolkit except GTK+

There are two wxWidgets ports to Mac OS. One of them, wxMac, exists in two versions: Classic and Carbon. The Classic version is the only one to work on Mac OS version 8. The Carbon version may be built either as CFM or Mach-O (binary format, like ELF) and the former may run under OS 9 while the latter only runs under OS X. Finally, there is a new Cocoa port which can only be used under OS X. To summarize:

- If you want to test for all Mac platforms, classic and OS X, you should test both `__WXMAC__` and `__WXCOCOA__`.

- If you want to test for any GUI Mac port under OS X, use `__WXOSX__`.

- If you want to test for any port under Mac OS X, including, for example, wxGTK and also wxBase, use `__DARWIN__` (see below).

The convention is to use the `__WX` prefix for these symbols, although this has not always been followed.

Operating systems

__APPLE__	any Mac OS version
__AIX__	AIX
__BSD__	Any *BSD system
__CYGWIN__	Cygwin: Unix on Win32
__DARWIN__	Mac OS X using the BSD Unix C library (as opposed to using the Metrowerks MSL C/C++ library)
__DATA_GENERAL__	DG-UX
__DOS_GENERAL__	DOS (used with wxMGL only)
__FREEBSD__	FreeBSD
__HPUX__	HP-UX (Unix)
__GNU__	GNU Hurd
__LINUX__	Linux
__MACH__	Mach-O Architecture (Mac OS X only builds)
__OSF__	OSF/1
__PALMOS__	PalmOS
__SGI__	IRIX
__SOLARIS__	Solaris
__SUN__	Any Sun
__SUNOS__	Sun OS
__SVR4__	SystemV R4
__SYSV__	SystemV generic
__ULTRIX__	Ultrix
__UNIX__	any Unix
__UNIX_LIKE__	Unix, BeOS or VMS
__VMS__	VMS
__WINDOWS__	any Windows

Hardware architectures (CPU)

Note that not all of these symbols are always defined, it depends on the compiler used.

__ALPHA__	DEC Alpha architecture
__INTEL__	Intel i386 or compatible
__IA64__	Intel 64 bit architecture
__POWERPC__	Motorola Power PC

Hardware type

__SMARTPHONE__	Generic mobile devices with phone buttons and a small display
__PDA__	Personal digital assistant, usually with touch screen
__HANDHELD__	Small but powerful computer, usually with a keyboard
__POCKETPC__	Microsoft-powered PocketPC devices with touch-screen
__WINCE_STANDARDSDK__	Microsoft-powered Windows CE devices, for generic Windows CE applications
__WINCE_NET__	Microsoft-powered Windows CE .NET devices (_WIN32_WCE is 400 or greater)
WIN32_PLATFORM_WFSP	Microsoft-powered smartphone

Compilers

__BORLANDC__	Borland C++. The value of the macro corresponds to the compiler version: 500 is 5.0.
__DJGPP__	DJGPP
__DIGITALMARS__	Digital Mars
__GNUG__	Gnu C++ on any platform, see also *wxCHECK_GCC_VERSION* (p. 1588)
__GNUWIN32__	Gnu-Win32 compiler, see also *wxCHECK_W32API_VERSION* (p. 1589)
__MINGW32__	MinGW
__MWERKS__	CodeWarrior MetroWerks compiler
__SUNCC__	Sun CC
__SYMANTECC__	Symantec C++

__VISAGECPP__	IBM Visual Age (OS/2)
__VISUALC__	Microsoft Visual C++. The value of this macro corresponds to the compiler version: 1020 for 4.2 (the first supported version), 1100 for 5.0, 1200 for 6.0 and so on
__XLC__	AIX compiler
__WATCOMC__	Watcom C++. The value of this macro corresponds to the compiler version, 1100 is 11.0 and 1200 is OpenWatcom.
_WIN32_WCE	Windows CE version

Miscellaneous

__WXWINDOWS__	always defined in wxWidgets applications, see also *wxCHECK_VERSION* (p. 1588)
__WXDEBUG__	defined in debug mode, undefined in release mode
wxUSE_XXX	if defined as 1, feature XXX is active (the symbols of this form are always defined, use #if and not #ifdef to test for them)
_UNICODE and UNICODE	both are defined if wxUSE_UNICODE is set to 1
wxUSE_GUI	this particular feature test macro is defined to 1when compiling or using the library with the GUI features activated, if it is defined as 0, only wxBase is available.
wxUSE_BASE	only used by wxWidgets internally (defined as 1 when building wxBase code, either as a standalone library or as part of the monolithic wxWidgets library, defined as 0 when building GUI library only)
wxNO_RTTI	is defined if the compiler RTTI support has been switched off
wxNO_EXCEPTIONS	is defined if the compiler support for C++ exceptions has been switched off
wxNO_THREADS	if this macro is defined, the compilation options don't include compiler flags needed for multithreaded code generation. This implies that wxUSE_THREADS is 0 and also that other (non wx-based) threading packages cannot be

used neither.

Standard event identifiers

wxWidgets defines a special identifier value `wxID_ANY` which is used in the following two situations:

- when creating a new window you may specify `wxID_ANY` to let wxWidgets assign an unused identifier to it automatically

- when installing an event handler using either the event table macros or *wxEvtHandler::Connect* (p. 488), you may use it to indicate that you want to handle the events coming from any control, regardless of its identifier

wxWidgets also defines a few standard command identifiers which may be used by the user code and also are sometimes used by wxWidgets itself. These reserved identifiers are all in the range between `wxID_LOWEST` and `wxID_HIGHEST` and, accordingly, the user code should avoid defining its own constants in this range.

```
wxID_LOWEST = 4999,

wxID_OPEN,
wxID_CLOSE,
wxID_NEW,
wxID_SAVE,
wxID_SAVEAS,
wxID_REVERT,
wxID_EXIT,
wxID_UNDO,
wxID_REDO,
wxID_HELP,
wxID_PRINT,
wxID_PRINT_SETUP,
wxID_PREVIEW,
wxID_ABOUT,
wxID_HELP_CONTENTS,
wxID_HELP_COMMANDS,
wxID_HELP_PROCEDURES,
wxID_HELP_CONTEXT,
wxID_CLOSE_ALL,

wxID_CUT = 5030,
wxID_COPY,
wxID_PASTE,
wxID_CLEAR,
wxID_FIND,
wxID_DUPLICATE,
wxID_SELECTALL,
wxID_DELETE,
wxID_REPLACE,
wxID_REPLACE_ALL,
wxID_PROPERTIES,

wxID_VIEW_DETAILS,
wxID_VIEW_LARGEICONS,
wxID_VIEW_SMALLICONS,
wxID_VIEW_LIST,
```

```
                wxID_VIEW_SORTDATE,
                wxID_VIEW_SORTNAME,
                wxID_VIEW_SORTSIZE,
                wxID_VIEW_SORTTYPE,

                wxID_FILE1 = 5050,
                wxID_FILE2,
                wxID_FILE3,
                wxID_FILE4,
                wxID_FILE5,
                wxID_FILE6,
                wxID_FILE7,
                wxID_FILE8,
                wxID_FILE9,

                // Standard button IDs
                wxID_OK = 5100,
                wxID_CANCEL,
                wxID_APPLY,
                wxID_YES,
                wxID_NO,
                wxID_STATIC,
                wxID_FORWARD,
                wxID_BACKWARD,
                wxID_DEFAULT,
                wxID_MORE,
                wxID_SETUP,
                wxID_RESET,
                wxID_CONTEXT_HELP,
                wxID_YESTOALL,
                wxID_NOTOALL,
                wxID_ABORT,
                wxID_RETRY,
                wxID_IGNORE,

                wxID_UP,
                wxID_DOWN,
                wxID_HOME,
                wxID_REFRESH,
                wxID_STOP,
                wxID_INDEX,

                wxID_BOLD,
                wxID_ITALIC,
                wxID_JUSTIFY_CENTER,
                wxID_JUSTIFY_FILL,
                wxID_JUSTIFY_RIGHT,
                wxID_JUSTIFY_LEFT,
                wxID_UNDERLINE,
                wxID_INDENT,
                wxID_UNINDENT,
                wxID_ZOOM_100,
                wxID_ZOOM_FIT,
                wxID_ZOOM_IN,
                wxID_ZOOM_OUT,
                wxID_UNDELETE,
                wxID_REVERT_TO_SAVED,

                // System menu IDs (used by wxUniv):
                wxID_SYSTEM_MENU = 5200,
                wxID_CLOSE_FRAME,
                wxID_MOVE_FRAME,
                wxID_RESIZE_FRAME,
```

```
      wxID_MAXIMIZE_FRAME,
      wxID_ICONIZE_FRAME,
      wxID_RESTORE_FRAME,

      // IDs used by generic file dialog (13 consecutive starting from this
value)
      wxID_FILEDLGG = 5900,

      wxID_HIGHEST = 5999
```

Keycodes

Keypresses are represented by an enumerated type, wxKeyCode. The possible values are the ASCII character codes, plus the following:

```
WXK_BACK    = 8
WXK_TAB     = 9
WXK_RETURN  = 13
WXK_ESCAPE  = 27
WXK_SPACE   = 32
WXK_DELETE  = 127

WXK_START   = 300
WXK_LBUTTON
WXK_RBUTTON
WXK_CANCEL
WXK_MBUTTON
WXK_CLEAR
WXK_SHIFT
WXK_CONTROL
WXK_MENU
WXK_PAUSE
WXK_CAPITAL
WXK_PRIOR
WXK_NEXT
WXK_END
WXK_HOME
WXK_LEFT
WXK_UP
WXK_RIGHT
WXK_DOWN
WXK_SELECT
WXK_PRINT
WXK_EXECUTE
WXK_SNAPSHOT
WXK_INSERT
WXK_HELP
WXK_NUMPAD0
WXK_NUMPAD1
WXK_NUMPAD2
WXK_NUMPAD3
WXK_NUMPAD4
WXK_NUMPAD5
WXK_NUMPAD6
WXK_NUMPAD7
WXK_NUMPAD8
WXK_NUMPAD9
WXK_MULTIPLY
WXK_ADD
```

```
WXK_SEPARATOR
WXK_SUBTRACT
WXK_DECIMAL
WXK_DIVIDE
WXK_F1
WXK_F2
WXK_F3
WXK_F4
WXK_F5
WXK_F6
WXK_F7
WXK_F8
WXK_F9
WXK_F10
WXK_F11
WXK_F12
WXK_F13
WXK_F14
WXK_F15
WXK_F16
WXK_F17
WXK_F18
WXK_F19
WXK_F20
WXK_F21
WXK_F22
WXK_F23
WXK_F24
WXK_NUMLOCK
WXK_SCROLL
WXK_PAGEUP,
WXK_PAGEDOWN,

WXK_NUMPAD_SPACE,
WXK_NUMPAD_TAB,
WXK_NUMPAD_ENTER,
WXK_NUMPAD_F1,
WXK_NUMPAD_F2,
WXK_NUMPAD_F3,
WXK_NUMPAD_F4,
WXK_NUMPAD_HOME,
WXK_NUMPAD_LEFT,
WXK_NUMPAD_UP,
WXK_NUMPAD_RIGHT,
WXK_NUMPAD_DOWN,
WXK_NUMPAD_PRIOR,
WXK_NUMPAD_PAGEUP,
WXK_NUMPAD_NEXT,
WXK_NUMPAD_PAGEDOWN,
WXK_NUMPAD_END,
WXK_NUMPAD_BEGIN,
WXK_NUMPAD_INSERT,
WXK_NUMPAD_DELETE,
WXK_NUMPAD_EQUAL,
WXK_NUMPAD_MULTIPLY,
WXK_NUMPAD_ADD,
WXK_NUMPAD_SEPARATOR,
WXK_NUMPAD_SUBTRACT,
WXK_NUMPAD_DECIMAL,
WXK_NUMPAD_DIVIDE,

// the following key codes are only generated under Windows currently
WXK_WINDOWS_LEFT,
```

```
        WXK_WINDOWS_RIGHT,
        WXK_WINDOWS_MENU,
        WXK_COMMAND,

        // Hardware-specific buttons
        WXK_SPECIAL1 = 193,
        WXK_SPECIAL2,
        WXK_SPECIAL3,
        WXK_SPECIAL4,
        WXK_SPECIAL5,
        WXK_SPECIAL6,
        WXK_SPECIAL7,
        WXK_SPECIAL8,
        WXK_SPECIAL9,
        WXK_SPECIAL10,
        WXK_SPECIAL11,
        WXK_SPECIAL12,
        WXK_SPECIAL13,
        WXK_SPECIAL14,
        WXK_SPECIAL15,
        WXK_SPECIAL16,
        WXK_SPECIAL17,
        WXK_SPECIAL18,
        WXK_SPECIAL19,
        WXK_SPECIAL20
```

Language identifiers

The following wxLanguage constants may be used to specify the language in
wxLocale::Init (p. 888) and are returned by *wxLocale::GetSystemLanguage* (p. 887):

wxLANGUAGE_DEFAULT   user's default language as obtained from the
operating system

wxLANGUAGE_UNKNOWN   returned by *GetSystemLanguage* (p. 887) if it
fails to detect the default language

wxLANGUAGE_USER_DEFINED   user defined languages' integer identifiers
should start from this

wxLANGUAGE_ABKHAZIAN

wxLANGUAGE_AFAR

wxLANGUAGE_AFRIKAANS

wxLANGUAGE_ALBANIAN

wxLANGUAGE_AMHARIC

wxLANGUAGE_ARABIC

wxLANGUAGE_ARABIC_ALGERIA

wxLANGUAGE_ARABIC_BAHRAIN

wxLANGUAGE_ARABIC_EGYPT

wxLANGUAGE_ARABIC_IRAQ

wxLANGUAGE_ARABIC_JORDAN

wxLANGUAGE_ARABIC_KUWAIT

wxLANGUAGE_ARABIC_LEBANON

wxLANGUAGE_ARABIC_LIBYA

wxLANGUAGE_ARABIC_MOROCCO

wxLANGUAGE_ARABIC_OMAN

wxLANGUAGE_ARABIC_QATAR

wxLANGUAGE_ARABIC_SAUDI_ARABIA

wxLANGUAGE_ARABIC_SUDAN

wxLANGUAGE_ARABIC_SYRIA

wxLANGUAGE_ARABIC_TUNISIA

wxLANGUAGE_ARABIC_UAE

wxLANGUAGE_ARABIC_YEMEN

wxLANGUAGE_ARMENIAN

wxLANGUAGE_ASSAMESE

wxLANGUAGE_AYMARA

wxLANGUAGE_AZERI

wxLANGUAGE_AZERI_CYRILLIC

wxLANGUAGE_AZERI_LATIN

wxLANGUAGE_BASHKIR

wxLANGUAGE_BASQUE

wxLANGUAGE_BELARUSIAN

wxLANGUAGE_BENGALI

wxLANGUAGE_BHUTANI

wxLANGUAGE_BIHARI

wxLANGUAGE_BISLAMA

wxLANGUAGE_BRETON

wxLANGUAGE_BULGARIAN

wxLANGUAGE_BURMESE

wxLANGUAGE_CAMBODIAN

wxLANGUAGE_CATALAN

wxLANGUAGE_CHINESE

wxLANGUAGE_CHINESE_SIMPLIFIED

wxLANGUAGE_CHINESE_TRADITIONAL

wxLANGUAGE_CHINESE_HONGKONG

wxLANGUAGE_CHINESE_MACAU

wxLANGUAGE_CHINESE_SINGAPORE

wxLANGUAGE_CHINESE_TAIWAN

wxLANGUAGE_CORSICAN

wxLANGUAGE_CROATIAN

wxLANGUAGE_CZECH

wxLANGUAGE_DANISH

wxLANGUAGE_DUTCH

wxLANGUAGE_DUTCH_BELGIAN

wxLANGUAGE_ENGLISH

wxLANGUAGE_ENGLISH_UK

wxLANGUAGE_ENGLISH_US

wxLANGUAGE_ENGLISH_AUSTRALIA

wxLANGUAGE_ENGLISH_BELIZE

wxLANGUAGE_ENGLISH_BOTSWANA

wxLANGUAGE_ENGLISH_CANADA

wxLANGUAGE_ENGLISH_CARIBBEAN

wxLANGUAGE_ENGLISH_DENMARK

wxLANGUAGE_ENGLISH_EIRE

wxLANGUAGE_ENGLISH_JAMAICA

wxLANGUAGE_ENGLISH_NEW_ZEALAND

wxLANGUAGE_ENGLISH_PHILIPPINES

wxLANGUAGE_ENGLISH_SOUTH_AFRICA

wxLANGUAGE_ENGLISH_TRINIDAD

wxLANGUAGE_ENGLISH_ZIMBABWE

wxLANGUAGE_ESPERANTO

wxLANGUAGE_ESTONIAN

wxLANGUAGE_FAEROESE

wxLANGUAGE_FARSI

wxLANGUAGE_FIJI

wxLANGUAGE_FINNISH

wxLANGUAGE_FRENCH

wxLANGUAGE_FRENCH_BELGIAN

wxLANGUAGE_FRENCH_CANADIAN

wxLANGUAGE_FRENCH_LUXEMBOURG

wxLANGUAGE_FRENCH_MONACO

wxLANGUAGE_FRENCH_SWISS

wxLANGUAGE_FRISIAN

wxLANGUAGE_GALICIAN

wxLANGUAGE_GEORGIAN

wxLANGUAGE_GERMAN

wxLANGUAGE_GERMAN_AUSTRIAN

wxLANGUAGE_GERMAN_BELGIUM

wxLANGUAGE_GERMAN_LIECHTENSTEIN

wxLANGUAGE_GERMAN_LUXEMBOURG

wxLANGUAGE_GERMAN_SWISS

wxLANGUAGE_GREEK

wxLANGUAGE_GREENLANDIC

wxLANGUAGE_GUARANI

wxLANGUAGE_GUJARATI

wxLANGUAGE_HAUSA

wxLANGUAGE_HEBREW

wxLANGUAGE_HINDI

wxLANGUAGE_HUNGARIAN

wxLANGUAGE_ICELANDIC

wxLANGUAGE_INDONESIAN

wxLANGUAGE_INTERLINGUA

wxLANGUAGE_INTERLINGUE

wxLANGUAGE_INUKTITUT

wxLANGUAGE_INUPIAK

wxLANGUAGE_IRISH

wxLANGUAGE_ITALIAN

wxLANGUAGE_ITALIAN_SWISS

wxLANGUAGE_JAPANESE

wxLANGUAGE_JAVANESE

wxLANGUAGE_KANNADA

wxLANGUAGE_KASHMIRI

wxLANGUAGE_KASHMIRI_INDIA

wxLANGUAGE_KAZAKH

wxLANGUAGE_KERNEWEK

wxLANGUAGE_KINYARWANDA

wxLANGUAGE_KIRGHIZ

wxLANGUAGE_KIRUNDI

wxLANGUAGE_KONKANI

wxLANGUAGE_KOREAN

wxLANGUAGE_KURDISH

wxLANGUAGE_LAOTHIAN

wxLANGUAGE_LATIN

wxLANGUAGE_LATVIAN

wxLANGUAGE_LINGALA

wxLANGUAGE_LITHUANIAN

wxLANGUAGE_MACEDONIAN

wxLANGUAGE_MALAGASY

wxLANGUAGE_MALAY

wxLANGUAGE_MALAYALAM

wxLANGUAGE_MALAY_BRUNEI_DARUSSALAM

wxLANGUAGE_MALAY_MALAYSIA

wxLANGUAGE_MALTESE

wxLANGUAGE_MANIPURI

wxLANGUAGE_MAORI

wxLANGUAGE_MARATHI

wxLANGUAGE_MOLDAVIAN

wxLANGUAGE_MONGOLIAN

wxLANGUAGE_NAURU

wxLANGUAGE_NEPALI

wxLANGUAGE_NEPALI_INDIA

wxLANGUAGE_NORWEGIAN_BOKMAL

wxLANGUAGE_NORWEGIAN_NYNORSK

wxLANGUAGE_OCCITAN

wxLANGUAGE_ORIYA

wxLANGUAGE_OROMO

wxLANGUAGE_PASHTO

wxLANGUAGE_POLISH

wxLANGUAGE_PORTUGUESE

wxLANGUAGE_PORTUGUESE_BRAZILIAN

wxLANGUAGE_PUNJABI

wxLANGUAGE_QUECHUA

wxLANGUAGE_RHAETO_ROMANCE

wxLANGUAGE_ROMANIAN

wxLANGUAGE_RUSSIAN

wxLANGUAGE_RUSSIAN_UKRAINE

wxLANGUAGE_SAMOAN

wxLANGUAGE_SANGHO

wxLANGUAGE_SANSKRIT

wxLANGUAGE_SCOTS_GAELIC

wxLANGUAGE_SERBIAN

wxLANGUAGE_SERBIAN_CYRILLIC

wxLANGUAGE_SERBIAN_LATIN

wxLANGUAGE_SERBO_CROATIAN

wxLANGUAGE_SESOTHO

wxLANGUAGE_SETSWANA

wxLANGUAGE_SHONA

wxLANGUAGE_SINDHI

wxLANGUAGE_SINHALESE

wxLANGUAGE_SISWATI

wxLANGUAGE_SLOVAK

wxLANGUAGE_SLOVENIAN

wxLANGUAGE_SOMALI

wxLANGUAGE_SPANISH

wxLANGUAGE_SPANISH_ARGENTINA

wxLANGUAGE_SPANISH_BOLIVIA

wxLANGUAGE_SPANISH_CHILE

wxLANGUAGE_SPANISH_COLOMBIA

wxLANGUAGE_SPANISH_COSTA_RICA

wxLANGUAGE_SPANISH_DOMINICAN_REPUBLIC

wxLANGUAGE_SPANISH_ECUADOR

wxLANGUAGE_SPANISH_EL_SALVADOR

wxLANGUAGE_SPANISH_GUATEMALA

wxLANGUAGE_SPANISH_HONDURAS

wxLANGUAGE_SPANISH_MEXICAN

wxLANGUAGE_SPANISH_MODERN

wxLANGUAGE_SPANISH_NICARAGUA

wxLANGUAGE_SPANISH_PANAMA

wxLANGUAGE_SPANISH_PARAGUAY

wxLANGUAGE_SPANISH_PERU

wxLANGUAGE_SPANISH_PUERTO_RICO

wxLANGUAGE_SPANISH_URUGUAY

wxLANGUAGE_SPANISH_US

wxLANGUAGE_SPANISH_VENEZUELA

wxLANGUAGE_SUNDANESE

wxLANGUAGE_SWAHILI

wxLANGUAGE_SWEDISH

wxLANGUAGE_SWEDISH_FINLAND

wxLANGUAGE_TAGALOG

wxLANGUAGE_TAJIK

wxLANGUAGE_TAMIL

wxLANGUAGE_TATAR

wxLANGUAGE_TELUGU

wxLANGUAGE_THAI

wxLANGUAGE_TIBETAN

wxLANGUAGE_TIGRINYA

wxLANGUAGE_TONGA

wxLANGUAGE_TSONGA

wxLANGUAGE_TURKISH

wxLANGUAGE_TURKMEN

wxLANGUAGE_TWI

wxLANGUAGE_UIGHUR

wxLANGUAGE_UKRAINIAN

wxLANGUAGE_URDU

wxLANGUAGE_URDU_INDIA

wxLANGUAGE_URDU_PAKISTAN

wxLANGUAGE_UZBEK

wxLANGUAGE_UZBEK_CYRILLIC

wxLANGUAGE_UZBEK_LATIN

wxLANGUAGE_VIETNAMESE

wxLANGUAGE_VOLAPUK

wxLANGUAGE_WELSH

wxLANGUAGE_WOLOF

wxLANGUAGE_XHOSA

wxLANGUAGE_YIDDISH

wxLANGUAGE_YORUBA

wxLANGUAGE_ZHUANG

wxLANGUAGE_ZULU

Stock items

Window IDs for which stock buttons are created (see *wxButton constructor* (p. 117)):

wxID_ADD	"Add"
wxID_APPLY	"&Apply"
wxID_BOLD	"&Bold"
wxID_CANCEL	"&Cancel"
wxID_CLEAR	"&Clear"
wxID_CLOSE	"&Close"
wxID_COPY	"&Copy"
wxID_CUT	"Cu&t"

wxID_DELETE	"&Delete"
wxID_FIND	"&Find"
wxID_REPLACE	"Find and rep&lace"
wxID_BACKWARD	"&Back"
wxID_DOWN	"&Down"
wxID_FORWARD	"&Forward"
wxID_UP	"&Up"
wxID_HELP	"&Help"
wxID_HOME	"&Home"
wxID_INDENT	"Indent"
wxID_INDEX	"&Index"
wxID_ITALIC	"&Italic"
wxID_JUSTIFY_CENTER	"Centered"
wxID_JUSTIFY_FILL	"Justified"
wxID_JUSTIFY_LEFT	"Align Left"
wxID_JUSTIFY_RIGHT	"Align Right"
wxID_NEW	"&New"
wxID_NO	"&No"
wxID_OK	"&OK"
wxID_OPEN	"&Open"
wxID_PASTE	"&Paste"
wxID_PREFERENCES	"&Preferences"
wxID_PRINT	"&Print"
wxID_PREVIEW	"Print previe&w"
wxID_PROPERTIES	"&Properties"
wxID_EXIT	"&Quit"
wxID_REDO	"&Redo"
wxID_REFRESH	"Refresh"
wxID_REMOVE	"Remove"

wxID_REVERT_TO_SAVED	"Revert to Saved"
wxID_SAVE	"&Save"
wxID_SAVEAS	"Save &As..."
wxID_STOP	"&Stop"
wxID_UNDELETE	"Undelete"
wxID_UNDERLINE	"&Underline"
wxID_UNDO	"&Undo"
wxID_UNINDENT	"&Unindent"
wxID_YES	"&Yes"
wxID_ZOOM_100	"&Actual Size"
wxID_ZOOM_FIT	"Zoom to &Fit"
wxID_ZOOM_IN	"Zoom &In"
wxID_ZOOM_OUT	"Zoom &Out"

Chapter 10 Classes by category

A classification of wxWidgets classes by category.

Managed windows

There are several types of window that are directly controlled by the window manager (such as MS Windows, or the Motif Window Manager). Frames and dialogs are similar in wxWidgets, but only dialogs may be modal.

wxTopLevelWindow (p. 1419)   Any top level window, dialog or frame

wxDialog (p. 406)   Dialog box

wxFrame (p. 579)   Normal frame

wxMDIChildFrame (p. 916)   MDI child frame

wxMDIParentFrame (p. 920)   MDI parent frame

wxMiniFrame (p. 977)   A frame with a small title bar

wxSplashScreen (p. 1235)   Splash screen class

wxPropertySheetDialog (p. 1081)   Property sheet dialog

wxTipWindow (p. 1397)   Shows text in a small window

wxWizard (p. 1549)   A wizard dialog

See also **Common dialogs**.

Miscellaneous windows

The following are a variety of classes that are derived from wxWindow.

wxPanel (p. 1030)   A window whose colour changes according to current user settings

wxScrolledWindow (p. 1150)   Window with automatically managed scrollbars

wxGrid (p. 619)   A grid (table) window

wxSplitterWindow (p. 1239)   Window which can be split vertically or horizontally

wxStatusBar (p. 1267)   Implements the status bar on a frame

wxToolBar (p. 1401)   Toolbar class

wxNotebook (p. 999)   Notebook class

wxListbook (p. 846)	Similar to notebook but using list control
wxChoicebook (p. 143)	Similar to notebook but using choice control
wxSashWindow (p. 1132)	Window with four optional sashes that can be dragged
wxSashLayoutWindow (p. 1129)	Window that can be involved in an IDE-like layout arrangement
wxVScrolledWindow (p. 1484)	As wxScrolledWindow but supports lines of variable height
wxWizardPage (p. 1555)	A base class for the page in wizard dialog.
wxWizardPageSimple (p. 1557)	A page in wizard dialog.

Common dialogs

Overview (p. 1798)

Common dialogs are ready-made dialog classes which are frequently used in an application.

wxDialog (p. 406)	Base class for common dialogs
wxColourDialog (p. 171)	Colour chooser dialog
wxDirDialog (p. 424)	Directory selector dialog
wxFileDialog (p. 511)	File selector dialog
wxFindReplaceDialog (p. 553)	Text search/replace dialog
wxMultiChoiceDialog (p. 993)	Dialog to get one or more selections from a list
wxSingleChoiceDialog (p. 1169)	Dialog to get a single selection from a list and return the string
wxTextEntryDialog (p. 1361)	Dialog to get a single line of text from the user
wxPasswordEntryDialog (p. 1034)	Dialog to get a password from the user
wxFontDialog (p. 571)	Font chooser dialog
wxPageSetupDialog (p. 1018)	Standard page setup dialog
wxPrintDialog (p. 1057)	Standard print dialog
wxMessageDialog (p. 969)	Simple message box dialog
wxWizard (p. 1549)	A wizard dialog.

Controls

Typically, these are small windows which provide interaction with the user. Controls that

are not static can have *validators* (p. 1461) associated with them.

wxControl (p. 215)	The base class for controls
wxButton (p. 117)	Push button control, displaying text
wxBitmapButton (p. 92)	Push button control, displaying a bitmap
wxToggleButton (p. 1398)	A button which stays pressed when clicked by user.
wxCalendarCtrl (p. 121)	Control showing an entire calendar month
wxCheckBox (p. 134)	Checkbox control
wxCheckListBox (p. 137)	A listbox with a checkbox to the left of each item
wxChoice (p. 140)	Choice control (a combobox without the editable area)
wxComboBox (p. 172)	A choice with an editable area
wxDatePickerCtrl (p. 248)	Small date picker control
wxGauge (p. 599)	A control to represent a varying quantity, such as time remaining
wxGenericDirCtrl (p. 608)	A control for displaying a directory tree
wxHtmlListBox (p. 734)	A listbox showing HTML content
wxStaticBox (p. 1260)	A static, or group box for visually grouping related controls
wxListBox (p. 847)	A list of strings for single or multiple selection
wxListCtrl (p. 852)	A control for displaying lists of strings and/or icons, plus a multicolumn report view
wxListView (p. 879)	A simpler interface (*façade* for wxListCtrl in report mode
wxTextCtrl (p. 1340)	Single or multiline text editing control
wxTreeCtrl (p. 1424)	Tree (hierarchy) control
wxScrollBar (p. 1143)	Scrollbar control
wxSpinButton (p. 1227)	A spin or 'up-down' control
wxSpinCtrl (p. 1231)	A spin control - i.e. spin button and text control
wxStaticText (p. 1264)	One or more lines of non-editable text
wxStaticBitmap (p. 1258)	A control to display a bitmap
wxRadioBox (p. 1089)	A group of radio buttons

wxRadioButton (p. 1096)	A round button to be used with others in a mutually exclusive way
wxSlider (p. 1192)	A slider that can be dragged by the user
wxVListBox (p. 1477)	A listbox supporting variable height rows

Menus

wxMenu (p. 940)	Displays a series of menu items for selection
wxMenuBar (p. 953)	Contains a series of menus for use with a frame
wxMenuItem (p. 964)	Represents a single menu item

Window layout

There are two different systems for laying out windows (and dialogs in particular). One is based upon so-called sizers and it requires less typing, thinking and calculating and will in almost all cases produce dialogs looking equally well on all platforms, the other is based on so-called constraints and is deprecated, though still available.

Sizer overview (p. 1771) describes sizer-based layout.

These are the classes relevant to sizer-based layout.

wxSizer (p. 1176)	Abstract base class
wxGridSizer (p. 683)	A sizer for laying out windows in a grid with all fields having the same size
wxFlexGridSizer (p. 554)	A sizer for laying out windows in a flexible grid
wxGridBagSizer (p. 656)	Another grid sizer that lets you specify the cell an item is in, and items can span rows and/or columns.
wxBoxSizer (p. 102)	A sizer for laying out windows in a row or column
wxStaticBoxSizer (p. 1262)	Same as wxBoxSizer, but with a surrounding static box

Constraints overview (p. 1768) describes constraints-based layout.

These are the classes relevant to constraints-based window layout.

wxIndividualLayoutConstraint (p. 811)	Represents a single constraint dimension
wxLayoutConstraints (p. 837)	Represents the constraints for a window class

Device contexts

Overview (p. 1791)

Device contexts are surfaces that may be drawn on, and provide an abstraction that allows parameterisation of your drawing code by passing different device contexts.

wxBufferedDC (p. 111)	A helper device context for double buffered drawing.
wxBufferedPaintDC (p. 112)	A helper device context for double buffered drawing inside **OnPaint**.
wxClientDC (p. 147)	A device context to access the client area outside **OnPaint** events
wxPaintDC (p. 1024)	A device context to access the client area inside **OnPaint** events
wxWindowDC (p. 1547)	A device context to access the non-client area
wxScreenDC (p. 1142)	A device context to access the entire screen
wxDC (p. 365)	The device context base class
wxMemoryDC (p. 935)	A device context for drawing into bitmaps
wxMetafileDC (p. 973)	A device context for drawing into metafiles
wxMirrorDC (p. 979)	A proxy device context allowing for simple mirroring.
wxPostScriptDC (p. 1046)	A device context for drawing into PostScript files
wxPrinterDC (p. 1065)	A device context for drawing to printers

Graphics device interface

Bitmaps overview (p. 1788)

These classes are related to drawing on device contexts and windows.

wxColour (p. 164)	Represents the red, blue and green elements of a colour
wxDCClipper (p. 385)	Wraps the operations of setting and destroying the clipping region
wxBitmap (p. 79)	Represents a bitmap
wxBrush (p. 103)	Used for filling areas on a device context
wxBrushList (p. 109)	The list of previously-created brushes
wxCursor (p. 226)	A small, transparent bitmap representing the cursor
wxFont (p. 558)	Represents fonts
wxFontList (p. 574)	The list of previously-created fonts
wxIcon (p. 766)	A small, transparent bitmap for assigning to frames and drawing on device contexts
wxImage (p. 779)	A platform-independent image class

wxImageList (p. 806)	A list of images, used with some controls
wxMask (p. 906)	Represents a mask to be used with a bitmap for transparent drawing
wxPen (p. 1036)	Used for drawing lines on a device context
wxPenList (p. 1043)	The list of previously-created pens
wxPalette (p. 1027)	Represents a table of indices into RGB values
wxRegion (p. 1111)	Represents a simple or complex region on a window or device context
wxRendererNative (p. 1122)	Abstracts high-level drawing primitives

Events

Overview (p. 1751)

An event object contains information about a specific event. Event handlers (usually member functions) have a single, event argument.

wxActivateEvent (p. 30)	A window or application activation event
wxCalendarEvent (p. 130)	Used with *wxCalendarCtrl* (p. 121)
wxCalculateLayoutEvent (p. 120)	Used to calculate window layout
wxCloseEvent (p. 153)	A close window or end session event
wxCommandEvent (p. 180)	An event from a variety of standard controls
wxContextMenuEvent (p. 211)	An event generated when the user issues a context menu command
wxDialUpEvent (p. 416)	Event send by *wxDialUpManager* (p. 416)
wxDropFilesEvent (p. 466)	A drop files event
wxEraseEvent (p. 482)	An erase background event
wxEvent (p. 483)	The event base class
wxFindDialogEvent (p. 550)	Event sent by*wxFindReplaceDialog* (p. 553)
wxFocusEvent (p. 557)	A window focus event
wxKeyEvent (p. 830)	A keypress event
wxIconizeEvent (p. 776)	An iconize/restore event
wxIdleEvent (p. 776)	An idle event
wxInitDialogEvent (p. 813)	A dialog initialisation event

wxJoystickEvent (p. 827)   A joystick event

wxListEvent (p. 871)   A list control event

wxMaximizeEvent (p. 908)   A maximize event

wxMenuEvent (p. 962)   A menu event

wxMouseCaptureChangedEvent (p. 981)   A mouse capture changed event

wxMouseEvent (p. 982)   A mouse event

wxMoveEvent (p. 992)   A move event

wxNotebookEvent (p. 1008)   A notebook control event

wxNotifyEvent (p. 1011)   A notification event, which can be vetoed

wxPaintEvent (p. 1025)   A paint event

wxProcessEvent (p. 1078)   A process ending event

wxQueryLayoutInfoEvent (p. 1087) Used to query layout information

wxScrollEvent (p. 1159)   A scroll event from sliders, stand-alone scrollbars and spin buttons

wxScrollWinEvent (p. 1162)   A scroll event from scrolled windows

wxSizeEvent (p. 1175)   A size event

wxSocketEvent (p. 1221)   A socket event

wxSpinEvent (p. 1234)   An event from *wxSpinButton* (p. 1227)

wxSysColourChangedEvent (p. 1314)   A system colour change event

wxTimerEvent (p. 1388)   A timer expiration event

wxTreeEvent (p. 1443)   A tree control event

wxUpdateUIEvent (p. 1448)   A user interface update event

wxWizardEvent (p. 1554)   A wizard event

Validators

Overview (p. 1765)

These are the window validators, used for filtering and validating user input.

wxValidator (p. 1461)   Base validator class

wxTextValidator (p. 1374)   Text control validator class

wxGenericValidator (p. 612)   Generic control validator class

Data structures

These are the data structure classes supported by wxWidgets.

Run-time class information system

wxWidgets supports run-time manipulation of class information, and dynamic creation of objects given class names.

Logging features

Overview (p. 1743)

wxWidgets provides several classes and functions for message logging. Please see the *wxLog overview* (p. 1743) for more details.

Debugging features

Overview (p. 1746)

wxWidgets supports some aspects of debugging an application through classes, functions and macros.

wxDebugReportUpload (p. 404)	Class for uploading compressed debug reports via HTTP.
wxDebugReportPreview (p. 403)	Abstract base class for previewing the contents of a debug report.
wxDebugReportPreviewStd (p. 404)	Standard implementation of wxDebugReportPreview.

Networking classes

wxWidgets provides its own classes for socket based networking.

wxDialUpManager (p. 416)	Provides functions to check the status of network connection and to establish one
wxIPV4address (p. 819)	Represents an Internet address
wxIPaddress (p. 817)	Represents an Internet address
wxSocketBase (p. 1203)	Represents a socket base object
wxSocketClient (p. 1219)	Represents a socket client
wxSocketServer (p. 1223)	Represents a socket server
wxSocketEvent (p. 1221)	A socket event
wxFTP (p. 592)	FTP protocol class
wxHTTP (p. 765)	HTTP protocol class
wxURL (p. 1459)	Represents a Universal Resource Locator

Interprocess communication

Overview (p. 1841)

wxWidgets provides simple interprocess communications facilities based on Windows DDE, but available on most platforms using TCP.

wxClient (p. 145), *wxDDEClient* (p. 386)	Represents a client
wxConnection (p. 207), *wxDDEConnection* (p. 387)	Represents the connection between a client and a server
wxServer (p. 1167), *wxDDEServer* (p. 391)	Represents a server

Document/view framework

Overview (p. 1802)

wxWidgets supports a document/view framework which provides housekeeping for a document-centric application.

Printing framework

A printing and previewing framework is implemented to make it relatively straightforward
to provide document printing facilities.

Drag and drop and clipboard classes

wxBitmapDataObject (p. 97)	Bitmap data object class
wxCustomDataObject (p. 232)	Custom data object class
wxClipboard (p. 150)	Clipboard class
wxDropTarget (p. 471)	Drop target class
wxFileDropTarget (p. 516)	File drop target class
wxTextDropTarget (p. 1360)	Text drop target class
wxDropSource (p. 468)	Drop source class

File related classes

wxWidgets has several small classes to work with disk files, see *file classes overview* (p. 1741) for more details.

wxFileName (p. 521)	Operations with the file name and attributes
wxDir (p. 420)	Class for enumerating files/subdirectories.
wxDirTraverser (p. 426)	Class used together with wxDir for recursively enumerating the files/subdirectories
wxFile (p. 502)	Low-level file input/output class.
wxFFile (p. 494)	Another low-level file input/output class.
wxTempFile (p. 1331)	Class to safely replace an existing file
wxTextFile (p. 1363)	Class for working with text files as with arrays of lines
wxStandardPaths (p. 1254)	Paths for standard directories

Stream classes

wxWidgets has its own set of stream classes, as an alternative to often buggy standard stream libraries, and to provide enhanced functionality.

wxStreamBase (p. 1275)	Stream base class
wxStreamBuffer (p. 1277)	Stream buffer class
wxInputStream (p. 814)	Input stream class
wxOutputStream (p. 1016)	Output stream class
wxCountingOutputStream (p. 222)	Stream class for querying what size a stream would have.
wxFilterInputStream (p. 549)	Filtered input stream class
wxFilterOutputStream (p. 549)	Filtered output stream class

Threading classes

wxWidgets provides a set of classes to make use of the native thread capabilities of the various platforms.

wxMutexLocker (p. 997)	Mutex locker utility class
wxCriticalSection (p. 223)	Critical section class
wxCriticalSectionLocker (p. 224)	Critical section locker utility class
wxCondition (p. 189)	Condition class
wxSemaphore (p. 1163)	Semaphore class

HTML classes

wxWidgets provides a set of classes to display text in HTML format. These class include a help system based on the HTML widget.

wxHtmlHelpController (p. 722)	HTML help controller class
wxHtmlWindow (p. 749)	HTML window class
wxHtmlEasyPrinting (p. 718)	Simple class for printing HTML
wxHtmlPrintout (p. 741)	Generic HTML wxPrintout class
wxHtmlParser (p. 736)	Generic HTML parser class
wxHtmlTagHandler (p. 746)	HTML tag handler, pluginable into wxHtmlParser
wxHtmlWinParser (p. 758)	HTML parser class for wxHtmlWindow
wxHtmlWinTagHandler (p. 764)	HTML tag handler, pluginable into wxHtmlWinParser

Virtual file system classes

wxWidgets provides a set of classes that implement an extensible virtual file system, used internally by the HTML classes.

wxFSFile (p. 590)	Represents a file in the virtual file system
wxFileSystem (p. 540)	Main interface for the virtual file system
wxFileSystemHandler (p. 542)	Class used to announce file system type

XML-based resource system classes

XML-based resource system (XRC) overview (p. 1777)

Resources allow your application to create controls and other user interface elements from specifications stored in an XML format.

wxXmlResource (p. 1558)	The main class for working with resources.
wxXmlResourceHandler (p. 1563)	The base class for XML resource handlers.

Online help

wxHelpController (p. 696)	Family of classes for controlling help windows
wxHtmlHelpController (p. 722)	HTML help controller class
wxContextHelp (p. 212)	Class to put application into context-sensitive help mode
wxContextHelpButton (p. 213)	Button class for putting application into context-sensitive help mode
wxHelpProvider (p. 703)	Abstract class for context-sensitive help provision
wxSimpleHelpProvider (p. 1168)	Class for simple context-sensitive help provision
wxHelpControllerHelpProvider (p. 701)	Class for context-sensitive help provision via a help controller
wxToolTip (p. 1418)	Class implementing tooltips

Database classes

Database classes overview (p. 1820)

wxWidgets provides a set of classes for accessing Microsoft's ODBC (Open Database Connectivity) product, donated by Remstar. This is known as wxODBC.

wxDb (p. 284)	ODBC database connection
wxDbTable (p. 324)	Provides access to a database table
wxDbInf (p. 324)	
wxDbTableInf (p. 360)	
wxDbColDef (p. 315)	
wxDbColInf (p. 317)	
wxDbColDataPtr (p. 315)	
wxDbColFor (p. 316)	
wxDbConnectInf (p. 318)	
wxDbIdxDef (p. 323)	

Miscellaneous

wxApp (p. 31)	Application class
wxCaret (p. 131)	A caret (cursor) object
wxCmdLineParser (p. 155)	Command line parser class
wxConfig (p. 193)	Classes for configuration reading/writing (using either INI files or registry)

wxDllLoader (p. 431)   Class to work with shared libraries.

wxGLCanvas (p. 614)   Canvas that you can render OpenGL calls to.

wxGLContext (p. 617)   Class to ease sharing of OpenGL data resources.

wxLayoutAlgorithm (p. 835)   An alternative window layout facility

wxProcess (p. 1073)   Process class

wxTimer (p. 1386)   Timer class

wxStopWatch (p. 1274)   Stop watch class

wxMimeTypesManager (p. 974)   MIME-types manager class

wxSystemSettings (p. 1318)   System settings class for obtaining various global parameters

wxSystemOptions (p. 1315)   System options class for run-time configuration

wxAcceleratorTable (p. 19)   Accelerator table

wxAutomationObject (p. 75)   OLE automation class

wxFontMapper (p. 575)   Font mapping, finding suitable font for given encoding

wxEncodingConverter (p. 478)   Encoding conversions

wxCalendarDateAttr (p. 127)   Used with *wxCalendarCtrl* (p. 121)

wxQuantize (p. 1085)   Class to perform quantization, or colour reduction

wxSingleInstanceChecker (p. 1171) Check that only single program instance is running

Chapter 11 Topic overviews

This chapter contains a selection of topic overviews.

Changes since 2.4.x

Listed here are the deprecated and incompatible changes made to wxWidgets.

For other changes (such as additional features, bug fixes, etc.) see the changes.txt file located in the docs directory of your wxWidgets directory.

Incompatible changes since 2.4.x

wxEvent and its derivatives losing public variable members

wxEvent (p. 483) and its *derivatives* (p. 1751) do not have their public variable members public any more. Please use Get/Set accessors.

New window repainting behaviour

Windows are no longer fully repainted when resized; use the new style **wxFULL_REPAINT_ON_RESIZE** to force this (**wxNO_FULL_REPAINT_ON_RESIZE** still exists but doesn't do anything any more, this behaviour is default now).

Window class member changes

wxWindow::m_font and m_backgroundColour/m_foregroundColour are no longer always set, use *GetFont()* (p. 1508), *GetBack* (p. 1503)/*ForegroundColour()* (p. 1508) to access them, and they will be dynamically determined if necessary.

Sizers Internal Overhaul

The sizers (p. 1771) have had some fundamental internal changes in the 2.5.2 and 2.5.3 releases intended to make them do more of the "Right Thing" but also be as backwards compatible as possible. First a bit about how things used to work:

- The size that a window had when *Add()* (p. 1178)ed to the sizer was assumed to be its minimal size, and that size would always be used by default when calculating layout size and positions, and the sizer itself would keep track of that minimal size.

- If the window item was *Add()* (p. 1178)ed with the **wxADJUST_MINSIZE** flag then when layout was calculated the item's *GetBestSize* (p. 1504) would be used to reset the minimal size that the sizer used.

The main thrust of the new sizer changes was to make behaviour like **wxADJUST_MINSIZE** be the default, and also to push the tracking of the minimal size

to the window itself (since it knows its own needs) instead of having the sizer take care of it. Consequently these changes were made:

- The **wxFIXED_MINSIZE** flag was added to allow for the old behaviour. When this flag is used the size a window has when *Add()* (p. 1178)ed to the sizer will be treated as its minimal size and it will not be readjusted on each layout.

- The minimum size stored in wxWindow and settable with *SetSizeHints* (p. 1539) or *SetMinSize* (p. 1531) will by default be used by the sizer (if it was set) as the minimal size of the sizer item. If the minsize was not set (or was only partially set) then the window's best size is fetched and it is used instead of (or blended with) the minsize. *wxWindow::GetBestFittingSize* (p. 1504) was added to facilitate getting the size to be used by the sizers.

- The best size of a window is cached so it doesn't need to be recalculated on every layout. *wxWindow::InvalidateBestSize* (p. 1516) was added and should be called (usually just internally in control methods) whenever something is done that would make the best size change.

- All *wxControls* (p. 215) were changed to set the minsize to what is passed to the constructor or their Create method, and also to set the real size of the control to the blending of the minsize and bestsize. *wxWindow::SetBestFittingSize* (p. 1529) was added to help with this, although most controls don't need to call it directly because it is called indirectly via the *SetInitialBestSize* (p. 1531) called in the base classes.

At this time, the only situation known not to work the same as before is the following:

```
win = new SomeWidget(parent);
win->SetSize(SomeNonDefaultSize);
sizer->Add(win);
```

In this case the old code would have used the new size as the minimum, but now the sizer will use the default size as the minimum rather than the size set later. It is an easy fix though, just move the specification of the size to the constructor (assuming that SomeWidget will set its minsize there like the rest of the controls do) or call *SetMinSize* (p. 1531) instead of *SetSize* (p. 1538).

In order to fit well with this new scheme of things, all *wxControls* (p. 215) or custom controls should do the following things. (Depending on how they are used you may also want to do the same thing for non-control custom windows.)

- Either override or inherit a meaningful *DoGetBestSize* (p. 1499) method that calculates whatever size is "best" for the control. Once that size is calculated then there should normally be a call to *CacheBestSize* (p. 1494) to save it for later use, unless for some reason you want the best size to be recalculated on every layout.

- Any method that changes the attributes of the control such that the best size will change should call *InvalidateBestSize* (p. 1516) so it will be recalculated the next time it is needed.

- The control's constructor and/or Create method should ensure that the minsize

is set to the size passed in, and that the control is sized to a blending of the min size and best size. This can be done by calling *SetBestFittingSize* (p. 1529).

Massive wxURL Rewrite

wxURL (p. 1459) has undergone some radical changes.

- Many accessors of *wxURL* (p. 1459) - GetHostName, GetProtocolName, and GetPath, have been replaced by its parent's (*wxURI* (p. 1452)) counterparts - *GetServer* (p. 1456), *GetScheme* (p. 1456), and *GetPath* (p. 1455), respectively.

- ConvertToValidURI has been replaced by *wxURI* (p. 1452). Do not use ConvertToValidURI for future applications.

- ConvertFromURI has been replaced by *wxURI::Unescape* (p. 1458).

Minor incompatible changes since 2.4.x

- no initialization/cleanup can be done in wxApp's constructor or destructor because they are now called much earlier/later than before; please move any exiting code from there to *wxApp::OnInit()* (p. 38)/*OnExit()* (p. 37)

- also, *OnExit()* (p. 37) is not called if *OnInit()* (p. 38) fails

- finally the program exit code is *OnRun()* (p. 38) return value, not *OnExit()* (p. 37) one

- `wxTheApp` can't be assigned to any longer, use *wxApp::SetInstance()* (p. 40) instead

- *wxFileType::GetIcon()* (p. 547) returns *wxIconLocation* (p. 775), not *wxIcon* (p. 766)

- wxColourDatabase is not a *wxList* (p. 840) any more, use AddColour to add new colours

- wxWindow::Clear() is now called ClearBackground()

- pointer returned by wxFont::GetNativeFontInfo() must not be deleted now

- wxMouseEvent::Moving() doesn't return true if mouse is being dragged any more

- (most) controls now inherit parents colours by default, override ShouldInheritColours() to return false if you don't want this to happen

- *wxApp::SendIdleEvents()* (p. 39) now takes 2 arguments

- wxTabView::GetLayers() changed return type from *wxList&* (p. 840) to wxTabLayerList& (when WXWIN_COMPATIBILITY_2_4 == 0)

- wxID_SEPARATOR (id used for the menu separators) value changed from -1 to -2

- wxGetNumberFromUser() is now in separate wx/numdlg.h, not wx/textdlg.h

- wxChoice and wxCombobox now handle their size in the same way as in all the other ports under MSW, new code is actually correct but different from weird stuff they were doing before so the behaviour of your programs might change

- *wxTaskBarIcon* (p. 1323) objects must now be destroyed before the application can exit. Previously, the application terminated if there were no top level windows; now it terminates if there are no top level windows or taskbar icons left.

- *wxZlibInputStream* (p. 1579) is not by default compatible with the output of the 2.4.x version of *wxZlibOutputStream* (p. 1580). However, there is a compatibility mode, switched on by passing wxZLIB_24COMPATIBLE to the constructor.

- when WXWIN_COMPATIBILITY_2_4 == 0 *wxHashTable* (p. 693) uses a new implementation not using *wxList* (p. 840) keyed interface (the same used when wxUSE_STL == 1), the only incompatibility being that Next() returns a wxHashTable::Node* instead of a wxNode*.

- non-const wxDC methods GetBackground(), GetBrush(), GetFont() and GetPen() as well as wxWindow methods GetFont() and GetCursor() don't exist any more, please fix your code -- it never worked correctly anyhow if you modified the objects returned by these methods so you should simply switch to using const methods.

- *wxWindow::GetFont()* (p. 1508) now returns *wxFont* (p. 558) object instead of reference

- EVT_XXX macros are now type-safe; code that uses wrong type for event handler's argument will no longer compile.

- Identical functionality of wxFileDialog::ParseWildcard, wxGenericDirCtrl::ParseFilter, Motif and MSW parsing native dialogs is now accessible in ::wxParseCommonDialogsFilter

- wxNotebookSizer and wxBookCtrlSizer are now deprecated -- they are no longer needed, you can treat wxNotebook as any other control and put it directly into the sizer that was wxNotebookSizer's parent sizer in old code.

- *wxFile* (p. 502) methods now return either wxFileOffset or wxFileSize_t which may be a 64 bit integer type, even on 32 bit platforms, instead of off_t and so the return value of *wxFile::Length()* (p. 507), for example, shouldn't be assigned to off_t variable any more (the compiler might warn you about this)

- wxListItem::m_data is now of type wxUIntPtr, not long, for compatibility with 64 bit systems

- wxSizer::Add/Insert returns pointer to wxSizerItem just added so conditions writeen with if(Add(..)==true) will not work. Use if(Add(..)) instead.

- New *wxBrush::IsHatch()* (p. 107) checking for brush type replaces IS_HATCH macro.

- wxSystemSettings::GetSystem*() members deprecated and replaced with wxSystemSettings::Get*().

Deprecated changes since 2.4.x

- wxURL::GetInputStream() and similar functionality has been deprecated in favor of other ways of connecting, such as though sockets or wxFileSystem.

- wxDocManager::GetNoHistoryFiles() renamed to *GetHistoryFilesCount()* (p. 440)

- wxSizer::Remove(wxWindow *), use Detach() instead (it is more clear)

- wxSizer::Set/GetOption(): use Set/GetProportion() instead

- wxKeyEvent::KeyCode(): use GetKeyCode instead

- wxList::Number, First, Last, Nth: use GetCount, GetFirst/Last, Item instead

- wxNode::Next, Previous, Data: use GetNext, GetPrevious, GetData instead

- wxListBase::operator wxList&(): use typesafe lists instead

- wxTheFontMapper: use *wxFontMapper::Get()* (p. 577) instead

- wxStringHashTable: use wxHashMap instead

- wxHashTableLong: use wxHashMap instead

- wxArrayString::GetStringArray: use wxCArrayString or alternative wxWidgets methods taking wxArrayString

- wxArrayString::Remove(index, count): use RemoveAt instead

- wxTreeItemId conversion to long is deprecated and shouldn't be used

- wxTreeCtrl::GetFirst/NextChild() 2nd argument now has type wxTreeItemIdValue and not long, please change declarations of "cookie"s in your code accordingly -- otherwise your code won't work on 64 bit platforms

- (MSW only) wxWindow::GetUseCtl3D(), GetTransparentBackground() and SetTransparent() as well as wxNO_3D and wxUSER_COLOURS styles

- *wxList* (p. 840) keyed interface: use *wxHashMap* (p. 685) instead

- wxColourDatabase::FindColour(): use *Find()* (p. 170) instead (NOTE: it has a different return type)

- wxHashTable::Next: use wxHashTable::Node* or wxHashTable::compatibility_iterator to store the return value

- wxWave class; use wxSound instead

- The wxHIDE_READONLY flag for wxFileDialog was not implemented and has now been removed

- wxTaskBarIcon::OnXXX() virtual methods: use events instead

- obsolete and not used wxUSE_GENERIC_DIALOGS_IN_MSW has been removed

- wxDbTable::wxDbTable with wxChar* deprecated, same with wxString& instead.

Notes on using the reference

In the descriptions of the wxWidgets classes and their member functions, note that descriptions of inherited member functions are not duplicated in derived classes unless their behaviour is different. So in using a class such as wxScrolledWindow, be aware that wxWindow functions may be relevant.

Note also that arguments with default values may be omitted from a function call, for brevity. Size and position arguments may usually be given a value of -1 (the default), in which case wxWidgets will choose a suitable value.

Most strings are returned as wxString objects. However, for remaining char * return values, the strings are allocated and deallocated by wxWidgets. Therefore, return values should always be copied for long-term use, especially since the same buffer is often used by wxWidgets.

The member functions are given in alphabetical order except for constructors and destructors which appear first.

Writing a wxWidgets application: a rough guide

To set a wxWidgets application going, you will need to derive a *wxApp* (p. 31) class and override *wxApp::OnInit* (p. 38).

An application must have a top-level *wxFrame* (p. 579) or *wxDialog* (p. 406) window. Each frame may contain one or more instances of classes such as *wxPanel* (p. 1030), *wxSplitterWindow* (p. 1239) or other windows and controls.

A frame can have a *wxMenuBar* (p. 953), a *wxToolBar* (p. 1401), a status line, and a *wxIcon* (p. 766) for when the frame is iconized.

A *wxPanel* (p. 1030) is used to place controls (classes derived from *wxControl* (p. 215)) which are used for user interaction. Examples of controls are *wxButton* (p. 117), *wxCheckBox* (p. 134), *wxChoice* (p. 140), *wxListBox* (p. 847), *wxRadioBox* (p. 1089), *wxSlider* (p. 1192).

Instances of *wxDialog* (p. 406) can also be used for controls and they have the advantage of not requiring a separate frame.

Instead of creating a dialog box and populating it with items, it is possible to choose one

of the convenient common dialog classes, such as *wxMessageDialog* (p. 969) and *wxFileDialog* (p. 511).

You never draw directly onto a window - you use a *device context* (DC). *wxDC* (p. 365) is the base for *wxClientDC* (p. 147), *wxPaintDC* (p. 1024), *wxMemoryDC* (p. 935), *wxPostScriptDC* (p. 1046), *wxMemoryDC* (p. 935), *wxMetafileDC* (p. 973) and *wxPrinterDC* (p. 1065). If your drawing functions have **wxDC** as a parameter, you can pass any of these DCs to the function, and thus use the same code to draw to several different devices. You can draw using the member functions of **wxDC**, such as *wxDC::DrawLine* (p. 372) and *wxDC::DrawText* (p. 375). Control colour on a window (*wxColour* (p. 164)) with brushes (*wxBrush* (p. 103)) and pens (*wxPen* (p. 1036)).

To intercept events, you add a DECLARE_EVENT_TABLE macro to the window class declaration, and put a BEGIN_EVENT_TABLE ... END_EVENT_TABLE block in the implementation file. Between these macros, you add event macros which map the event (such as a mouse click) to a member function. These might override predefined event handlers such as for *wxKeyEvent* (p. 830) and *wxMouseEvent* (p. 982).

Most modern applications will have an on-line, hypertext help system; for this, you need wxHelp and the *wxHelpController* (p. 696) class to control wxHelp.

GUI applications aren't all graphical wizardry. List and hash table needs are catered for by *wxList* (p. 840) and *wxHashMap* (p. 685). You will undoubtedly need some platform-independent *file functions* (p. 1599), and you may find it handy to maintain and search a list of paths using *wxPathList* (p. 1035). There's a *miscellany* (p. 1630) of operating system and other functions.

See also *Classes by Category* (p. 1682) for a list of classes.

wxWidgets Hello World sample

As many people have requested a mini-sample to be published here so that some quick judgment concerning syntax and basic principles can be made, you can now look at wxWidgets' "Hello World":

You have to include wxWidgets' header files, of course. This can be done on a file by file basis (such as #include "wx/window.h") or using one global include (#include "wx/wx.h"). This is also useful on platforms which support precompiled headers such as all major compilers on the Windows platform.

```
//
// file name: hworld.cpp
//
//    purpose: wxWidgets "Hello world"
//

// For compilers that support precompilation, includes "wx/wx.h".
#include "wx/wxprec.h"

#ifdef __BORLANDC__
    #pragma hdrstop
#endif
```

```
#ifndef WX_PRECOMP
    #include "wx/wx.h"
#endif
```

Practically every app should define a new class derived from wxApp. By overriding wxApp's OnInit() the program can be initialized, e.g. by creating a new main window.

```
class MyApp: public wxApp
{
    virtual bool OnInit();
};
```

The main window is created by deriving a class from wxFrame and giving it a menu and a status bar in its constructor. Also, any class that wishes to respond to any "event" (such as mouse clicks or messages from the menu or a button) must declare an event table using the macro below. Finally, the way to react to such events must be done in "handlers". In our sample, we react to two menu items, one for "Quit" and one for displaying an "About" window. These handlers should not be virtual.

```
class MyFrame: public wxFrame
{
public:
    MyFrame(const wxString& title, const wxPoint& pos, const wxSize&
size);

    void OnQuit(wxCommandEvent& event);
    void OnAbout(wxCommandEvent& event);

private:
    DECLARE_EVENT_TABLE()
};
```

In order to be able to react to a menu command, it must be given a unique identifier such as a const or an enum.

```
enum
{
    ID_Quit = 1,
    ID_About,
};
```

We then proceed to actually implement an event table in which the events are routed to their respective handler functions in the class MyFrame. There are predefined macros for routing all common events, ranging from the selection of a list box entry to a resize event when a user resizes a window on the screen. If -1 is given as the ID, the given handler will be invoked for any event of the specified type, so that you could add just one entry in the event table for all menu commands or all button commands etc. The origin of the event can still be distinguished in the event handler as the (only) parameter in an event handler is a reference to a wxEvent object, which holds various information about the event (such as the ID of and a pointer to the class, which emitted the event).

```
BEGIN_EVENT_TABLE(MyFrame, wxFrame)
    EVT_MENU(ID_Quit,  MyFrame::OnQuit)
    EVT_MENU(ID_About, MyFrame::OnAbout)
END_EVENT_TABLE()
```

As in all programs there must be a "main" function. Under wxWidgets main is

implemented using this macro, which creates an application instance and starts the program.

```
IMPLEMENT_APP(MyApp)
```

As mentioned above, wxApp::OnInit() is called upon startup and should be used to initialize the program, maybe showing a "splash screen" and creating the main window (or several). The frame should get a title bar text ("Hello World") and a position and start-up size. One frame can also be declared to be the top window. Returning true indicates a successful initialization.

```
bool MyApp::OnInit()
{
    MyFrame *frame = new MyFrame( "Hello World", wxPoint(50,50),
wxSize(450,340) );
    frame->Show( true );
    SetTopWindow( frame );
    return true;
}
```

In the constructor of the main window (or later on) we create a menu with two menu items as well as a status bar to be shown at the bottom of the main window. Both have to be "announced" to the frame with respective calls.

```
MyFrame::MyFrame(const wxString& title, const wxPoint& pos, const wxSize&
size)
       : wxFrame((wxFrame *)NULL, -1, title, pos, size)
{
    wxMenu *menuFile = new wxMenu;

    menuFile->Append( ID_About, "&About..." );
    menuFile->AppendSeparator();
    menuFile->Append( ID_Quit, "E&xit" );

    wxMenuBar *menuBar = new wxMenuBar;
    menuBar->Append( menuFile, "&File" );

    SetMenuBar( menuBar );

    CreateStatusBar();
    SetStatusText( "Welcome to wxWidgets!" );
}
```

Here are the actual event handlers. MyFrame::OnQuit() closes the main window by calling Close(). The parameter true indicates that other windows have no veto power such as after asking "Do you really want to close?". If there is no other main window left, the application will quit.

```
void MyFrame::OnQuit(wxCommandEvent& WXUNUSED(event))
{
    Close( true );
}
```

MyFrame::OnAbout() will display a small window with some text in it. In this case a typical "About" window with information about the program.

```
void MyFrame::OnAbout(wxCommandEvent& WXUNUSED(event))
{
```

```
      wxMessageBox( "This is a wxWidgets' Hello world sample",
                    "About Hello World", wxOK | wxICON_INFORMATION );
}
```

wxWidgets samples

Probably the best way to learn wxWidgets is by reading the source of some 50+ samples provided with it. Many aspects of wxWidgets programming can be learnt from them, but sometimes it is not simple to just choose the right sample to look at. This overview aims at describing what each sample does/demonstrates to make it easier to find the relevant one if a simple grep through all sources didn't help. They also provide some notes about using the samples and what features of wxWidgets are they supposed to test.

There are currently more than 50 different samples as part of wxWidgets and this list is not complete. You should start your tour of wxWidgets with the *minimal sample* (p. 1707) which is the wxWidgets version of "Hello, world!". It shows the basic structure of wxWidgets program and is the most commented sample of all - looking at its source code is recommended.

The next most useful sample is probably the *controls* (p. 1708)one which shows many of wxWidgets standard controls, such as buttons, listboxes, checkboxes, comboboxes etc.

Other, more complicated controls, have their own samples. In this category you may find the following samples showing the corresponding controls:

wxCalendarCtrl (p. 1708)	Calendar a.k.a. date picker control
wxListCtrl (p. 1713)	List view control
wxTreeCtrl (p. 1716)	Tree view control
wxGrid (p. 1711)	Grid control

Finally, it might be helpful to do a search in the entire sample directory if you can't find the sample showing the control you are interested in by name. Most classes contained in wxWidgets occur in at least one of the samples.

Minimal sample

The minimal sample is what most people will know under the term Hello World, i.e. a minimal program that doesn't demonstrate anything apart from what is needed to write a program that will display a "hello" dialog. This is usually a good starting point for learning how to use wxWidgets.

Art provider sample

The `artprov` sample shows how you can customize the look of standard wxWidgets dialogs by replacing default bitmaps/icons with your own versions. It also shows how you can use wxArtProvider to get stock bitmaps for use in your application.

Calendar sample

This font shows the *calendar control* (p. 121) in action. It shows how to configure the control (see the different options in the calendar menu) and also how to process the notifications from it.

Checklist sample

This sample demonstrates use of the *wxCheckListBox* (p. 137)class intercepting check, select and double click events. It also tests use of various methods modifying the control, such as by deleting items from it or inserting new ones (these functions are actually implemented in the parent class *wxListBox* (p. 847) so the sample tests that class as well). The layout of the dialog is created using a *wxBoxSizer* (p. 102)demonstrating a simple dynamic layout.

Config sample

This sample demonstrates the *wxConfig* (p. 193) classes in a platform independent way, i.e. it uses text based files to store a given configuration under Unix and uses the Registry under Windows.

See *wxConfig overview* (p. 1748) for the descriptions of all features of this class.

Controls sample

The controls sample is the main test program for most simple controls used in wxWidgets. The sample tests their basic functionality, events, placement, modification in terms of colour and font as well as the possibility to change the controls programmatically, such as adding an item to a list box etc. Apart from that, the sample uses a *wxNotebook* (p. 999) and tests most features of this special control (using bitmap in the tabs, using*wxSizers* (p. 1176) and *constraints* (p. 837) within notebook pages, advancing pages programmatically and vetoing a page change by intercepting the *wxNotebookEvent* (p. 1008).

The various controls tested are listed here:

wxButton (p. 117)	Push button control, displaying text
wxBitmapButton (p. 92)	Push button control, displaying a bitmap
wxCheckBox (p. 134)	Checkbox control
wxChoice (p. 140)	Choice control (a combobox without the editable area)
wxComboBox (p. 172)	A choice with an editable area
wxGauge (p. 599)	A control to represent a varying quantity, such as time remaining

wxStaticBox (p. 1260)	A static, or group box for visually grouping related controls
wxListBox (p. 847)	A list of strings for single or multiple selection
wxSpinCtrl	A spin ctrl with a text field and a 'up-down' control
wxSpinButton (p. 1227)	A spin or 'up-down' control
wxStaticText (p. 1264)	One or more lines of non-editable text
wxStaticBitmap (p. 1258)	A control to display a bitmap
wxRadioBox (p. 1089)	A group of radio buttons
wxRadioButton (p. 1096)	A round button to be used with others in a mutually exclusive way
wxSlider (p. 1192)	A slider that can be dragged by the user

Database sample

The database sample is a small test program showing how to use the ODBC classes written by Remstar Intl. Obviously, this sample requires a database with ODBC support to be correctly installed on your system.

DebugRpt sample

This sample shows how to use *wxDebugReport* (p. 398) class to generate a debug report in case of a program crash or otherwise. On start up, it proposes to either crash itself (by dereferencing a NULL pointer) or generate debug report without doing it. Next it initializes the debug report with standard information adding a custom file to it (just a timestamp) and allows to view the information gathered using *wxDebugReportPreview* (p. 403).

For the report processing part of the sample to work you should make available a Web server accepting form uploads, otherwise *wxDebugReportUpload* (p. 404) will report an error.

Dialogs sample

This sample shows how to use the common dialogs available from wxWidgets. These dialogs are described in detail in the *Common dialogs overview* (p. 1798).

Dialup sample

This sample shows the *wxDialUpManager* (p. 416)class. In the status bar, it displays the information gathered through its interface: in particular, the current connection status

(online or offline) and whether the connection is permanent (in which case a string 'LAN' appears in the third status bar field - but note that you may be on a LAN not connected to the Internet, in which case you will not see this) or not.

Using the menu entries, you may also dial or hang up the line if you have a modem attached and (this only makes sense for Windows) list the available connections.

DnD sample

This sample shows both clipboard and drag and drop in action. It is quite non trivial and may be safely used as a basis for implementing the clipboard and drag and drop operations in a real-life program.

When you run the sample, its screen is split in several parts. On the top, there are two listboxes which show the standard derivations of *wxDropTarget* (p. 471):*wxTextDropTarget* (p. 1360) and*wxFileDropTarget* (p. 516).

The middle of the sample window is taken by the log window which shows what is going on (of course, this only works in debug builds) and may be helpful to see the sequence of steps of data transfer.

Finally, the last part is used for dragging text from it to either one of the listboxes (only one will accept it) or another application. The last functionality available from the main frame is to paste a bitmap from the clipboard (or, in the case of the Windows version, also a metafile) - it will be shown in a new frame.

So far, everything we mentioned was implemented with minimal amount of code using standard wxWidgets classes. The more advanced features are demonstrated if you create a shape frame from the main frame menu. A shape is a geometric object which has a position, size and color. It models some application-specific data in this sample. A shape object supports its own private *wxDataFormat* (p. 234) which means that you may cut and paste it or drag and drop (between one and the same or different shapes) from one sample instance to another (or the same). However, chances are that no other program supports this format and so shapes can also be rendered as bitmaps which allows them to be pasted/dropped in many other applications (and, under Windows, also as metafiles which are supported by most of Windows programs as well - try Write/Wordpad, for example).

Take a look at DnDShapeDataObject class to see how you may use*wxDataObject* (p. 239) to achieve this.

Dynamic sample

This sample is a very small sample that demonstrates use of the*wxEvtHandler::Connect* (p. 488) method. This method should be used whenever it is not known at compile time which control will receive which event or which controls are actually going to be in a dialog or frame. This is most typically the case for any scripting language that would work as a wrapper for wxWidgets or programs where forms or similar datagrams can be created by the users.

See also the *event sample* (p. 1711)

Event sample

The event sample demonstrates various features of the wxWidgets events. It shows using dynamic events and connecting/disconnecting the event handlers during run time and also using*PushEventHandler()* (p. 1522) and*PopEventHandler()* (p. 1520).

It replaces the old dynamic sample.

Except(ions) sample

This very simple sample shows how to use C++ exceptions in wxWidgets programs, i.e. where to catch the exception which may be thrown by the program code. It doesn't do anything very exciting by itself, you need to study its code to understand what goes on.

You need to build the library with `wxUSE_EXCEPTIONS` being set to 1and compile your code with C++ exceptions support to be able to build this sample.

Exec sample

The exec sample demonstrates the *wxExecute* (p. 1592) and*wxShell* (p. 1596) functions. Both of them are used to execute the external programs and the sample shows how to do this synchronously (waiting until the program terminates) or asynchronously (notification will come later).

It also shows how to capture the output of the child process in both synchronous and asynchronous cases and how to kill the processes with*wxProcess::Kill* (p. 1076) and test for their existence with*wxProcess::Exists* (p. 1076).

Font sample

The font sample demonstrates *wxFont* (p. 558),*wxFontEnumerator* (p. 573) and*wxFontMapper* (p. 575) classes. It allows you to see the fonts available (to wxWidgets) on the computer and shows all characters of the chosen font as well.

Grid sample

TODO.

HTML samples

Eight HTML samples (you can find them in directory `samples/html`) cover all features of the HTML sub-library.

Test demonstrates how to create *wxHtmlWindow* (p. 749)and also shows most supported HTML tags.

Widget shows how you can embed ordinary controls or windows within an HTML page. It also nicely explains how to write new tag handlers and extend the library to work with unsupported tags.

About may give you an idea how to write good-looking About boxes.

Zip demonstrates use of virtual file systems in wxHTML. The zip archives handler (ships with wxWidgets) allows you to access HTML pages stored in a compressed archive as if they were ordinary files.

Virtual is yet another virtual file systems demo. This one generates pages at run-time. You may find it useful if you need to display some reports in your application.

Printing explains use of *wxHtmlEasyPrinting* (p. 718)class which serves as as-simple-as-possible interface for printing HTML documents without much work. In fact, only few function calls are sufficient.

Help and **Helpview** are variations on displaying HTML help (compatible with MS HTML Help Workshop). *Help* shows how to embed*wxHtmlHelpController* (p. 722) in your application while *Helpview* is a simple tool that only pops up the help window and displays help books given at command line.

Image sample

The image sample demonstrates use of the *wxImage* (p. 779) class and shows how to download images in a variety of formats, currently PNG, GIF, TIFF, JPEG, BMP, PNM and PCX. The top of the sample shows two rectangles, one of which is drawn directly in the window, the other one is drawn into a*wxBitmap* (p. 79), converted to a wxImage, saved as a PNG image and then reloaded from the PNG file again so that conversions between wxImage and wxBitmap as well as loading and saving PNG files are tested.

At the bottom of the main frame there is a test for using a monochrome bitmap by drawing into a *wxMemoryDC* (p. 935). The bitmap is then drawn specifying the foreground and background colours with*wxDC::SetTextForeground* (p. 385) and*wxDC::SetTextBackground* (p. 385) (on the left). The bitmap is then converted to a wxImage and the foreground colour (black) is replaced with red using *wxImage::Replace* (p. 795).

Internat(ionalization) sample

The not very clearly named internat sample demonstrates the wxWidgets internationalization (i18n for short from now on) features. To be more precise, it only shows localization support, i.e. support for translating the program messages into another language while true i18n would also involve changing the other aspects of the programs behaviour.

More information about this sample can be found in the `readme.txt` file in its directory.

Please see also *i18n overview* (p. 1736).

Layout sample

The layout sample demonstrates the two different layout systems offered by wxWidgets. When starting the program, you will see a frame with some controls and some graphics. The controls will change their size whenever you resize the entire frame and the exact behaviour of the size changes is determined using the *wxLayoutConstraints* (p. 837)class. See also the *overview* (p. 1768) and the*wxIndividualLayoutConstraint* (p. 811)class for further information.

The menu in this sample offers two more tests, one showing how to use a *wxBoxSizer* (p. 102) in a simple dialog and the other one showing how to use sizers in connection with a *wxNotebook* (p. 999)class. See also *wxSizer* (p. 1176).

Listctrl sample

This sample shows the *wxListCtrl* (p. 852) control. Different modes supported by the control (list, icons, small icons, report) may be chosen from the menu.

The sample also provides some timings for adding/deleting/sorting a lot of (several thousands) items into the control.

Mediaplayer sample

This sample demonstrates how to use all the features of*wxMediaCtrl* (p. 927) and play various types of sound, video, and other files.

It replaces the old dynamic sample.

Notebook sample

This samples shows *wxBookCtrl* (p. 1797) family of controls. Although initially it was written to demonstrate *wxNotebook* (p. 999)only, it can now be also used to see *wxListbook* (p. 846) and*wxChoicebook* (p. 143) in action. Test each of the controls, their orientation, images and pages using commands through menu.

Render sample

This sample shows how to replace the default wxWidgets*renderer* (p. 1122) and also how to write a shared library (DLL) implementing a renderer and load and unload it during the run-time.

Rotate sample

This is a simple example which demonstrates how to rotate an image with the *wxImage::Rotate* (p. 796) method. The rotation can be done without interpolation (left mouse button) which will be faster, or with interpolation (right mouse button) which is slower but gives better results.

Scroll subwindow sample

This sample demonstrates use of the *wxScrolledWindow* (p. 1150)class including placing subwindows into it and drawing simple graphics. It uses the*SetTargetWindow* (p. 1159) method and thus the effect of scrolling does not show in the scrolled window itself, but in one of its subwindows.

Additionally, this samples demonstrates how to optimize drawing operations in wxWidgets, in particular using the *wxWindow::IsExposed* (p. 1516) method with the aim to prevent unnecessary drawing in the window and thus reducing or removing flicker on screen.

Sockets sample

The sockets sample demonstrates how to use the communication facilities provided by *wxSocket* (p. 1203). There are two different applications in this sample: a server, which is implemented using a*wxSocketServer* (p. 1223) object, and a client, which is implemented as a *wxSocketClient* (p. 1219).

The server binds to the local address, using TCP port number 3000, sets up an event handler to be notified of incoming connection requests (**wxSOCKET_CONNECTION** events), and sits there, waiting for clients (*listening*, in socket parlance). For each accepted connection, a new *wxSocketBase* (p. 1203) object is created. These socket objects are independent from the server that created them, so they set up their own event handler, and then request to be notified of **wxSOCKET_INPUT** (incoming data) or **wxSOCKET_LOST**(connection closed at the remote end) events. In the sample, the event handler is the same for all connections; to find out which socket the event is addressed to, the *GetSocket* (p. 1222) function is used.

Although it might take some time to get used to the event-oriented system upon which wxSocket is built, the benefits are many. See, for example, that the server application, while being single-threaded (and of course without using fork() or ugly select() loops) can handle an arbitrary number of connections.

The client starts up unconnected, so you can use the Connect... option to specify the address of the server you are going to connect to (the TCP port number is hard-coded as 3000). Once connected, a number of tests are possible. Currently, three tests are implemented. They show how to use the basic IO calls in *wxSocketBase* (p. 1203), such as *Read* (p. 1213), *Write* (p. 1217),*ReadMsg* (p. 1214) and *WriteMsg* (p. 1218), and how to set up the correct IO flags depending on what you are going to do. See the comments in the code for more information. Note that because both clients and connection objects in the server set up an event handler to catch **wxSOCKET_LOST** events, each one is immediately notified if the other end closes the connection.

There is also a URL test which shows how to use the *wxURL* (p. 1459) class to fetch

data from a given URL.

The sockets sample is work in progress. Some things to do:

- More tests for basic socket functionality.

- More tests for protocol classes (wxProtocol and its descendants).

- Tests for the recently added (and still in alpha stage) datagram sockets.

- New samples which actually do something useful (suggestions accepted).

Sound sample

The `sound` sample shows how to use *wxSound* (p. 1225) for simple audio output (e.g. notifications).

Statbar sample

This sample shows how to create and use wxStatusBar. Although most of the samples have a statusbar, they usually only create a default one and only do it once.

Here you can see how to recreate the statusbar (with possibly different number of fields) and how to use it to show icons/bitmaps and/or put arbitrary controls into it.

Text sample

This sample demonstrates four features: firstly the use and many variants of the *wxTextCtrl* (p. 1340) class (single line, multi line, read only, password, ignoring TAB, ignoring ENTER).

Secondly it shows how to intercept a *wxKeyEvent* (p. 830) in both the raw form using the `EVT_KEY_UP` and `EVT_KEY_DOWN` macros and the higher level from using the `EVT_CHAR` macro. All characters will be logged in a log window at the bottom of the main window. By pressing some of the function keys, you can test some actions in the text ctrl as well as get statistics on the text ctrls, which is useful for testing if these statistics actually are correct.

Thirdly, on platforms which support it, the sample will offer to copy text to the*wxClipboard* (p. 150) and to paste text from it. The GTK version will use the so called PRIMARY SELECTION, which is the pseudo clipboard under X and best known from pasting text to the XTerm program.

Last not least: some of the text controls have tooltips and the sample also shows how tooltips can be centrally disabled and their latency controlled.

Thread sample

This sample demonstrates use of threads in connection with GUI programs. There are two fundamentally different ways to use threads in GUI programs and either way has to take care of the fact that the GUI library itself usually is not multi-threading safe, i.e. that it might crash if two threads try to access the GUI class simultaneously. One way to prevent that is have a normal GUI program in the main thread and some worker threads which work in the background. In order to make communication between the main thread and the worker threads possible, wxWidgets offers the *wxPostEvent* (p. 1638)function and this sample makes use of this function.

The other way to use a so called Mutex (such as those offered in the *wxMutex* (p. 994)class) that prevent threads from accessing the GUI classes as long as any other thread accesses them. For this, wxWidgets has the *wxMutexGuiEnter* (p. 1598)and *wxMutexGuiLeave* (p. 1599) functions, both of which are used and tested in the sample as well.

See also *Multithreading overview* (p. 1816) and *wxThread* (p. 1377).

Toolbar sample

The toolbar sample shows the *wxToolBar* (p. 1401) class in action.

The following things are demonstrated:

- Creating the toolbar using *wxToolBar::AddTool* (p. 1405)and *wxToolBar::AddControl* (p. 1404): see MyApp::InitToolbar in the sample.

- Using `EVT_UPDATE_UI` handler for automatically enabling/disabling toolbar buttons without having to explicitly call EnableTool. This is done in MyFrame::OnUpdateCopyAndCut.

- Using *wxToolBar::DeleteTool* (p. 1406) and*wxToolBar::InsertTool* (p. 1412) to dynamically update the toolbar.

Some buttons in the main toolbar are check buttons, i.e. they stay checked when pressed. On the platforms which support it, the sample also adds a combobox to the toolbar showing how you can use arbitrary controls and not only buttons in it.

If you toggle another toolbar in the sample (using `Ctrl-A`) you will also see the radio toolbar buttons in action: the first three buttons form a radio group, i.e. checking any of them automatically unchecks the previously checked one.

Treectrl sample

This sample demonstrates using the *wxTreeCtrl* (p. 1424) class. Here you may see how to process various notification messages sent by this control and also when they occur (by looking at the messages in the text control in the bottom part of the frame).

Adding, inserting and deleting items and branches from the tree as well as sorting (in default alphabetical order as well as in custom one) is demonstrated here as well - try the corresponding menu entries.

Wizard sample

This sample shows the so-called wizard dialog (implemented using*wxWizard* (p. 1549) and related classes). It shows almost all features supported:

- Using bitmaps with the wizard and changing them depending on the page shown (notice that wxValidationPage in the sample has a different image from the other ones)

- Using *TransferDataFromWindow* (p. 1545)to verify that the data entered is correct before passing to the next page (done in wxValidationPage which forces the user to check a checkbox before continuing).

- Using more elaborated techniques to allow returning to the previous page, but not continuing to the next one or vice versa (in wxRadioboxPage)

- This (wxRadioboxPage) page also shows how the page may process the `Cancel` button itself instead of relying on the wizard parent to do it.

- Normally, the order of the pages in the wizard is known at compile-time, but sometimes it depends on the user choices: wxCheckboxPage shows how to dynamically decide which page to display next (see also*wxWizardPage* (p. 1555))

wxApp overview

Classes: *wxApp* (p. 31)

A wxWidgets application does not have a *main* procedure; the equivalent is the *OnInit* (p. 38) member defined for a class derived from wxApp. *OnInit* will usually create a top window as a bare minimum.

Unlike in earlier versions of wxWidgets, OnInit does not return a frame. Instead it returns a boolean value which indicates whether processing should continue (true) or not (false). You call *wxApp::SetTopWindow* (p. 41) to let wxWidgets know about the top window.

Note that the program's command line arguments, represented by *argc* and *argv*, are available from within wxApp member functions.

An application closes by destroying all windows. Because all frames must be destroyed for the application to exit, it is advisable to use parent frames wherever possible when creating new frames, so that deleting the top level frame will automatically delete child frames. The alternative is to explicitly delete child frames in the top-level frame's *wxCloseEvent* (p. 153) handler.

In emergencies the *wxExit* (p. 1594) function can be called to kill the application however normally the application shuts down automatically, *see below* (p. 1718).

An example of defining an application follows:

```
class DerivedApp : public wxApp
{
```

```
public:
  virtual bool OnInit();
};

IMPLEMENT_APP(DerivedApp)

bool DerivedApp::OnInit()
{
  wxFrame *the_frame = new wxFrame(NULL, ID_MYFRAME, argv[0]);
  ...
  the_frame->Show(true);
  SetTopWindow(the_frame);

  return true;
}
```

Note the use of IMPLEMENT_APP(appClass), which allows wxWidgets to dynamically create an instance of the application object at the appropriate point in wxWidgets initialization. Previous versions of wxWidgets used to rely on the creation of a global application object, but this is no longer recommended, because required global initialization may not have been performed at application object construction time.

You can also use DECLARE_APP(appClass) in a header file to declare the wxGetApp function which returns a reference to the application object. Otherwise you can only use the global `wxTheApp` pointer which is of type `wxApp *`.

Application shutdown

The application normally shuts down when the last of its top level windows is closed. This is normally the expected behaviour and means that it is enough to call *Close()* (p. 1496) in response to the "`Exit`" menu command if your program has a single top level window. If this behaviour is not desirable *wxApp::SetExitOnFrameDelete* (p. 40) can be called to change it. Note that starting from wxWidgets 2.3.3 such logic doesn't apply for the windows shown before the program enters the main loop: in other words, you can safely show a dialog from *wxApp::OnInit* (p. 38) and not be afraid that your application terminates when this dialog -- which is the last top level window for the moment -- is closed.

Another aspect of the application shutdown is *OnExit* (p. 37) which is called when the application exits but *before* wxWidgets cleans up its internal structures. You should delete all wxWidgets object that you created by the time OnExit finishes. In particular, do **not** destroy them from application class' destructor!

For example, this code may crash:

```
class MyApp : public wxApp
{
 public:
    wxCHMHelpController m_helpCtrl;
    ...
};
```

The reason for that is that `m_helpCtrl` is a member object and is thus destroyed from MyApp destructor. But MyApp object is deleted after wxWidgets structures that

wxCHMHelpController depends on were uninitialized! The solution is to destroy HelpCtrl in *OnExit*:

```
class MyApp : public wxApp
{
 public:
    wxCHMHelpController *m_helpCtrl;
    ...
};

bool MyApp::OnInit()
{
  ...
  m_helpCtrl = new wxCHMHelpController;
  ...
}

int MyApp::OnExit()
{
  delete m_helpCtrl;
  return 0;
}
```

Runtime class information (aka RTTI) overview

Classes: *wxObject* (p. 1012), *wxClassInfo* (p. 144).

One of the failings of C++ used to be that no run-time information was provided about a class and its position in the inheritance hierarchy. Another, which still persists, is that instances of a class cannot be created just by knowing the name of a class, which makes facilities such as persistent storage hard to implement.

Most C++ GUI frameworks overcome these limitations by means of a set of macros and functions and wxWidgets is no exception. As it originated before the addition of RTTI to the C++ standard and as support for it is still missing from some (albeit old) compilers, wxWidgets doesn't (yet) use it, but provides its own macro-based RTTI system.

In the future, the standard C++ RTTI will be used though and you're encouraged to use whenever possible the *wxDynamicCast()* (p. 1646) macro which, for the implementations that support it, is defined just as dynamic_cast<> and uses wxWidgets RTTI for all the others. This macro is limited to wxWidgets classes only and only works with pointers (unlike the real dynamic_cast<> which also accepts references).

Each class that you wish to be known to the type system should have a macro such as DECLARE_DYNAMIC_CLASS just inside the class declaration. The macro IMPLEMENT_DYNAMIC_CLASS should be in the implementation file. Note that these are entirely optional; use them if you wish to check object types, or create instances of classes using the class name. However, it is good to get into the habit of adding these macros for all classes.

Variations on these *macros* (p. 1641) are used for multiple inheritance, and abstract classes that cannot be instantiated dynamically or otherwise.

DECLARE_DYNAMIC_CLASS inserts a static wxClassInfo declaration into the class,

initialized by IMPLEMENT_DYNAMIC_CLASS. When initialized, the wxClassInfo object inserts itself into a linked list (accessed through wxClassInfo::first and wxClassInfo::next pointers). The linked list is fully created by the time all global initialisation is done.

IMPLEMENT_DYNAMIC_CLASS is a macro that not only initialises the static wxClassInfo member, but defines a global function capable of creating a dynamic object of the class in question. A pointer to this function is stored in wxClassInfo, and is used when an object should be created dynamically.

wxObject::IsKindOf (p. 1014) uses the linked list of wxClassInfo. It takes a wxClassInfo argument, so use CLASSINFO(className) to return an appropriate wxClassInfo pointer to use in this function.

The function *wxCreateDynamicObject* (p. 1645) can be used to construct a new object of a given type, by supplying a string name. If you have a pointer to the wxClassInfo object instead, then you can simply call *wxClassInfo::CreateObject* (p. 144).

wxClassInfo

Runtime class information (aka RTTI) overview (p. 1719)

Class: *wxClassInfo* (p. 144)

This class stores meta-information about classes. An application may use macros such as DECLARE_DYNAMIC_CLASS and IMPLEMENT_DYNAMIC_CLASS to record run-time information about a class, including:

- its position in the inheritance hierarchy;

- the base class name(s) (up to two base classes are permitted);

- a string representation of the class name;

- a function that can be called to construct an instance of this class.

The DECLARE_... macros declare a static wxClassInfo variable in a class, which is initialized by macros of the form IMPLEMENT_... in the implementation C++ file. Classes whose instances may be constructed dynamically are given a global constructor function which returns a new object.

You can get the wxClassInfo for a class by using the CLASSINFO macro, e.g. CLASSINFO(wxFrame). You can get the wxClassInfo for an object using wxObject::GetClassInfo.

See also *wxObject* (p. 1012) and *wxCreateDynamicObject* (p. 1645).

Example

In a header file frame.h:

```
class wxFrame : public wxWindow
```

```
    {
    DECLARE_DYNAMIC_CLASS(wxFrame)

    private:
        wxString m_title;

    public:
        ...
    };
```

In a C++ file frame.cpp:

```
    IMPLEMENT_DYNAMIC_CLASS(wxFrame, wxWindow)

    wxFrame::wxFrame()
    {
    ...
    }
```

wxString overview

Classes: *wxString* (p. 1284), *wxArrayString* (p. 65), *wxStringTokenizer* (p. 1312)

Introduction

wxString is a class which represents a character string of arbitrary length (limited by *MAX_INT* which is usually 2147483647 on 32 bit machines) and containing arbitrary characters. The ASCII NUL character is allowed, but be aware that in the current string implementation some methods might not work correctly in this case.

wxString works with both ASCII (traditional, 7 or 8 bit, characters) as well as Unicode (wide characters) strings.

This class has all the standard operations you can expect to find in a string class: dynamic memory management (string extends to accommodate new characters), construction from other strings, C strings and characters, assignment operators, access to individual characters, string concatenation and comparison, substring extraction, case conversion, trimming and padding (with spaces), searching and replacing and both C-like *Printf()* (p. 1301) and stream-like insertion functions as well as much more - see *wxString* (p. 1284) for a list of all functions.

Comparison of wxString to other string classes

The advantages of using a special string class instead of working directly with C strings are so obvious that there is a huge number of such classes available. The most important advantage is the need to always remember to allocate/free memory for C strings; working with fixed size buffers almost inevitably leads to buffer overflows. At last, C++ has a standard string class (std::string). So why the need for wxString?

There are several advantages:

1. **Efficiency** This class was made to be as efficient as possible: both in terms of size (each wxString objects takes exactly the same space as a *char ** pointer, sing *reference counting* (p. 1724)) and speed. It also provides performance *statistics gathering code* (p. 1724) which may be enabled to fine tune the memory allocation strategy for your particular application - and the gain might be quite big.

2. **Compatibility** This class tries to combine almost full compatibility with the old wxWidgets 1.xx wxString class, some reminiscence to MFC CString class and 90% of the functionality of std::string class.

3. **Rich set of functions** Some of the functions present in wxString are very useful but don't exist in most of other string classes: for example, *AfterFirst* (p. 1293), *BeforeLast* (p. 1294), *operator<<* (p. 1306) or *Printf* (p. 1301). Of course, all the standard string operations are supported as well.

4. **Unicode** wxString is Unicode friendly: it allows to easily convert to and from ANSI and Unicode strings in any build mode (see the *Unicode overview* (p. 1730) for more details) and maps to either `string` or `wstring` transparently depending on the current mode.

5. **Used by wxWidgets** And, of course, this class is used everywhere inside wxWidgets so there is no performance loss which would result from conversions of objects of any other string class (including std::string) to wxString internally by wxWidgets.

However, there are several problems as well. The most important one is probably that there are often several functions to do exactly the same thing: for example, to get the length of the string either one of length(), *Len()* (p. 1299) or *Length()* (p. 1299) may be used. The first function, as almost all the other functions in lowercase, is std::string compatible. The second one is "native" wxString version and the last one is wxWidgets 1.xx way. So the question is: which one is better to use? And the answer is that:

The usage of std::string compatible functions is strongly advised! It will both make your code more familiar to other C++ programmers (who are supposed to have knowledge of std::string but not of wxString), let you reuse the same code in both wxWidgets and other programs (by just typedefing wxString as std::string when used outside wxWidgets) and by staying compatible with future versions of wxWidgets which will probably start using std::string sooner or later too.

In the situations where there is no corresponding std::string function, please try to use the new wxString methods and not the old wxWidgets 1.xx variants which are deprecated and may disappear in future versions.

Some advice about using wxString

Probably the main trap with using this class is the implicit conversion operator to *const char **. It is advised that you use *c_str()* (p. 1294)instead to clearly indicate when the conversion is done. Specifically, the danger of this implicit conversion may be seen in the following code fragment:

```
// this function converts the input string to uppercase, output it to the
screen
// and returns the result
const char *SayHELLO(const wxString& input)
{
    wxString output = input.Upper();

    printf("Hello, %s!\n", output);

    return output;
}
```

There are two nasty bugs in these three lines. First of them is in the call to the *printf()* function. Although the implicit conversion to C strings is applied automatically by the compiler in the case of

```
        puts(output);
```

because the argument of *puts()* is known to be of the type *const char **, this is **not** done for *printf()* which is a function with variable number of arguments (and whose arguments are of unknown types). So this call may do anything at all (including displaying the correct string on screen), although the most likely result is a program crash. The solution is to use *c_str()* (p. 1294): just replace this line with

```
        printf("Hello, %s!\n", output.c_str());
```

The second bug is that returning *output* doesn't work. The implicit cast is used again, so the code compiles, but as it returns a pointer to a buffer belonging to a local variable which is deleted as soon as the function exits, its contents is totally arbitrary. The solution to this problem is also easy: just make the function return wxString instead of a C string.

This leads us to the following general advice: all functions taking string arguments should take *const wxString&* (this makes assignment to the strings inside the function faster because of *reference counting* (p. 1724)) and all functions returning strings should return *wxString* - this makes it safe to return local variables.

Other string related functions and classes

As most programs use character strings, the standard C library provides quite a few functions to work with them. Unfortunately, some of them have rather counter-intuitive behaviour (like strncpy() which doesn't always terminate the resulting string with a NULL) and are in general not very safe (passing NULL to them will probably lead to program crash). Moreover, some very useful functions are not standard at all. This is why in addition to all wxString functions, there are also a few global string functions which try to correct these problems: *wxIsEmpty()* (p. 1610) verifies whether the string is empty (returning `true` for `NULL` pointers), *wxStrlen()* (p. 1611) also handles NULLs correctly and returns 0 for them and *wxStricmp()* (p. 1611) is just a platform-independent version of case-insensitive string comparison function known either as stricmp() or strcasecmp() on different platforms.

The `<wx/string.h>` header also defines *wxSnprintf* (p. 1611) and *wxVsnprintf* (p. 1613) functions which should be used instead of the inherently dangerous standard

`sprintf()` and which use `snprintf()` instead which does buffer size checks whenever possible. Of course, you may also use *wxString::Printf* (p. 1301) which is also safe.

There is another class which might be useful when working with wxString: *wxStringTokenizer* (p. 1312). It is helpful when a string must be broken into tokens and replaces the standard C library *strtok()* function.

And the very last string-related class is *wxArrayString* (p. 65): it is just a version of the "template" dynamic array class which is specialized to work with strings. Please note that this class is specially optimized (using its knowledge of the internal structure of wxString) for storing strings and so it is vastly better from a performance point of view than a wxObjectArray of wxStrings.

Reference counting and why you shouldn't care about it

wxString objects use a technique known as *copy on write* (COW). This means that when a string is assigned to another, no copying really takes place: only the reference count on the shared string data is incremented and both strings share the same data.

But as soon as one of the two (or more) strings is modified, the data has to be copied because the changes to one of the strings shouldn't be seen in the others. As data copying only happens when the string is written to, this is known as COW.

What is important to understand is that all this happens absolutely transparently to the class users and that whether a string is shared or not is not seen from the outside of the class - in any case, the result of any operation on it is the same.

Probably the unique case when you might want to think about reference counting is when a string character is taken from a string which is not a constant (or a constant reference). In this case, due to C++ rules, the "read-only" *operator[]* (which is the same as *GetChar()* (p. 1297)) cannot be chosen and the "read/write" *operator[]* (the same as *GetWritableChar()* (p. 1297)) is used instead. As the call to this operator may modify the string, its data is unshared (COW is done) and so if the string was really shared there is some performance loss (both in terms of speed and memory consumption). In the rare cases when this may be important, you might prefer using *GetChar()* (p. 1297) instead of the array subscript operator for this reasons. Please note that *at()* (p. 1289) method has the same problem as the subscript operator in this situation and so using it is not really better. Also note that if all string arguments to your functions are passed as *const wxString&* (see the section *Some advice* (p. 1722)) this situation will almost never arise because for constant references the correct operator is called automatically.

Tuning wxString for your application

> **Note:** this section is strictly about performance issues and is absolutely not necessary to read for using wxString class. Please skip it unless you feel familiar with profilers and relative tools. If you do read it, please also read the preceding section about *reference counting* (p. 1724).

For the performance reasons wxString doesn't allocate exactly the amount of memory

needed for each string. Instead, it adds a small amount of space to each allocated block which allows it to not reallocate memory (a relatively expensive operation) too often as when, for example, a string is constructed by subsequently adding one character at a time to it, as for example in:

```
// delete all vowels from the string
wxString DeleteAllVowels(const wxString& original)
{
    wxString result;

    size_t len = original.length();
    for ( size_t n = 0; n < len; n++ )
    {
        if ( strchr("aeuio", tolower(original[n])) == NULL )
            result += original[n];
    }

    return result;
}
```

This is quite a common situation and not allocating extra memory at all would lead to very bad performance in this case because there would be as many memory (re)allocations as there are consonants in the original string. Allocating too much extra memory would help to improve the speed in this situation, but due to a great number of wxString objects typically used in a program would also increase the memory consumption too much.

The very best solution in precisely this case would be to use *Alloc()* (p. 1292) function to preallocate, for example, len bytes from the beginning - this will lead to exactly one memory allocation being performed (because the result is at most as long as the original string).

However, using Alloc() is tedious and so wxString tries to do its best. The default algorithm assumes that memory allocation is done in granularity of at least 16 bytes (which is the case on almost all of wide-spread platforms) and so nothing is lost if the amount of memory to allocate is rounded up to the next multiple of 16. Like this, no memory is lost and 15 iterations from 16 in the example above won't allocate memory but use the already allocated pool.

The default approach is quite conservative. Allocating more memory may bring important performance benefits for programs using (relatively) few very long strings. The amount of memory allocated is configured by the setting of *EXTRA_ALLOC* in the file string.cpp during compilation (be sure to understand why its default value is what it is before modifying it!). You may try setting it to greater amount (say twice nLen) or to 0 (to see performance degradation which will follow) and analyse the impact of it on your program. If you do it, you will probably find it helpful to also define WXSTRING_STATISTICS symbol which tells the wxString class to collect performance statistics and to show them on stderr on program termination. This will show you the average length of strings your program manipulates, their average initial length and also the percent of times when memory wasn't reallocated when string concatenation was done but the already preallocated memory was used (this value should be about 98% for the default allocation policy, if it is less than 90% you should really consider fine tuning wxString for your application).

It goes without saying that a profiler should be used to measure the precise difference the change to EXTRA_ALLOC makes to your program.

Buffer classes overview

wxWidgets uses two classes of classes for dealing with buffers in memory.

The first is one for dealing with character buffers, namely wxCharBuffer for char pointer or multi-byte c strings and wxWCharBuffer for wchar_t pointer or wide character c strings.

Secondly, wxWidgets uses, although only rarely currently, wxMemoryBuffer for dealing with raw buffers in memory.

wxXCharBuffer Overview

General Usage

As mentioned, wxCharBuffer and its wide character variant wxWCharBuffer deal with c strings in memory. They have two constructors, one in which you pass the c string you want them to have a copy of, and another where you specify the size of the buffer in memory in characters you want.

wxCharBuffer and its variant only contain the c string as a member, so they can be used safely to c functions with variable arguments such as printf. They also contain standard assignment, character access operators and a copy constructor.

Destruction

It should be noted that on destruction wxCharBuffer and its wide character variant delete the c string that hold onto. If you want to get the pointer to the buffer and don't want wxCharBuffer to delete it on destruction, use the member function release to do so.

Date and time classes overview

Classes: *wxDateTime* (p. 257), *wxDateSpan* (p. 252), *wxTimeSpan* (p. 1389), *wxCalendarCtrl* (p. 121)

Introduction

wxWidgets provides a set of powerful classes to work with dates and times. Some of the supported features of *wxDateTime* (p. 257) class are:

Wide range	The range of supported dates goes from about 4714 B.C. to some 480 million years in the future.

Precision	Not using floating point calculations anywhere ensures that the date calculations don't suffer from rounding errors.
Many features	Not only all usual calculations with dates are supported, but also more exotic week and year day calculations, work day testing, standard astronomical functions, conversion to and from strings in either strict or free format.
Efficiency	Objects of wxDateTime are small (8 bytes) and working with them is fast

All date/time classes at a glance

There are 3 main classes declared in `<wx/datetime.h>`: except *wxDateTime* (p. 257) itself which represents an absolute moment in time, there are also two classes - *wxTimeSpan* (p. 1389) and *wxDateSpan* (p. 252) - which represent the intervals of time.

There are also helper classes which are used together with wxDateTime: *wxDateTimeHolidayAuthority* (p. 284) which is used to determine whether a given date is a holiday or not and *wxDateTimeWorkDays* (p. 284) which is a derivation of this class for which (only) Saturdays and Sundays are the holidays. See more about these classes in the discussion of the *holidays* (p. 1730).

Finally, in other parts of this manual you may find mentions of wxDate and wxTime classes. *These classes* (p. 1730) are obsolete and superseded by wxDateTime.

wxDateTime characteristics

wxDateTime (p. 257) stores the time as a signed number of milliseconds since the Epoch which is fixed, by convention, to Jan 1, 1970 - however this is not visible to the class users (in particular, dates prior to the Epoch are handled just as well (or as bad) as the dates after it). But it does mean that the best resolution which can be achieved with this class is 1 millisecond.

The size of wxDateTime object is 8 bytes because it is represented as a 64 bit integer. The resulting range of supported dates is thus approximatively 580 million years, but due to the current limitations in the Gregorian calendar support, only dates from Nov 24, 4714BC are supported (this is subject to change if there is sufficient interest in doing it).

Finally, the internal representation is time zone independent (always in GMT) and the time zones only come into play when a date is broken into year/month/day components. See more about *timezones* (p. 1729) below.

Currently, the only supported calendar is Gregorian one (which is used even for the dates prior to the historic introduction of this calendar which was first done on Oct 15, 1582 but is, generally speaking, country, and even region, dependent). Future versions will probably have Julian calendar support as well and support for other calendars (Maya, Hebrew, Chinese...) is not ruled out.

Difference between wxDateSpan and wxTimeSpan

While there is only one logical way to represent an absolute moment in the time (and hence only one wxDateTime class), there are at least two methods to describe a time interval.

First, there is the direct and self-explaining way implemented by *wxTimeSpan* (p. 1389): it is just a difference in milliseconds between two moments in time. Adding or subtracting such an interval to wxDateTime is always well-defined and is a fast operation.

But in the daily life other, calendar-dependent time interval specifications are used. For example, 'one month later' is commonly used. However, it is clear that this is not the same as wxTimeSpan of 60*60*24*31 seconds because 'one month later' Feb 15 is Mar 15 and not Mar 17 or Mar 16 (depending on whether the year is leap or not).

This is why there is another class for representing such intervals called *wxDateSpan* (p. 252). It handles these sort of operations in the most natural way possible, but note that manipulating with intervals of this kind is not always well-defined. Consider, for example, Jan 31 + '1 month': this will give Feb 28 (or 29), i.e. the last day of February and not the non-existent Feb 31. Of course, this is what is usually wanted, but you still might be surprised to notice that now subtracting back the same interval from Feb 28 will result in Jan 28 and **not** Jan 31 we started with!

So, unless you plan to implement some kind of natural language parsing in the program, you should probably use wxTimeSpan instead of wxDateSpan (which is also more efficient). However, wxDateSpan may be very useful in situations when you do need to understand what 'in a month' means (of course, it is just `wxDateTime::Now() + wxDateSpan::Month()`).

Date arithmetics

Many different operations may be performed with the dates, however not all of them make sense. For example, multiplying a date by a number is an invalid operation, even though multiplying either of the time span classes by a number is perfectly valid.

Here is what can be done:

Addition	a wxTimeSpan or wxDateSpan can be added to wxDateTime resulting in a new wxDateTime object and also 2 objects of the same span class can be added together giving another object of the same class.
Subtraction	the same types of operations as above are allowed and, additionally, a difference between two wxDateTime objects can be taken and this will yield wxTimeSpan.
Multiplication	a wxTimeSpan or wxDateSpan object can be multiplied by an integer number resulting in an object of the same type.

Unary minus   a wxTimeSpan or wxDateSpan object may finally be negated giving an interval of the same magnitude but of opposite time direction.

For all these operations there are corresponding global (overloaded) operators and also member functions which are synonyms for them: Add(), Subtract() and Multiply(). Unary minus as well as composite assignment operations (like +=) are only implemented as members and Neg() is the synonym for unary minus.

Time zone considerations

Although the time is always stored internally in GMT, you will usually work in the local time zone. Because of this, all wxDateTime constructors and setters which take the broken down date assume that these values are for the local time zone. Thus, `wxDateTime(1, wxDateTime::Jan, 1970)` will not correspond to the wxDateTime Epoch unless you happen to live in the UK.

All methods returning the date components (year, month, day, hour, minute, second...) will also return the correct values for the local time zone by default, so, generally, doing the natural things will lead to natural and correct results.

If you only want to do this, you may safely skip the rest of this section. However, if you want to work with different time zones, you should read it to the end.

In this (rare) case, you are still limited to the local time zone when constructing wxDateTime objects, i.e. there is no way to construct a wxDateTime corresponding to the given date in, say, Pacific Standard Time. To do it, you will need to call *ToTimezone* (p. 283) or *MakeTimezone* (p. 284) methods to adjust the date for the target time zone. There are also special versions of these functions *ToGMT* (p. 284) and *MakeGMT* (p. 284) for the most common case - when the date should be constructed in GMT.

You also can just retrieve the value for some time zone without converting the object to it first. For this you may pass TimeZone argument to any of the methods which are affected by the time zone (all methods getting date components and the date formatting ones, for example). In particular, the Format() family of methods accepts a TimeZone parameter and this allows to simply print time in any time zone.

To see how to do it, the last issue to address is how to construct a TimeZone object which must be passed to all these methods. First of all, you may construct it manually by specifying the time zone offset in seconds from GMT, but usually you will just use one of the *symbolic time zone names* (p. 257) and let the conversion constructor do the job. I.e. you would just write

```
wxDateTime dt(...whatever...);
printf("The time is %s in local time zone", dt.FormatTime().c_str());
printf("The time is %s in GMT", dt.FormatTime(wxDateTime::GMT).c_str());
```

Daylight saving time (DST)

DST (a.k.a. 'summer time') handling is always a delicate task which is better left to the

operating system which is supposed to be configured by the administrator to behave correctly. Unfortunately, when doing calculations with date outside of the range supported by the standard library, we are forced to deal with these issues ourselves.

Several functions are provided to calculate the beginning and end of DST in the given year and to determine whether it is in effect at the given moment or not, but they should not be considered as absolutely correct because, first of all, they only work more or less correctly for only a handful of countries (any information about other ones appreciated!) and even for them the rules may perfectly well change in the future.

The time zone handling *methods* (p. 1729) use these functions too, so they are subject to the same limitations.

wxDateTime and Holidays

TODO.

Compatibility

The old classes for date/time manipulations ported from wxWidgets version 1.xx are still included but are reimplemented in terms of wxDateTime. However, using them is strongly discouraged because they have a few quirks/bugs and were not 'Y2K' compatible.

Unicode support in wxWidgets

This section briefly describes the state of the Unicode support in wxWidgets. Read it if you want to know more about how to write programs able to work with characters from languages other than English.

What is Unicode?

Starting with release 2.1 wxWidgets has support for compiling in Unicode mode on the platforms which support it. Unicode is a standard for character encoding which addresses the shortcomings of the previous, 8 bit standards, by using at least 16 (and possibly 32) bits for encoding each character. This allows to have at least 65536 characters (what is called the BMP, or basic multilingual plane) and possible 2^32 of them instead of the usual 256 and is sufficient to encode all of the world languages at once. More details about Unicode may be found at `www.unicode.org`.

As this solution is obviously preferable to the previous ones (think of incompatible encodings for the same language, locale chaos and so on), many modern operating systems support it. The probably first example is Windows NT which uses only Unicode internally since its very first version.

Writing internationalized programs is much easier with Unicode and, as the support for it improves, it should become more and more so. Moreover, in the Windows NT/2000

case, even the program which uses only standard ASCII can profit from using Unicode because they will work more efficiently - there will be no need for the system to convert all strings the program uses to/from Unicode each time a system call is made.

Unicode and ANSI modes

As not all platforms supported by wxWidgets support Unicode (fully) yet, in many cases it is unwise to write a program which can only work in Unicode environment. A better solution is to write programs in such way that they may be compiled either in ANSI (traditional) mode or in the Unicode one.

This can be achieved quite simply by using the means provided by wxWidgets. Basically, there are only a few things to watch out for:

- Character type (`char` or `wchar_t`)

- Literal strings (i.e. `"Hello, world!"` or `'*'`)

- String functions (`strlen()`, `strcpy()`, ...)

- Special preprocessor tokens (`__FILE__`, `__DATE__` and `__TIME__`)

Let's look at them in order. First of all, each character in an Unicode program takes 2 bytes instead of usual one, so another type should be used to store the characters (`char` only holds 1 byte usually). This type is called `wchar_t` which stands for *wide-character type*.

Also, the string and character constants should be encoded using wide characters (`wchar_t` type) which typically take 2 or 4 bytes instead of `char` which only takes one. This is achieved by using the standard C (and C++) way: just put the letter `'L'` after any string constant and it becomes a *long* constant, i.e. a wide character one. To make things a bit more readable, you are also allowed to prefix the constant with `'L'` instead of putting it after it.

Of course, the usual standard C functions don't work with `wchar_t` strings, so another set of functions exists which do the same thing but accept `wchar_t *` instead of `char *`. For example, a function to get the length of a wide-character string is called `wcslen()` (compare with `strlen()` - you see that the only difference is that the "str" prefix standing for "string" has been replaced with "wcs" standing for "wide-character string").

And finally, the standard preprocessor tokens enumerated above expand to ANSI strings but it is more likely that Unicode strings are wanted in the Unicode build. wxWidgets provides the macros `__TFILE__`, `__TDATE__` and `__TTIME__` which behave exactly as the standard ones except that they produce ANSI strings in ANSI build and Unicode ones in the Unicode build.

To summarize, here is a brief example of how a program which can be compiled in both ANSI and Unicode modes could look like:

```
#ifdef __UNICODE__
```

```
    wchar_t wch = L'*';
    const wchar_t *ws = L"Hello, world!";
    int len = wcslen(ws);

    wprintf(L"Compiled at %s\n", __TDATE__);
#else // ANSI
    char ch = '*';
    const char *s = "Hello, world!";
    int len = strlen(s);

    printf("Compiled at %s\n", __DATE__);
#endif // Unicode/ANSI
```

Of course, it would be nearly impossibly to write such programs if it had to be done this way (try to imagine the number of `#ifdef UNICODE` an average program would have had!). Luckily, there is another way - see the next section.

Unicode support in wxWidgets

In wxWidgets, the code fragment from above should be written instead:

```
    wxChar ch = wxT('*');
    wxString s = wxT("Hello, world!");
    int len = s.Len();
```

What happens here? First of all, you see that there are no more `#ifdef`s at all. Instead, we define some types and macros which behave differently in the Unicode and ANSI builds and allow us to avoid using conditional compilation in the program itself.

We have a `wxChar` type which maps either on `char` or `wchar_t` depending on the mode in which program is being compiled. There is no need for a separate type for strings though, because the standard *wxString* (p. 1284) supports Unicode, i.e. it stores either ANSI or Unicode strings depending on the compile mode.

Finally, there is a special *wxT()* (p. 1612) macro which should enclose all literal strings in the program. As it is easy to see comparing the last fragment with the one above, this macro expands to nothing in the (usual) ANSI mode and prefixes `'L'` to its argument in the Unicode mode.

The important conclusion is that if you use `wxChar` instead of `char`, avoid using C style strings and use `wxString` instead and don't forget to enclose all string literals inside *wxT()* (p. 1612) macro, your program automatically becomes (almost) Unicode compliant!

Just let us state once again the rules:

- Always use `wxChar` instead of `char`

- Always enclose literal string constants in *wxT()* (p. 1612) macro unless they're already converted to the right representation (another standard wxWidgets macro *_()* (p. 1613) does it, for example, so there is no need for `wxT()` in this case) or you intend to pass the constant directly to an external function which doesn't accept wide-character strings.

- Use `wxString` instead of C style strings.

Unicode and the outside world

We have seen that it was easy to write Unicode programs using wxWidgets types and macros, but it has been also mentioned that it isn't quite enough. Although everything works fine inside the program, things can get nasty when it tries to communicate with the outside world which, sadly, often expects ANSI strings (a notable exception is the entire Win32 API which accepts either Unicode or ANSI strings and which thus makes it unnecessary to ever perform any conversions in the program). GTK 2.0 only accepts UTF-8 strings.

To get a ANSI string from a wxString, you may use the mb_str() function which always returns an ANSI string (independently of the mode - while the usual *c_str()* (p. 1294) returns a pointer to the internal representation which is either ASCII or Unicode). More rarely used, but still useful, is wc_str() function which always returns the Unicode string.

Unicode-related compilation settings

You should define `wxUSE_UNICODE` to 1 to compile your program in Unicode mode. Note that it currently only works in Win32 and GTK 2.0 and that some parts of wxWidgets are not Unicode-compliant yet. If you compile your program in ANSI mode you can still define `wxUSE_WCHAR_T` to get some limited support for `wchar_t` type.

This will allow your program to perform conversions between Unicode strings and ANSI ones (using *wxMBConv classes* (p. 1733)) and construct wxString objects from Unicode strings (presumably read from some external file or elsewhere).

wxMBConv classes overview

Classes: *wxMBConv* (p. 909), wxMBConvLibc, *wxMBConvUTF7* (p. 912), *wxMBConvUTF8* (p. 913), *wxCSConv* (p. 225), *wxMBConvUTF16* (p. 914), *wxMBConvUTF32* (p. 915)

The wxMBConv classes in wxWidgets enable an Unicode-aware application to easily convert between Unicode and the variety of 8-bit encoding systems still in use.

Background: The need for conversion

As programs are becoming more and more globalized, and users exchange documents across country boundaries as never before, applications increasingly need to take into account all the different character sets in use around the world. It is no longer enough to just depend on the default byte-sized character set that computers have traditionally used.

A few years ago, a solution was proposed: the Unicode standard. Able to contain the complete set of characters in use in one unified global coding system, it would resolve

the character set problems once and for all.

But it hasn't happened yet, and the migration towards Unicode has created new challenges, resulting in "compatibility encodings" such as UTF-8. A large number of systems out there still depends on the old 8-bit encodings, hampered by the huge amounts of legacy code still widely deployed. Even sending Unicode data from one Unicode-aware system to another may need encoding to an 8-bit multibyte encoding (UTF-7 or UTF-8 is typically used for this purpose), to pass unhindered through any traditional transport channels.

Background: The wxString class

If you have compiled wxWidgets in Unicode mode, the wxChar type will become identical to wchar_t rather than char, and a wxString stores wxChars. Hence, all wxString manipulation in your application will then operate on Unicode strings, and almost as easily as working with ordinary char strings (you just need to remember to use the wxT() macro to encapsulate any string literals).

But often, your environment doesn't want Unicode strings. You could be sending data over a network, or processing a text file for some other application. You need a way to quickly convert your easily-handled Unicode data to and from a traditional 8-bit encoding. And this is what the wxMBConv classes do.

wxMBConv classes

The base class for all these conversions is the wxMBConv class (which itself implements standard libc locale conversion). Derived classes include wxMBConvLibc, several different wxMBConvUTFxxx classes, and wxCSConv, which implement different kinds of conversions. You can also derive your own class for your own custom encoding and use it, should you need it. All you need to do is override the MB2WC and WC2MB methods.

wxMBConv objects

Several of the wxWidgets-provided wxMBConv classes have predefined instances (wxConvLibc, wxConvFileName, wxConvUTF7, wxConvUTF8, wxConvLocal). You can use these predefined objects directly, or you can instantiate your own objects.

A variable, wxConvCurrent, points to the conversion object that the user interface is supposed to use, in the case that the user interface is not Unicode-based (like with GTK+ 1.2). By default, it points to wxConvLibc or wxConvLocal, depending on which works best on the current platform.

wxCSConv

The wxCSConv class is special because when it is instantiated, you can tell it which character set it should use, which makes it meaningful to keep many instances of them around, each with a different character set (or you can create a wxCSConv instance on

the fly).

The predefined wxCSConv instance, wxConvLocal, is preset to use the default user character set, but you should rarely need to use it directly, it is better to go through wxConvCurrent.

Converting strings

Once you have chosen which object you want to use to convert your text, here is how you would use them with wxString. These examples all assume that you are using a Unicode build of wxWidgets, although they will still compile in a non-Unicode build (they just won't convert anything).

Example 1: Constructing a wxString from input in current encoding.

```
wxString str(input_data, *wxConvCurrent);
```

Example 2: Input in UTF-8 encoding.

```
wxString str(input_data, wxConvUTF8);
```

Example 3: Input in KOI8-R. Construction of wxCSConv instance on the fly.

```
wxString str(input_data, wxCSConv(wxT("koi8-r")));
```

Example 4: Printing a wxString to stdout in UTF-8 encoding.

```
puts(str.mb_str(wxConvUTF8));
```

Example 5: Printing a wxString to stdout in custom encoding. Using preconstructed wxCSConv instance.

```
wxCSConv cust(user_encoding);
printf("Data: %s\n", (const char*) str.mb_str(cust));
```

Note: Since mb_str() returns a temporary wxCharBuffer to hold the result of the conversion, you need to explicitly cast it to const char* if you use it in a vararg context (like with printf).

Converting buffers

If you have specialized needs, or just don't want to use wxString, you can also use the conversion methods of the conversion objects directly. This can even be useful if you need to do conversion in a non-Unicode build of wxWidgets; converting a string from UTF-8 to the current encoding should be possible by doing this:

```
wxString str(wxConvUTF8.cMB2WC(input_data), *wxConvCurrent);
```

Here, cMB2WC of the UTF8 object returns a wxWCharBuffer containing a Unicode string. The wxString constructor then converts it back to an 8-bit character set using the passed conversion object, *wxConvCurrent. (In a Unicode build of wxWidgets, the constructor ignores the passed conversion object and retains the Unicode data.)

This could also be done by first making a wxString of the original data:

```
wxString input_str(input_data);
wxString str(input_str.wc_str(wxConvUTF8), *wxConvCurrent);
```

To print a wxChar buffer to a non-Unicode stdout:

```
printf("Data: %s\n", (const char*) wxConvCurrent->cWX2MB(unicode_data));
```

If you need to do more complex processing on the converted data, you may want to store the temporary buffer in a local variable:

```
const wxWX2MBbuf tmp_buf = wxConvCurrent->cWX2MB(unicode_data);
const char *tmp_str = (const char*) tmp_buf;
printf("Data: %s\n", tmp_str);
process_data(tmp_str);
```

If a conversion had taken place in cWX2MB (i.e. in a Unicode build), the buffer will be deallocated as soon as tmp_buf goes out of scope. (The macro wxWX2MBbuf reflects the correct return value of cWX2MB (either char* or wxCharBuffer), except for the const.)

Internationalization

Although internationalization of an application (i18n for short) involves far more than just translating its text messages to another message - date, time and currency formats need changing too, some languages are written left to right and others right to left, character encoding may differ and many other things may need changing too - it is a necessary first step. wxWidgets provides facilities for message translation with its *wxLocale* (p. 882) class and is itself fully translated into several languages. Please consult wxWidgets home page for the most up-to-date translations - and if you translate it into one of the languages not done yet, your translations would be gratefully accepted for inclusion into future versions of the library!

The wxWidgets approach to i18n closely follows the GNU gettext package. wxWidgets uses the message catalogs which are binary compatible with gettext catalogs and this allows to use all of the programs in this package to work with them. But note that no additional libraries are needed during run-time, however, so you have only the message catalogs to distribute and nothing else.

During program development you will need the gettext package for working with message catalogs. **Warning:** gettext versions < 0.10 are known to be buggy, so you should find a later version of it!

There are two kinds of message catalogs: source catalogs which are text files with extension .po and binary catalogs which are created from the source ones with *msgfmt* program (part of gettext package) and have the extension .mo. Only the binary files are needed during program execution.

The program i18n involves several steps:

1. Translating the strings in the program text using *wxGetTranslation* (p. 1610) or

equivalently the *_()* (p. 1613) macro.

2. Extracting the strings to be translated from the program: this uses the work done in the previous step because `xgettext` program used for string extraction recognises the standard _() as well as (using its `-k` option) our wxGetTranslation and extracts all strings inside the calls to these functions. Alternatively, you may use `-a` option to extract all the strings, but it will usually result in many strings being found which don't have to be translated at all. This will create a text message catalog - a .po file.

3. Translating the strings extracted in the previous step to other language(s). It involves editing the .po file.

4. Compiling the .po file into .mo file to be used by the program.

5. Setting the appropriate locale in your program to use the strings for the given language: see *wxLocale* (p. 882).

If you want your app to run under MacOS X with internationlization as described above you'll need to make one modification to the Info.plist file which describes the contents of the "application bundle". This file (an XML text file in UTF-8 format) should have a CFBundleDevelopmentRegion entry describing the language of the developer - mostly English - and normally MacOS X will query the bundle for the presence of certain resource directories to find out which languages are supported (e.g. the directory German.lproj for German). Since wxWidgets based applications don't use these directories for storing resource information (they store the translation in the mo files instead) the application needs to be told explicitly which langauges are supported. This is done by adding a CFBundleLocalizations entry to Info.plist. This can look like this:

```
<key>CFBundleDevelopmentRegion</key>
<string>English</string>
<key>CFBundleLocalizations</key>
<array>
      <string>en</string>
      <string>de</string>
      <string>fr</string>
</array>
```

See also the GNU gettext documentation linked from `docs/html/index.htm` in your wxWidgets distribution.

See also *Writing non-English applications* (p. 1737). It focuses on handling charsets related problems.

Finally, take a look at the *i18n sample* (p. 1712) which shows you how all this looks in practice.

Writing non-English applications

This article describes how to write applications that communicate with the user in a language other than English. Unfortunately many languages use different charsets under Unix and Windows (and other platforms, to make the situation even more complicated).

These charsets usually differ in so many characters that it is impossible to use the same texts under all platforms.

The wxWidgets library provides a mechanism that helps you avoid distributing many identical, only differently encoded, packages with your application (e.g. help files and menu items in iso8859-13 and windows-1257). Thanks to this mechanism you can, for example, distribute only iso8859-13 data and it will be handled transparently under all systems.

Please read *Internationalization* (p. 1736) which describes the locales concept.

In the following text, wherever *iso8859-2* and *windows-1250* are used, any encodings are meant and any encodings may be substituted there.

Locales

The best way to ensure correctly displayed texts in a GUI across platforms is to use locales. Write your in-code messages in English or without diacritics and put real messages into the message catalog (see *Internationalization* (p. 1736)).

A standard .po file begins with a header like this:

```
# SOME DESCRIPTIVE TITLE.
# Copyright (C) YEAR Free Software Foundation, Inc.
# FIRST AUTHOR <EMAIL@ADDRESS>, YEAR.
#
msgid ""
msgstr ""
"Project-Id-Version: PACKAGE VERSION\n"
"POT-Creation-Date: 1999-02-19 16:03+0100\n"
"PO-Revision-Date: YEAR-MO-DA HO:MI+ZONE\n"
"Last-Translator: FULL NAME <EMAIL@ADDRESS>\n"
"Language-Team: LANGUAGE <LL@li.org>\n"
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=CHARSET\n"
"Content-Transfer-Encoding: ENCODING\n"
```

Note this particular line:

```
"Content-Type: text/plain; charset=CHARSET\n"
```

It specifies the charset used by the catalog. All strings in the catalog are encoded using this charset.

You have to fill in proper charset information. Your .po file may look like this after doing so:

```
# SOME DESCRIPTIVE TITLE.
# Copyright (C) YEAR Free Software Foundation, Inc.
# FIRST AUTHOR <EMAIL@ADDRESS>, YEAR.
#
msgid ""
msgstr ""
"Project-Id-Version: PACKAGE VERSION\n"
"POT-Creation-Date: 1999-02-19 16:03+0100\n"
"PO-Revision-Date: YEAR-MO-DA HO:MI+ZONE\n"
"Last-Translator: FULL NAME <EMAIL@ADDRESS>\n"
"Language-Team: LANGUAGE <LL@li.org>\n"
```

```
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=iso8859-2\n"
"Content-Transfer-Encoding: 8bit\n"
```

(Make sure that the header is **not** marked as *fuzzy*.)

wxWidgets is able to use this catalog under any supported platform (although iso8859-2 is a Unix encoding and is normally not understood by Windows).

How is this done? When you tell the wxLocale class to load a message catalog that contains a correct header, it checks the charset. The catalog is then converted to the charset used (see*wxLocale::GetSystemEncoding* (p. 887) and*wxLocale::GetSystemEncodingName* (p. 887)) by the user's operating system. This is the default behaviour of the*wxLocale* (p. 882) class; you can disable it by **not** passing`wxLOCALE_CONV_ENCODING` to *wxLocale::Init* (p. 888).

Non-English strings or 8-bit characters in the source code

By convention, you should only use characters without diacritics (i.e. 7-bit ASCII strings) for msgids in the source code and write them in English.

If you port software to wxWindows, you may be confronted with legacy source code containing non-English string literals. Instead of translating the strings in the source code to English and putting the original strings into message catalog, you may configure wxWidgets to use non-English msgids and translate to English using message catalogs:

1. you use the program `xgettext` to extract the strings from the source code, specify the option `--from-code=<source code charset>`.

2. the source code language and charset as arguments to*wxLocale::AddCatalog* (p. 883). For example:`locale.AddCatalog(_T("myapp"), wxLANGUAGE_GERMAN, _T("iso-8859-1"));`

Font mapping

You can use *wxMBConv classes* (p. 1733) and *wxFontMapper* (p. 575) to display text:

```
if (!wxFontMapper::Get()->IsEncodingAvailable(enc, facename))
{
    wxFontEncoding alternative;
    if (wxFontMapper::Get()->GetAltForEncoding(enc, &alternative,
                                                facename, false))
    {
        wxCSConv convFrom(wxFontMapper::Get()->GetEncodingName(enc));
        wxCSConv convTo(wxFontMapper::Get()->GetEncodingName(alternative));
        text = wxString(text.mb_str(convFrom), convTo);
    }
    else
        ...failure (or we may try iso8859-1/7bit ASCII)...
}
...display text...
```

Converting data

You may want to store all program data (created documents etc.) in the same encoding,

let's say `utf-8`. You can use*wxCSConv* (p. 225) class to convert data to the encoding used by the system your application is running on (see*wxLocale::GetSystemEncoding* (p. 887)).

Help files

If you're using *wxHtmlHelpController* (p. 722) there is no problem at all. You only need to make sure that all the HTML files contain the META tag, e.g.

```
<meta http-equiv="Content-Type" content="text/html; charset=iso8859-2">
```

and that the hhp project file contains one additional line in the `OPTIONS`section:

```
Charset=iso8859-2
```

This additional entry tells the HTML help controller what encoding is used in contents and index tables.

Container classes overview

Classes: *wxList* (p. 840), *wxArray* (p. 53)

wxWidgets uses itself several container classes including doubly-linked lists and dynamic arrays (i.e. arrays which expand automatically when they become full). For both historical and portability reasons wxWidgets does not use STL which provides the standard implementation of many container classes in C++. First of all, wxWidgets has existed since well before STL was written, and secondly we don't believe that today compilers can deal really well with all of STL classes (this is especially true for some less common platforms). Of course, the compilers are evolving quite rapidly and hopefully their progress will allow to base future versions of wxWidgets on STL - but this is not yet the case.

wxWidgets container classes don't pretend to be as powerful or full as STL ones, but they are quite useful and may be compiled with absolutely any C++ compiler. They're used internally by wxWidgets, but may, of course, be used in your programs as well if you wish.

The list classes in wxWidgets are doubly-linked lists which may either own the objects they contain (meaning that the list deletes the object when it is removed from the list or the list itself is destroyed) or just store the pointers depending on whether you called or not *wxList::DeleteContents* (p. 843) method.

Dynamic arrays resemble C arrays but with two important differences: they provide run-time range checking in debug builds and they automatically expand the allocated memory when there is no more space for new items. They come in two sorts: the "plain" arrays which store either built-in types such as "char", "int" or "bool" or the pointers to arbitrary objects, or "object arrays" which own the object pointers to which they store.

For the same portability reasons, the container classes implementation in wxWidgets does not use templates, but is rather based on C preprocessor i.e. is done with the macros: *WX_DECLARE_LIST* and *WX_DEFINE_LIST* for the linked lists and

WX_DECLARE_ARRAY, *WX_DECLARE_OBJARRAY* and *WX_DEFINE_OBJARRAY* for the dynamic arrays. The "DECLARE" macro declares a new container class containing the elements of given type and is needed for all three types of container classes: lists, arrays and objarrays. The "DEFINE" classes must be inserted in your program in a place where the **full declaration of container element class is in scope** (i.e. not just forward declaration), otherwise destructors of the container elements will not be called! As array classes never delete the items they contain anyhow, there is no WX_DEFINE_ARRAY macro for them.

Examples of usage of these macros may be found in *wxList* (p. 840) and *wxArray* (p. 53) documentation.

Finally, wxWidgets predefines several commonly used container classes. wxList is defined for compatibility with previous versions as a list containing wxObjects and wxStringList as a list of C-style strings (char *), both of these classes are deprecated and should not be used in new programs. The following array classes are defined: wxArrayInt, wxArrayLong, wxArrayPtrVoid and wxArrayString. The first three store elements of corresponding types, but wxArrayString is somewhat special: it is an optimized version of wxArray which uses its knowledge about *wxString* (p. 1284) reference counting schema.

File classes and functions overview

Classes: *wxFile* (p. 502), *wxDir* (p. 420), *wxTempFile* (p. 1331), *wxTextFile* (p. 1363)

Functions: see *file functions* (p. 1599).

wxWidgets provides some functions and classes to facilitate working with files. As usual, the accent is put on cross-platform features which explains, for example, the *wxTextFile* (p. 1363) class which may be used to convert between different types of text files (DOS/Unix/Mac).

wxFile may be used for low-level IO. It contains all the usual functions to work with files (opening/closing, reading/writing, seeking, and so on) but compared with using standard C functions, has error checking (in case of an error a message is logged using *wxLog* (p. 889) facilities) and closes the file automatically in the destructor which may be quite convenient.

wxTempFile is a very small file designed to make replacing the files contents safer - see its *documentation* (p. 1331) for more details.

wxTextFile is a general purpose class for working with small text files on line by line basis. It is especially well suited for working with configuration files and program source files. It can be also used to work with files with "non native" line termination characters and write them as "native" files if needed (in fact, the files may be written in any format).

wxDir is a helper class for enumerating the files or subdirectories of a directory. It may be used to enumerate all files, only files satisfying the given template mask or only non-hidden files.

wxStreams overview

Classes: *wxStreamBase* (p. 1275), *wxStreamBuffer* (p. 1277), *wxInputStream* (p. 814), *wxOutputStream* (p. 1016), *wxFilterInputStream* (p. 549), *wxFilterOutputStream* (p. 549)

Purpose of wxStream

Standard C++ streams can cause problems on several platforms: they work quite well in most cases, but in the multi-threaded case, for example, they have many problems. Some Borland compilers refuse to work at all with them and using iostreams on Linux makes writing programs that are binary compatible across different Linux distributions, impossible.

Therefore, wxStreams have been added to wxWidgets so that applications can reliably compile and run on all supported platforms without dependence on a particular release of libg++.

wxStreams is divided in two main parts:

1. the core: wxStreamBase, wxStreamBuffer, wxInputStream, wxOutputStream, wxFilterIn/OutputStream

2. the "IO" classes: wxSocketIn/OutputStream, wxDataIn/OutputStream, wxFileIn/OutputStream, ...

wxStreamBase is the base definition of a stream. It defines, for example, the API of OnSysRead, OnSysWrite, OnSysSeek and OnSysTell. These functions are really implemented by the "IO" classes. wxInputStream and wxOutputStream inherit from it.

wxStreamBuffer is a cache manager for wxStreamBase: it manages a stream buffer linked to a stream. One stream can have multiple stream buffers but one stream have always one autoinitialized stream buffer.

wxInputStream is the base class for read-only streams. It implements Read, SeekI (I for Input), and all read or IO generic related functions. wxOutputStream does the same thing but it is for write-only streams.

wxFilterIn/OutputStream is the base class definition for stream filtering. Stream filtering means a stream which does no syscall but filters data which are passed to it and then pass them to another stream. For example, wxZLibInputStream is an inline stream decompressor.

The "IO" classes implements the specific parts of the stream. This could be nothing in the case of wxMemoryIn/OutputStream which bases itself on wxStreamBuffer. This could also be a simple link to the a true syscall (for example read(...), write(...)).

Generic usage: an example

Usage is simple. We can take the example of wxFileInputStream and here is some sample code:

```
    ...
     // The constructor initializes the stream buffer and open the file
   descriptor
```

```
        // associated to the name of the file.
        wxFileInputStream in_stream("the_file_to_be_read");

        // Ok, read some bytes ... nb_datas is expressed in bytes.
        in_stream.Read(data, nb_datas);
        if (in_stream.LastError() != wxSTREAM_NOERROR) {
          // Oh oh, something bad happens.
          // For a complete list, look into the documentation at wxStreamBase.
        }

        // You can also inline all like this.
        if (in_stream.Read(data, nb_datas).LastError() != wxSTREAM_NOERROR) {
          // Do something.
        }

        // You can also get the last number of bytes REALLY put into the buffer.
        size_t really_read = in_stream.LastRead();

        // Ok, moves to the beginning of the stream. SeekI returns the last
       position
        // in the stream counted from the beginning.
        off_t old_position = in_stream.SeekI(0, wxFromBeginning);

        // What is my current position ?
        off_t position = in_stream.TellI();

        // wxFileInputStream will close the file descriptor on destruction.
```

wxLog classes overview

Classes: *wxLog* (p. 889),
wxLogStderr (p. 900),
wxLogStream (p. 900),
wxLogTextCtrl (p. 901),
wxLogWindow (p. 901),
wxLogGui (p. 898),
wxLogNull (p. 898),
wxLogChain (p. 896),
wxLogPassThrough (p. 899),
wxStreamToTextRedirector (p. 1283)

This is a general overview of logging classes provided by wxWidgets. The word logging here has a broad sense, including all of the program output, not only non interactive messages. The logging facilities included in wxWidgets provide the base *wxLog* class which defines the standard interface for a *log target* as well as several standard implementations of it and a family of functions to use with them.

First of all, no knowledge of *wxLog* classes is needed to use them. For this, you should only know about *wxLogXXX()* functions. All of them have the same syntax as *printf()* or *vprintf()* , i.e. they take the format string as the first argument and respectively a variable number of arguments or a variable argument list pointer. Here are all of them:

- **wxLogFatalError** which is like *wxLogError*, but also terminates the program with the exit code 3 (using *abort()* standard function). Unlike for all the other logging functions, this function can't be overridden by a log target.

- **wxLogError** is the function to use for error messages, i.e. the messages that must be shown to the user. The default processing is to pop up a message box to inform the user about it.

- **wxLogWarning** for warnings - they are also normally shown to the user, but don't interrupt the program work.

- **wxLogMessage** is for all normal, informational messages. They also appear in a message box by default (but it can be changed, see below).

- **wxLogVerbose** is for verbose output. Normally, it is suppressed, but might be activated if the user wishes to know more details about the program progress (another, but possibly confusing name for the same function is **wxLogInfo**).

- **wxLogStatus** is for status messages - they will go into the status bar of the active or specified (as the first argument) *wxFrame* (p. 579) if it has one.

- **wxLogSysError** is mostly used by wxWidgets itself, but might be handy for logging errors after system call (API function) failure. It logs the specified message text as well as the last system error code (*errno* or *::GetLastError()* depending on the platform) and the corresponding error message. The second form of this function takes the error code explicitly as the first argument.

- **wxLogDebug** is **the** right function for debug output. It only does anything at all in the debug mode (when the preprocessor symbol__WXDEBUG__ is defined) and expands to nothing in release mode (otherwise).**Tip:** under Windows, you must either run the program under debugger or use a 3rd party program such as DbgView (`http://www.sysinternals.com`) to actually see the debug output.

- **wxLogTrace** as **wxLogDebug** only does something in debug build. The reason for making it a separate function from it is that usually there are a lot of trace messages, so it might make sense to separate them from other debug messages which would be flooded in them. Moreover, the second version of this function takes a trace mask as the first argument which allows to further restrict the amount of messages generated.

The usage of these functions should be fairly straightforward, however it may be asked why not use the other logging facilities, such as C standard stdio functions or C++ streams. The short answer is that they're all very good generic mechanisms, but are not really adapted for wxWidgets, while the log classes are. Some of advantages in using wxWidgets log functions are:

- **Portability** It is a common practice to use *printf()*statements or cout/cerr C++ streams for writing out some (debug or otherwise) information. Although it works just fine under Unix, these messages go strictly nowhere under Windows where the stdout of GUI programs is not assigned to anything. Thus, you might view *wxLogMessage()* as a simple substitute for *printf()*.

```
You can also redirect the wxLogXXX calls to cout by just writing:  wxLog
*logger=new wxLogStream(&cout);
  wxLog::SetActiveTarget(logger);
```

Finally, there is also a possibility to redirect the output sent to *cout* to a *wxTextCtrl* (p. 1340) by using the *wxStreamToTextRedirector* (p. 1283) class.

- **Flexibility** The output of wxLog functions can be redirected or suppressed entirely based on their importance, which is either impossible or difficult to do with traditional methods. For example, only error messages, or only error messages and warnings might be logged, filtering out all informational messages.

- **Completeness** Usually, an error message should be presented to the user when some operation fails. Let's take a quite simple but common case of a file error: suppose that you're writing your data file on disk and there is not enough space. The actual error might have been detected inside wxWidgets code (say, in *wxFile::Write*), so the calling function doesn't really know the exact reason of the failure, it only knows that the data file couldn't be written to the disk. However, as wxWidgets uses *wxLogError()* in this situation, the exact error code (and the corresponding error message) will be given to the user together with "high level" message about data file writing error.

After having enumerated all the functions which are normally used to log the messages, and why would you want to use them we now describe how all this works.

wxWidgets has the notion of a *log target*: it is just a class deriving from *wxLog* (p. 889). As such, it implements the virtual functions of the base class which are called when a message is logged. Only one log target is *active* at any moment, this is the one used by *wxLogXXX()*functions. The normal usage of a log object (i.e. object of a class derived from wxLog) is to install it as the active target with a call to *SetActiveTarget()* and it will be used automatically by all subsequent calls to *wxLogXXX()* functions.

To create a new log target class you only need to derive it from wxLog and implement one (or both) of *DoLog()* and *DoLogString()* in it. The second one is enough if you're happy with the standard wxLog message formatting (prepending "Error:" or "Warning:", timestamping &c) but just want to send the messages somewhere else. The first one may be overridden to do whatever you want but you have to distinguish between the different message types yourself.

There are some predefined classes deriving from wxLog and which might be helpful to see how you can create a new log target class and, of course, may also be used without any change. There are:

- **wxLogStderr** This class logs messages to a *FILE **, using stderr by default as its name suggests.

- **wxLogStream** This class has the same functionality as wxLogStderr, but uses *ostream* and cerr instead of *FILE ** and stderr.

- **wxLogGui** This is the standard log target for wxWidgets applications (it is used by default if you don't do anything) and provides the most reasonable handling of all types of messages for given platform.

- **wxLogWindow** This log target provides a "log console" which collects all

messages generated by the application and also passes them to the previous active log target. The log window frame has a menu allowing user to clear the log, close it completely or save all messages to file.

- • **wxLogNull** The last log class is quite particular: it doesn't do anything. The objects of this class may be instantiated to (temporarily) suppress output of *wxLogXXX()* functions. As an example, trying to open a non-existing file will usually provoke an error message, but if for some reasons it is unwanted, just use this construction:

```
wxFile file;

// wxFile.Open() normally complains if file can't be opened, we don't
want it
  {
    wxLogNull logNo;
    if ( !file.Open("bar") )
      ... process error ourselves ...
  } // ~wxLogNull called, old log sink restored

wxLogMessage("..."); // ok
```

The log targets can also be combined: for example you may wish to redirect the messages somewhere else (for example, to a log file) but also process them as normally. For this the *wxLogChain* (p. 896) and *wxLogPassThrough* (p. 899) can be used.

Debugging overview

Classes, functions and macros: *wxDebugContext* (p. 392), *wxObject* (p. 1012), *wxLog* (p. 889), *Log functions* (p. 1648), *Debug macros* (p. 1657)

Various classes, functions and macros are provided in wxWidgets to help you debug your application. Most of these are only available if you compile both wxWidgets, your application and *all* libraries that use wxWidgets with the __WXDEBUG__ symbol defined. You can also test the __WXDEBUG__ symbol in your own applications to execute code that should be active only in debug mode.

wxDebugContext

wxDebugContext (p. 392) is a class that never gets instantiated, but ties together various static functions and variables. It allows you to dump all objects to that stream, write statistics about object allocation, and check memory for errors.

It is good practice to define a *wxObject::Dump* (p. 1013) member function for each class you derive from a wxWidgets class, so that *wxDebugContext::Dump* (p. 393) can call it and give valuable information about the state of the application.

If you have difficulty tracking down a memory leak, recompile in debugging mode and call *wxDebugContext::Dump* (p. 393) and *wxDebugContext::PrintStatistics* (p. 395) at appropriate places. They will tell you what objects have not yet been deleted, and what

kinds of object they are. In fact, in debug mode wxWidgets will automatically detect memory leaks when your application is about to exit, and if there are any leaks, will give you information about the problem. (How much information depends on the operating system and compiler -- some systems don't allow all memory logging to be enabled). See the memcheck sample for example of usage.

For wxDebugContext to do its work, the *new* and *delete* operators for wxObject have been redefined to store extra information about dynamically allocated objects (but not statically declared objects). This slows down a debugging version of an application, but can find difficult-to-detect memory leaks (objects are not deallocated), overwrites (writing past the end of your object) and underwrites (writing to memory in front of the object).

If debugging mode is on and the symbols wxUSE_GLOBAL_MEMORY_OPERATORS and wxUSE_DEBUG_NEW_ALWAYS are set to 1 in setup.h, 'new' is defined to be:

```
#define new new(__FILE__,__LINE__)
```

All occurrences of 'new' in wxWidgets and your own application will use the overridden form of the operator with two extra arguments. This means that the debugging output (and error messages reporting memory problems) will tell you what file and on what line you allocated the object. Unfortunately not all compilers allow this definition to work properly, but most do.

Debug macros

You should also use *debug macros* (p. 1657) as part of a 'defensive programming' strategy, scattering wxASSERTs liberally to test for problems in your code as early as possible. Forward thinking will save a surprising amount of time in the long run.

wxASSERT (p. 1657) is used to pop up an error message box when a condition is not true. You can also use *wxASSERT_MSG* (p. 1658) to supply your own helpful error message. For example:

```
void MyClass::MyFunction(wxObject* object)
{
    wxASSERT_MSG( (object != NULL), "object should not be NULL in
MyFunction!" );

    ...
};
```

The message box allows you to continue execution or abort the program. If you are running the application inside a debugger, you will be able to see exactly where the problem was.

Logging functions

You can use the *wxLogDebug* (p. 1651) and *wxLogTrace* (p. 1651) functions to output debugging information in debug mode; it will do nothing for non-debugging code.

wxDebugContext overview

Debugging overview (p. 1746)

Class: *wxDebugContext* (p. 392)

wxDebugContext is a class for performing various debugging and memory tracing operations.

This class has only static data and function members, and there should be no instances. Probably the most useful members are SetFile (for directing output to a file, instead of the default standard error or debugger output); Dump (for dumping the dynamically allocated objects) and PrintStatistics (for dumping information about allocation of objects). You can also call Check to check memory blocks for integrity.

Here's an example of use. The SetCheckpoint ensures that only the allocations done after the checkpoint will be dumped.

```
wxDebugContext::SetCheckpoint();

wxDebugContext::SetFile("c:\\temp\\debug.log");

wxString *thing = new wxString;

char *ordinaryNonObject = new char[1000];

wxDebugContext::Dump();
wxDebugContext::PrintStatistics();
```

You can use wxDebugContext if __WXDEBUG__ is defined, or you can use it at any other time (if wxUSE_DEBUG_CONTEXT is set to 1 in setup.h). It is not disabled in non-debug mode because you may not wish to recompile wxWidgets and your entire application just to make use of the error logging facility.

Note: wxDebugContext::SetFile has a problem at present, so use the default stream instead. Eventually the logging will be done through the wxLog facilities instead.

wxConfig classes overview

Classes: *wxConfig* (p. 193)

This overview briefly describes what the config classes are and what they are for. All the details about how to use them may be found in the description of the *wxConfigBase* (p. 193) class and the documentation of the file, registry and INI file based implementations mentions all the features/limitations specific to each one of these versions.

The config classes provide a way to store some application configuration information. They were especially designed for this usage and, although may probably be used for many other things as well, should be limited to it. It means that this information should be:

1. Typed, i.e. strings or numbers for the moment. You can not store binary data, for example.

2. Small. For instance, it is not recommended to use the Windows registry for

amounts of data more than a couple of kilobytes.

3. Not performance critical, neither from speed nor from a memory consumption point of view.

On the other hand, the features provided make them very useful for storing all kinds of small to medium volumes of hierarchically-organized, heterogeneous data. In short, this is a place where you can conveniently stuff all your data (numbers and strings) organizing it in a tree where you use the filesystem-like paths to specify the location of a piece of data. In particular, these classes were designed to be as easy to use as possible.

From another point of view, they provide an interface which hides the differences between the Windows registry and the standard Unix text format configuration files. Other (future) implementations of wxConfigBase might also understand GTK resource files or their analogues on the KDE side.

In any case, each implementation of wxConfigBase does its best to make the data look the same way everywhere. Due to limitations of the underlying physical storage, it may not implement 100% of the base class functionality.

There are groups of entries and the entries themselves. Each entry contains either a string or a number (or a boolean value; support for other types of data such as dates or timestamps is planned) and is identified by the full path to it: something like /MyApp/UserPreferences/Colors/Foreground. The previous elements in the path are the group names, and each name may contain an arbitrary number of entries and subgroups. The path components are **always** separated with a slash, even though some implementations use the backslash internally. Further details (including how to read/write these entries) may be found in the documentation for *wxConfigBase* (p. 193).

wxFileSystem

The wxHTML library uses a **virtual file systems** mechanism similar to the one used in Midnight Commander, Dos Navigator, FAR or almost any modern file manager. It allows the user to access data stored in archives as if they were ordinary files. On-the-fly generated files that exist only in memory are also supported.

Classes

Three classes are used in order to provide virtual file systems mechanism:

- The *wxFSFile* (p. 590) class provides information about opened file (name, input stream, mime type and anchor).

- The *wxFileSystem* (p. 540) class is the interface. Its main methods are ChangePathTo() and OpenFile(). This class is most often used by the end user.

- The *wxFileSystemHandler* (p. 542) is the core of virtual file systems mechanism. You can derive your own handler and pass it to the VFS mechanism. You can derive your own handler and pass it to wxFileSystem's AddHandler() method. In the new handler you only need to override the OpenFile() and CanOpen()

methods.

Locations

Locations (aka filenames aka addresses) are constructed from four parts:

- **protocol** - handler can recognize if it is able to open a file by checking its protocol. Examples are "http", "file" or "ftp".

- **right location** - is the name of file within the protocol. In "http://www.wxwidgets.org/index.html" the right location is "//www.wxwidgets.org/index.html".

- **anchor** - an anchor is optional and is usually not present. In "index.htm#chapter2" the anchor is "chapter2".

- **left location** - this is usually an empty string. It is used by 'local' protocols such as ZIP. See Combined Protocols paragraph for details.

Combined Protocols

The left location precedes the protocol in the URL string. It is not used by global protocols like HTTP but it becomes handy when nesting protocols - for example you may want to access files in a ZIP archive:

file:archives/cpp_doc.zip#zip:reference/fopen.htm#syntax

In this example, the protocol is "zip", right location is "reference/fopen.htm", anchor is "syntax" and left location is "file:archives/cpp_doc.zip".

There are **two** protocols used in this example: "zip" and "file".

File Systems Included in wxHTML

The following virtual file system handlers are part of wxWidgets so far:

wxInternetFSHandler	A handler for accessing documents via HTTP or FTP protocols. Include file is <wx/fs_inet.h>.
wxZipFSHandler	A handler for ZIP archives. Include file is <wx/fs_zip.h>. URL is in form "archive.zip#zip:filename".
wxMemoryFSHandler	This handler allows you to access data stored in memory (such as bitmaps) as if they were regular files. See *wxMemoryFSHandler documentation* (p. 936) for details. Include file is <wx/fs_mem.h>. URL is prefixed with memory:, e.g. "memory:myfile.htm"

In addition, wxFileSystem itself can access local files.

Initializing file system handlers

Use *wxFileSystem::AddHandler* (p. 540) to initialize a handler, for example:

```
#include <wx/fs_mem.h>

...

bool MyApp::OnInit()
{
    wxFileSystem::AddHandler(new wxMemoryFSHandler);
...
}
```

Event handling overview

Classes: *wxEvtHandler* (p. 486), *wxWindow* (p. 1490), *wxEvent* (p. 483)

Introduction

Before version 2.0 of wxWidgets, events were handled by the application either by supplying callback functions, or by overriding virtual member functions such as **OnSize**.

From wxWidgets 2.0, *event tables* are used instead, with a few exceptions.

An event table is placed in an implementation file to tell wxWidgets how to map events to member functions. These member functions are not virtual functions, but they are all similar in form: they take a single wxEvent-derived argument, and have a void return type.

Here's an example of an event table.

```
BEGIN_EVENT_TABLE(MyFrame, wxFrame)
  EVT_MENU    (wxID_EXIT, MyFrame::OnExit)
  EVT_MENU    (DO_TEST,   MyFrame::DoTest)
  EVT_SIZE    (          MyFrame::OnSize)
  EVT_BUTTON  (BUTTON1,   MyFrame::OnButton1)
END_EVENT_TABLE()
```

The first two entries map menu commands to two different member functions. The EVT_SIZE macro doesn't need a window identifier, since normally you are only interested in the current window's size events.

The EVT_BUTTON macro demonstrates that the originating event does not have to come from the window class implementing the event table -- if the event source is a button within a panel within a frame, this will still work, because event tables are searched up through the hierarchy of windows for the command events. In this case, the button's event table will be searched, then the parent panel's, then the frame's.

As mentioned before, the member functions that handle events do not have to be virtual. Indeed, the member functions should not be virtual as the event handler ignores that the functions are virtual, i.e. overriding a virtual member function in a derived class will not have any effect. These member functions take an event argument, and the class of event differs according to the type of event and the class of the originating window. For

size events, *wxSizeEvent* (p. 1175) is used. For menu commands and most control commands (such as button presses), *wxCommandEvent* (p. 180) is used. When controls get more complicated, then specific event classes are used, such as *wxTreeEvent* (p. 1443) for events from *wxTreeCtrl* (p. 1424) windows.

As well as the event table in the implementation file, there must also be a DECLARE_EVENT_TABLE macro somewhere in the class declaration. For example:

```
class MyFrame : public wxFrame
{
public:
  ...
  void OnExit(wxCommandEvent& event);
  void OnSize(wxSizeEvent& event);

protected:
  int       m_count;
  ...

  DECLARE_EVENT_TABLE()
};
```

Note that this macro may occur in any section of the class (public, protected or private) but that it is probably better to insert it at the end, as shown, because this macro implicitly changes the access to protected which may be quite unexpected if there is anything following it.

Finally, if you don't like using macros for static initialization of the event tables you may also use *wxEvtHandler::Connect* (p. 488) to connect the events to the handlers dynamically, during run-time. See the *event sample* (p. 1711) for an example of doing it.

How events are processed

When an event is received from the windowing system, wxWidgets calls *wxEvtHandler::ProcessEvent* (p. 491) on the first event handler object belonging to the window generating the event.

It may be noted that wxWidgets' event processing system implements something very close to virtual methods in normal C++, i.e. it is possible to alter the behaviour of a class by overriding its event handling functions. In many cases this works even for changing the behaviour of native controls. For example it is possible to filter out a number of key events sent by the system to a native text control by overriding wxTextCtrl and defining a handler for key events using EVT_KEY_DOWN. This would indeed prevent any key events from being sent to the native control - which might not be what is desired. In this case the event handler function has to call Skip() so as to indicate that the search for the event handler should continue.

To summarize, instead of explicitly calling the base class version as you would have done with C++ virtual functions (i.e. *wxTextCtrl::OnChar()*), you should instead call *Skip* (p. 486).

In practice, this would look like this if the derived text control only accepts 'a' to 'z' and 'A' to 'Z':

```
void MyTextCtrl::OnChar(wxKeyEvent& event)
{
    if ( isalpha( event.KeyCode() ) )
    {
        // key code is within legal range. we call event.Skip() so the
        // event can be processed either in the base wxWidgets class
        // or the native control.

        event.Skip();
    }
    else
    {
        // illegal key hit. we don't call event.Skip() so the
        // event is not processed anywhere else.

        wxBell();
    }
}
```

The normal order of event table searching by ProcessEvent is as follows:

1. If the object is disabled (via a call to *wxEvtHandler::SetEvtHandlerEnabled* (p. 493)) the function skips to step (6).

2. If the object is a wxWindow, **ProcessEvent** is recursively called on the window's *wxValidator* (p. 1461). If this returns true, the function exits.

3. **SearchEventTable** is called for this event handler. If this fails, the base class table is tried, and so on until no more tables exist or an appropriate function was found, in which case the function exits.

4. The search is applied down the entire chain of event handlers (usually the chain has a length of one). If this succeeds, the function exits.

5. If the object is a wxWindow and the event is set to set to propagate (in the library only wxCommandEvent based events are set to propagate), **ProcessEvent** is recursively applied to the parent window's event handler. If this returns true, the function exits.

6. Finally, **ProcessEvent** is called on the wxApp object.

Pay close attention to Step 5. People often overlook or get confused by this powerful feature of the wxWidgets event processing system. To put it a different way, events set to propagate (*See: wxEvent::ShouldPropagate* (p. 486)) (most likely derived either directly or indirectly from wxCommandEvent) will travel up the containment hierarchy from child to parent until the maximal propagation level is reached or an event handler is found that doesn't call *event.Skip()* (p. 486).

Finally, there is another additional complication (which, in fact, simplifies life of wxWidgets programmers significantly): when propagating the command events upwards to the parent window, the event propagation stops when it reaches the parent dialog, if any. This means that you don't risk to get unexpected events from the dialog controls (which might be left unprocessed by the dialog itself because it doesn't care about them) when a modal dialog is popped up. The events do propagate beyond the frames, however. The rationale for this choice is that there are only a few frames in a typical application and their parent-child relation are well understood by the programmer while it

may be very difficult, if not impossible, to track down all the dialogs which may be popped up in a complex program (remember that some are created automatically by wxWidgets). If you need to specify a different behaviour for some reason, you can use *SetExtraStyle(wxWS_EX_BLOCK_EVENTS)* (p. 1533) explicitly to prevent the events from being propagated beyond the given window or unset this flag for the dialogs which have it on by default.

Typically events that deal with a window as a window (size, motion, paint, mouse, keyboard, etc.) are sent only to the window. Events that have a higher level of meaning and/or are generated by the window itself, (button click, menu select, tree expand, etc.) are command events and are sent up to the parent to see if it is interested in the event.

Note that your application may wish to override ProcessEvent to redirect processing of events. This is done in the document/view framework, for example, to allow event handlers to be defined in the document or view. To test for command events (which will probably be the only events you wish to redirect), you may use *wxEvent::IsCommandEvent* (p. 485) for efficiency, instead of using the slower run-time type system.

As mentioned above, only command events are recursively applied to the parents event handler in the library itself. As this quite often causes confusion for users, here is a list of system events which will NOT get sent to the parent's event handler:

wxEvent (p. 483)	The event base class
wxActivateEvent (p. 30)	A window or application activation event
wxCloseEvent (p. 153)	A close window or end session event
wxEraseEvent (p. 482)	An erase background event
wxFocusEvent (p. 557)	A window focus event
wxKeyEvent (p. 830)	A keypress event
wxIdleEvent (p. 776)	An idle event
wxInitDialogEvent (p. 813)	A dialog initialisation event
wxJoystickEvent (p. 827)	A joystick event
wxMenuEvent (p. 962)	A menu event
wxMouseEvent (p. 982)	A mouse event
wxMoveEvent (p. 992)	A move event
wxPaintEvent (p. 1025)	A paint event
wxQueryLayoutInfoEvent (p. 1087)	Used to query layout information
wxSetCursorEvent (p. 1166)	Used for special cursor processing based on current mouse position
wxSizeEvent (p. 1175)	A size event

wxScrollWinEvent (p. 1162)	A scroll event sent by a scrolled window (not a scroll bar)
wxSysColourChangedEvent (p. 1314)	A system colour change event

In some cases, it might be desired by the programmer to get a certain number of system events in a parent window, for example all key events sent to, but not used by, the native controls in a dialog. In this case, a special event handler will have to be written that will override ProcessEvent() in order to pass all events (or any selection of them) to the parent window.

Pluggable event handlers

In fact, you don't have to derive a new class from a window class if you don't want to. You can derive a new class from wxEvtHandler instead, defining the appropriate event table, and then call *wxWindow::SetEventHandler* (p. 1532) (or, preferably, *wxWindow::PushEventHandler* (p. 1522)) to make this event handler the object that responds to events. This way, you can avoid a lot of class derivation, and use the same event handler object to handle events from instances of different classes. If you ever have to call a window's event handler manually, use the GetEventHandler function to retrieve the window's event handler and use that to call the member function. By default, GetEventHandler returns a pointer to the window itself unless an application has redirected event handling using SetEventHandler or PushEventHandler.

One use of PushEventHandler is to temporarily or permanently change the behaviour of the GUI. For example, you might want to invoke a dialog editor in your application that changes aspects of dialog boxes. You can grab all the input for an existing dialog box, and edit it 'in situ', before restoring its behaviour to normal. So even if the application has derived new classes to customize behaviour, your utility can indulge in a spot of body-snatching. It could be a useful technique for on-line tutorials, too, where you take a user through a serious of steps and don't want them to diverge from the lesson. Here, you can examine the events coming from buttons and windows, and if acceptable, pass them through to the original event handler. Use PushEventHandler/PopEventHandler to form a chain of event handlers, where each handler processes a different range of events independently from the other handlers.

Window identifiers

Window identifiers are integers, and are used to uniquely determine window identity in the event system (though you can use it for other purposes). In fact, identifiers do not need to be unique across your entire application just so long as they are unique within a particular context you're interested in, such as a frame and its children. You may use the `wxID_OK` identifier, for example, on any number of dialogs so long as you don't have several within the same dialog.

If you pass `wxID_ANY` to a window constructor, an identifier will be generated for you automatically by wxWidgets. This is useful when you don't care about the exact identifier either because you're not going to process the events from the control being created at all or because you process the events from all controls in one place (in which case you

should specify `wxID_ANY` in the event table or *wxEvtHandler::Connect* (p. 488) call as well. The automatically generated identifiers are always negative and so will never conflict with the user-specified identifiers which must be always positive.

The following standard identifiers are supplied. You can use wxID_HIGHEST to determine the number above which it is safe to define your own identifiers. Or, you can use identifiers below wxID_LOWEST.

```
#define wxID_ANY                -1

#define wxID_LOWEST             4999

#define wxID_OPEN               5000
#define wxID_CLOSE              5001
#define wxID_NEW                5002
#define wxID_SAVE               5003
#define wxID_SAVEAS             5004
#define wxID_REVERT             5005
#define wxID_EXIT               5006
#define wxID_UNDO               5007
#define wxID_REDO               5008
#define wxID_HELP               5009
#define wxID_PRINT              5010
#define wxID_PRINT_SETUP        5011
#define wxID_PREVIEW            5012
#define wxID_ABOUT              5013
#define wxID_HELP_CONTENTS      5014
#define wxID_HELP_COMMANDS      5015
#define wxID_HELP_PROCEDURES    5016
#define wxID_HELP_CONTEXT       5017

#define wxID_CUT                5030
#define wxID_COPY               5031
#define wxID_PASTE              5032
#define wxID_CLEAR              5033
#define wxID_FIND               5034
#define wxID_DUPLICATE          5035
#define wxID_SELECTALL          5036
#define wxID_DELETE             5037
#define wxID_REPLACE            5038
#define wxID_REPLACE_ALL        5039
#define wxID_PROPERTIES         5040

#define wxID_VIEW_DETAILS       5041
#define wxID_VIEW_LARGEICONS    5042
#define wxID_VIEW_SMALLICONS    5043
#define wxID_VIEW_LIST          5044
#define wxID_VIEW_SORTDATE      5045
#define wxID_VIEW_SORTNAME      5046
#define wxID_VIEW_SORTSIZE      5047
#define wxID_VIEW_SORTTYPE      5048

#define wxID_FILE1              5050
#define wxID_FILE2              5051
#define wxID_FILE3              5052
#define wxID_FILE4              5053
#define wxID_FILE5              5054
#define wxID_FILE6              5055
#define wxID_FILE7              5056
#define wxID_FILE8              5057
#define wxID_FILE9              5058
```

```
#define wxID_OK              5100
#define wxID_CANCEL          5101
#define wxID_APPLY           5102
#define wxID_YES             5103
#define wxID_NO              5104
#define wxID_STATIC          5105

#define wxID_HIGHEST         5999
```

Event macros summary

Macros listed by event class

The documentation for specific event macros is organised by event class. Please refer to these sections for details.

wxActivateEvent (p. 30)	The EVT_ACTIVATE and EVT_ACTIVATE_APP macros intercept activation and deactivation events.
wxCommandEvent (p. 180)	A range of commonly-used control events.
wxCloseEvent (p. 153)	The EVT_CLOSE macro handles window closure called via *wxWindow::Close* (p. 1496).
wxDropFilesEvent (p. 466)	The EVT_DROP_FILES macros handles file drop events.
wxEraseEvent (p. 482)	The EVT_ERASE_BACKGROUND macro is used to handle window erase requests.
wxFocusEvent (p. 557)	The EVT_SET_FOCUS and EVT_KILL_FOCUS macros are used to handle keyboard focus events.
wxKeyEvent (p. 830)	EVT_CHAR, EVT_KEY_DOWN and EVT_KEY_UP macros handle keyboard input for any window.
wxIdleEvent (p. 776)	The EVT_IDLE macro handle application idle events (to process background tasks, for example).
wxInitDialogEvent (p. 813)	The EVT_INIT_DIALOG macro is used to handle dialog initialisation.
wxListEvent (p. 871)	These macros handle *wxListCtrl* (p. 852) events.
wxMenuEvent (p. 962)	These macros handle special menu

events (not menu commands).

wxMouseEvent (p. 982)	Mouse event macros can handle either individual mouse events or all mouse events.
wxMoveEvent (p. 992)	The EVT_MOVE macro is used to handle a window move.
wxPaintEvent (p. 1025)	The EVT_PAINT macro is used to handle window paint requests.
wxScrollEvent (p. 1159)	These macros are used to handle scroll events from *wxScrollBar* (p. 1143), *wxSlider* (p. 1192),and *wxSpinButton* (p. 1227).
wxSetCursorEvent (p. 1166)	The EVT_SET_CURSOR macro is used for special cursor processing.
wxSizeEvent (p. 1175)	The EVT_SIZE macro is used to handle a window resize.
wxSplitterEvent (p. 1237)	The EVT_SPLITTER_SASH_POS_CHANGED, EVT_SPLITTER_UNSPLIT and EVT_SPLITTER_DCLICK macros are used to handle the various splitter window events.
wxSysColourChangedEvent (p. 1314)	The EVT_SYS_COLOUR_CHANGED macro is used to handle events informing the application that the user has changed the system colours (Windows only).
wxTreeEvent (p. 1443)	These macros handle *wxTreeCtrl* (p. 1424) events.
wxUpdateUIEvent (p. 1448)	The EVT_UPDATE_UI macro is used to handle user interface update pseudo-events, which are generated to give the application the chance to update the visual state of menus, toolbars and controls.

Custom event summary

General approach

Since version 2.2.x of wxWidgets, each event type is identified by ID which is given to the event type *at runtime* which makes it possible to add new event types to the library or application without risking ID clashes (two different event types mistakingly getting the

same event ID). This event type ID is stored in a struct of type **const wxEventType**.

In order to define a new event type, there are principally two choices. One is to define a entirely new event class (typically deriving from*wxEvent* (p. 483) or *wxCommandEvent* (p. 180). The other is to use the existing event classes and give them an new event type. You'll have to define and declare a new event type using either way, and this is done using the following macros:

```
// in the header of the source file
DECLARE_EVENT_TYPE(name, value)

// in the implementation
DEFINE_EVENT_TYPE(name)
```

You can ignore the *value* parameter of the DECLARE_EVENT_TYPE macro since it used only for backwards compatibility with wxWidgets 2.0.x based applications where you have to give the event type ID an explicit value.

Using existing event classes

If you just want to use a *wxCommandEvent* (p. 180) with a new event type, you can then use one of the generic event table macros listed below, without having to define a new macro yourself. This also has the advantage that you won't have to define a new *wxEvent::Clone()* (p. 484)method for posting events between threads etc. This could look like this in your code:

```
DECLARE_EVENT_TYPE(wxEVT_MY_EVENT, -1)

DEFINE_EVENT_TYPE(wxEVT_MY_EVENT)

// user code intercepting the event

BEGIN_EVENT_TABLE(MyFrame, wxFrame)
  EVT_MENU     (wxID_EXIT, MyFrame::OnExit)
  // ....
  EVT_COMMAND  (ID_MY_WINDOW, wxEVT_MY_EVENT, MyFrame::OnMyEvent)
END_EVENT_TABLE()

void MyFrame::OnMyEvent( wxCommandEvent &event )
{
    // do something
    wxString text = event.GetText();
}


// user code sending the event

void MyWindow::SendEvent()
{
    wxCommandEvent event( wxEVT_MY_EVENT, GetId() );
    event.SetEventObject( this );
    // Give it some contents
    event.SetText( wxT("Hallo") );
    // Send it
    GetEventHandler()->ProcessEvent( event );
}
```

Generic event table macros

EVT_CUSTOM(event, id, func)   Allows you to add a custom event table entry by specifying the event identifier (such as wxEVT_SIZE), the window identifier, and a member function to call.

EVT_CUSTOM_RANGE(event, id1, id2, func)   The same as EVT_CUSTOM, but responds to a range of window identifiers.

EVT_COMMAND(id, event, func)   The same as EVT_CUSTOM, but expects a member function with a wxCommandEvent argument.

EVT_COMMAND_RANGE(id1, id2, event, func)   The same as EVT_CUSTOM_RANGE, but expects a member function with a wxCommandEvent argument.

EVT_NOTIFY(id, event, func)   The same as EVT_CUSTOM, but expects a member function with a wxNotifyEvent argument.

EVT_NOTIFY_RANGE(id1, id2, event, func)   The same as EVT_CUSTOM_RANGE, but expects a member function with a wxNotifyEvent argument.

Defining your own event class

Under certain circumstances, it will be required to define your own event class e.g. for sending more complex data from one place to another. Apart from defining your event class, you will also need to define your own event table macro (which is quite long). Watch out to put in enough casts to the inherited event function. Here is an example, taken mostly from the *wxPlot* library, which is in the *contrib* section of the wxWidgets sources.

```
// code defining event

class wxPlotEvent: public wxNotifyEvent
{
public:
    wxPlotEvent( wxEventType commandType = wxEVT_NULL, int id = 0 );

    // accessors
    wxPlotCurve *GetCurve()
        { return m_curve; }

    // required for sending with wxPostEvent()
    wxEvent* Clone();

private:
    wxPlotCurve    *m_curve;
};

DECLARE_EVENT_MACRO( wxEVT_PLOT_ACTION, -1 )

typedef void (wxEvtHandler::*wxPlotEventFunction)(wxPlotEvent&);
```

```
#define EVT_PLOT(id, fn) \
    DECLARE_EVENT_TABLE_ENTRY( wxEVT_PLOT_ACTION, id, -1, \
    (wxObjectEventFunction) (wxEventFunction) (wxCommandEventFunction)
(wxNotifyEventFunction) \
    wxStaticCastEvent( wxPlotEventFunction, & fn ), (wxObject *) NULL ),


// code implementing the event type and the event class

DEFINE_EVENT_TYPE( wxEVT_PLOT_ACTION )

wxPlotEvent::wxPlotEvent( ...


// user code intercepting the event

BEGIN_EVENT_TABLE(MyFrame, wxFrame)
  EVT_PLOT  (ID_MY_WINDOW,  MyFrame::OnPlot)
END_EVENT_TABLE()

void MyFrame::OnPlot( wxPlotEvent &event )
{
    wxPlotCurve *curve = event.GetCurve();
}


// user code sending the event

void MyWindow::SendEvent()
{
    wxPlotEvent event( wxEVT_PLOT_ACTION, GetId() );
    event.SetEventObject( this );
    event.SetCurve( m_curve );
    GetEventHandler()->ProcessEvent( event );
}
```

C++ exceptions overview

Introduction

wxWidgets had been started long before the exceptions were introduced in C++ so it is not very surprising that it is not built around using them as some more modern C++ libraries are. For instance, the library doesn't throw exceptions to signal about the errors. Moreover, up to (and including) the version 2.4 of wxWidgets, even using the exceptions in the user code was dangerous because the library code wasn't exception-safe and so an exception propagating through it could result in memory and/or resource leaks, and also not very convenient.

Starting from the version 2.5.1 wxWidgets becomes more exception-friendly. It still doesn't use the exceptions by itself but it should be now safe to use the exceptions in the user code and the library tries to help you with this. Please note that making the library exception-safe is still work in progress.

Strategies for exceptions handling

There are several choice for using the exceptions in wxWidgets programs. First of all, you may not use them at all. As stated above, the library doesn't throw any exceptions by itself and so you don't have to worry about exceptions at all unless your own code throws them. This is, of course, the simplest solution but may be not the best one to deal with all possible errors.

Another strategy is to use exceptions only to signal truly fatal errors. In this case you probably don't expect to recover from them and the default behaviour -- to simply terminate the program -- may be appropriate. If it is not, you may override *OnUnhandledException()* (p. 38) in your wxApp-derived class to perform any clean up tasks. Note, however, that any information about the exact exception type is lost when this function is called, so if you need you should override *OnRun()* (p. 38) and add a try/catch clause around the call of the base class version. This would allow you to catch any exceptions generated during the execution of the main event loop. To deal with the exceptions which may arise during the program startup and/or shutdown you should insert try/catch clauses in *OnInit()* (p. 38) and/or *OnExit()* (p. 37) as well.

Finally, you may also want to continue running even when certain exceptions occur. If all of your exceptions may happen only in the event handlers of a single class (or only in the classes derived from it), you may centralize your exception handling code in *ProcessEvent* (p. 491) method of this class. If this is impractical, you may also consider overriding the *wxApp::HandleEvent()* (p. 42) which allows you to handle all the exceptions thrown by any event handler.

Technicalities

To use any kind of exception support in the library you need to build it with `wxUSE_EXCEPTIONS` set to 1. This should be the case by default but if it isn't, you should edit the `include/wx/msw/setup.h` file under Windows or run `configure` with `--enable-exceptions` argument under Unix.

On the other hand, if you do *not* plan to use exceptions, setting this flag to 0 or using `--disable-exceptions` could result in a leaner and slightly faster library.

As for any other library feature, there is a *sample* (p. 1711) showing how to use it. Please look at its sources for further information.

Window styles

Window styles are used to specify alternative behaviour and appearances for windows, when they are created. The symbols are defined in such a way that they can be combined in a 'bit-list' using the C++ *bitwise-or* operator. For example:

```
wxCAPTION | wxMINIMIZE_BOX | wxMAXIMIZE_BOX | wxTHICK_FRAME
```

For the window styles specific to each window class, please see the documentation for the window. Most windows can use the generic styles listed for *wxWindow* (p. 1490) in

addition to their own styles.

Window deletion overview

Classes: *wxCloseEvent* (p. 153), *wxWindow* (p. 1490)

Window deletion can be a confusing subject, so this overview is provided to help make it clear when and how you delete windows, or respond to user requests to close windows.

What is the sequence of events in a window deletion?

When the user clicks on the system close button or system close command, in a frame or a dialog, wxWidgets calls *wxWindow::Close* (p. 1496). This in turn generates an EVT_CLOSE event: see *wxCloseEvent* (p. 153).

It is the duty of the application to define a suitable event handler, and decide whether or not to destroy the window. If the application is for some reason forcing the application to close (*wxCloseEvent::CanVeto* (p. 154) returns false), the window should always be destroyed, otherwise there is the option to ignore the request, or maybe wait until the user has answered a question before deciding whether it is safe to close. The handler for EVT_CLOSE should signal to the calling code if it does not destroy the window, by calling *wxCloseEvent::Veto* (p. 155). Calling this provides useful information to the calling code.

The wxCloseEvent handler should only call *wxWindow::Destroy* (p. 1499) to delete the window, and not use the **delete** operator. This is because for some window classes, wxWidgets delays actual deletion of the window until all events have been processed, since otherwise there is the danger that events will be sent to a non-existent window.

As reinforced in the next section, calling Close does not guarantee that the window will be destroyed. Call *wxWindow::Destroy* (p. 1499) if you want to be certain that the window is destroyed.

How can the application close a window itself?

Your application can either use *wxWindow::Close* (p. 1496) event just as the framework does, or it can call *wxWindow::Destroy* (p. 1499) directly. If using Close(), you can pass a true argument to this function to tell the event handler that we definitely want to delete the frame and it cannot be vetoed.

The advantage of using Close instead of Destroy is that it will call any clean-up code defined by the EVT_CLOSE handler; for example it may close a document contained in a window after first asking the user whether the work should be saved. Close can be vetoed by this process (return false), whereas Destroy definitely destroys the window.

What is the default behaviour?

The default close event handler for wxDialog simulates a Cancel command, generating a wxID_CANCEL event. Since the handler for this cancel event might itself call **Close**, there is a check for infinite looping. The default handler for wxID_CANCEL hides the dialog (if modeless) or calls EndModal(wxID_CANCEL) (if modal). In other words, by default, the dialog *is not destroyed* (it might have been created on the stack, so the

assumption of dynamic creation cannot be made).

The default close event handler for wxFrame destroys the frame using Destroy().

What should I do when the user calls up Exit from a menu?

You can simply call *wxWindow::Close* (p. 1496) on the frame. This will invoke your own close event handler which may destroy the frame.

You can do checking to see if your application can be safely exited at this point, either from within your close event handler, or from within your exit menu command handler. For example, you may wish to check that all files have been saved. Give the user a chance to save and quit, to not save but quit anyway, or to cancel the exit command altogether.

What should I do to upgrade my 1.xx OnClose to 2.0?

In wxWidgets 1.xx, the **OnClose** function did not actually delete 'this', but signaled to the calling function (either **Close**, or the wxWidgets framework) to delete or not delete the window.

To update your code, you should provide an event table entry in your frame or dialog, using the EVT_CLOSE macro. The event handler function might look like this:

```
void MyFrame::OnCloseWindow(wxCloseEvent& event)
{
  if (MyDataHasBeenModified())
  {
    wxMessageDialog* dialog = new wxMessageDialog(this,
      "Save changed data?", "My app", wxYES_NO|wxCANCEL);

    int ans = dialog->ShowModal();
    dialog->Destroy();

    switch (ans)
    {
      case wxID_YES:      // Save, then destroy, quitting app
        SaveMyData();
        this->Destroy();
        break;
      case wxID_NO:       // Don't save; just destroy, quitting app
        this->Destroy();
        break;
      case wxID_CANCEL:   // Do nothing - so don't quit app.
      default:
        if (!event.CanVeto()) // Test if we can veto this deletion
          this->Destroy();    // If not, destroy the window anyway.
        else
          event.Veto();     // Notify the calling code that we didn't
delete the frame.
        break;
    }
  }
}
```

How do I exit the application gracefully?

A wxWidgets application automatically exits when the designated top window, or the last

frame or dialog, is destroyed. Put any application-wide cleanup code in *wxApp::OnExit* (p. 37) (this is a virtual function, not an event handler).

Do child windows get deleted automatically?

Yes, child windows are deleted from within the parent destructor. This includes any children that are themselves frames or dialogs, so you may wish to close these child frame or dialog windows explicitly from within the parent close handler.

What about other kinds of window?

So far we've been talking about 'managed' windows, i.e. frames and dialogs. Windows with parents, such as controls, don't have delayed destruction and don't usually have close event handlers, though you can implement them if you wish. For consistency, continue to use the *wxWindow::Destroy* (p. 1499) function instead of the **delete** operator when deleting these kinds of windows explicitly.

wxDialog overview

Classes: *wxDialog* (p. 406)

A dialog box is similar to a panel, in that it is a window which can be used for placing controls, with the following exceptions:

1. A surrounding frame is implicitly created.

2. Extra functionality is automatically given to the dialog box, such as tabbing between items (currently Windows only).

3. If the dialog box is *modal*, the calling program is blocked until the dialog box is dismissed.

Under Windows 3, modal dialogs have to be emulated using modeless dialogs and a message loop. This is because Windows 3 expects the contents of a modal dialog to be loaded from a resource file or created on receipt of a dialog initialization message. This is too restrictive for wxWidgets, where any window may be created and displayed before its contents are created.

For a set of dialog convenience functions, including file selection, see *Dialog functions* (p. 1613).

See also *wxPanel* (p. 1030) and *wxWindow* (p. 1490) for inherited member functions. Validation of data in controls is covered in *Validator overview* (p. 1765).

wxValidator overview

Classes: *wxValidator* (p. 1461), *wxTextValidator* (p. 1374), *wxGenericValidator* (p. 612)

The aim of the validator concept is to make dialogs very much easier to write. A validator is an object that can be plugged into a control (such as a wxTextCtrl), and mediates

between C++ data and the control, transferring the data in either direction and validating it. It also is able to intercept events generated by the control, providing filtering behaviour without the need to derive a new control class.

You can use a stock validator, such as *wxTextValidator* (p. 1374) (which does text control data transfer, validation and filtering) and *wxGenericValidator* (p. 612) (which does data transfer for a range of controls); or you can write your own.

Example

Here is an example of wxTextValidator usage.

```
wxTextCtrl *txt1 = new wxTextCtrl(this, -1, wxT(""),
  wxPoint(10, 10), wxSize(100, 80), 0,
  wxTextValidator(wxFILTER_ALPHA, &g_data.m_string));
```

In this example, the text validator object provides the following functionality:

1. It transfers the value of g_data.m_string (a wxString variable) to the wxTextCtrl when the dialog is initialised.

2. It transfers the wxTextCtrl data back to this variable when the dialog is dismissed.

3. It filters input characters so that only alphabetic characters are allowed.

The validation and filtering of input is accomplished in two ways. When a character is input, wxTextValidator checks the character against the allowed filter flag (wxFILTER_ALPHA in this case). If the character is inappropriate, it is vetoed (does not appear) and a warning beep sounds. The second type of validation is performed when the dialog is about to be dismissed, so if the default string contained invalid characters already, a dialog box is shown giving the error, and the dialog is not dismissed.

Anatomy of a validator

A programmer creating a new validator class should provide the following functionality.

A validator constructor is responsible for allowing the programmer to specify the kind of validation required, and perhaps a pointer to a C++ variable that is used for storing the data for the control. If such a variable address is not supplied by the user, then the validator should store the data internally.

The *wxValidator::Validate* (p. 1463) member function should return true if the data in the control (not the C++ variable) is valid. It should also show an appropriate message if data was not valid.

The *wxValidator::TransferToWindow* (p. 1463) member function should transfer the data from the validator or associated C++ variable to the control.

The *wxValidator::TransferFromWindow* (p. 1463) member function should transfer the data from the control to the validator or associated C++ variable.

There should be a copy constructor, and a *wxValidator::Clone* (p. 1462) function which returns a copy of the validator object. This is important because validators are passed by

reference to window constructors, and must therefore be cloned internally.

You can optionally define event handlers for the validator, to implement filtering. These handlers will capture events before the control itself does.

For an example implementation, see the valtext.h and valtext.cpp files in the wxWidgets library.

How validators interact with dialogs

For validators to work correctly, validator functions must be called at the right times during dialog initialisation and dismissal.

When a *wxDialog::Show* (p. 415) is called (for a modeless dialog) or *wxDialog::ShowModal* (p. 415) is called (for a modal dialog), the function *wxWindow::InitDialog* (p. 1516) is automatically called. This in turn sends an initialisation event to the dialog. The default handler for the wxEVT_INIT_DIALOG event is defined in the wxWindow class to simply call the function *wxWindow::TransferDataToWindow* (p. 1545). This function finds all the validators in the window's children and calls the TransferToWindow function for each. Thus, data is transferred from C++ variables to the dialog just as the dialog is being shown.

> If you are using a window or panel instead of a dialog, you will need to call *wxWindow::InitDialog* (p. 1516) explicitly before showing the window.

When the user clicks on a button, for example the OK button, the application should first call *wxWindow::Validate* (p. 1547), which returns false if any of the child window validators failed to validate the window data. The button handler should return immediately if validation failed. Secondly, the application should call *wxWindow::TransferDataFromWindow* (p. 1545) and return if this failed. It is then safe to end the dialog by calling EndModal (if modal) or Show (if modeless).

In fact, wxDialog contains a default command event handler for the wxID_OK button. It goes like this:

```
void wxDialog::OnOK(wxCommandEvent& event)
{
    if ( Validate() && TransferDataFromWindow() )
    {
        if ( IsModal() )
            EndModal(wxID_OK);
        else
        {
            SetReturnCode(wxID_OK);
            this->Show(false);
        }
    }
}
```

So if using validators and a normal OK button, you may not even need to write any code for handling dialog dismissal.

If you load your dialog from a resource file, you will need to iterate through the controls setting validators, since validators can't be specified in a dialog resource.

Constraints overview

Classes: *wxLayoutConstraints* (p. 837), *wxIndividualLayoutConstraint* (p. 811).

Note: constraints are now deprecated and you should use *sizers* (p. 1771) instead.

Objects of class wxLayoutConstraint can be associated with a window to define the way it is laid out, with respect to its siblings or the parent.

The class consists of the following eight constraints of class wxIndividualLayoutConstraint, some or all of which should be accessed directly to set the appropriate constraints.

- **left:** represents the left hand edge of the window

- **right:** represents the right hand edge of the window

- **top:** represents the top edge of the window

- **bottom:** represents the bottom edge of the window

- **width:** represents the width of the window

- **height:** represents the height of the window

- **centreX:** represents the horizontal centre point of the window

- **centreY:** represents the vertical centre point of the window

The constraints are initially set to have the relationship wxUnconstrained, which means that their values should be calculated by looking at known constraints. To calculate the position and size of the control, the layout algorithm needs to know exactly 4 constraints (as it has 4 numbers to calculate from them), so you should always set exactly 4 of the constraints from the above table.

If you want the controls height or width to have the default value, you may use a special value for the constraint: wxAsIs. If the constraint is wxAsIs, the dimension will not be changed which is useful for the dialog controls which often have the default size (e.g. the buttons whose size is determined by their label).

The constrains calculation is done in *wxWindow::Layout* (p. 1517) function which evaluates constraints. To call it you can either call*wxWindow::SetAutoLayout* (p. 1527) if the parent window is a frame, panel or a dialog to tell default OnSize handlers to call Layout automatically whenever the window size changes, or override OnSize and call Layout yourself (note that you do have to call *Layout* (p. 1517) yourself if the parent window is not a frame, panel or dialog).

Constraint layout: more details

By default, windows do not have a wxLayoutConstraints object. In this case, much layout must be done explicitly, by performing calculations in OnSize members, except for the case of frames that have exactly one subwindow (not counting toolbar and statusbar

which are also positioned by the frame automatically), where wxFrame::OnSize takes care of resizing the child to always fill the frame.

To avoid the need for these rather awkward calculations, the user can create a wxLayoutConstraints object and associate it with a window with wxWindow::SetConstraints. This object contains a constraint for each of the window edges, two for the centre point, and two for the window size. By setting some or all of these constraints appropriately, the user can achieve quite complex layout by defining relationships between windows.

In wxWidgets, each window can be constrained relative to either its *siblings* on the same window, or the *parent*. The layout algorithm therefore operates in a top-down manner, finding the correct layout for the children of a window, then the layout for the grandchildren, and so on. Note that this differs markedly from native Motif layout, where constraints can ripple upwards and can eventually change the frame window or dialog box size. We assume in wxWidgets that the *user* is always 'boss' and specifies the size of the outer window, to which subwindows must conform. Obviously, this might be a limitation in some circumstances, but it suffices for most situations, and the simplification avoids some of the nightmarish problems associated with programming Motif.

When the user sets constraints, many of the constraints for windows edges and dimensions remain unconstrained. For a given window, the wxWindow::Layout algorithm first resets all constraints in all children to have unknown edge or dimension values, and then iterates through the constraints, evaluating them. For unconstrained edges and dimensions, it tries to find the value using known relationships that always hold. For example, an unconstrained *width* may be calculated from the *left* and *right edges*, if both are currently known. For edges and dimensions with user-supplied constraints, these constraints are evaluated if the inputs of the constraint are known.

The algorithm stops when all child edges and dimension are known (success), or there are unknown edges or dimensions but there has been no change in this cycle (failure).

It then sets all the window positions and sizes according to the values it has found.

Because the algorithm is iterative, the order in which constraints are considered is irrelevant, however you may reduce the number of iterations (and thus speed up the layout calculations) by creating the controls in such order that as many constraints as possible can be calculated during the first iteration. For example, if you have 2 buttons which you'd like to position in the lower right corner, it is slightly more efficient to first create the second button and specify that its right border IsSameAs(parent, wxRight) and then create the first one by specifying that it should be LeftOf() the second one than to do in a more natural left-to-right order.

Window layout examples

Example 1: subwindow layout

This example specifies a panel and a window side by side, with a text subwindow below it.

```
    frame->panel = new wxPanel(frame, -1, wxPoint(0, 0), wxSize(1000, 500),
0);
```

```
   frame->scrollWindow = new MyScrolledWindow(frame, -1, wxPoint(0, 0),
wxSize(400, 400), wxRETAINED);
   frame->text_window = new MyTextWindow(frame, -1, wxPoint(0, 250),
wxSize(400, 250));

   // Set constraints for panel subwindow
   wxLayoutConstraints *c1 = new wxLayoutConstraints;

   c1->left.SameAs        (frame, wxLeft);
   c1->top.SameAs         (frame, wxTop);
   c1->right.PercentOf    (frame, wxWidth, 50);
   c1->height.PercentOf   (frame, wxHeight, 50);

   frame->panel->SetConstraints(c1);

   // Set constraints for scrollWindow subwindow
   wxLayoutConstraints *c2 = new wxLayoutConstraints;

   c2->left.SameAs        (frame->panel, wxRight);
   c2->top.SameAs         (frame, wxTop);
   c2->right.SameAs       (frame, wxRight);
   c2->height.PercentOf   (frame, wxHeight, 50);

   frame->scrollWindow->SetConstraints(c2);

   // Set constraints for text subwindow
   wxLayoutConstraints *c3 = new wxLayoutConstraints;
   c3->left.SameAs        (frame, wxLeft);
   c3->top.Below          (frame->panel);
   c3->right.SameAs       (frame, wxRight);
   c3->bottom.SameAs      (frame, wxBottom);

   frame->text_window->SetConstraints(c3);
```

Example 2: panel item layout

This example sizes a button width to 80 percent of the panel width, and centres it horizontally. A listbox and multitext item are placed below it. The listbox takes up 40 percent of the panel width, and the multitext item takes up the remainder of the width. Margins of 5 pixels are used.

```
       // Create some panel items
       wxButton *btn1 = new wxButton(frame->panel, -1, "A button") ;

       wxLayoutConstraints *b1 = new wxLayoutConstraints;
       b1->centreX.SameAs    (frame->panel, wxCentreX);
       b1->top.SameAs        (frame->panel, wxTop, 5);
       b1->width.PercentOf   (frame->panel, wxWidth, 80);
       b1->height.PercentOf  (frame->panel, wxHeight, 10);
       btn1->SetConstraints(b1);

       wxListBox *list = new wxListBox(frame->panel, -1, "A list",
                                  wxPoint(-1, -1), wxSize(200, 100));

       wxLayoutConstraints *b2 = new wxLayoutConstraints;
       b2->top.Below         (btn1, 5);
       b2->left.SameAs       (frame->panel, wxLeft, 5);
       b2->width.PercentOf   (frame->panel, wxWidth, 40);
       b2->bottom.SameAs     (frame->panel, wxBottom, 5);
       list->SetConstraints(b2);
```

```
   wxTextCtrl *mtext = new wxTextCtrl(frame->panel, -1, "Multiline text",
"Some text",
                       wxPoint(-1, -1), wxSize(150, 100),
wxTE_MULTILINE);

   wxLayoutConstraints *b3 = new wxLayoutConstraints;
   b3->top.Below        (btn1, 5);
   b3->left.RightOf     (list, 5);
   b3->right.SameAs     (frame->panel, wxRight, 5);
   b3->bottom.SameAs    (frame->panel, wxBottom, 5);
   mtext->SetConstraints(b3);
```

Sizer overview

Classes: *wxSizer* (p. 1176), *wxGridSizer* (p. 683), *wxFlexGridSizer* (p. 554), *wxBoxSizer* (p. 102), *wxStaticBoxSizer* (p. 1262), *CreateButtonSizer* (p. 1776)

Sizers, as represented by the wxSizer class and its descendants in the wxWidgets class hierarchy, have become the method of choice to define the layout of controls in dialogs in wxWidgets because of their ability to create visually appealing dialogs independent of the platform, taking into account the differences in size and style of the individual controls. Unlike the original wxWidgets Dialog Editor, editors such as wxDesigner, DialogBlocks, wxrcedit, XRCed and wxWorkshop create dialogs based exclusively on sizers, practically forcing the user to create platform independent layouts without compromises.

The next section describes and shows what can be done with sizers. The following sections briefly describe how to program with individual sizer classes.

For information about the new wxWidgets resource system, which can describe sizer-based dialogs, see the *XML-based resource system overview* (p. 1777).

The idea behind sizers

The layout algorithm used by sizers in wxWidgets is closely related to layout systems in other GUI toolkits, such as Java's AWT, the GTK toolkit or the Qt toolkit. It is based upon the idea of individual subwindows reporting their minimal required size and their ability to get stretched if the size of the parent window has changed. This will most often mean that the programmer does not set the start-up size of a dialog, the dialog will rather be assigned a sizer and this sizer will be queried about the recommended size. This sizer in turn will query its children (which can be normal windows, empty space or other sizers) so that a hierarchy of sizers can be constructed. Note that wxSizer does not derive from wxWindow and thus does not interfere with tab ordering and requires very few resources compared to a real window on screen.

What makes sizers so well fitted for use in wxWidgets is the fact that every control reports its own minimal size and the algorithm can handle differences in font sizes or different window (dialog item) sizes on different platforms without problems. For example, if the standard font as well as the overall design of Linux/GTK widgets requires more space than on Windows, the initial dialog size will automatically be bigger on Linux/GTK than on Windows.

There are currently five different kinds of sizers available in wxWidgets. Each represents either a certain way to lay out dialog items in a dialog or it fulfills a special task such as wrapping a static box around a dialog item (or another sizer). These sizers will be discussed one by one in the text below. For more detailed information on how to use sizers programmatically, please refer to the section *Programming with Sizers* (p. 1774).

Common features

All sizers are containers, that is, they are used to lay out one dialog item (or several dialog items), which they contain. Such items are sometimes referred to as the children of the sizer. Independent of how the individual sizers lay out their children, all children have certain features in common:

A minimal size: This minimal size is usually identical to the initial size of the controls and may either be set explicitly in the wxSize field of the control constructor or may be calculated by wxWidgets, typically by setting the height and/or the width of the item to -1. Note that only some controls can calculate their size (such as a checkbox) whereas others (such as a listbox) don't have any natural width or height and thus require an explicit size. Some controls can calculate their height, but not their width (e.g. a single line text control):

PRESS F9 TO FORMAT PICTURE

PRESS F9 TO FORMAT PICTURE

PRESS F9 TO FORMAT PICTURE

A border: The border is just empty space and is used to separate dialog items in a dialog. This border can either be all around, or at any combination of sides such as only above and below the control. The thickness of this border must be set explicitly, typically 5 points. The following samples show dialogs with only one dialog item (a button) and a border of 0, 5, and 10 pixels around the button:

PRESS F9 TO FORMAT PICTURE

PRESS F9 TO FORMAT PICTURE

PRESS F9 TO FORMAT PICTURE

An alignment: Often, a dialog item is given more space than its minimal size plus its border. Depending on what flags are used for the respective dialog item, the dialog item can be made to fill out the available space entirely, i.e. it will grow to a size larger than the minimal size, or it will be moved to either the centre of the available space or to either side of the space. The following sample shows a listbox and three buttons in a horizontal box sizer; one button is centred, one is aligned at the top, one is aligned at the bottom:

PRESS F9 TO FORMAT PICTURE

A stretch factor: If a sizer contains more than one child and it is offered more space than its children and their borders need, the question arises how to distribute the surplus space among the children. For this purpose, a stretch factor may be assigned to each child, where the default value of 0 indicates that the child will not get more space than its

requested minimum size. A value of more than zero is interpreted in relation to the sum of all stretch factors in the children of the respective sizer, i.e. if two children get a stretch factor of 1, they will get half the extra space each *independent of whether one control has a minimal sizer inferior to the other or not*. The following sample shows a dialog with three buttons, the first one has a stretch factor of 1 and thus gets stretched, whereas the other two buttons have a stretch factor of zero and keep their initial width:

PRESS F9 TO FORMAT PICTURE

Within wxDesigner, this stretch factor gets set from the *Option* menu.

Hiding controls using sizers

You can hide controls contained in sizers the same way you would hide any control, using the *wxWindow::Show* (p. 1544) method.

However, wxSizer also offers a separate method which can tell the sizer not to consider that control in its size calculations. To hide a window using the sizer, call *wxSizer::Show* (p. 1185). You must then call Layout on the sizer to force an update.

This is useful when hiding parts of the interface, since you can avoid removing the controls from the sizer and having to add them back later.

Note: This is supported only by wxBoxSizer and wxFlexGridSizer.

wxBoxSizer

wxBoxSizer (p. 102) can lay out its children either vertically or horizontally, depending on what flag is being used in its constructor. When using a vertical sizer, each child can be centered, aligned to the right or aligned to the left. Correspondingly, when using a horizontal sizer, each child can be centered, aligned at the bottom or aligned at the top. The stretch factor described in the last paragraph is used for the main orientation, i.e. when using a horizontal box sizer, the stretch factor determines how much the child can be stretched horizontally. The following sample shows the same dialog as in the last sample, only the box sizer is a vertical box sizer now:

PRESS F9 TO FORMAT PICTURE

wxStaticBoxSizer

wxStaticBoxSixer (p. 1262) is the same as a wxBoxSizer, but surrounded by a static box. Here is a sample:

PRESS F9 TO FORMAT PICTURE

wxGridSizer

wxGridSizer (p. 683) is a two-dimensional sizer. All children are given the same size, which is the minimal size required by the biggest child, in this case the text control in the left bottom border. Either the number of columns or the number or rows is fixed and the grid sizer will grow in the respectively other orientation if new children are added:

PRESS F9 TO FORMAT PICTURE

For programming information, see *wxGridSizer* (p. 683).

wxFlexGridSizer

Another two-dimensional sizer derived from wxGridSizer. The width of each column and the height of each row are calculated individually according to the minimal requirements from the respectively biggest child. Additionally, columns and rows can be declared to be stretchable if the sizer is assigned a size different from the one it requested. The following sample shows the same dialog as the one above, but using a flex grid sizer:

PRESS F9 TO FORMAT PICTURE

Programming with wxBoxSizer

The basic idea behind a *wxBoxSizer* (p. 102) is that windows will most often be laid out in rather simple basic geometry, typically in a row or a column or several hierarchies of either.

As an example, we will construct a dialog that will contain a text field at the top and two buttons at the bottom. This can be seen as a top-hierarchy column with the text at the top and buttons at the bottom and a low-hierarchy row with an OK button to the left and a Cancel button to the right. In many cases (particularly dialogs under Unix and normal frames) the main window will be resizable by the user and this change of size will have to get propagated to its children. In our case, we want the text area to grow with the dialog, whereas the button shall have a fixed size. In addition, there will be a thin border around all controls to make the dialog look nice and - to make matter worse - the buttons shall be centred as the width of the dialog changes.

It is the unique feature of a box sizer, that it can grow in both directions (height and width) but can distribute its growth in the main direction (horizontal for a row) *unevenly* among its children. In our example case, the vertical sizer is supposed to propagate all its height changes to only the text area, not to the button area. This is determined by the *proportion* parameter when adding a window (or another sizer) to a sizer. It is interpreted as a weight factor, i.e. it can be zero, indicating that the window may not be resized at all, or above zero. If several windows have a value above zero, the value is interpreted relative to the sum of all weight factors of the sizer, so when adding two windows with a value of 1, they will both get resized equally much and each half as much as the sizer owning them. Then what do we do when a column sizer changes its width? This behaviour is controlled by *flags* (the second parameter of the Add() function): Zero or no flag indicates that the window will preserve it is original size, wxGROW flag (same as wxEXPAND) forces the window to grow with the sizer, and wxSHAPED flag tells the window to change it is size proportionally, preserving original aspect ratio. When wxGROW flag is not used, the item can be aligned within available space. wxALIGN_LEFT, wxALIGN_TOP, wxALIGN_RIGHT, wxALIGN_BOTTOM, wxALIGN_CENTER_HORIZONTAL and wxALIGN_CENTER_VERTICAL do what they say. wxALIGN_CENTRE (same as wxALIGN_CENTER) is defined as (wxALIGN_CENTER_HORIZONTAL | wxALIGN_CENTER_VERTICAL). Default alignment is wxALIGN_LEFT | wxALIGN_TOP.

As mentioned above, any window belonging to a sizer may have border, and it can be specified which of the four sides may have this border, using the wxTOP, wxLEFT,

wxRIGHT and wxBOTTOM constants or wxALL for all directions (and you may also use wxNORTH, wxWEST etc instead). These flags can be used in combination with the alignment flags above as the second parameter of the Add() method using the binary or operator |. The sizer of the border also must be made known, and it is the third parameter in the Add() method. This means, that the entire behaviour of a sizer and its children can be controlled by the three parameters of the Add() method.

```
      // we want to get a dialog that is stretchable because it
      // has a text ctrl at the top and two buttons at the bottom

      MyDialog::MyDialog(wxFrame *parent, wxWindowID id, const wxString &title )
              : wxDialog(parent, id, title, wxDefaultPosition, wxDefaultSize,
                         wxDEFAULT_DIALOG_STYLE | wxRESIZE_BORDER)
      {
        wxBoxSizer *topsizer = new wxBoxSizer( wxVERTICAL );

        // create text ctrl with minimal size 100x60
        topsizer->Add(
          new wxTextCtrl( this, -1, "My text.", wxDefaultPosition,
      wxSize(100,60), wxTE_MULTILINE),
          1,              // make vertically stretchable
          wxEXPAND |     // make horizontally stretchable
          wxALL,         //   and make border all around
          10 );          // set border width to 10


        wxBoxSizer *button_sizer = new wxBoxSizer( wxHORIZONTAL );
        button_sizer->Add(
          new wxButton( this, wxID_OK, "OK" ),
          0,            // make horizontally unstretchable
          wxALL,        // make border all around (implicit top alignment)
          10 );         // set border width to 10
        button_sizer->Add(
          new wxButton( this, wxID_CANCEL, "Cancel" ),
          0,            // make horizontally unstretchable
          wxALL,        // make border all around (implicit top alignment)
          10 );         // set border width to 10

        topsizer->Add(
          button_sizer,
          0,                  // make vertically unstretchable
          wxALIGN_CENTER ); // no border and centre horizontally

        SetSizer( topsizer );      // use the sizer for layout

        topsizer->SetSizeHints( this );  // set size hints to honour minimum
      size
      }
```

Note that the new way of specifying flags to wxSizer is via *wxSizerFlags* (p. 1185). This class greatly eases the burden of passing flags to a wxSizer.

Here's how you'd do the previous example with wxSizerFlags:

```
      // we want to get a dialog that is stretchable because it
      // has a text ctrl at the top and two buttons at the bottom

      MyDialog::MyDialog(wxFrame *parent, wxWindowID id, const wxString &title )
              : wxDialog(parent, id, title, wxDefaultPosition, wxDefaultSize,
                         wxDEFAULT_DIALOG_STYLE | wxRESIZE_BORDER)
      {
```

```
        wxBoxSizer *topsizer = new wxBoxSizer( wxVERTICAL );

        // create text ctrl with minimal size 100x60 that is horizontally and
        // vertically stretchable with a border width of 10
        topsizer->Add(
          new wxTextCtrl( this, -1, "My text.", wxDefaultPosition,
    wxSize(100,60), wxTE_MULTILINE),
          wxSizerFlags(1).Align().Expand().Border(wxALL, 10));

        wxBoxSizer *button_sizer = new wxBoxSizer( wxHORIZONTAL );

        //create two buttons that are horizontally unstretchable,
        // with an all-around border with a width of 10 and implicit top
    alignment
      button_sizer->Add(
          new wxButton( this, wxID_OK, "OK" ),
          wxSizerFlags(0).Align().Border(wxALL, 10));

      button_sizer->Add(
          new wxButton( this, wxID_CANCEL, "Cancel" ),
          wxSizerFlags(0).Align().Border(wxALL, 10));

      //create a sizer with no border and centered horizontally
      topsizer->Add(
          button_sizer,
          wxSizerFlags(0).Center() );

      SetSizer( topsizer );        // use the sizer for layout

      topsizer->SetSizeHints( this );   // set size hints to honour minimum
    size
    }
```

Programming with wxGridSizer

wxGridSizer (p. 683) is a sizer which lays out its children in a two-dimensional table with all table fields having the same size, i.e. the width of each field is the width of the widest child, the height of each field is the height of the tallest child.

Programming with wxFlexGridSizer

wxFlexGridSizer (p. 554) is a sizer which lays out its children in a two-dimensional table with all table fields in one row having the same height and all fields in one column having the same width, but all rows or all columns are not necessarily the same height or width as in the *wxGridSizer* (p. 683).

Programming with wxStaticBoxSizer

wxStaticBoxSizer (p. 1262) is a sizer derived from wxBoxSizer but adds a static box around the sizer. Note that this static box has to be created separately.

CreateButtonSizer

As a convenience, CreateButtonSizer (long flags) can be used to create a standard button sizer in which standard buttons are displayed. The following flags can be passed to this function:

```
wxYES_NO // Add Yes/No subpanel
wxYES    // return wxID_YES
wxNO     // return wxID_NO
wxNO_DEFAULT // make the wxNO button the default, otherwise wxYES or
wxOK button will be default

wxOK     // return wxID_OK
wxCANCEL // return wxID_CANCEL
wxHELP   // return wxID_HELP

wxFORWARD   // return wxID_FORWARD
wxBACKWARD  // return wxID_BACKWARD
wxSETUP     // return wxID_SETUP
wxMORE      // return wxID_MORE
```

XML-based resource system overview

Classes: *wxXmlResource* (p. 1558), *wxXmlResourceHandler* (p. 1563)

The XML-based resource system, known as XRC, allows user interface elements such as dialogs, menu bars and toolbars, to be stored in text files and loaded into the application at run-time. XRC files can also be compiled into binary XRS files or C++ code (the former makes it possible to store all resources in a single file and the latter is useful when you want to embed the resources into the executable).

There are several advantages to using XRC resources.

- Recompiling and linking an application is not necessary if the resources change.

- If you use a dialog designer that generates C++ code, it can be hard to reintegrate this into existing C++ code. Separation of resources and code is a more elegant solution.

- You can choose between different alternative resource files at run time, if necessary.

- The XRC format uses sizers for flexibility, allowing dialogs to be resizable and highly portable.

- The XRC format is a wxWidgets standard, and can be generated or postprocessed by any program that understands it. As it is based on the XML standard, existing XML editors can be used for simple editing purposes.

XRC was written by Vaclav Slavik.

XRC concepts

These are the typical steps for using XRC files in your application.

- Include the appropriate headers: normally "wx/xrc/xmlres.h" will suffice;

- If you are going to use *XRS files* (p. 1778), install wxFileSystem ZIP handler first with `wxFileSystem::AddHandler(new wxZipFSHandler);`

- call `wxXmlResource::Get()->InitAllHandlers()` from your wxApp::OnInit function, and then call `wxXmlResource::Get()->Load("myfile.xrc")` to load the resource file;

- to create a dialog from a resource, create it using the default constructor, and then load it using for example `wxXmlResource::Get()->LoadDialog(&dlg, this, "dlg1");`

- set up event tables as usual but use the `XRCID(str)` macro to translate from XRC string names to a suitable integer identifier, for example `EVT_MENU(XRCID("quit"), MyFrame::OnQuit).`

To create an XRC file, you can use one of the following methods.

- Create the file by hand;

- use wxDesigner (`http://www.roebling.de`), a commercial dialog designer/RAD tool;

- use DialogBlocks (`http://www.anthemion.co.uk/dialogblocks`), a commercial dialog editor;

- use XRCed (`http://xrced.sf.net`), a wxPython-based dialog editor that you can find in the `wxPython/tools` subdirectory of the wxWidgets CVS archive;

- use wxGlade (`http://wxglade.sf.net`), a GUI designer written in wxPython. At the moment it can generate Python, C++ and XRC;

- convert WIN32 RC files to XRC with the tool in `contrib/utils/convertrc`.

A complete list of third-party tools that write to XRC can be found at www.wxwidgets.org/lnk_tool.htm (`http://www.wxwidgets.org/lnk_tool.in`).

It is highly recommended that you use a resource editing tool, since it's fiddly writing XRC files by hand.

You can use *wxXmlResource::Load* (p. 1561) in a number of ways. You can pass an XRC file (XML-based text resource file) or a *zip-compressed file* (p. 1778) (extension ZIP or XRS) containing other XRC.

You can also use *embedded C++ resources* (p. 1779)

Using binary resource files

To compile binary resource files, use the command-line wxrc utility. It takes one or more file parameters (the input XRC files) and the following switches and options:

- -h (--help): show a help message

- -v (--verbose): show verbose logging information

- -c (--cpp-code): write C++ source rather than a XRS file

- -e (--extra-cpp-code): if used together with -c, generates C++ header file containing class definitions for the windows defined by the XRC file (see special subsection)

- -u (--uncompressed): do not compress XML files (C++ only)

- -g (--gettext): output underscore-wrapped strings that poEdit or gettext can scan. Outputs to stdout, or a file if -o is used

- -n (--function) <name>: specify C++ function name (use with -c)

- -o (--output) <filename>: specify the output file, such as resource.xrs or resource.cpp

- -l (--list-of-handlers) <filename>: output a list of necessary handlers to this file

```
For example:  % wxrc resource.xrc
  % wxrc resource.xrc -o resource.xrs
  % wxrc resource.xrc -v -c -o resource.cpp
```

Note

XRS file is essentially a renamed ZIP archive which means that you can manipulate it with standard ZIP tools. Note that if you are using XRS files, you have to initialize the *wxFileSystem* (p. 540) ZIP handler first! It is a simple thing to do:

```
#include <wx/filesys.h>
#include <wx/fs_zip.h>
...
wxFileSystem::AddHandler(new wxZipFSHandler);
```

Using embedded resources

It is sometimes useful to embed resources in the executable itself instead of loading an external file (e.g. when your app is small and consists only of one exe file). XRC provides means to convert resources into regular C++ file that can be compiled and included in the executable.

Use the -c switch to wxrc utility to produce C++ file with embedded resources. This file will contain a function called *InitXmlResource* (unless you override this with a command line switch). Use it to load the resource:

```
extern void InitXmlResource(); // defined in generated file
...
wxXmlResource::Get()->InitAllHandlers();
```

```
    InitXmlResource();
    ...
```

XRC C++ sample

This is the C++ source file (xrcdemo.cpp) for the XRC sample.

```cpp
#include "wx/wx.h"
#include "wx/image.h"
#include "wx/xrc/xmlres.h"

// the application icon
#if defined(__WXGTK__) || defined(__WXMOTIF__) || defined(__WXMAC__)
    #include "rc/appicon.xpm"
#endif

// ---------------------------------------------------------------------------
-----
// private classes
// ---------------------------------------------------------------------------
-----

// Define a new application type, each program should derive a class from
wxApp
class MyApp : public wxApp
{
public:
    // override base class virtuals
    // ----------------------------

    // this one is called on application startup and is a good place for
the app
    // initialization (doing it here and not in the ctor allows to have an
error
    // return: if OnInit() returns false, the application terminates)
    virtual bool OnInit();
};

// Define a new frame type: this is going to be our main frame
class MyFrame : public wxFrame
{
public:
    // ctor(s)
    MyFrame(const wxString& title, const wxPoint& pos, const wxSize&
size);

    // event handlers (these functions should _not_ be virtual)
    void OnQuit(wxCommandEvent& event);
    void OnAbout(wxCommandEvent& event);
    void OnDlg1(wxCommandEvent& event);
    void OnDlg2(wxCommandEvent& event);

private:
    // any class wishing to process wxWidgets events must use this macro
    DECLARE_EVENT_TABLE()
};

// ---------------------------------------------------------------------------
-----
// event tables and other macros for wxWidgets
```

```
// -------------------------------------------------------------------------
-----

BEGIN_EVENT_TABLE(MyFrame, wxFrame)
    EVT_MENU(XRCID("menu_quit"),  MyFrame::OnQuit)
    EVT_MENU(XRCID("menu_about"), MyFrame::OnAbout)
    EVT_MENU(XRCID("menu_dlg1"), MyFrame::OnDlg1)
    EVT_MENU(XRCID("menu_dlg2"), MyFrame::OnDlg2)
END_EVENT_TABLE()

IMPLEMENT_APP(MyApp)

// -------------------------------------------------------------------------
-----
// the application class
// -------------------------------------------------------------------------
-----

// 'Main program' equivalent: the program execution "starts" here
bool MyApp::OnInit()
{
    wxImage::AddHandler(new wxGIFHandler);
    wxXmlResource::Get()->InitAllHandlers();
    wxXmlResource::Get()->Load("rc/resource.xrc");

    MyFrame *frame = new MyFrame("XML resources demo",
                                 wxPoint(50, 50), wxSize(450, 340));
    frame->Show(true);
    return true;
}

// -------------------------------------------------------------------------
-----
// main frame
// -------------------------------------------------------------------------
-----

// frame constructor
MyFrame::MyFrame(const wxString& title, const wxPoint& pos, const wxSize&
size)
       : wxFrame((wxFrame *)NULL, -1, title, pos, size)
{
    SetIcon(wxICON(appicon));

    SetMenuBar(wxXmlResource::Get()->LoadMenuBar("mainmenu"));
    SetToolBar(wxXmlResource::Get()->LoadToolBar(this, "toolbar"));
}

// event handlers
void MyFrame::OnQuit(wxCommandEvent& WXUNUSED(event))
{
    // true is to force the frame to close
    Close(true);
}

void MyFrame::OnAbout(wxCommandEvent& WXUNUSED(event))
{
    wxString msg;
    msg.Printf( _T("This is the about dialog of XML resources demo.\n")
                _T("Welcome to %s"), wxVERSION_STRING);

    wxMessageBox(msg, "About XML resources demo", wxOK |
wxICON_INFORMATION, this);
```

```
}

void MyFrame::OnDlg1(wxCommandEvent& WXUNUSED(event))
{
    wxDialog dlg;
    wxXmlResource::Get()->LoadDialog(&dlg, this, "dlg1");
    dlg.ShowModal();
}

void MyFrame::OnDlg2(wxCommandEvent& WXUNUSED(event))
{
    wxDialog dlg;
    wxXmlResource::Get()->LoadDialog(&dlg, this, "dlg2");
    dlg.ShowModal();
}
```

XRC resource file sample

This is the XML file (resource.xrc) for the XRC sample.

```
<?xml version="1.0"?>
<resource version="2.3.0.1">
  <object class="wxMenuBar" name="mainmenu">
    <style>wxMB_DOCKABLE</style>
    <object class="wxMenu" name="menu_file">
      <label>_File</label>
      <style>wxMENU_TEAROFF</style>
      <object class="wxMenuItem" name="menu_about">
        <label>_About...</label>
        <bitmap>filesave.gif</bitmap>
      </object>
      <object class="separator"/>
      <object class="wxMenuItem" name="menu_dlg1">
        <label>Dialog 1</label>
      </object>
      <object class="wxMenuItem" name="menu_dlg2">
        <label>Dialog 2</label>
      </object>
      <object class="separator"/>
      <object class="wxMenuItem" name="menu_quit">
        <label>E_xit\tAlt-X</label>
      </object>
    </object>
  </object>
  <object class="wxToolBar" name="toolbar">
    <style>wxTB_FLAT|wxTB_DOCKABLE</style>
    <margins>2,2</margins>
    <object class="tool" name="menu_open">
      <bitmap>fileopen.gif</bitmap>
      <tooltip>Open catalog</tooltip>
    </object>
    <object class="tool" name="menu_save">
      <bitmap>filesave.gif</bitmap>
      <tooltip>Save catalog</tooltip>
    </object>
    <object class="tool" name="menu_update">
      <bitmap>update.gif</bitmap>
      <tooltip>Update catalog - synchronize it with sources</tooltip>
    </object>
    <separator/>
```

```
      <object class="tool" name="menu_quotes">
        <bitmap>quotes.gif</bitmap>
        <toggle>1</toggle>
        <tooltip>Display quotes around the string?</tooltip>
      </object>
      <object class="separator"/>
      <object class="tool" name="menu_fuzzy">
        <bitmap>fuzzy.gif</bitmap>
        <tooltip>Toggled if selected string is fuzzy translation</tooltip>
        <toggle>1</toggle>
      </object>
    </object>
  </object>
  <object class="wxDialog" name="dlg1">
    <object class="wxBoxSizer">
      <object class="sizeritem">
        <object class="wxBitmapButton">
          <bitmap>fuzzy.gif</bitmap>
          <focus>fileopen.gif</focus>
        </object>
      </object>
      <object class="sizeritem">
        <object class="wxPanel">
          <object class="wxStaticText">
            <label>fdgdfgdfgdfg</label>
          </object>
          <style>wxSUNKEN_BORDER</style>
        </object>
        <flag>wxALIGN_CENTER</flag>
      </object>
      <object class="sizeritem">
        <object class="wxButton">
          <label>Buttonek</label>
        </object>
        <border>10d</border>
        <flag>wxALL</flag>
      </object>
      <object class="sizeritem">
        <object class="wxHtmlWindow">
          <htmlcode>&lt;h1&gt;Hi,&lt;/h1&gt;man</htmlcode>
          <size>100,45d</size>
        </object>
      </object>
      <object class="sizeritem">
        <object class="wxNotebook">
          <object class="notebookpage">
            <object class="wxPanel">
              <object class="wxBoxSizer">
                <object class="sizeritem">
                  <object class="wxHtmlWindow">
                    <htmlcode>Hello, we are inside a
&lt;u&gt;NOTEBOOK&lt;/u&gt;...</htmlcode>
                    <size>50,50d</size>
                  </object>
                  <option>1</option>
                </object>
              </object>
            </object>
            <label>Page</label>
          </object>
          <object class="notebookpage">
            <object class="wxPanel">
              <object class="wxBoxSizer">
                <object class="sizeritem">
```

```
                <object class="wxHtmlWindow">
                  <htmlcode>Hello, we are inside a
&lt;u&gt;NOTEBOOK&lt;/u&gt;...</htmlcode>
                  <size>50,50d</size>
                </object>
              </object>
            </object>
          </object>
          <label>Page 2</label>
        </object>
        <usenotebooksizer>1</usenotebooksizer>
      </object>
      <flag>wxEXPAND</flag>
    </object>
    <orient>wxVERTICAL</orient>
  </object>
</object>
<object class="wxDialog" name="dlg2">
  <object class="wxBoxSizer">
    <orient>wxVERTICAL</orient>
    <object class="sizeritem" name="dfgdfg">
      <object class="wxTextCtrl">
        <size>200,200d</size>
        <style>wxTE_MULTILINE|wxSUNKEN_BORDER</style>
        <value>Hello, this is an ordinary multiline\n
textctrl....</value>
      </object>
      <option>1</option>
      <flag>wxEXPAND|wxALL</flag>
      <border>10</border>
    </object>
    <object class="sizeritem">
      <object class="wxBoxSizer">
        <object class="sizeritem">
          <object class="wxButton" name="wxID_OK">
            <label>Ok</label>
            <default>1</default>
          </object>
        </object>
        <object class="sizeritem">
          <object class="wxButton" name="wxID_CANCEL">
            <label>Cancel</label>
          </object>
          <border>10</border>
          <flag>wxLEFT</flag>
        </object>
      </object>
      <flag>wxLEFT|wxRIGHT|wxBOTTOM|wxALIGN_RIGHT</flag>
      <border>10</border>
    </object>
  </object>
  <title>Second testing dialog</title>
</object>
</resource>
```

XRC file format

Please see Technical Note 14 (docs/tech/tn0014.txt) in your wxWidgets distribution.

C++ header file generation

Using the `-e` switch together with `-c`, a C++ header file is written containing class definitions for the GUI windows defined in the XRC file. This code generation can make it easier to use XRC and automate program development. The classes can be used as basis for development, freeing the programmer from dealing with most of the XRC specifics (e.g. `XRCCTRL`).

For each top level window defined in the XRC file a C++ class definition is generated, containing as class members the named widgets of the window. A default constructor for each class is also generated. Inside the constructor all XRC loading is done and all class members representing widgets are initialized.

A simple example will help understand how the scheme works. Suppose you have a XRC file defining a top level window `TestWnd_Base`, which subclasses `wxFrame` (any other class like `wxDialog` will do also), and has subwidgets `wxTextCtrl` A and `wxButton` B. The XRC file and corresponding class definition in the header file will be something like:

```
<?xml version="1.0"?>
<resource version="2.3.0.1">
    <object class="wxFrame" name="TestWnd_Base">
        <size>-1,-1</size>
        <title>Test</title>
        <object class="wxBoxSizer">
            <orient>wxHORIZONTAL</orient>
            <object class="sizeritem">
                <object class="wxTextCtrl" name="A">
                    <label>Test label</label>
                </object>
            </object>
            <object class="sizeritem">
                <object class="wxButton" name="B">
                    <label>Test button</label>
                </object>
            </object>
        </object>
    </object>
</resource>


class TestWnd_Base : public wxFrame {
protected:
 wxTextCtrl* A;
 wxButton* B;

private:
 void InitWidgetsFromXRC(){
  wxXmlResource::Get()->LoadObject(this,NULL,"TestWnd","wxFrame");
  A = XRCCTRL(*this,"A",wxTextCtrl);
  B = XRCCTRL(*this,"B",wxButton);
 }
public:
TestWnd::TestWnd(){
  InitWidgetsFromXRC();
 }
};
```

The generated window class can be used as basis for the full window class. The class members which represent widgets may be accessed by name instead of using XRCCTRL every time you wish to reference them (note that they are `protected` class members), though you must still use `XRCID` to refer to widget IDs in the event table.

Example:

```
#include "resource.h"

class TestWnd : public TestWnd_Base {
 public:
        TestWnd(){
                // A, B already initialised at this point
                A->SetValue("Updated in TestWnd::TestWnd");
                B->SetValue("Nice :)");
        }
        void OnBPressed(wxEvent& event){
                Close();
        }
        DECLARE_EVENT_TABLE();
};

BEGIN_EVENT_TABLE(TestWnd,TestWnd_Base)
EVT_BUTTON(XRCID("B"),TestWnd::OnBPressed)
END_EVENT_TABLE()
```

Adding new resource handlers

Coming soon.

Scrolling overview

Classes: *wxWindow* (p. 1490), *wxScrolledWindow* (p. 1150), *wxIcon* (p. 766), *wxScrollBar* (p. 1143).

Scrollbars come in various guises in wxWidgets. All windows have the potential to show a vertical scrollbar and/or a horizontal scrollbar: it is a basic capability of a window. However, in practice, not all windows do make use of scrollbars, such as a single-line wxTextCtrl.

Because any class derived from wxWindow (p. 1490) may have scrollbars, there are functions to manipulate the scrollbars and event handlers to intercept scroll events. But just because a window generates a scroll event, doesn't mean that the window necessarily handles it and physically scrolls the window. The base class wxWindow in fact doesn't have any default functionality to handle scroll events. If you created a wxWindow object with scrollbars, and then clicked on the scrollbars, nothing at all would happen. This is deliberate, because the *interpretation* of scroll events varies from one window class to another.

wxScrolledWindow (p. 1150) (formerly wxCanvas) is an example of a window that adds functionality to make scrolling really work. It assumes that scrolling happens in

consistent units, not different-sized jumps, and that page size is represented by the visible portion of the window. It is suited to drawing applications, but perhaps not so suitable for a sophisticated editor in which the amount scrolled may vary according to the size of text on a given line. For this, you would derive from wxWindow and implement scrolling yourself. *wxGrid* (p. 619) is an example of a class that implements its own scrolling, largely because columns and rows can vary in size.

The scrollbar model

The function *wxWindow::SetScrollbar* (p. 1536) gives a clue about the way a scrollbar is modeled. This function takes the following arguments:

orientation	Which scrollbar: wxVERTICAL or wxHORIZONTAL.
position	The position of the scrollbar in scroll units.
visible	The size of the visible portion of the scrollbar, in scroll units.
range	The maximum position of the scrollbar.
refresh	Whether the scrollbar should be repainted.

orientation determines whether we're talking about the built-in horizontal or vertical scrollbar.
position is simply the position of the 'thumb' (the bit you drag to scroll around). It is given in scroll units, and so is relative to the total range of the scrollbar.

visible gives the number of scroll units that represents the portion of the window currently visible. Normally, a scrollbar is capable of indicating this visually by showing a different length of thumb.

range is the maximum value of the scrollbar, where zero is the start position. You choose the units that suit you, so if you wanted to display text that has 100 lines, you would set this to 100. Note that this doesn't have to correspond to the number of pixels scrolled - it is up to you how you actually show the contents of the window.

refresh just indicates whether the scrollbar should be repainted immediately or not.

An example

Let's say you wish to display 50 lines of text, using the same font. The window is sized so that you can only see 16 lines at a time.

You would use:

```
SetScrollbar(wxVERTICAL, 0, 16, 50);
```

Note that with the window at this size, the thumb position can never go above 50 minus 16, or 34.

You can determine how many lines are currently visible by dividing the current view size

by the character height in pixels.

When defining your own scrollbar behaviour, you will always need to recalculate the scrollbar settings when the window size changes. You could therefore put your scrollbar calculations and SetScrollbar call into a function named AdjustScrollbars, which can be called initially and also from your *wxSizeEvent* (p. 1175) handler function.

Bitmaps and icons overview

Classes: *wxBitmap* (p. 79), *wxBitmapHandler* (p. 98), *wxIcon* (p. 766), *wxCursor* (p. 226).

The wxBitmap class encapsulates the concept of a platform-dependent bitmap, either monochrome or colour. Platform-specific methods for creating a wxBitmap object from an existing file are catered for, and this is an occasion where conditional compilation will sometimes be required.

A bitmap created dynamically or loaded from a file can be selected into a memory device context (instance of *wxMemoryDC* (p. 935)). This enables the bitmap to be copied to a window or memory device context using *wxDC::Blit* (p. 367), or to be used as a drawing surface.

See *wxMemoryDC* (p. 935) for an example of drawing onto a bitmap.

All wxWidgets platforms support XPMs for small bitmaps and icons. You may include the XPM inline as below, since it's C code, or you can load it at run-time.

```
#include "mondrian.xpm"
```

Sometimes you wish to use a .ico resource on Windows, and XPMs on other platforms (for example to take advantage of Windows' support for multiple icon resolutions). A macro, *wxICON* (p. 1625), is available which creates an icon using an XPM on the appropriate platform, or an icon resource on Windows.

```
wxIcon icon(wxICON(mondrian));

// Equivalent to:

#if defined(__WXGTK__) || defined(__WXMOTIF__)
wxIcon icon(mondrian_xpm);
#endif

#if defined(__WXMSW__)
wxIcon icon("mondrian");
#endif
```

There is also a corresponding *wxBITMAP* (p. 1623) macro which allows to create the bitmaps in much the same way as *wxICON* (p. 1625) creates icons. It assumes that bitmaps live in resources under Windows or OS2 and XPM files under all other platforms (for XPMs, the corresponding file must be included before this macro is used, of course, and the name of the bitmap should be the same as the resource name under Windows with _xpm suffix). For example:

```
// an easy and portable way to create a bitmap
wxBitmap bmp(wxBITMAP(bmpname));

// which is roughly equivalent to the following
#if defined(__WXMSW__) || defined(__WXPM__)
    wxBitmap bmp("bmpname", wxBITMAP_TYPE_RESOURCE);
#else // Unix
    wxBitmap bmp(bmpname_xpm, wxBITMAP_TYPE_XPM);
#endif
```

You should always use wxICON and wxBITMAP macros because they work for any platform (unlike the code above which doesn't deal with wxMac, wxX11, ...) and are more short and clear than versions with `#ifdef`s. Even better, use the same XPMs on all platforms.

Supported bitmap file formats

The following lists the formats handled on different platforms. Note that missing or partially-implemented formats are automatically supplemented by the *wxImage* (p. 779) to load the data, and then converting it to wxBitmap form. Note that using wxImage is the preferred way to load images in wxWidgets, with the exception of resources (XPM-files or native Windows resources). Writing an image format handler for wxImage is also far easier than writing one for wxBitmap, because wxImage has exactly one format on all platforms whereas wxBitmap can store pixel data very differently, depending on colour depths and platform.

wxBitmap

Under Windows, wxBitmap may load the following formats:

- Windows bitmap resource (wxBITMAP_TYPE_BMP_RESOURCE)

- Windows bitmap file (wxBITMAP_TYPE_BMP)

- XPM data and file (wxBITMAP_TYPE_XPM)

- All formats that are supported by the *wxImage* (p. 779) class.

Under wxGTK, wxBitmap may load the following formats:

- XPM data and file (wxBITMAP_TYPE_XPM)

- All formats that are supported by the *wxImage* (p. 779) class.

Under wxMotif and wxX11, wxBitmap may load the following formats:

- XBM data and file (wxBITMAP_TYPE_XBM)

- XPM data and file (wxBITMAP_TYPE_XPM)

- All formats that are supported by the *wxImage* (p. 779) class.

wxIcon

Under Windows, wxIcon may load the following formats:

- • Windows icon resource (wxBITMAP_TYPE_ICO_RESOURCE)

- • Windows icon file (wxBITMAP_TYPE_ICO)

- • XPM data and file (wxBITMAP_TYPE_XPM)

Under wxGTK, wxIcon may load the following formats:

- • XPM data and file (wxBITMAP_TYPE_XPM)

- • All formats that are supported by the *wxImage* (p. 779) class.

Under wxMotif and wxX11, wxIcon may load the following formats:

- • XBM data and file (wxBITMAP_TYPE_XBM)

- • XPM data and file (wxBITMAP_TYPE_XPM)

- • All formats that are supported by the *wxImage* (p. 779) class.

wxCursor

Under Windows, wxCursor may load the following formats:

- • Windows cursor resource (wxBITMAP_TYPE_CUR_RESOURCE)

- • Windows cursor file (wxBITMAP_TYPE_CUR)

- • Windows icon file (wxBITMAP_TYPE_ICO)

- • Windows bitmap file (wxBITMAP_TYPE_BMP)

Under wxGTK, wxCursor may load the following formats (in additional to stock cursors):

- • None (stock cursors only).

Under wxMotif and wxX11, wxCursor may load the following formats:

- • XBM data and file (wxBITMAP_TYPE_XBM)

Bitmap format handlers

To provide extensibility, the functionality for loading and saving bitmap formats is not implemented in the wxBitmap class, but in a number of handler classes, derived from wxBitmapHandler. There is a static list of handlers which wxBitmap examines when a file load/save operation is requested. Some handlers are provided as standard, but if you have special requirements, you may wish to initialise the wxBitmap class with some extra handlers which you write yourself or receive from a third party.

To add a handler object to wxBitmap, your application needs to include the header which implements it, and then call the static function *wxBitmap::AddHandler* (p. 83).

Note: bitmap handlers are not implemented on all platforms, and new ones rarely need to be implemented since wxImage can be used for loading most formats, as noted earlier.

Device context overview

Classes: *wxBufferedDC* (p. 111), *wxBufferedPaintDC* (p. 112), *wxDC* (p. 365), *wxPostScriptDC* (p. 1046), *wxMetafileDC* (p. 973), *wxMemoryDC* (p. 935), *wxPrinterDC* (p. 1065), *wxScreenDC* (p. 1142), *wxClientDC* (p. 147), *wxPaintDC* (p. 1024), *wxWindowDC* (p. 1547).

A wxDC is a *device context* onto which graphics and text can be drawn. The device context is intended to represent a number of output devices in a generic way, with the same API being used throughout.

Some device contexts are created temporarily in order to draw on a window. This is true of *wxScreenDC* (p. 1142), *wxClientDC* (p. 147), *wxPaintDC* (p. 1024), and *wxWindowDC* (p. 1547). The following describes the differences between these device contexts and when you should use them.

- **wxScreenDC.** Use this to paint on the screen, as opposed to an individual window.

- **wxClientDC.** Use this to paint on the client area of window (the part without borders and other decorations), but do not use it from within an *wxPaintEvent* (p. 1025).

- **wxPaintDC.** Use this to paint on the client area of a window, but *only* from within a *wxPaintEvent* (p. 1025).

- **wxWindowDC.** Use this to paint on the whole area of a window, including decorations. This may not be available on non-Windows platforms.

To use a client, paint or window device context, create an object on the stack with the window as argument, for example:

```
void MyWindow::OnMyCmd(wxCommandEvent& event)
{
  wxClientDC dc(window);
  DrawMyPicture(dc);
}
```

Try to write code so it is parameterised by wxDC - if you do this, the same piece of code may write to a number of different devices, by passing a different device context. This doesn't work for everything (for example not all device contexts support bitmap drawing) but will work most of the time.

wxFont overview

Class: *wxFont* (p. 558), *wxFontDialog* (p. 571)

A font is an object which determines the appearance of text, primarily when drawing text to a window or device context. A font is determined by the following parameters (not all of them have to be specified, of course):

Point size	This is the standard way of referring to text size.
Family	Supported families are: **wxDEFAULT, wxDECORATIVE, wxROMAN, wxSCRIPT, wxSWISS, wxMODERN**. **wxMODERN** is a fixed pitch font; the others are either fixed or variable pitch.
Style	The value can be **wxNORMAL, wxSLANT** or **wxITALIC**.
Weight	The value can be **wxNORMAL, wxLIGHT** or **wxBOLD**.
Underlining	The value can be true or false.
Face name	An optional string specifying the actual typeface to be used. If NULL, a default typeface will chosen based on the family.
Encoding	The font encoding (see **wxFONTENCODING_XXX**constants and the *font overview* (p. 1793) for more details)

Specifying a family, rather than a specific typeface name, ensures a degree of portability across platforms because a suitable font will be chosen for the given font family, however it doesn't allow to choose a font precisely as the parameters above don't suffice, in general, to identify all the available fonts and this is where using the native font descriptions may be helpful - see below.

Under Windows, the face name can be one of the installed fonts on the user's system. Since the choice of fonts differs from system to system, either choose standard Windows fonts, or if allowing the user to specify a face name, store the family name with any file that might be transported to a different Windows machine or other platform.

> **Note:** There is currently a difference between the appearance of fonts on the two platforms, if the mapping mode is anything other than wxMM_TEXT. Under X, font size is always specified in points. Under MS Windows, the unit for text is points but the text is scaled according to the current mapping mode. However, user scaling on a device context will also scale fonts under both environments.

Native font information

An alternative way of choosing fonts is to use the native font description. This is the only acceptable solution if the user is allowed to choose the font using the *wxFontDialog* (p. 571) because the selected font cannot be described using only the family name and so, if only family name is stored permanently, the user would almost surely see a different font in the program later.

Instead, you should store the value returned by *wxFont::GetNativeFontInfoDesc* (p. 564) and pass it to *wxFont::SetNativeFontInfo* (p. 567) later to recreate exactly the same font.

Note that the contents of this string depends on the platform and shouldn't be used for any other purpose (in particular, it is not meant to be shown to the user). Also please note that although the native font information is currently implemented for Windows and Unix (GTK+ and Motif) ports only, all the methods are available for all the ports and should be used to make your program work correctly when they are implemented later.

Font encoding overview

wxWidgets has support for multiple font encodings starting from release 2.2. By encoding we mean here the mapping between the character codes and the letters. Probably the most well-known encoding is (7 bit) ASCII one which is used almost universally now to represent the letters of the English alphabet and some other common characters. However, it is not enough to represent the letters of foreign alphabets and here other encodings come into play. Please note that we will only discuss 8-bit fonts here and not *Unicode* (p. 1730).

Font encoding support is ensured by several classes: *wxFont* (p. 558) itself, but also *wxFontEnumerator* (p. 573) and *wxFontMapper* (p. 575). wxFont encoding support is reflected by a (new) constructor parameter *encoding* which takes one of the following values (elements of enumeration type `wxFontEncoding`):

wxFONTENCODING_SYSTEM   The default encoding of the underlying operating system (notice that this might be a "foreign" encoding for foreign versions of Windows 9x/NT).

wxFONTENCODING_DEFAULT   The applications default encoding as returned by *wxFont::GetDefaultEncoding* (p. 563). On program startup, the applications default encoding is the same as wxFONTENCODING_SYSTEM, but may be changed to make all the fonts created later to use it (by default).

wxFONTENCODING_ISO8859_1..15   ISO8859 family encodings which are usually used by all non-Microsoft operating systems

wxFONTENCODING_KOI8   Standard Cyrillic encoding for the Internet (but see also wxFONTENCODING_ISO8859_5 and wxFONTENCODING_CP1251)

wxFONTENCODING_CP1250   Microsoft analogue of ISO8859-2

wxFONTENCODING_CP1251   Microsoft analogue of ISO8859-5

wxFONTENCODING_CP1252   Microsoft analogue of ISO8859-1

As you may see, Microsoft's encoding partly mirror the standard ISO8859 ones, but there are (minor) differences even between ISO8859-1 (Latin1, ISO encoding for Western Europe) and CP1251 (WinLatin1, standard code page for English versions of Windows) and there are more of them for other encodings.

The situation is particularly complicated with Cyrillic encodings for which (more than) three incompatible encodings exist: KOI8 (the old standard, widely used on the Internet),

ISO8859-5 (ISO standard for Cyrillic) and CP1251 (WinCyrillic).

This abundance of (incompatible) encodings should make it clear that using encodings is less easy than it might seem. The problems arise both from the fact that the standard encodings for the given language (say Russian, which is written in Cyrillic) are different on different platforms and because the fonts in the given encoding might just not be installed (this is especially a problem with Unix, or, in general, non-Win32 systems).

To clarify, the *wxFontEnumerator* (p. 573) class may be used to enumerate both all available encodings and to find the facename(s) in which the given encoding exists. If you can find the font in the correct encoding with wxFontEnumerator then your troubles are over, but, unfortunately, sometimes this is not enough. For example, there is no standard way (that I know of, please tell me if you do!) to find a font on a Windows system for KOI8 encoding (only for WinCyrillic one which is quite different), so *wxFontEnumerator* (p. 573) will never return one, even if the user has installed a KOI8 font on his system.

To solve this problem, a *wxFontMapper* (p. 575) class is provided. This class stores the mapping between the encodings and the font face names which support them in *wxConfig* (p. 1748) object. Of course, it would be fairly useless if it tried to determine these mappings by itself, so, instead, it (optionally) asks the user and remembers his answers so that the next time the program will automatically choose the correct font.

All these topics are illustrated by the *font sample* (p. 1711); please refer to it and the documentation of the classes mentioned here for further explanations.

wxSplitterWindow overview

Classes: *wxSplitterWindow* (p. 1239)

The following screenshot shows the appearance of a splitter window with a vertical split.

PRESS F9 TO FORMAT PICTURE

The style wxSP_3D has been used to show a 3D border and 3D sash.

Example

The following fragment shows how to create a splitter window, creating two subwindows and hiding one of them.

```
    splitter = new wxSplitterWindow(this, -1, wxPoint(0, 0), wxSize(400,
400), wxSP_3D);

    leftWindow = new MyWindow(splitter);
    leftWindow->SetScrollbars(20, 20, 50, 50);

    rightWindow = new MyWindow(splitter);
    rightWindow->SetScrollbars(20, 20, 50, 50);
    rightWindow->Show(false);
```

```
        splitter->Initialize(leftWindow);

    // Set this to prevent unsplitting
//    splitter->SetMinimumPaneSize(20);
```

The next fragment shows how the splitter window can be manipulated after creation.

```
void MyFrame::OnSplitVertical(wxCommandEvent& event)
{
        if ( splitter->IsSplit() )
            splitter->Unsplit();
        leftWindow->Show(true);
        rightWindow->Show(true);
        splitter->SplitVertically( leftWindow, rightWindow );
}

void MyFrame::OnSplitHorizontal(wxCommandEvent& event)
{
        if ( splitter->IsSplit() )
            splitter->Unsplit();
        leftWindow->Show(true);
        rightWindow->Show(true);
        splitter->SplitHorizontally( leftWindow, rightWindow );
}

void MyFrame::OnUnsplit(wxCommandEvent& event)
{
        if ( splitter->IsSplit() )
            splitter->Unsplit();
}
```

wxTreeCtrl overview

Classes: *wxTreeCtrl* (p. 1424), *wxImageList* (p. 806)

The tree control displays its items in a tree like structure. Each item has its own (optional) icon and a label. An item may be either collapsed (meaning that its children are not visible) or expanded (meaning that its children are shown). Each item in the tree is identified by its *itemId* which is of opaque data type *wxTreeItemId*. You can test whether an item is valid by calling wxTreeItemId::IsOk.

The items text and image may be retrieved and changed with *GetItemText* (p. 1434)/*SetItemText* (p. 1442) and *GetItemImage* (p. 1434)/*SetItemImage* (p. 1441). In fact, an item may even have two images associated with it: the normal one and another one for selected state which is set/retrieved with *SetItemSelectedImage* (p. 1441)/*GetItemSelectedImage* (p. 1436) functions, but this functionality might be unavailable on some platforms.

Tree items have several attributes: an item may be selected or not, visible or not, bold or not. It may also be expanded or collapsed. All these attributes may be retrieved with the corresponding functions: *IsSelected* (p. 1438), *IsVisible* (p. 1438), *IsBold* (p. 1438) and

IsExpanded (p. 1438). Only one item at a time may be selected, selecting another one (with *SelectItem* (p. 1439)) automatically unselects the previously selected one.

In addition to its icon and label, a user-specific data structure may be associated with all tree items. If you wish to do it, you should derive a class from *wxTreeItemData* which is a very simple class having only one function *GetId()* which returns the id of the item this data is associated with. This data will be freed by the control itself when the associated item is deleted (all items are deleted when the control is destroyed), so you shouldn't delete it yourself (if you do it, you should call *SetItemData(NULL)* (p. 1440) to prevent the tree from deleting the pointer second time). The associated data may be retrieved with *GetItemData()* (p. 1433) function.

Working with trees is relatively straightforward if all the items are added to the tree at the moment of its creation. However, for large trees it may be very inefficient. To improve the performance you may want to delay adding the items to the tree until the branch containing the items is expanded: so, in the beginning, only the root item is created (with *AddRoot* (p. 1428)). Other items are added when EVT_TREE_ITEM_EXPANDING event is received: then all items lying immediately under the item being expanded should be added, but, of course, only when this event is received for the first time for this item - otherwise, the items would be added twice if the user expands/collapses/re-expands the branch.

The tree control provides functions for enumerating its items. There are 3 groups of enumeration functions: for the children of a given item, for the sibling of the given item and for the visible items (those which are currently shown to the user: an item may be invisible either because its branch is collapsed or because it is scrolled out of view). Child enumeration functions require the caller to give them a *cookie* parameter: it is a number which is opaque to the caller but is used by the tree control itself to allow multiple enumerations to run simultaneously (this is explicitly allowed). The only thing to remember is that the *cookie* passed to *GetFirstChild* (p. 1432) and to *GetNextChild* (p. 1434) should be the same variable (and that nothing should be done with it by the user code).

Among other features of the tree control are: item sorting with *SortChildren* (p. 1442) which uses the user-defined comparison function *OnCompareItems* (p. 1439) (by default the comparison is the alphabetic comparison of tree labels), hit testing (determining to which portion of the control the given point belongs, useful for implementing drag-and-drop in the tree) with *HitTest* (p. 1437) and editing of the tree item labels in place (see *EditLabel* (p. 1430)).

Finally, the tree control has a keyboard interface: the cursor navigation (arrow) keys may be used to change the current selection. <HOME> and <END> are used to go to the first/last sibling of the current item. '+', '-' and '*' expand, collapse and toggle the current branch. Note, however, that <DEL> and <INS> keys do nothing by default, but it is usual to associate them with deleting item from a tree and inserting a new one into it.

wxListCtrl overview

Classes: *wxListCtrl* (p. 852), *wxImageList* (p. 806)

Sorry, this topic has yet to be written.

wxImageList overview

Classes: *wxImageList* (p. 806)

An image list is a list of images that may have transparent areas. The class helps an application organise a collection of images so that they can be referenced by integer index instead of by pointer.

Image lists are used in *wxNotebook* (p. 999), *wxListCtrl* (p. 852), *wxTreeCtrl* (p. 852) and some other control classes.

wxBookCtrl overview

Classes: *wxNotebook* (p. 999), *wxListbook* (p. 846), *wxChoicebook* (p. 143)

Introduction

A book control is a convenient way of displaying multiple pages of information, displayed one page at a time. wxWidgets has three variants of this control:

- *wxNotebook* (p. 999): uses a row of tabs

- *wxListbook* (p. 846): controlled by a *wxListCtrl* (p. 852)

- *wxChoicebook* (p. 143): controlled by a *wxChoice* (p. 140)

Best book

wxBookCtrl is mapped to the class best suited for a given platform. Currently it provides *wxChoicebook* (p. 143) for smartphones equipped with WinCE, and *wxNotebook* (p. 999) for all other platforms. The mapping consists of:

wxBookCtrl   wxChoicebook or wxNotebook

wxBookCtrlEvent   wxChoicebookEvent or wxNotebookEvent

wxEVT_COMMAND_BOOKCTRL_PAGE_CHANGED
                                 wxEVT_COMMAND_CHOICEBOOK_PAGE_CHANGED
                                 or wxEVT_COMMAND_NOTEBOOK_PAGE_CHANGED

wxEVT_COMMAND_BOOKCTRL_PAGE_CHANGING
                                 wxEVT_COMMAND_CHOICEBOOK_PAGE_CHANGING
                                 or wxEVT_COMMAND_NOTEBOOK_PAGE_CHANGING

EVT_BOOKCTRL_PAGE_CHANGED(id, fn)
                                 EVT_CHOICEBOOK_PAGE_CHANGED(id, fn) or
                                 EVT_NOTEBOOK_PAGE_CHANGED(id, fn)

EVT_BOOKCTRL_PAGE_CHANGING(id, fn)

	EVT_CHOICEBOOK_PAGE_CHANGING(id, fn) or EVT_NOTEBOOK_PAGE_CHANGING(id, fn)
wxBC_TOP	**wxCHB_TOP** or **wxNB_TOP**
wxBC_BOTTOM	**wxCHB_BOTTOM** or **wxNB_BOTTOM**
wxBC_LEFT	**wxCHB_LEFT** or **wxNB_LEFT**
wxBC_RIGHT	**wxCHB_RIGHT** or **wxNB_RIGHT**
wxBC_DEFAULT	**wxCHB_DEFAULT** or **wxNB_DEFAULT**

See `samples/widgets` for an example of wxBookCtrl usage.

Common dialogs overview

Classes: *wxColourDialog* (p. 171), *wxFontDialog* (p. 571), *wxPrintDialog* (p. 1057), *wxFileDialog* (p. 511), *wxDirDialog* (p. 424), *wxTextEntryDialog* (p. 1361), *wxPasswordEntryDialog* (p. 1034), *wxMessageDialog* (p. 969), *wxSingleChoiceDialog* (p. 1169), *wxMultiChoiceDialog* (p. 993)

Common dialog classes and functions encapsulate commonly-needed dialog box requirements. They are all 'modal', grabbing the flow of control until the user dismisses the dialog, to make them easy to use within an application.

Some dialogs have both platform-dependent and platform-independent implementations, so that if underlying windowing systems do not provide the required functionality, the generic classes and functions can stand in. For example, under MS Windows, wxColourDialog uses the standard colour selector. There is also an equivalent called wxGenericColourDialog for other platforms, and a macro defines wxColourDialog to be the same as wxGenericColourDialog on non-MS Windows platforms. However, under MS Windows, the generic dialog can also be used, for testing or other purposes.

wxColourDialog overview

Classes: *wxColourDialog* (p. 171), *wxColourData* (p. 167)

The wxColourDialog presents a colour selector to the user, and returns with colour information.

The MS Windows colour selector

Under Windows, the native colour selector common dialog is used. This presents a dialog box with three main regions: at the top left, a palette of 48 commonly-used colours is shown. Under this, there is a palette of 16 'custom colours' which can be set by the application if desired. Additionally, the user may open up the dialog box to show a right-hand panel containing controls to select a precise colour, and add it to the custom colour palette.

The generic colour selector

Under non-MS Windows platforms, the colour selector is a simulation of most of the features of the MS Windows selector. Two palettes of 48 standard and 16 custom colours are presented, with the right-hand area containing three sliders for the user to select a colour from red, green and blue components. This colour may be added to the custom colour palette, and will replace either the currently selected custom colour, or the first one in the palette if none is selected. The RGB colour sliders are not optional in the generic colour selector. The generic colour selector is also available under MS Windows; use the name wxGenericColourDialog.

Example

In the samples/dialogs directory, there is an example of using the wxColourDialog class. Here is an excerpt, which sets various parameters of a wxColourData object, including a grey scale for the custom colours. If the user did not cancel the dialog, the application retrieves the selected colour and uses it to set the background of a window.

```
wxColourData data;
data.SetChooseFull(true);
for (int i = 0; i < 16; i++)
{
  wxColour colour(i*16, i*16, i*16);
  data.SetCustomColour(i, colour);
}

wxColourDialog dialog(this, &data);
if (dialog.ShowModal() == wxID_OK)
{
  wxColourData retData = dialog.GetColourData();
  wxColour col = retData.GetColour();
  wxBrush brush(col, wxSOLID);
  myWindow->SetBackground(brush);
  myWindow->Clear();
  myWindow->Refresh();
}
```

wxFontDialog overview

Classes: *wxFontDialog* (p. 571), *wxFontData* (p. 569)

The wxFontDialog presents a font selector to the user, and returns with font and colour information.

The MS Windows font selector

Under Windows, the native font selector common dialog is used. This presents a dialog box with controls for font name, point size, style, weight, underlining, strikeout and text foreground colour. A sample of the font is shown on a white area of the dialog box. Note that in the translation from full MS Windows fonts to wxWidgets font conventions, strikeout is ignored and a font family (such as Swiss or Modern) is deduced from the actual font name (such as Arial or Courier).

The generic font selector

Under non-MS Windows platforms, the font selector is simpler. Controls for font family, point size, style, weight, underlining and text foreground colour are provided, and a

sample is shown upon a white background. The generic font selector is also available under MS Windows; use the name wxGenericFontDialog.

Example

In the samples/dialogs directory, there is an example of using the wxFontDialog class. The application uses the returned font and colour for drawing text on a canvas. Here is an excerpt:

```
wxFontData data;
data.SetInitialFont(canvasFont);
data.SetColour(canvasTextColour);

wxFontDialog dialog(this, &data);
if (dialog.ShowModal() == wxID_OK)
{
  wxFontData retData = dialog.GetFontData();
  canvasFont = retData.GetChosenFont();
  canvasTextColour = retData.GetColour();
  myWindow->Refresh();
}
```

wxPrintDialog overview

Classes: *wxPrintDialog* (p. 1057), *wxPrintData* (p. 1051)

This class represents the print and print setup common dialogs. You may obtain a *wxPrinterDC* (p. 1065) device context from a successfully dismissed print dialog.

The samples/printing example shows how to use it: see *Printing overview* (p. 1815) for an excerpt from this example.

wxFileDialog overview

Classes: *wxFileDialog* (p. 511)

Pops up a file selector box. In Windows and GTK2.4+, this is the common file selector dialog. In X, this is a file selector box with somewhat less functionality. The path and filename are distinct elements of a full file pathname. If path is "", the current directory will be used. If filename is "", no default filename will be supplied. The wildcard determines what files are displayed in the file selector, and file extension supplies a type extension for the required filename. Flags may be a combination of wxOPEN, wxSAVE, wxOVERWRITE_PROMPT, wxHIDE_READONLY, wxFILE_MUST_EXIST, wxMULTIPLE, wxCHANGE_DIR or 0.

Both the X and Windows versions implement a wildcard filter. Typing a filename containing wildcards (*, ?) in the filename text item, and clicking on Ok, will result in only those files matching the pattern being displayed. In the X version, supplying no default name will result in the wildcard filter being inserted in the filename text item; the filter is ignored if a default name is supplied.

The wildcard may be a specification for multiple types of file with a description for each,

such as:

```
"BMP files (*.bmp)|*.bmp|GIF files (*.gif)|*.gif"
```

wxDirDialog overview

Classes: *wxDirDialog* (p. 424)

This dialog shows a directory selector dialog, allowing the user to select a single directory.

wxTextEntryDialog overview

Classes: *wxTextEntryDialog* (p. 1361)

This is a dialog with a text entry field. The value that the user entered is obtained using *wxTextEntryDialog::GetValue* (p. 1362).

wxPasswordEntryDialog overview

Classes: *wxPasswordEntryDialog* (p. 1034)

This is a dialog with a password entry field. The value that the user entered is obtained using *wxTextEntryDialog::GetValue* (p. 1362).

wxMessageDialog overview

Classes: *wxMessageDialog* (p. 969)

This dialog shows a message, plus buttons that can be chosen from OK, Cancel, Yes, and No. Under Windows, an optional icon can be shown, such as an exclamation mark or question mark.

The return value of *wxMessageDialog::ShowModal* (p. 971) indicates which button the user pressed.

wxSingleChoiceDialog overview

Classes: *wxSingleChoiceDialog* (p. 1169)

This dialog shows a list of choices, plus OK and (optionally) Cancel. The user can select one of them. The selection can be obtained from the dialog as an index, a string or client data.

wxMultiChoiceDialog overview

Classes: *wxMultiChoiceDialog* (p. 993)

This dialog shows a list of choices, plus OK and (optionally) Cancel. The user can select one or more of them.

Document/view overview

Classes: *wxDocument* (p. 454), *wxView* (p. 1473), *wxDocTemplate* (p. 448), *wxDocManager* (p. 435), *wxDocParentFrame* (p. 447), *wxDocChildFrame* (p. 433), *wxDocMDIParentFrame* (p. 446), *wxDocMDIChildFrame* (p. 444), *wxCommand* (p. 178), *wxCommandProcessor* (p. 185)

The document/view framework is found in most application frameworks, because it can dramatically simplify the code required to build many kinds of application.

The idea is that you can model your application primarily in terms of *documents* to store data and provide interface-independent operations upon it, and *views* to visualise and manipulate the data. Documents know how to do input and output given stream objects, and views are responsible for taking input from physical windows and performing the manipulation on the document data. If a document's data changes, all views should be updated to reflect the change.

The framework can provide many user-interface elements based on this model. Once you have defined your own classes and the relationships between them, the framework takes care of popping up file selectors, opening and closing files, asking the user to save modifications, routing menu commands to appropriate (possibly default) code, even some default print/preview functionality and support for command undo/redo. The framework is highly modular, allowing overriding and replacement of functionality and objects to achieve more than the default behaviour.

These are the overall steps involved in creating an application based on the document/view framework:

1. Define your own document and view classes, overriding a minimal set of member functions e.g. for input/output, drawing and initialization.

2. Define any subwindows (such as a scrolled window) that are needed for the view(s). You may need to route some events to views or documents, for example OnPaint needs to be routed to wxView::OnDraw.

3. Decide what style of interface you will use: Microsoft's MDI (multiple document child frames surrounded by an overall frame), SDI (a separate, unconstrained frame for each document), or single-window (one document open at a time, as in Windows Write).

4. Use the appropriate wxDocParentFrame and wxDocChildFrame classes. Construct an instance of wxDocParentFrame in your wxApp::OnInit, and a wxDocChildFrame (if not single-window) when you initialize a view. Create menus using standard menu ids (such as wxID_OPEN, wxID_PRINT).

5. Construct a single wxDocManager instance at the beginning of your

wxApp::OnInit, and then as many wxDocTemplate instances as necessary to define relationships between documents and views. For a simple application, there will be just one wxDocTemplate.

If you wish to implement Undo/Redo, you need to derive your own class(es) from wxCommand and use wxCommandProcessor::Submit instead of directly executing code. The framework will take care of calling Undo and Do functions as appropriate, so long as the wxID_UNDO and wxID_REDO menu items are defined in the view menu.

Here are a few examples of the tailoring you can do to go beyond the default framework behaviour:

- Override wxDocument::OnCreateCommandProcessor to define a different Do/Undo strategy, or a command history editor.

- Override wxView::OnCreatePrintout to create an instance of a derived *wxPrintout* (p. 1065) class, to provide multi-page document facilities.

- Override wxDocManager::SelectDocumentPath to provide a different file selector.

- Limit the maximum number of open documents and the maximum number of undo commands.

Note that to activate framework functionality, you need to use some or all of the wxWidgets *predefined command identifiers* (p. 1807) in your menus.

wxPerl note: The document/view framework is available in wxPerl. To use it, you will need the following statements in your application code:

```
use Wx::DocView;
use Wx ':docview';   # import constants (optional)
```

wxDocument overview

Document/view framework overview (p. 1802)

Class: *wxDocument* (p. 454)

The wxDocument class can be used to model an application's file-based data. It is part of the document/view framework supported by wxWidgets, and cooperates with the *wxView* (p. 1473), *wxDocTemplate* (p. 448) and *wxDocManager* (p. 435) classes.

Using this framework can save a lot of routine user-interface programming, since a range of menu commands -- such as open, save, save as -- are supported automatically. The programmer just needs to define a minimal set of classes and member functions for the framework to call when necessary. Data, and the means to view and edit the data, are explicitly separated out in this model, and the concept of multiple *views* onto the same data is supported.

Note that the document/view model will suit many but not all styles of application. For

example, it would be overkill for a simple file conversion utility, where there may be no call for *views* on *documents* or the ability to open, edit and save files. But probably the majority of applications are document-based.

See the example application in `samples/docview`.

To use the abstract wxDocument class, you need to derive a new class and override at least the member functions SaveObject and LoadObject. SaveObject and LoadObject will be called by the framework when the document needs to be saved or loaded.

Use the macros DECLARE_DYNAMIC_CLASS and IMPLEMENT_DYNAMIC_CLASS in order to allow the framework to create document objects on demand. When you create a *wxDocTemplate* (p. 448) object on application initialization, you should pass CLASSINFO(YourDocumentClass) to the wxDocTemplate constructor so that it knows how to create an instance of this class.

If you do not wish to use the wxWidgets method of creating document objects dynamically, you must override wxDocTemplate::CreateDocument to return an instance of the appropriate class.

wxView overview

Document/view framework overview (p. 1802)

Class: *wxView* (p. 1473)

The wxView class can be used to model the viewing and editing component of an application's file-based data. It is part of the document/view framework supported by wxWidgets, and cooperates with the *wxDocument* (p. 454), *wxDocTemplate* (p. 448)and *wxDocManager* (p. 435) classes.

See the example application in `samples/docview`.

To use the abstract wxView class, you need to derive a new class and override at least the member functions OnCreate, OnDraw, OnUpdate and OnClose. You will probably want to override OnMenuCommand to respond to menu commands from the frame containing the view.

Use the macros DECLARE_DYNAMIC_CLASS and IMPLEMENT_DYNAMIC_CLASS in order to allow the framework to create view objects on demand. When you create a *wxDocTemplate* (p. 448) object on application initialization, you should pass CLASSINFO(YourViewClass) to the wxDocTemplate constructor so that it knows how to create an instance of this class.

If you do not wish to use the wxWidgets method of creating view objects dynamically, you must override wxDocTemplate::CreateView to return an instance of the appropriate class.

wxDocTemplate overview

Document/view framework overview (p. 1802)

Class: *wxDocTemplate* (p. 448)

The wxDocTemplate class is used to model the relationship between a document class and a view class. The application creates a document template object for each document/view pair. The list of document templates managed by the wxDocManager instance is used to create documents and views. Each document template knows what file filters and default extension are appropriate for a document/view combination, and how to create a document or view.

For example, you might write a small doodling application that can load and save lists of line segments. If you had two views of the data -- graphical, and a list of the segments -- then you would create one document class DoodleDocument, and two view classes (DoodleGraphicView and DoodleListView). You would also need two document templates, one for the graphical view and another for the list view. You would pass the same document class and default file extension to both document templates, but each would be passed a different view class. When the user clicks on the Open menu item, the file selector is displayed with a list of possible file filters -- one for each wxDocTemplate. Selecting the filter selects the wxDocTemplate, and when a file is selected, that template will be used for creating a document and view.

For the case where an application has one document type and one view type, a single document template is constructed, and dialogs will be appropriately simplified.

wxDocTemplate is part of the document/view framework supported by wxWidgets, and cooperates with the *wxView* (p. 1473), *wxDocument* (p. 454) and *wxDocManager* (p. 435) classes.

See the example application in `samples/docview`.

To use the wxDocTemplate class, you do not need to derive a new class. Just pass relevant information to the constructor including CLASSINFO(YourDocumentClass) and CLASSINFO(YourViewClass) to allow dynamic instance creation. If you do not wish to use the wxWidgets method of creating document objects dynamically, you must override wxDocTemplate::CreateDocument and wxDocTemplate::CreateView to return instances of the appropriate class.

NOTE: the document template has nothing to do with the C++ template construct.

wxDocManager overview

Document/view framework overview (p. 1802)

Class: *wxDocManager* (p. 435)

The wxDocManager class is part of the document/view framework supported by wxWidgets, and cooperates with the *wxView* (p. 1473), *wxDocument* (p. 454) and *wxDocTemplate* (p. 448) classes.

A wxDocManager instance coordinates documents, views and document templates. It keeps a list of document and template instances, and much functionality is routed through this object, such as providing selection and file dialogs. The application can use this class 'as is' or derive a class and override some members to extend or change the

functionality. Create an instance of this class near the beginning of your application initialization, before any documents, views or templates are manipulated.

There may be multiple wxDocManager instances in an application.

See the example application in `samples/docview`.

wxCommand overview

Document/view framework overview (p. 1802)

Classes: *wxCommand* (p. 178), *wxCommandProcessor* (p. 185)

wxCommand is a base class for modelling an application command, which is an action usually performed by selecting a menu item, pressing a toolbar button or any other means provided by the application to change the data or view.

Instead of the application functionality being scattered around switch statements and functions in a way that may be hard to read and maintain, the functionality for a command is explicitly represented as an object which can be manipulated by a framework or application. When a user interface event occurs, the application *submits* a command to a *wxCommandProcessor* (p. 1806) object to execute and store.

The wxWidgets document/view framework handles Undo and Redo by use of wxCommand and wxCommandProcessor objects. You might find further uses for wxCommand, such as implementing a macro facility that stores, loads and replays commands.

An application can derive a new class for every command, or, more likely, use one class parameterized with an integer or string command identifier.

wxCommandProcessor overview

Document/view framework overview (p. 1802)

Classes: *wxCommandProcessor* (p. 185), *wxCommand* (p. 178)

wxCommandProcessor is a class that maintains a history of wxCommand instances, with undo/redo functionality built-in. Derive a new class from this if you want different behaviour.

wxFileHistory overview

Document/view framework overview (p. 1802)

Classes: *wxFileHistory* (p. 517), *wxDocManager* (p. 435)

wxFileHistory encapsulates functionality to record the last few files visited, and to allow the user to quickly load these files using the list appended to the File menu.

Although wxFileHistory is used by wxDocManager, it can be used independently. You may wish to derive from it to allow different behaviour, such as popping up a scrolling list of files.

By calling wxFileHistory::FileHistoryUseMenu you can associate a file menu with the file history, that will be used for appending the filenames. They are appended using menu identifiers in the range wxID_FILE1 to wxID_FILE9.

In order to respond to a file load command from one of these identifiers, you need to handle them using an event handler, for example:

```
BEGIN_EVENT_TABLE(wxDocParentFrame, wxFrame)
    EVT_MENU(wxID_EXIT, wxDocParentFrame::OnExit)
    EVT_MENU_RANGE(wxID_FILE1, wxID_FILE9, wxDocParentFrame::OnMRUFile)
END_EVENT_TABLE()

void wxDocParentFrame::OnExit(wxCommandEvent& WXUNUSED(event))
{
    Close();
}

void wxDocParentFrame::OnMRUFile(wxCommandEvent& event)
{
    wxString f(m_docManager->GetHistoryFile(event.GetId() -
wxID_FILE1));
    if (f != "")
        (void)m_docManager->CreateDocument(f, wxDOC_SILENT);
}
```

wxWidgets predefined command identifiers

To allow communication between the application's menus and the document/view framework, several command identifiers are predefined for you to use in menus. The framework recognizes them and processes them if you forward commands from wxFrame::OnMenuCommand (or perhaps from toolbars and other user interface constructs).

- wxID_OPEN (5000)

- wxID_CLOSE (5001)

- wxID_NEW (5002)

- wxID_SAVE (5003)

- wxID_SAVEAS (5004)

- wxID_REVERT (5005)

- wxID_EXIT (5006)

- wxID_UNDO (5007)

- wxID_REDO (5008)

- wxID_HELP (5009)

- wxID_PRINT (5010)

- wxID_PRINT_SETUP (5011)

- wxID_PREVIEW (5012)

Toolbar overview

Classes: *wxToolBar* (p. 1401)

The toolbar family of classes allows an application to use toolbars in a variety of configurations and styles.

The toolbar is a popular user interface component and contains a set of bitmap buttons or toggles. A toolbar gives faster access to an application's facilities than menus, which have to be popped up and selected rather laboriously.

Instead of supplying one toolbar class with a number of different implementations depending on platform, wxWidgets separates out the classes. This is because there are a number of different toolbar styles that you may wish to use simultaneously, and also, future toolbar implementations will emerge which cannot all be shoe-horned into the one class.

For each platform, the symbol **wxToolBar** is defined to be one of the specific toolbar classes.

The following is a summary of the toolbar classes and their differences.

- **wxToolBarBase.** This is a base class with pure virtual functions, and should not be used directly.

- **wxToolBarSimple.** A simple toolbar class written entirely with generic wxWidgets functionality. A simple 3D effect for buttons is possible, but it is not consistent with the Windows look and feel. This toolbar can scroll, and you can have arbitrary numbers of rows and columns.

- **wxToolBarMSW.** This class implements an old-style Windows toolbar, only on Windows. There are small, three-dimensional buttons, which do not (currently) reflect the current Windows colour settings: the buttons are grey. This is the default wxToolBar on 16-bit windows.

- **wxToolBar95.** Uses the native Windows 95 toolbar class. It dynamically adjusts its background and button colours according to user colour settings. CreateTools must be called after the tools have been added. No absolute positioning is supported but you can specify the number of rows, and add tool separators with **AddSeparator**. Tooltips are supported. **OnRightClick** is not supported. This is the default wxToolBar on Windows 95, Windows NT 4 and

above. With the style wxTB_FLAT, the flat toolbar look is used, with a border that is highlighted when the cursor moves over the buttons.

A toolbar might appear as a single row of images under the menubar, or it might be in a separate frame layout in several rows and columns. The class handles the layout of the images, unless explicit positioning is requested.

A tool is a bitmap which can either be a button (there is no 'state', it just generates an event when clicked) or it can be a toggle. If a toggle, a second bitmap can be provided to depict the 'on' state; if the second bitmap is omitted, either the inverse of the first bitmap will be used (for monochrome displays) or a thick border is drawn around the bitmap (for colour displays where inverting will not have the desired result).

The Windows-specific toolbar classes expect 16-colour bitmaps that are 16 pixels wide and 15 pixels high. If you want to use a different size, call **SetToolBitmapSize** as the demo shows, before adding tools to the button bar. Don't supply more than one bitmap for each tool, because the toolbar generates all three images (normal, depressed and checked) from the single bitmap you give it.

Using the toolbar library

Include "`wx/toolbar.h`", or if using a class directly, one of:

- "`wx/msw/tbarmsw.h` for wxToolBarMSW

- "`wx/msw/tbar95.h` for wxToolBar95

- "`wx/tbarsmpl.h` for wxToolBarSimple

Example of toolbar use are given in the sample program "toolbar". The source is given below. In fact it is out of date because recommended practise is to use event handlers (using EVT_MENU or EVT_TOOL) instead of overriding OnLeftClick.

```
/////////////////////////////////////////////////////////////////////////
///
// Name:          test.cpp
// Purpose:       wxToolBar sample
// Author:        Julian Smart
// Modified by:
// Created:       04/01/98
// RCS-ID:        $Id: ttoolbar.tex,v 1.11 2005/02/22 15:09:55 ABX Exp $
// Copyright:     (c) Julian Smart
// License:       wxWindows license
/////////////////////////////////////////////////////////////////////////
///

// For compilers that support precompilation, includes "wx/wx.h".
#include "wx/wxprec.h"

#ifdef __BORLANDC__
#pragma hdrstop
#endif

#ifndef WX_PRECOMP
#include "wx/wx.h"
```

```
      #endif

      #include "wx/toolbar.h"
      #include <wx/log.h>

      #include "test.h"

      #if defined(__WXGTK__) || defined(__WXMOTIF__)
      #include "mondrian.xpm"
      #include "bitmaps/new.xpm"
      #include "bitmaps/open.xpm"
      #include "bitmaps/save.xpm"
      #include "bitmaps/copy.xpm"
      #include "bitmaps/cut.xpm"
      #include "bitmaps/print.xpm"
      #include "bitmaps/preview.xpm"
      #include "bitmaps/help.xpm"
      #endif

      IMPLEMENT_APP(MyApp)

      // The `main program' equivalent, creating the windows and returning the
      // main frame
      bool MyApp::OnInit(void)
      {
        // Create the main frame window
        MyFrame* frame = new MyFrame((wxFrame *) NULL, -1, (const wxString)
      "wxToolBar Sample",
            wxPoint(100, 100), wxSize(450, 300));

        // Give it a status line
        frame->CreateStatusBar();

        // Give it an icon
        frame->SetIcon(wxICON(mondrian));

        // Make a menubar
        wxMenu *fileMenu = new wxMenu;
        fileMenu->Append(wxID_EXIT, "E&xit", "Quit toolbar sample" );

        wxMenu *helpMenu = new wxMenu;
        helpMenu->Append(wxID_HELP, "&About", "About toolbar sample");

        wxMenuBar* menuBar = new wxMenuBar;

        menuBar->Append(fileMenu, "&File");
        menuBar->Append(helpMenu, "&Help");

        // Associate the menu bar with the frame
        frame->SetMenuBar(menuBar);

        // Create the toolbar
        frame->CreateToolBar(wxNO_BORDER|wxHORIZONTAL|wxTB_FLAT, ID_TOOLBAR);

        frame->GetToolBar()->SetMargins( 2, 2 );

        InitToolbar(frame->GetToolBar());

        // Force a resize. This should probably be replaced by a call to a
      wxFrame
        // function that lays out default decorations and the remaining content
      window.
        wxSizeEvent event(wxSize(-1, -1), frame->GetId());
```

```
   frame->OnSize(event);
   frame->Show(true);

   frame->SetStatusText("Hello, wxWidgets");

   SetTopWindow(frame);

   return true;
}

bool MyApp::InitToolbar(wxToolBar* toolBar)
{
   // Set up toolbar
   wxBitmap* toolBarBitmaps[8];

#ifdef __WXMSW__
   toolBarBitmaps[0] = new wxBitmap("icon1");
   toolBarBitmaps[1] = new wxBitmap("icon2");
   toolBarBitmaps[2] = new wxBitmap("icon3");
   toolBarBitmaps[3] = new wxBitmap("icon4");
   toolBarBitmaps[4] = new wxBitmap("icon5");
   toolBarBitmaps[5] = new wxBitmap("icon6");
   toolBarBitmaps[6] = new wxBitmap("icon7");
   toolBarBitmaps[7] = new wxBitmap("icon8");
#else
   toolBarBitmaps[0] = new wxBitmap( new_xpm );
   toolBarBitmaps[1] = new wxBitmap( open_xpm );
   toolBarBitmaps[2] = new wxBitmap( save_xpm );
   toolBarBitmaps[3] = new wxBitmap( copy_xpm );
   toolBarBitmaps[4] = new wxBitmap( cut_xpm );
   toolBarBitmaps[5] = new wxBitmap( preview_xpm );
   toolBarBitmaps[6] = new wxBitmap( print_xpm );
   toolBarBitmaps[7] = new wxBitmap( help_xpm );
#endif

#ifdef __WXMSW__
   int width = 24;
#else
   int width = 16;
#endif
   int currentX = 5;

   toolBar->AddTool(wxID_NEW, *(toolBarBitmaps[0]), wxNullBitmap, false,
currentX, -1, (wxObject *) NULL, "New file");
   currentX += width + 5;
   toolBar->AddTool(wxID_OPEN, *(toolBarBitmaps[1]), wxNullBitmap, false,
currentX, -1, (wxObject *) NULL, "Open file");
   currentX += width + 5;
   toolBar->AddTool(wxID_SAVE, *(toolBarBitmaps[2]), wxNullBitmap, false,
currentX, -1, (wxObject *) NULL, "Save file");
   currentX += width + 5;
   toolBar->AddSeparator();
   toolBar->AddTool(wxID_COPY, *(toolBarBitmaps[3]), wxNullBitmap, false,
currentX, -1, (wxObject *) NULL, "Copy");
   currentX += width + 5;
   toolBar->AddTool(wxID_CUT, *(toolBarBitmaps[4]), wxNullBitmap, false,
currentX, -1, (wxObject *) NULL, "Cut");
   currentX += width + 5;
   toolBar->AddTool(wxID_PASTE, *(toolBarBitmaps[5]), wxNullBitmap, false,
currentX, -1, (wxObject *) NULL, "Paste");
   currentX += width + 5;
   toolBar->AddSeparator();
   toolBar->AddTool(wxID_PRINT, *(toolBarBitmaps[6]), wxNullBitmap, false,
```

```
currentX, -1, (wxObject *) NULL, "Print");
  currentX += width + 5;
  toolBar->AddSeparator();
  toolBar->AddTool(wxID_HELP, *(toolBarBitmaps[7]), wxNullBitmap, false,
currentX, -1, (wxObject *) NULL, "Help");

  toolBar->Realize();

  // Can delete the bitmaps since they're reference counted
  int i;
  for (i = 0; i < 8; i++)
    delete toolBarBitmaps[i];

  return true;
}

// wxID_HELP will be processed for the 'About' menu and the toolbar help
button.

BEGIN_EVENT_TABLE(MyFrame, wxFrame)
    EVT_MENU(wxID_EXIT, MyFrame::OnQuit)
    EVT_MENU(wxID_HELP, MyFrame::OnAbout)
    EVT_CLOSE(MyFrame::OnCloseWindow)
    EVT_TOOL_RANGE(wxID_OPEN, wxID_PASTE, MyFrame::OnToolLeftClick)
    EVT_TOOL_ENTER(wxID_OPEN, MyFrame::OnToolEnter)
END_EVENT_TABLE()

// Define my frame constructor
MyFrame::MyFrame(wxFrame* parent, wxWindowID id, const wxString& title,
const wxPoint& pos,
        const wxSize& size, long style):
  wxFrame(parent, id, title, pos, size, style)
{
  m_textWindow = new wxTextCtrl(this, -1, "", wxPoint(0, 0), wxSize(-1, -
1), wxTE_MULTILINE);
}

void MyFrame::OnQuit(wxCommandEvent& WXUNUSED(event))
{
    Close(true);
}

void MyFrame::OnAbout(wxCommandEvent& WXUNUSED(event))
{
    (void)wxMessageBox("wxWidgets toolbar sample", "About wxToolBar");
}

// Define the behaviour for the frame closing
// - must delete all frames except for the main one.
void MyFrame::OnCloseWindow(wxCloseEvent& WXUNUSED(event))
{
  Destroy();
}

void MyFrame::OnToolLeftClick(wxCommandEvent& event)
{
  wxString str;
  str.Printf("Clicked on tool %d", event.GetId());
  SetStatusText(str);
}

void MyFrame::OnToolEnter(wxCommandEvent& event)
{
```

```
      if (event.GetSelection() > -1)
      {
        wxString str;
        str.Printf("This is tool number %d", event.GetSelection());
        SetStatusText(str);
      }
      else
        SetStatusText("");
    }
```

wxGrid classes overview

Classes: *wxGrid* (p. 619)

Introduction

wxGrid and its related classes are used for displaying and editing tabular data.

Getting started: a simple example

For simple applications you need only refer to the wxGrid class in your code. This
example shows how you might create a grid in a frame or dialog constructor and
illustrates some of the formatting functions.

```
        // Create a wxGrid object

        grid = new wxGrid( this,
                           -1,
                           wxPoint( 0, 0 ),
                           wxSize( 400, 300 ) );

        // Then we call CreateGrid to set the dimensions of the grid
        // (100 rows and 10 columns in this example)
        grid->CreateGrid( 100, 10 );

        // We can set the sizes of individual rows and columns
        // in pixels
        grid->SetRowSize( 0, 60 );
        grid->SetColSize( 0, 120 );

        // And set grid cell contents as strings
        grid->SetCellValue( 0, 0, "wxGrid is good" );

        // We can specify that some cells are read-only
        grid->SetCellValue( 0, 3, "This is read-only" );
        grid->SetReadOnly( 0, 3 );

        // Colours can be specified for grid cell contents
        grid->SetCellValue(3, 3, "green on grey");
        grid->SetCellTextColour(3, 3, *wxGREEN);
        grid->SetCellBackgroundColour(3, 3, *wxLIGHT_GREY);

        // We can specify the some cells will store numeric
```

```
// values rather than strings. Here we set grid column 5
// to hold floating point values displayed with width of 6
// and precision of 2
grid->SetColFormatFloat(5, 6, 2);
grid->SetCellValue(0, 6, "3.1415");
```

A more complex example

Yet to be written

How the wxGrid classes relate to each other

Yet to be written

Keyboard and mouse actions

Yet to be written

wxTipProvider overview

Many "modern" Windows programs have a feature (some would say annoyance) of presenting the user tips at program startup. While this is probably useless to the advanced users of the program, the experience shows that the tips may be quite helpful for the novices and so more and more programs now do this.

For a wxWidgets programmer, implementing this feature is extremely easy. To show a tip, it is enough to just call *wxShowTip* (p. 1622) function like this:

```
if ( ...show tips at startup?... )
{
    wxTipProvider *tipProvider = wxCreateFileTipProvider("tips.txt",
0);
    wxShowTip(windowParent, tipProvider);
    delete tipProvider;
}
```

Of course, you need to get the text of the tips from somewhere - in the example above, the text is supposed to be in the file tips.txt from where it is read by the *tip provider*. The tip provider is just an object of a class deriving from *wxTipProvider* (p. 1395). It has to implement one pure virtual function of the base class: *GetTip* (p. 1396). In the case of the tip provider created by *wxCreateFileTipProvider* (p. 1614), the tips are just the lines of the text file.

If you want to implement your own tip provider (for example, if you wish to hardcode the tips inside your program), you just have to derive another class from wxTipProvider and pass a pointer to the object of this class to wxShowTip - then you don't need wxCreateFileTipProvider at all.

You will probably want to save somewhere the index of the tip last shown - so that the program doesn't always show the same tip on startup. As you also need to remember whether to show tips or not (you shouldn't do it if the user unchecked "Show tips on startup" checkbox in the dialog), you will probably want to store both the index of the last shown tip (as returned by *wxTipProvider::GetCurrentTip* (p. 1397) and the flag telling whether to show the tips at startup at all.

In a tips.txt file, lines that begin with a # character are considered comments and are automatically skipped. Blank lines and lines only having spaces are also skipped.

You can easily add runtime-translation capacity by placing each line of the tips.txt file inside the usual translation macro. For example, your tips.txt file would look like this:

```
_("This is my first tip")
_("This is my second tip")
```

Now add your tips.txt file into the list of files that gettext searches for translatable strings. The tips will thus get included into your generated .po file catalog and be translated at runtime along with the rest of your application's translatable strings. Note1: Each line in the tips.txt file needs to strictly begin with exactly the 3 characters of underscore-parenthesis-doublequote, and end with doublequote-parenthesis, as shown above. Note2: Remember to escape any doublequote characters within the tip string with a backslash-doublequote.

See the dialogs program in your samples folder for a working example inside a program.

Printing overview

Classes: *wxPrintout* (p. 1065), *wxPrinter* (p. 1062), *wxPrintPreview* (p. 1069), *wxPrinterDC* (p. 1065), *wxPrintDialog* (p. 1057), *wxPrintData* (p. 1051), *wxPrintDialogData* (p. 1058), *wxPageSetupDialog* (p. 1018), *wxPageSetupDialogData* (p. 1019)

The printing framework relies on the application to provide classes whose member functions can respond to particular requests, such as 'print this page' or 'does this page exist in the document?'. This method allows wxWidgets to take over the housekeeping duties of turning preview pages, calling the print dialog box, creating the printer device context, and so on: the application can concentrate on the rendering of the information onto a device context.

The *document/view framework* (p. 1802) creates a default wxPrintout object for every view, calling wxView::OnDraw to achieve a prepackaged print/preview facility.

A document's printing ability is represented in an application by a derived wxPrintout class. This class prints a page on request, and can be passed to the Print function of a wxPrinter object to actually print the document, or can be passed to a wxPrintPreview object to initiate previewing. The following code (from the printing sample) shows how easy it is to initiate printing, previewing and the print setup dialog, once the wxPrintout functionality has been defined. Notice the use of MyPrintout for both printing and previewing. All the preview user interface functionality is taken care of by wxWidgets.

For details on how MyPrintout is defined, please look at the printout sample code.

```
        case WXPRINT_PRINT:
        {
          wxPrinter printer;
          MyPrintout printout("My printout");
          printer.Print(this, &printout, true);
          break;
        }
        case WXPRINT_PREVIEW:
        {
          // Pass two printout objects: for preview, and possible printing.
          wxPrintPreview *preview = new wxPrintPreview(new MyPrintout, new
      MyPrintout);
          wxPreviewFrame *frame = new wxPreviewFrame(preview, this, "Demo
      Print Preview", wxPoint(100, 100), wxSize(600, 650));
          frame->Centre(wxBOTH);
          frame->Initialize();
          frame->Show(true);
          break;
        }
```

Printing under Unix (GTK+)

Printing under Unix has always been a cause of problems as Unix does not provide a standard way to display text and graphics on screen and print it to a printer using the same application programming interface - instead, displaying on screen is done via the X11 library while printing has to be done with using PostScript commands. This was particularly difficult to handle for the case of fonts with the result that only a selected number of application could offer WYSIWYG under Unix. Equally, wxWidgets offered its own printing implementation using PostScript which never really matched the screen display.

Starting with version 2.8.X, the GNOME project provides printing support through the libgnomeprint and libgnomeprintui libraries by which especially the font problem is mostly solved. Beginning with version 2.5.4, the GTK+ port of wxWidgets can make use of these libraries if wxWidgets is configured accordingly and if the libraries are present. You need to configure wxWidgets with the*configure --with-gnomeprint* switch and you application will then search for the GNOME print libraries at runtime. If they are found, printing will be done through these, otherwise the application will fall back to the old PostScript printing code. Note that the application will not require the GNOME print libraries to be installed in order to run (there will be no dependency on these libraries).

It is expected that the printing code that is currently implemented in the GNOME print libraries will be moved into GTK+ later.

Multithreading overview

Classes: *wxThread* (p. 1377), *wxMutex* (p. 994), *wxCriticalSection* (p. 223), *wxCondition* (p. 189)

wxWidgets provides a complete set of classes encapsulating objects necessary in

multithreaded (MT) programs: the *thread* (p. 1377) class itself and different synchronization objects: *mutexes* (p. 994) and *critical sections* (p. 223) with *conditions* (p. 189). The thread API in wxWidgets resembles to POSIX1.c threads API (a.k.a. pthreads), although several functions have different names and some features inspired by Win32 thread API are there as well.

These classes will hopefully make writing MT programs easier and they also provide some extra error checking (compared to the native (be it Win32 or Posix) thread API), however it is still an non-trivial undertaking especially for large projects. Before starting an MT application (or starting to add MT features to an existing one) it is worth asking oneself if there is no easier and safer way to implement the same functionality. Of course, in some situations threads really make sense (classical example is a server application which launches a new thread for each new client), but in others it might be a very poor choice (example: launching a separate thread when doing a long computation to show a progress dialog). Other implementation choices are available: for the progress dialog example it is far better to do the calculations in the *idle handler* (p. 776) or even simply do everything at once but call *wxWindow::Update()* (p. 1546) periodically to update the screen.

If you do decide to use threads in your application, it is strongly recommended that no more than one thread calls GUI functions. The thread sample shows that it *is* possible for many different threads to call GUI functions at once (all the threads created in the sample access GUI), but it is a very poor design choice for anything except an example. The design which uses one GUI thread and several worker threads which communicate with the main one using events is much more robust and will undoubtedly save you countless problems (example: under Win32 a thread can only access GDI objects such as pens, brushes, &c created by itself and not by the other threads).

For communication between secondary threads and the main thread, you may use *wxEvtHandler::AddPendingEvent* (p. 487)or its short version *wxPostEvent* (p. 1638). These functions have thread safe implementation so that they can be used as they are for sending events from one thread to another. However there is no built in method to send messages to the worker threads and you will need to use the available synchronization classes to implement the solution which suits your needs yourself. In particular, please note that it is *not* enough to derive your class from *wxThread* (p. 1377) and *wxEvtHandler* (p. 486) to send messages to it: in fact, this does*not* work at all.

Drag and drop overview

Classes: *wxDataObject* (p. 239), *wxTextDataObject* (p. 1359), *wxDropSource* (p. 468), *wxDropTarget* (p. 471), *wxTextDropTarget* (p. 1360), *wxFileDropTarget* (p. 516)

Note that wxUSE_DRAG_AND_DROP must be defined in setup.h in order to use drag and drop in wxWidgets.

See also: *wxDataObject overview* (p. 1819) and *DnD sample* (p. 1710)

It may be noted that data transfer to and from the clipboard is quite similar to data transfer with drag and drop and the code to implement these two types is almost the same. In particular, both data transfer mechanisms store data in some kind of *wxDataObject* (p. 239)and identify its format(s) using the *wxDataFormat* (p. 234)class.

To be a *drag source*, i.e. to provide the data which may be dragged by the user elsewhere, you should implement the following steps:

- **Preparation:** First of all, a data object must be created and initialized with the data you wish to drag. For example:

```
wxTextDataObject my_data("This text will be dragged.");
```

- **Drag start:** To start the dragging process (typically in response to a mouse click) you must call *wxDropSource::DoDragDrop* (p. 470)like this:

```
wxDropSource dragSource( this );
dragSource.SetData( my_data );
wxDragResult result = dragSource.DoDragDrop( TRUE );
```

- **Dragging:** The call to DoDragDrop() blocks the program until the user releases the mouse button (unless you override the *GiveFeedback* (p. 470) function to do something special). When the mouse moves in a window of a program which understands the same drag-and-drop protocol (any program under Windows or any program supporting the XDnD protocol under X Windows), the corresponding *wxDropTarget* (p. 471) methods are called - see below.

- **Processing the result:** DoDragDrop() returns an *effect code* which is one of the values of `wxDragResult` enum (explained *here* (p. 471)):

```
switch (result)
{
    case wxDragCopy: /* copy the data */ break;
    case wxDragMove: /* move the data */ break;
    default:         /* do nothing */ break;
}
```

To be a *drop target*, i.e. to receive the data dropped by the user you should follow the instructions below:

- **Initialization:** For a window to be a drop target, it needs to have an associated *wxDropTarget* (p. 471) object. Normally, you will call *wxWindow::SetDropTarget* (p. 1532) during window creation associating your drop target with it. You must derive a class from wxDropTarget and override its pure virtual methods. Alternatively, you may derive from *wxTextDropTarget* (p. 1360) or*wxFileDropTarget* (p. 516) and override their OnDropText() or OnDropFiles() method.

- **Drop:** When the user releases the mouse over a window, wxWidgets asks the associated wxDropTarget object if it accepts the data. For this, a *wxDataObject* (p. 239) must be associated with the drop target and this data object will be responsible for the format negotiation between the drag source and the drop target. If all goes well, then *OnData* (p. 472) will get called and the

wxDataObject belonging to the drop target can get filled with data.

- **The end:** After processing the data, DoDragDrop() returns either wxDragCopy or wxDragMove depending on the state of the keys <Ctrl>, <Shift> and <Alt> at the moment of the drop. There is currently no way for the drop target to change this return code.

wxDataObject overview

Classes: *wxDataObject* (p. 239), *wxClipboard* (p. 150), *wxDataFormat* (p. 234), *wxDropSource* (p. 468), *wxDropTarget* (p. 471)

See also: *Drag and drop overview* (p. 1817) and *DnD sample* (p. 1710)

This overview discusses data transfer through clipboard or drag and drop. In wxWidgets, these two ways to transfer data (either between different applications or inside one and the same) are very similar which allows to implement both of them using almost the same code - or, in other words, if you implement drag and drop support for your application, you get clipboard support for free and vice versa.

At the heart of both clipboard and drag and drop operations lies the *wxDataObject* (p. 239) class. The objects of this class (or, to be precise, classes derived from it) represent the data which is being carried by the mouse during drag and drop operation or copied to or pasted from the clipboard. wxDataObject is a "smart" piece of data because it knows which formats it supports (see GetFormatCount and GetAllFormats) and knows how to render itself in any of them (see GetDataHere). It can also receive its value from the outside in a format it supports if it implements the SetData method. Please see the documentation of this class for more details.

Both clipboard and drag and drop operations have two sides: the source and target, the data provider and the data receiver. These which may be in the same application and even the same window when, for example, you drag some text from one position to another in a word processor. Let us describe what each of them should do.

The data provider (source) duties

The data provider is responsible for creating a *wxDataObject* (p. 239) containing the data to be transferred. Then it should either pass it to the clipboard using *SetData* (p. 152) function or to *wxDropSource* (p. 468) and call *DoDragDrop* (p. 470) function.

The only (but important) difference is that the object for the clipboard transfer must always be created on the heap (i.e. using `new`) and it will be freed by the clipboard when it is no longer needed (indeed, it is not known in advance when, if ever, the data will be pasted from the clipboard). On the other hand, the object for drag and drop operation must only exist while *DoDragDrop* (p. 470) executes and may be safely deleted afterwards and so can be created either on heap or on stack (i.e. as a local variable).

Another small difference is that in the case of clipboard operation, the application usually knows in advance whether it copies or cuts (i.e. copies and deletes) data - in fact, this usually depends on which menu item the user chose. But for drag and drop it can only

know it after *DoDragDrop* (p. 470) returns (from its return value).

The data receiver (target) duties

To receive (paste in usual terminology) data from the clipboard, you should create a *wxDataObject* (p. 239) derived class which supports the data formats you need and pass it as argument to *wxClipboard::GetData* (p. 152). If it returns `false`, no data in (any of) the supported format(s) is available. If it returns `true`, the data has been successfully transferred to wxDataObject.

For drag and drop case, the *wxDropTarget::OnData* (p. 472) virtual function will be called when a data object is dropped, from which the data itself may be requested by calling *wxDropTarget::GetData* (p. 472) method which fills the data object.

Database classes overview

Following is a detailed overview of how to use the wxWidgets ODBC classes - *wxDb* (p. 284) and *wxDbTable* (p. 324) and their associated functions. These are the ODBC classes donated by Remstar International, and are collectively referred to herein as the wxODBC classes.

wxDb/wxDbTable wxODBC Overview

Classes: *wxDb* (p. 284), *wxDbTable* (p. 324)

The wxODBC classes were designed for database independence. Although SQL and ODBC both have standards which define the minimum requirements they must support to be in compliance with specifications, different database vendors may implement things slightly differently. One example of this is that Oracle requires all user names for the datasources to be supplied in uppercase characters. In situations like this, the wxODBC classes have been written to make this transparent to the programmer when using functions that require database-specific syntax.

Currently several major databases, along with other widely used databases, have been tested and supported through the wxODBC classes. The list of supported databases is certain to grow as more users start implementing software with these classes, but at the time of the writing of this document, users have successfully used the classes with the following datasources:

- DB2

- DBase (IV, V)**

- Firebird

- INFORMIX

- Interbase

- MS SQL Server (v7 - minimal testing)

- MS Access (97, 2000, 2002, and 2003)

- MySQL (2.x and 3.5 - use the 2.5x drivers though)

- Oracle (v7, v8, v8i)

- Pervasive SQL

- PostgreSQL

- Sybase (ASA and ASE)

- XBase Sequiter

- VIRTUOSO

An up-to-date list can be obtained by looking in the comments of the function *wxDb::Dbms* (p. 295) in db.cpp, or in the enumerated type *wxDBMS* (p. 285) in db.h.

**dBase is not truly an ODBC datasource, but there are drivers which can emulate much of the functionality of an ODBC connection to a dBase table. See the *wxODBC Known Issues* (p. 1834) section of this overview for details.

wxODBC Where To Start

First, if you are not familiar with SQL and ODBC, go to your local bookstore and pick up a good book on each. This documentation is not meant to teach you many details about SQL or ODBC, though you may learn some just from immersion in the subject.

If you have worked with non-SQL/ODBC datasources before, there are some things you will need to un-learn. First some terminology as these phrases will be used heavily in this section of the manual.

Datasource	(usually a database) that contains the data that will be accessed by the wxODBC classes.
Data table	The section of the datasource that contains the rows and columns of data.
ODBC driver	The middle-ware software that interprets the ODBC commands sent by your application and converts them to the SQL format expected by the target datasource.
Datasource connection	An open pipe between your application and the ODBC driver which in turn has a connection to the target datasource. Datasource connections can have a virtually unlimited number of wxDbTable instances using the same connect (dependent on the ODBC driver). A separate connection is not needed for each table (the exception is for isolating commits/rollbacks on different tables from

	affecting more than the desired table. See the class documentation on *wxDb::CommitTrans* (p. 293) and *wxDb::RollbackTrans* (p. 310).)
Rows	Similar to records in old relational databases, a row is a collection of one instance of each column of the data table that are all associated with each other.
Columns	Individual fields associated with each row of a data table.
Query	Request from the client to the datasource asking for the data that matches the requirements specified in the users request. When a query is performed, the datasource performs the lookup of the rows with satisfy the query, and creates a result set.
Result set	The data which matches the requirements specified in a query sent to the datasource. Dependent on drivers, a result set typically remains at the datasource (no data is transmitted to the ODBC driver) until the client actually instructs the ODBC driver to retrieve it.
Cursor	A logical pointer into the result set that a query generates, indicating the next record that will be returned to the client when a request for the next record is made.
Scrolling cursors	Scrolling refers to the movement of cursors through the result set. Cursors can always scroll forward sequentially in the result set (FORWARD ONLY scrolling cursors). With Forward only scrolling cursors, once a row in the result set has been returned to the ODBC driver and on to the client, there is no way to have the cursor move backward in the result set to look at the row that is previous to the current row in the result set. If BACKWARD scrolling cursors are supported by both the ODBC driver and the datasource that are being used, then backward scrolling cursor functions may be used (*wxDbTable::GetPrev* (p. 342), *wxDbTable::GetFirst* (p. 339), and *wxDbTable::GetLast* (p. 340)). If the datasource or the ODBC driver only support forward scrolling cursors, your program and logic must take this in to account.
Commit/Rollback	Commit will physically save insertions/deletions/updates, while rollback basically does an undo of everything done against the datasource connection that has not been previously committed. Note that Commit and Rollbacks are done on a connection, not on individual tables. All tables which use a shared connection to the datasource are all committed/rolled back at the same time when a call to *wxDb::CommitTrans* (p. 293) or *wxDb::RollbackTrans* (p. 310) is made.

Index   Indexes are datasource-maintained lookup structures that
                             allow the datasource to quickly locate data rows based on
                             the values of certain columns. Without indexes, the
                             datasource would need to do a sequential search of a
                             table every time a query request is made. Proper unique
                             key index construction can make datasource queries
                             nearly instantaneous.

Before you are able to read data from a data table in a datasource, you must have a
connection to the datasource. Each datasource connection may be used to open
multiple tables all on the same connection (number of tables open are dependent on the
driver, datasource configuration and the amount of memory on the client workstation).
Multiple connections can be opened to the same datasource by the same client
(number of concurrent connections is dependent on the driver and datasource
configuration).

When a query is performed, the client passes the query to the ODBC driver, and the
driver then translates it and passes it along to the datasource. The database engine (in
most cases - exceptions are text and dBase files) running on the machine hosting the
database does all the work of performing the search for the requested data. The client
simply waits for a status to come back through the ODBC driver from the datasource.

Depending on the ODBC driver, the result set either remains "queued" on the database
server side, or is transferred to the machine that the driver is queued on. The client
does not receive this data. The client must request some or all of the result set to be
returned before any data rows are returned to the client application.

Result sets do not need to include all columns of every row matching the query. In fact,
result sets can actually be joinings of columns from two or more data tables, may have
derived column values, or calculated values returned.

For each result set, a cursor is maintained (typically by the database) which keeps track
of where in the result set the user currently is. Depending on the database, ODBC
driver, and how you configured the wxWidgets ODBC settings in setup.h (see *wxODBC
- Compiling* (p. 1825)), cursors can be either forward or backward scrolling. At a
minimum, cursors must scroll forward. For example, if a query resulted in a result set
with 100 rows, as the data is read by the client application, it will read row 1, then 2,
then 3, etc. With forward only cursors, once the cursor has moved to the next row, the
previous row cannot be accessed again without re-querying the datasource for the
result set over again. Backward scrolling cursors allow you to request the previous row
from the result set, actually scrolling the cursor backward.

Backward scrolling cursors are not supported on all database/driver combinations. For
this reason, forward-only cursors are the default in the wxODBC classes. If your
datasource does support backward scrolling cursors and you wish to use them, make
the appropriate changes in setup.h to enable them (see *wxODBC - Compiling* (p.
1825)). For greatest portability between datasources, writing your program in such a
way that it only requires forward scrolling cursors is your best bet. On the other hand, if
you are focusing on using only datasources that support backward scrolling cursors,
potentially large performance benefits can be gained from using them.

There is a limit to the number of cursors that can be open on each connection to the

datasource, and usually a maximum number of cursors for the datasource itself. This is all dependent on the database. Each connection that is opened (each instance of a wxDb) opens a minimum of 5 cursors on creation that are required for things such as updates/deletions/rollbacks/queries. Cursors are a limited resource, so use care in creating large numbers of cursors.

Additional cursors can be created if necessary with the *wxDbTable::GetNewCursor* (p. 340) function. One example use for additional cursors is to track multiple scroll points in result sets. By creating a new cursor, a program could request a second result set from the datasource while still maintaining the original cursor position in the first result set.

Different than non-SQL/ODBC datasources, when a program performs an insertion, deletion, or update (or other SQL functions like altering tables, etc) through ODBC, the program must issue a "commit" to the datasource to tell the datasource that the action(s) it has been told to perform are to be recorded as permanent. Until a commit is performed, any other programs that query the datasource will not see the changes that have been made (although there are databases that can be configured to auto-commit). NOTE: With most datasources, until the commit is performed, any cursor that is open on that same datasource connection will be able to see the changes that are uncommitted. Check your database's documentation/configuration to verify this before relying on it though.

A rollback is basically an UNDO command on the datasource connection. When a rollback is issued, the datasource will flush all commands it has been told to do since the last commit that was performed.

NOTE: Commits/Rollbacks are done on datasource connections (wxDb instances) not on the wxDbTable instances. This means that if more than one table shares the same connection, and a commit or rollback is done on that connection, all pending changes for ALL tables using that connection are committed/rolled back.

wxODBC - Configuring your system for ODBC use

Before you are able to access a datasource, you must have installed and configured an ODBC driver. Doing this is system specific, so it will not be covered in detail here. But here are a few details to get you started.

Most database vendors provide at least a minimal ODBC driver with their database product. In practice, many of these drivers have proven to be slow and/or incomplete. Rumour has it that this is because the vendors do not want you using the ODBC interface to their products; they want you to use their applications to access the data.

Whatever the reason, for database-intensive applications, you may want to consider using a third-party ODBC driver for your needs. One example of a third-party set of ODBC drivers that has been heavily tested and used is Rogue Wave's drivers. Rogue Wave has drivers available for many different platforms and databases. Under Microsoft Windows, install the ODBC driver you are planning to use. You will then use the ODBC Administrator in the Control Panel to configure an instance of the driver for your intended datasource. Note that with all flavors of NT, this configuration can be set up as a System or User DSN (datasource name). Configuring it as a system resource will make it available to all users (if you are logged in as 'administrator'), otherwise the

datasource will only be available to the user who configured the DSN.

Under Unix, iODBC is used for implementation of the ODBC API. To compile the wxODBC classes, you must first obtain iODBC from http://www.iodbc.org (`www.iodbc.org`) and install it. (Note: wxWidgets currently includes a version of iODBC.) Then you must create the file "~/.odbc.ini" (or optionally create "/etc/odbc.ini" for access for all users on the system). This file contains the settings for your system/datasource. Below is an example section of a odbc.ini file for use with the "samples/db" sample program using MySQL:

```
[contacts]
Trace    = Off
TraceFile= stderr
Driver   = /usr/local/lib/libmyodbc.so
DSN      = contacts
SERVER   = 192.168.1.13
USER     = qet
PASSWORD =
PORT     = 3306
```

wxODBC - Compiling

The wxWidgets setup.h file has several settings in it pertaining to compiling the wxODBC classes.

wxUSE_ODBC   This must be set to 1 in order for the compiler to compile the wxODBC classes. Without setting this to 1, there will be no access to any of the wxODBC classes. The default is 0.

wxODBC_FWD_ONLY_CURSORS When a new database connection is requested, this setting controls the default of whether the connection allows only forward scrolling cursors, or forward and backward scrolling cursors (see the section in "WHERE TO START" on cursors for more information on cursors). This default can be overridden by passing a second parameter to either the *wxDbGetConnection* (p. 289) or *wxDb constructor* (p. 291). The default is 1.

wxODBC_BACKWARD_COMPATABILITY Between v2.0 and 2.2, massive renaming efforts were done to the ODBC classes to get naming conventions similar to those used throughout wxWidgets, as well as to preface all wxODBC classes names and functions with a wxDb preface. Because this renaming would affect applications written using the v2.0 names, this compile-time directive was added to allow those programs written for v2.0 to still compile using the old naming conventions. These deprecated names are all #define'd to their corresponding new function names at the end of the db.cpp/dbtable.cpp source files. These deprecated class/function names should not be used in future development, as at some point in the future they will be removed. The default is 0.

Under MS Windows

You are required to include the "odbc32.lib" provided by your compiler vendor in the list of external libraries to be linked in. If using the makefiles supplied with wxWidgets, this library should already be included for use with makefile.b32, makefile.vc, and makefile.g95.

MORE TO COME

Under Unix--with-odbc flag for configure

MORE TO COME

wxODBC - Basic Step-By-Step Guide

To use the classes in an application, there are eight basic steps:

- Define datasource connection information

- Get a datasource connection

- Create table definition

- Open the table

- Use the table

- Close the table

- Close the datasource connection

- Release the ODBC environment handle

Following each of these steps is detailed to explain the step, and to hopefully mention as many of the pitfalls that beginning users fall in to when first starting to use the classes. Throughout the steps, small snippets of code are provided to show the syntax of performing the step. A complete code snippet is provided at the end of this overview that shows a complete working flow of all these steps (see *wxODBC - Sample Code* (p. 1836)).

Define datasource connection information

To be able to connect to a datasource through the ODBC driver, a program must supply a minimum of three pieces of information: Datasource name, User ID, and Authorization string (password). A fourth piece of information, a default directory indicating where the data file is stored, is required for Text and dBase drivers for ODBC.

The wxWidgets data class wxDbConnectInf exists for holding all of these values, plus some others that may be desired.

The 'Henv' member is the environment handle used to access memory for use by the ODBC driver. Use of this member is described below in the "Getting a Connection to the

Datasource" section.

The 'Dsn' must exactly match the datasource name used to configure the ODBC datasource (in the ODBC Administrator (MSW only) or in the .odbc.ini file).

The 'Uid' is the User ID that is to be used to log in to the datasource. This User ID must already have been created and assigned rights within the datasource to which you are connecting. The user that the connection is establish by will determine what rights and privileges the datasource connection will allow the program to have when using the connection that this connection information was used to establish. Some datasources are case sensitive for User IDs, and though the wxODBC classes attempt to hide this from you by manipulating whatever data you pass in to match the datasource's needs, it is always best to pass the 'Uid' in the case that the datasource requires.

The 'AuthStr' is the password for the User ID specified in the 'Uid' member. As with the 'Uid', some datasources are case sensitive (in fact most are). The wxODBC classes do NOT try to manage the case of the 'AuthStr' at all. It is passed verbatim to the datasource, so you must use the case that the datasource is expecting.

The 'defaultDir' member is used with file based datasources (i.e. dBase, FoxPro, text files). It contains a full path to the location where the data table or file is located. When setting this value, use forward slashes '/' rather than backslashes ' avoid compatibility differences between ODBC drivers.

The other fields are currently unused. The intent of these fields are that they will be used to write our own ODBC Administrator type program that will work on both MSW and Un*x systems, regardless of the datasource. Very little work has been done on this to date.

Get a Datasource Connection

There are two methods of establishing a connection to a datasource. You may either manually create your own wxDb instance and open the connection, or you may use the caching functions provided with the wxODBC classes to create/maintain/delete the connections.

Regardless of which method you use, you must first have a fully populated wxDbConnectInf object. In the wxDbConnectInf instance, provide a valid Dns, Uid, and AuthStr (along with a 'defaultDir' if necessary). Before using this though, you must allocate an environment handle to the 'Henv' member.

```
wxDbConnectInf DbConnectInf;
DbConnectInf.SetDsn("MyDSN");
DbConnectInf.SetUserID("MyUserName");
DbConnectInf.SetPassword("MyPassword");
DbConnectInf.SetDefaultDir("");
```

To allocate an environment handle for the ODBC connection to use, the wxDbConnectInf class has a datasource independent method for creating the necessary handle:

```
if (DbConnectInf.AllocHenv())
{
    wxMessageBox("Unable to allocate an ODBC environment handle",
                "DB CONNECTION ERROR", wxOK | wxICON_EXCLAMATION);
```

```
            return;
      }
```

When the wxDbConnectInf::AllocHenv() function is called successfully, a value of true will be returned. A value of false means allocation failed, and the handle will be undefined.

A shorter form of doing the above steps is encapsulated into the long form of the constructor for wxDbConnectInf.

```
      wxDbConnectInf *DbConnectInf;

    DbConnectInf = new wxDbConnectInf(NULL, "MyDSN", "MyUserName",
                                   "MyPassword", "");
```

This shorthand form of initializing the constructor passes a NULL for the SQL environment handle, telling the constructor to allocate a handle during construction. This handle is also managed for the life of wxDbConnectInf instance, and is freed automatically upon destruction of the instance.

Once the wxDbConnectInf instance is initialized, you are ready to connect to the datasource.

To manually create datasource connections, you must create a wxDb instance, and then open it.

```
      wxDb *db = new wxDb(DbConnectInf->GetHenv());

      opened = db->Open(DbConnectInf);
```

The first line does the house keeping needed to initialize all the members of the wxDb class. The second line actually sends the request to the ODBC driver to open a connection to its associated datasource using the parameters supplied in the call to *wxDb::Open* (p. 307).

A more advanced form of opening a connection is to use the connection caching functions that are included with the wxODBC classes. The caching mechanisms perform the same functions as the manual approach to opening a connection, but they also manage each connection they have created, re-using them and cleaning them up when they are closed, without you needing to do the coding.

To use the caching function *wxDbGetConnection* (p. 289) to get a connection to a datasource, simply call it with a single parameter of the type wxDbConnectInf:

```
      db = wxDbGetConnection(DbConnectInf);
```

The wxDb pointer that is returned is both initialized and opened. If something failed in creating or opening the connection, the return value from *wxDbGetConnection* (p. 289) will be NULL.

The connection that is returned is either a new connection, or it is a "free" connection from the cache of connections that the class maintains that was no longer in use. Any wxDb instance created with a call to *wxDbGetConnection* (p. 289) is recorded in a linked list of established connections. When a program is finished with a connection, a call to *wxDbFreeConnection* (p. 289) is made, and the datasource connection will then be

tagged as FREE, making it available for the next call to *wxDbGetConnection* (p. 289) that needs a connection using the same connection information (Dsn, Uid, AuthStr). The cached connections remain cached until a call to *wxDbCloseConnections* (p. 289) is made, at which time all cached connections are closed and deleted.

Besides the obvious advantage of using the single command caching routine to obtain a datasource connection, using cached connections can be quite a performance boost as well. Each time that a new connection is created (not retrieved from the cache of free connections), the wxODBC classes perform many queries against the datasource to determine the datasource's datatypes and other fundamental behaviours. Depending on the hardware, network bandwidth, and datasource speed, this can in some cases take a few seconds to establish the new connection (with well-balanced systems, it should only be a fraction of a second). Re-using already established datasource connections rather than creating/deleting, creating/deleting connections can be quite a time-saver.

Another time-saver is the "copy connection" features of both *wxDb::Open* (p. 307) and *wxDbGetConnection* (p. 289). If manually creating a wxDb instance and opening it, you must pass an existing connection to the *wxDb::Open* (p. 307) function yourself to gain the performance benefit of copying existing connection settings. The *wxDbGetConnection* (p. 289) function automatically does this for you, checking the Dsn, Uid, and AuthStr parameters when you request a connection for any existing connections that use those same settings. If one is found, *wxDbGetConnection* (p. 289) copies the datasource settings for datatypes and other datasource specific information that was previously queried, rather than re-querying the datasource for all those same settings.

One final note on creating a connection. When a connection is created, it will default to only allowing cursor scrolling to be either forward only, or both backward and forward scrolling. The default behavior is determined by the setting `wxODBC_FWD_ONLY_CURSORS` in setup.h when you compile the wxWidgets library. The library default is to only support forward scrolling cursors only, though this can be overridden by parameters for wxDb() constructor or the *wxDbGetConnection* (p. 289) function. All datasources and ODBC drivers must support forward scrolling cursors. Many datasources support backward scrolling cursors, and many ODBC drivers support backward scrolling cursors. Before planning on using backward scrolling cursors, you must be certain that both your datasource and ODBC driver fully support backward scrolling cursors. See the small blurb about "Scrolling cursors" in the definitions at the beginning of this overview, or other details of setting the cursor behavior in the wxDb class documentation.

Create Table Definition

Data can be accessed in a datasource's tables directly through various functions of the wxDb class (see *wxDb::GetData* (p. 300)). But to make life much simpler, the wxDbTable class encapsulates all of the SQL specific API calls that would be necessary to do this, wrapping it in an intuitive class of APIs.

The first step in accessing data in a datasource's tables via the wxDbTable class is to create a wxDbTable instance.

```
table = new wxDbTable(db, tableName, numTableColumns, "",
                      !wxDB_QUERY_ONLY, "");
```

When you create the instance, you indicate the previously established datasource connection to be used to access the table, the name of the primary table that is to be accessed with the datasource's tables, how many columns of each row are going to be returned, the name of the view of the table that will actually be used to query against (works with Oracle only at this time), whether the data returned is for query purposes only, and finally the path to the table, if different than the path specified when connecting to the datasource.

Each of the above parameters are described in detail in the wxDbTable class' description, but one special note here about the fifth parameter - the queryOnly setting. If a wxDbTable instance is created as `wxDB_QUERY_ONLY`, then no inserts/deletes/updates can be performed using this instance of the wxDbTable. Any calls to *wxDb::CommitTrans* (p. 293) or *wxDb::RollbackTrans* (p. 310) against the datasource connection used by this wxDbTable instance are ignored by this instance. If the wxDbTable instance is created with `!wxDB_QUERY_ONLY` as shown above, then all the cursors and other overhead associated with being able to insert/update/delete data in the table are created, and thereby those operations can then be performed against the associated table with this wxDbTable instance.

If a table is to be accessed via a wxDbTable instance, and the table will only be read from, not written to, there is a performance benefit (not as many cursors need to be maintained/updated, hence speeding up access times), as well as a resource savings due to fewer cursors being created for the wxDbTable instance. Also, with some datasources, the number of simultaneous cursors is limited.

When defining the columns to be retrievable by the wxDbTable instance, you can specify anywhere from one column up to all columns in the table.

```
table->SetColDefs(0, "FIRST_NAME", DB_DATA_TYPE_VARCHAR, FirstName,
                  SQL_C_WXCHAR, sizeof(FirstName), true, true);
table->SetColDefs(1, "LAST_NAME", DB_DATA_TYPE_VARCHAR, LastName,
                  SQL_C_WXCHAR, sizeof(LastName), true, true);
```

Notice that column definitions start at index 0 and go up to one less than the number of columns specified when the wxDbTable instance was created (in this example, two columns - one with index 0, one with index 1).

The above lines of code "bind" the datasource columns specified to the memory variables in the client application. So when the application makes a call to *wxDbTable::GetNext* (p. 341) (or any other function that retrieves data from the result set), the variables that are bound to the columns will have the column value stored into them. See the *wxDbTable::SetColDefs* (p. 352) class documentation for more details on all the parameters for this function.

The bound memory variables have undefined data in them until a call to a function that retrieves data from a result set is made (e.g. *wxDbTable::GetNext* (p. 341),*wxDbTable::GetPrev* (p. 342), etc). The variables are not initialized to any data by the wxODBC classes, and they still contain undefined data after a call to *wxDbTable::Query* (p. 347). Only after a successful call to one of the ::GetXxxx() functions is made do the variables contain valid data.

It is not necessary to define column definitions for columns whose data is not going to be returned to the client. For example, if you want to query the datasource for all users

with a first name of 'GEORGE', but you only want the list of last names associated with those rows (why return the FIRST_NAME column every time when you already know it is 'GEORGE'), you would only have needed to define one column above.

You may have as many wxDbTable instances accessing the same table using the same wxDb instance as you desire. There is no limit imposed by the classes on this. All datasources supported (so far) also have no limitations on this.

Open the table

Opening the table is not technically doing anything with the datasource itself. Calling *wxDbTable::Open* (p. 345) simply does all the housekeeping of checking that the specified table exists, that the current connected user has at least SELECT privileges for accessing the table, setting up the requisite cursors, binding columns and cursors, and constructing the default INSERT statement that is used when a new row is inserted into the table (non-wxDB_QUERY_ONLY tables only).

```
if (!table->Open())
{
    // An error occurred opening (setting up) the table
}
```

The only reason that a call to *wxDbTable::Open* (p. 345) is likely to fail is if the user has insufficient privileges to even SELECT the table. Other problems could occur, such as being unable to bind columns, but these other reason point to some lack of resource (like memory). Any errors generated internally in the *wxDbTable::Open* (p. 345) function are logged to the error log if SQL logging is turned on for the classes.

Use the table

To use the table and the definitions that are now set up, we must first define what data we want the datasource to collect in to a result set, tell it where to get the data from, and in which sequence we want the data returned.

```
// the WHERE clause limits/specifies which rows in the table
// are to be returned in the result set
table->SetWhereClause("FIRST_NAME = 'GEORGE'");

// Result set will be sorted in ascending alphabetical
// order on the data in the 'LAST_NAME' column of each row
// If the same last name is in the table for two rows,
// sub-sort on the 'AGE' column
table->SetOrderByClause("LAST_NAME, AGE");

// No other tables (joins) are used for this query
table->SetFromClause("");
```

The above lines will be used to tell the datasource to return in the result all the rows in the table whose column "FIRST_NAME" contains the name 'GEORGE' (note the required use of the single quote around the string literal) and that the result set will return the rows sorted by ascending last names (ascending is the default, and can be overridden with the "DESC" keyword for datasources that support it - "LAST_NAME DESC").

Specifying a blank WHERE clause will result in the result set containing all rows in the datasource.

Specifying a blank ORDERBY clause means that the datasource will return the result set in whatever sequence it encounters rows which match the selection criteria. What this sequence is can be hard to determine. Typically it depends on the index that the datasource used to find the rows which match the WHERE criteria. BEWARE - relying on the datasource to return data in a certain sequence when you have not provided an ORDERBY clause will eventually cause a problem for your program. Databases can be tuned to be COST-based, SPEED-based, or some other basis for how it gets your result set. In short, if you need your result set returned in a specific sequence, ask for it that way by providing an ORDERBY clause.

Using an ORDERBY clause can be a performance hit, as the database must sort the items before making the result set available to the client. Creating efficient indexes that cause the data to be "found" in the correct ORDERBY sequence can be a big performance benefit. Also, in the large majority of cases, the database will be able to sort the records faster than your application can read all the records in (unsorted) and then sort them. Let the database do the work for you!

Notice in the example above, a column that is not included in the bound data columns ('AGE') will be used to sub-sort the result set.

The FROM clause in this example is blanked, as we are not going to be performing any table joins with this simple query. When the FROM clause is blank, it is assumed that all columns referenced are coming from the default table for the wxDbTable instance.

After the selection criteria have been specified, the program can now ask the datasource to perform the search and create a result set that can be retrieved:

```
// Instruct the datasource to perform a query based on the
// criteria specified above in the where/orderBy/from clauses.
if (!table->Query())
{
    // An error occurred performing the query
}
```

Typically, when an error occurs when calling *wxDbTable::Query* (p. 347), it is a syntax problem in the WHERE clause that was specified. The exact SQL (datasource-specific) reason for what caused the failure of *wxDbTable::Query* (p. 347) (and all other operations against the datasource can be found by parsing the table's database connection's "errorList[]" array member for the stored text of the error.

When the *wxDbTable::Query* (p. 347) returns true, the database was able to successfully complete the requested query using the provided criteria. This does not mean that there are any rows in the result set, it just mean that the query was successful.

IMPORTANT: The result created by the call to *wxDbTable::Query* (p. 347) can take one of two forms. It is either a snapshot of the data at the exact moment that the database determined the record matched the search criteria, or it is a pointer to the row that matched the selection criteria. Which form of behavior is datasource dependent. If it is a snapshot, the data may have changed since the result set was constructed, so beware if your datasource uses snapshots and call *wxDbTable::Refresh* (p. 352). Most larger brand databases do not use snapshots, but it is important to mention so that your application can handle it properly if your datasource does.

To retrieve the data, one of the data fetching routines must be used to request a row from the result set, and to store the data from the result set into the bound memory variables. After *wxDbTable::Query* (p. 347) has completed successfully, the default/current cursor is placed so it is pointing just before the first record in the result set. If the result set is empty (no rows matched the criteria), then any calls to retrieve data from the result set will return false.

```
wxString msg;

while (table->GetNext())
{
    msg.Printf("Row #%lu -- First Name : %s  Last Name is %s",
               table->GetRowNum(), FirstName, LastName);
    wxMessageBox(msg, "Data", wxOK | wxICON_INFORMATION, NULL);
}
```

The sample code above will read the next record in the result set repeatedly until the end of the result set has been reached. The first time that *wxDbTable::GetNext* (p. 341) is called right after the successful call to *wxDbTable::Query* (p. 347), it actually returns the first record in the result set.

When *wxDbTable::GetNext* (p. 341) is called and there are no rows remaining in the result set after the current cursor position, *wxDbTable::GetNext* (p. 341) (as well as all the other wxDbTable::GetXxxxx() functions) will return false.

Close the table

When the program is done using a wxDbTable instance, it is as simple as deleting the table pointer (or if declared statically, letting the variable go out of scope). Typically the default destructor will take care of all that is required for cleaning up the wxDbTable instance.

```
if (table)
{
    delete table;
    table = NULL;
}
```

Deleting a wxDbTable instance releases all of its cursors, deletes the column definitions and frees the SQL environment handles used by the table (but not the environment handle used by the datasource connection that the wxDbTable instance was using).

Close the datasource connection

After all tables that have been using a datasource connection have been closed (this can be verified by calling *wxDb::GetTableCount* (p. 304) and checking that it returns 0), then you may close the datasource connection. The method of doing this is dependent on whether the non-caching or caching method was used to obtain the datasource connection.

If the datasource connection was created manually (non-cached), closing the connection is done like this:

```
if (db)
{
    db->Close();
```

```
        delete db;
        db = NULL;
}
```

If the program used the *wxDbGetConnection* (p. 289) function to get a datasource connection, the following is the code that should be used to free the connection(s):

```
if (db)
{
    wxDbFreeConnection(db);
    db = NULL;
}
```

Note that the above code just frees the connection so that it can be re-used on the next call the *wxDbGetConnection* (p. 289). To actually dispose of the connection, releasing all of its resources (other than the environment handle), do the following:

```
wxDbCloseConnections();
```

Release the ODBC environment handle

Once all of the connections that used the ODBC environment handle (in this example it was stored in "DbConnectInf.Henv") have been closed, then it is safe to release the environment handle:

```
DbConnectInf->FreeHenv();
```

Or, if the long form of the constructor was used and the constructor was allowed to allocate its own SQL environment handle, leaving scope or destruction of the wxDbConnectInf will free the handle automatically.

```
delete DbConnectInf;
```

Remember to never release this environment handle if there are any connections still using the handle.

wxODBC - Known Issues

As with creating wxWidgets, writing the wxODBC classes was not the simple task of writing an application to run on a single type of computer system. The classes need to be cross-platform for different operating systems, and they also needed to take in to account different database manufacturers and different ODBC driver manufacturers. Because of all the possible combinations of OS/database/drivers, it is impossible to say that these classes will work perfectly with datasource ABC, ODBC driver XYZ, on platform LMN. You may run into some incompatibilities or unsupported features when moving your application from one environment to another. But that is what makes cross-platform programming fun. It also pinpoints one of the great things about open source software. It can evolve!

The most common difference between different database/ODBC driver manufacturers in regards to these wxODBC classes is the lack of standard error codes being returned to the calling program. Sometimes manufacturers have even changed the error codes between versions of their databases/drivers.

In all the tested databases, every effort has been made to determine the correct error codes and handle them in the class members that need to check for specific error codes (such as TABLE DOES NOT EXIST when you try to open a table that has not been created yet). Adding support for additional databases in the future requires adding an entry for the database in the *wxDb::Dbms* (p. 295) function, and then handling any error codes returned by the datasource that do not match the expected values.

Databases

Following is a list of known issues and incompatibilities that the wxODBC classes have between different datasources. An up to date listing of known issues can be seen in the comments of the source for *wxDb::Dbms* (p. 295).

ORACLE

- Currently the only database supported by the wxODBC classes to support VIEWS

DBASE

NOTE: dBase is not a true ODBC datasource. You only have access to as much functionality as the driver can emulate.

- Does not support the SQL_TIMESTAMP structure

- Supports only one cursor and one connect (apparently? with Microsoft driver only?)

- Does not automatically create the primary index if the 'keyField' param of SetColDef is true. The user must create ALL indexes from their program with calls to *wxDbTable::CreateIndex* (p. 332)

- Table names can only be 8 characters long

- Column names can only be 10 characters long

- Currently cannot CREATE a dBase table - bug or limitation of the drivers used??

- Currently cannot insert rows that have integer columns - bug??

SYBASE (all)

- To lock a record during QUERY functions, the reserved word 'HOLDLOCK' must be added after every table name involved in the query/join if that table's matching record(s) are to be locked

- Ignores the keywords 'FOR UPDATE'. Use the HOLDLOCK functionality described above

SYBASE (Enterprise)

- If a column is part of the Primary Key, the column cannot be NULL

- Maximum row size is somewhere in the neighborhood of 1920 bytes

mySQL

- If a column is part of the Primary Key, the column cannot be NULL.

- Cannot support selecting for update [*wxDbTable::CanSelectForUpdate* (p. 329)]. Always returns false.

- Columns that are part of primary or secondary keys must be defined as being NOT NULL when they are created. Some code is added in *wxDbTable::CreateIndex* (p. 332) to try to adjust the column definition if it is not defined correctly, but it is experimental (as of wxWidgets v2.2.1)

- Does not support sub-queries in SQL statements

POSTGRES

- Does not support the keywords 'ASC' or 'DESC' as of release v6.5.0

- Does not support sub-queries in SQL statements

DB2

- Columns which are part of a primary key must be declared as NOT NULL

UNICODE with wxODBC classes

As of v2.6 of wxWidgets, the wxODBC classes now fully support the compilation and use of the classes in a Unicode build of wxWidgets, assuming the compiler and OS on which the program will be compiled/run is Unicode capable.

The one major difference in writing code that can be compiled in either unicode or non-unicode builds that is specific to the wxODBC classes is to use the SQL_C_WXCHAR datatype for string columns rather than SQL_C_CHAR or SQL_C_WCHAR.

wxODBC - Sample Code

Simplest example of establishing/opening a connection to an ODBC datasource, binding variables to the columns for read/write usage, opening an existing table in the datasource, inserting a record, setting query parameters (where/orderBy/from), querying the datasource, reading each row of the result set, deleting a record, releasing the connection, then cleaning up.

NOTE: Very basic error handling is shown here, to reduce the size of the code and to make it more easily readable. The HandleError() function uses the wxDbLogExtendedErrorMsg() function for retrieving database error messages.

```
// --------------------------------------------------------------------
-----
// HEADERS
// --------------------------------------------------------------------
-----
#include "wx/log.h"          // #included to enable output of messages only
#include "wx/dbtable.h"
```

```
// ------------------------------------------------------------------------
-----
// FUNCTION USED FOR HANDLING/DISPLAYING ERRORS
// ------------------------------------------------------------------------
-----
// Very generic error handling function.
// If a connection to the database is passed in, then we retrieve all the
// database errors for the connection and add them to the displayed
message
int HandleError(wxString errmsg, wxDb *pDb=NULL)
{
    // Retrieve all the error message for the errors that occurred
    wxString allErrors;
    if (!pDb == NULL)
        // Get the database errors and append them to the error message
        allErrors = wxDbLogExtendedErrorMsg(errmsg.c_str(), pDb, 0, 0);
    else
        allErrors = errmsg;

    // Do whatever you wish with the error message here
    // wxLogDebug() is called inside wxDbLogExtendedErrorMsg() so this
    // console program will show the errors in the console window,
    // but these lines will show the errors in RELEASE builds also
    wxFprintf(stderr, wxT("\n%s\n"), allErrors.c_str());
    fflush(stderr);

    return 1;
}


// ------------------------------------------------------------------------
-----
// entry point
// ------------------------------------------------------------------------
-----
int main(int argc, char **argv)
{
wxDbConnectInf  *DbConnectInf   = NULL;    // DB connection information

wxDb            *db             = NULL;    // Database connection

wxDbTable       *table          = NULL;    // Data table to access
const wxChar    tableName[]     = wxT("USERS"); // Name of database table
const UWORD     numTableColumns = 2;        // Number table columns
wxChar          FirstName[50+1];            // column data: "FIRST_NAME"
wxChar          LastName[50+1];             // column data: "LAST_NAME"

wxString        msg;                        // Used for display messages

// ------------------------------------------------------------------------
// DEFINE THE CONNECTION HANDLE FOR THE DATABASE
// ------------------------------------------------------------------------
DbConnectInf = new wxDbConnectInf(NULL,
                                  wxT("CONTACTS-SqlServer"),
                                  wxT("sa"),
                                  wxT("abk"));

// Error checking....
if (!DbConnectInf || !DbConnectInf->GetHenv())
{
    return HandleError(wxT("DB ENV ERROR: Cannot allocate ODBC env
handle"));
}
```

```
// -------------------------------------------------------------------------
// GET A DATABASE CONNECTION
// -------------------------------------------------------------------------
db = wxDbGetConnection(DbConnectInf);

if (!db)
{
    return HandleError(wxT("CONNECTION ERROR - Cannot get DB
connection"));
}


// -------------------------------------------------------------------------
// DEFINE THE TABLE, AND THE COLUMNS THAT WILL BE ACCESSED
// -------------------------------------------------------------------------
table = new wxDbTable(db, tableName, numTableColumns, wxT(""),
                      !wxDB_QUERY_ONLY, wxT(""));
//
// Bind the columns that you wish to retrieve. Note that there must be
// 'numTableColumns' calls to SetColDefs(), to match the wxDbTable def
//
// Not all columns need to be bound, only columns whose values are to be
// returned back to the client.
//
table->SetColDefs(0, wxT("FIRST_NAME"), DB_DATA_TYPE_VARCHAR, FirstName,
                  SQL_C_WXCHAR, sizeof(FirstName), true, true);
table->SetColDefs(1, wxT("LAST_NAME"), DB_DATA_TYPE_VARCHAR, LastName,
                  SQL_C_WXCHAR, sizeof(LastName), true, true);


// -------------------------------------------------------------------------
// CREATE (or RECREATE) THE TABLE IN THE DATABASE
// -------------------------------------------------------------------------
if (!table->CreateTable(true))  //NOTE: No CommitTrans is required
{
    return HandleError(wxT("TABLE CREATION ERROR: "), table->GetDb());
}


// -------------------------------------------------------------------------
// OPEN THE TABLE FOR ACCESS
// -------------------------------------------------------------------------
if (!table->Open())
{
    return HandleError(wxT("TABLE OPEN ERROR: "), table->GetDb());
}


// -------------------------------------------------------------------------
// INSERT A NEW ROW INTO THE TABLE
// -------------------------------------------------------------------------
wxStrcpy(FirstName, wxT("JULIAN"));
wxStrcpy(LastName, wxT("SMART"));
if (!table->Insert())
{
    return HandleError(wxT("INSERTION ERROR: "), table->GetDb());
}

// Must commit the insert to write the data to the DB
table->GetDb()->CommitTrans();
```

```
// ----------------------------------------------------------------------
// RETRIEVE ROWS FROM THE TABLE BASED ON SUPPLIED CRITERIA
// ----------------------------------------------------------------------
// Set the WHERE clause to limit the result set to return
// all rows that have a value of 'JULIAN' in the FIRST_NAME
// column of the table.
table->SetWhereClause(wxT("FIRST_NAME = 'JULIAN'"));

// Result set will be sorted in ascending alphabetical
// order on the data in the 'LAST_NAME' column of each row
table->SetOrderByClause(wxT("LAST_NAME"));

// No other tables (joins) are used for this query
table->SetFromClause(wxT(""));

// Instruct the datasource to perform a query based on the
// criteria specified above in the where/orderBy/from clauses.
if (!table->Query())
{
    return HandleError(wxT("QUERY ERROR: "), table->GetDb());
}

// Loop through all rows matching the query criteria until
// there are no more records to read
while (table->GetNext())
{
    msg.Printf(wxT("Row #%lu -- First Name : %s  Last Name is %s"),
            table->GetRowNum(), FirstName, LastName);

    // Code to display 'msg' here
    wxLogMessage(wxT("\n%s\n"), msg.c_str());
}


// ----------------------------------------------------------------------
// DELETE A ROW FROM THE TABLE
// ----------------------------------------------------------------------
// Select the row which has FIRST_NAME of 'JULIAN' and LAST_NAME
// of 'SMART', then delete the retrieved row
//
if (!table->DeleteWhere(wxT("FIRST_NAME = 'JULIAN' and LAST_NAME =
'SMART'")))
{
    return HandleError(wxT("DELETION ERROR: "), table->GetDb());
}

// Must commit the deletion to the database
table->GetDb()->CommitTrans();


// ----------------------------------------------------------------------
// TAKE CARE OF THE ODBC CLASS INSTANCES THAT WERE BEING USED
// ----------------------------------------------------------------------
// If the wxDbTable instance was successfully created
// then delete it as we are done with it now.
wxDELETE(table);

// Free the cached connection
// (meaning release it back in to the cache of datasource
// connections) for the next time a call to wxDbGetConnection()
// is made.
wxDbFreeConnection(db);
```

```
db = NULL;


// ----------------------------------------------------------------------
// CLEANUP BEFORE EXITING APP
// ----------------------------------------------------------------------
// The program is now ending, so we need to close
// any cached connections that are still being
// maintained.
wxDbCloseConnections();

// Release the environment handle that was created
// for use with the ODBC datasource connections
wxDELETE(DbConnectInf);

wxUnusedVar(argc);  // Here just to prevent compiler warnings
wxUnusedVar(argv);  // Here just to prevent compiler warnings

return 0;
}
```

A selection of SQL commands

The following is a very brief description of some common SQL commands, with
examples.

See also

Database classes overview (p. 1820)

Create

Creates a table.

Example:

```
CREATE TABLE Book
 (BookNumber      INTEGER      PRIMARY KEY
 , CategoryCode   CHAR(2)      DEFAULT 'RO' NOT NULL
 , Title          VARCHAR(100) UNIQUE
 , NumberOfPages  SMALLINT
 , RetailPriceAmount NUMERIC(5,2)
 )
```

Insert

Inserts records into a table.

Example:

```
INSERT INTO Book
   (BookNumber, CategoryCode, Title)
   VALUES(5, 'HR', 'The Lark Ascending')
```

Select

The Select operation retrieves rows and columns from a table. The criteria for selection

and the columns returned may be specified.

Examples:

```
SELECT * FROM Book
```

Selects all rows and columns from table Book.

```
SELECT Title, RetailPriceAmount FROM Book WHERE RetailPriceAmount
> 20.0
```

Selects columns Title and RetailPriceAmount from table Book, returning only the rows that match the WHERE clause.

```
SELECT * FROM Book WHERE CatCode = 'LL' OR CatCode = 'RR'
```

Selects all columns from table Book, returning only the rows that match the WHERE clause.

```
SELECT * FROM Book WHERE CatCode IS NULL
```

Selects all columns from table Book, returning only rows where the CatCode column is NULL.

```
SELECT * FROM Book ORDER BY Title
```

Selects all columns from table Book, ordering by Title, in ascending order. To specify descending order, add DESC after the ORDER BY Title clause.

```
SELECT Title FROM Book WHERE RetailPriceAmount >= 20.0 AND
RetailPriceAmount <= 35.0
```

Selects records where RetailPriceAmount conforms to the WHERE expression.

Update

Updates records in a table.

Example:

```
UPDATE Incident SET X = 123 WHERE ASSET = 'BD34'
```

This example sets a field in column 'X' to the number 123, for the record where the column ASSET has the value 'BD34'.

Interprocess communication overview

Classes: *wxServer* (p. 1167),*wxConnection* (p. 387),*wxClient* (p. 145)wxWidgets has a number of different classes to help with interprocess communication and network programming. This section only discusses one family of classes -- the DDE-like protocol -- but here's a list of other useful classes:

- *wxSocketEvent* (p. 1221),*wxSocketBase* (p. 1203),*wxSocketClient* (p.

1219),*wxSocketServer* (p. 1223): classes for the low-level TCP/IP API.

- *wxProtocol* (p. 1083), *wxURL* (p. 1459), *wxFTP* (p. 592), *wxHTTP* (p. 765): classes for programming popular Internet protocols.

wxWidgets' DDE-like protocol is a high-level protocol based on Windows DDE. There are two implementations of this DDE-like protocol: one using real DDE running on Windows only, and another using TCP/IP (sockets) that runs on most platforms. Since the API and virtually all of the behaviour is the same apart from the names of the classes, you should find it easy to switch between the two implementations.

Notice that by including `<wx/ipc.h>` you may define convenient synonyms for the IPC classes: `wxServer` for either`wxDDEServer` or `wxTCPServer` depending on whether DDE-based or socket-based implementation is used and the same thing for `wxClient` and `wxConnection`.

By default, the DDE implementation is used under Windows. DDE works within one computer only. If you want to use IPC between different workstations you should define `wxUSE_DDE_FOR_IPC` as 0 before including this header -- this will force using TCP/IP implementation even under Windows.

The following description refers to wx... but remember that the equivalent wxTCP... and wxDDE... classes can be used in much the same way.

Three classes are central to the DDE-like API:

1. wxClient. This represents the client application, and is used only within a client program.

2. wxServer. This represents the server application, and is used only within a server program.

3. wxConnection. This represents the connection from the client to the server - both the client and the server use an instance of this class, one per connection. Most DDE transactions operate on this object.

Messages between applications are usually identified by three variables: connection object, topic name and item name. A data string is a fourth element of some messages. To create a connection (a conversation in Windows parlance), the client application uses wxClient::MakeConnection to send a message to the server object, with a string service name to identify the server and a topic name to identify the topic for the duration of the connection. Under Unix, the service name may be either an integer port identifier in which case an Internet domain socket will be used for the communications or a valid file name (which shouldn't exist and will be deleted afterwards) in which case a Unix domain socket is created.

SECURITY NOTE: Using Internet domain sockets is extremely insecure for IPC as there is absolutely no access control for them, use Unix domain sockets whenever possible!

The server then responds and either vetoes the connection or allows it. If allowed, both the server and client objects create wxConnection objects which persist until the connection is closed. The connection object is then used for sending and receiving subsequent messages between client and server - overriding virtual functions in your

class derived from wxConnection allows you to handle the DDE messages.

To create a working server, the programmer must:

1. Derive a class from wxConnection, providing handlers for various messages sent to the server side of a wxConnection (e.g. OnExecute, OnRequest, OnPoke). Only the handlers actually required by the application need to be overridden.

2. Derive a class from wxServer, overriding OnAcceptConnection to accept or reject a connection on the basis of the topic argument. This member must create and return an instance of the derived connection class if the connection is accepted.

3. Create an instance of your server object and call Create to activate it, giving it a service name.

To create a working client, the programmer must:

1. Derive a class from wxConnection, providing handlers for various messages sent to the client side of a wxConnection (e.g. OnAdvise). Only the handlers actually required by the application need to be overridden.

2. Derive a class from wxClient, overriding OnMakeConnection to create and return an instance of the derived connection class.

3. Create an instance of your client object.

4. When appropriate, create a new connection using *wxClient::MakeConnection* (p. 146), with arguments host name (processed in Unix only, use 'localhost' for local computer), service name, and topic name for this connection. The client object will call *OnMakeConnection* (p. 387) to create a connection object of the derived class if the connection is successful.

5. Use the wxConnection member functions to send messages to the server.

Data transfer

These are the ways that data can be transferred from one application to another. These are methods of wxConnection.

*   **Execute:** the client calls the server with a data string representing a command to be executed. This succeeds or fails, depending on the server's willingness to answer. If the client wants to find the result of the Execute command other than success or failure, it has to explicitly call Request.

*   **Request:** the client asks the server for a particular data string associated with a given item string. If the server is unwilling to reply, the return value is NULL. Otherwise, the return value is a string (actually a pointer to the connection buffer, so it should not be deallocated by the application).

*   **Poke:** The client sends a data string associated with an item string directly to

the server. This succeeds or fails.

- **Advise:** The client asks to be advised of any change in data associated with a particular item. If the server agrees, the server will send an OnAdvise message to the client along with the item and data.

The default data type is wxCF_TEXT (ASCII text), and the default data size is the length of the null-terminated string. Windows-specific data types could also be used on the PC.

Examples

See the sample programs *server* and *client* in the IPC samples directory. Run the server, then the client. This demonstrates using the Execute, Request, and Poke commands from the client, together with an Advise loop: selecting an item in the server list box causes that item to be highlighted in the client list box.

More DDE details

A wxClient object initiates the client part of a client-server DDE-like (Dynamic Data Exchange) conversation (available in both Windows and Unix).

To create a client which can communicate with a suitable server, you need to derive a class from wxConnection and another from wxClient. The custom wxConnection class will receive communications in a 'conversation' with a server. and the custom wxServer is required so that a user-overridden*wxClient::OnMakeConnection* (p. 387)member can return a wxConnection of the required class, when a connection is made.

For example:

```
class MyConnection: public wxConnection {
 public:
  MyConnection(void)::wxConnection() {}
  ~MyConnection(void) { }
  bool OnAdvise(const wxString& topic, const wxString& item, char *data,
int size, wxIPCFormat format)
  { wxMessageBox(topic, data); }
};

class MyClient: public wxClient {
 public:
  MyClient(void) {}
  wxConnectionBase *OnMakeConnection(void) { return new MyConnection; }
};
```

Here, **MyConnection** will respond to*OnAdvise* (p. 389) messages sent by the server by displaying a message box.

When the client application starts, it must create an instance of the derived wxClient. In the following, command line arguments are used to pass the host name (the name of the machine the server is running on) and the server name (identifying the server process). Calling*wxClient::MakeConnection* (p. 387) implicitly creates an instance of **MyConnection** if the request for a connection is accepted, and the client then requests

an *Advise* loop from the server (an Advise loop is where the server calls the client when data has changed).

```
wxString server = "4242";
wxString hostName;
wxGetHostName(hostName);

// Create a new client
MyClient *client = new MyClient;
connection = (MyConnection *)client->MakeConnection(hostName, server,
"IPC TEST");

if (!connection)
{
   wxMessageBox("Failed to make connection to server", "Client Demo
Error");
   return NULL;
}
connection->StartAdvise("Item");
```

wxHTML overview

This topic was written by Vaclav Slavik, the author of the wxHTML library.

The wxHTML library provides classes for parsing and displaying HTML.

It is not intended to be a high-end HTML browser. If you are looking for something like that try http://www.mozilla.org (`http://www.mozilla.org`).

wxHTML can be used as a generic rich text viewer - for example to display a nice About Box (like those of GNOME apps) or to display the result of database searching. There is a *wxFileSystem* (p. 540) class which allows you to use your own virtual file systems.

wxHtmlWindow supports tag handlers. This means that you can easily extend wxHtml library with new, unsupported tags. Not only that, you can even use your own application-specific tags! See `src/html/m_*.cpp` files for details.

There is a generic wxHtmlParser class, independent of wxHtmlWindow.

wxHTML quick start

Displaying HTML

First of all, you must include <wx/wxhtml.h>.

Class *wxHtmlWindow* (p. 749) (derived from wxScrolledWindow) is used to display HTML documents. It has two important methods: *LoadPage* (p. 753) and *SetPage* (p. 757). LoadPage loads and displays HTML file while SetPage displays directly the passed **string**. See the example:

```
mywin -> LoadPage("test.htm");
mywin -> SetPage("<html><body>"
                  "<h1>Error</h1>"
```

```
                        "Some error occurred :-H)"
                        "</body></hmtl>");
```

I think the difference is quite clear.

Displaying Help

See *wxHtmlHelpController* (p. 722).

Setting up wxHtmlWindow

Because wxHtmlWindow is derived from wxScrolledWindow and not from wxFrame, it doesn't have visible frame. But the user usually wants to see the title of HTML page displayed somewhere and the frame's titlebar is the ideal place for it.

wxHtmlWindow provides 2 methods in order to handle this: *SetRelatedFrame* (p. 757) and *SetRelatedStatusBar* (p. 758). See the example:

```
            html = new wxHtmlWindow(this);
            html -> SetRelatedFrame(this, "HTML : %%s");
            html -> SetRelatedStatusBar(0);
```

The first command associates the HTML object with its parent frame (this points to wxFrame object there) and sets the format of the title. Page title "Hello, world!" will be displayed as "HTML : Hello, world!" in this example.

The second command sets which frame's status bar should be used to display browser's messages (such as "Loading..." or "Done" or hypertext links).

Customizing wxHtmlWindow

You can customize wxHtmlWindow by setting font size, font face and borders (space between border of window and displayed HTML). Related functions:

- *SetFonts* (p. 756)

- *SetBorders* (p. 756)

- *ReadCustomization* (p. 755)

- *WriteCustomization* (p. 758)

The last two functions are used to store user customization info wxConfig stuff (for example in the registry under Windows, or in a dotfile under Unix).

HTML Printing

The wxHTML library provides printing facilities with several levels of complexity.

The easiest way to print an HTML document is to use *wxHtmlEasyPrinting class* (p. 718). It lets you print HTML documents with only one command and you don't have to worry about deriving from the wxPrintout class at all. It is only a simple wrapper around the *wxHtmlPrintout* (p. 741), normal wxWidgets printout class.

And finally there is the low level class *wxHtmlDCRenderer* (p. 716) which you can use to render HTML into a rectangular area on any DC. It supports rendering into multiple rectangles with the same width. (The most common use of this is placing one rectangle on each page or printing into two columns.)

Help Files Format

wxHTML library uses a reduced version of MS HTML Workshop format. Tex2RTF can produce these files when generating HTML, if you set **htmlWorkshopFiles** to **true** in your tex2rtf.ini file.

(See *wxHtmlHelpController* (p. 722) for help controller description.)

A **book** consists of three files: header file, contents file and index file. You can make a regular zip archive of these files, plus the HTML and any image files, for wxHTML (or helpview) to read; and the .zip file can optionally be renamed to .htb.

Header file (.hhp)

Header file must contain these lines (and may contain additional lines which are ignored) :

```
Contents file=<filename.hhc>
Index file=<filename.hhk>
Title=<title of your book>
Default topic=<default page to be displayed.htm>
```

All filenames (including the Default topic) are relative to the location of .hhp file.

Localization note: In addition, .hhp file may contain line

```
Charset=<rfc_charset>
```

which specifies what charset (e.g. "iso8859_1") was used in contents and index files. Please note that this line is incompatible with MS HTML Help Workshop and it would either silently remove it or complain with some error. See also *Writing non-English applications* (p. 1737).

Contents file (.hhc)

Contents file has HTML syntax and it can be parsed by regular HTML parser. It contains exactly one list (`<ul>....</ul>` statement):

```
<ul>

  <li> <object type="text/sitemap">
          <param name="Name" value="@topic name@">
          <param name="ID" value=@numeric_id@>
          <param name="Local" value="@filename.htm@">
      </object>
  <li> <object type="text/sitemap">
          <param name="Name" value="@topic name@">
          <param name="ID" value=@numeric_id@>
          <param name="Local" value="@filename.htm@">
      </object>
```

```
   ...

</ul>
```

You can modify value attributes of param tags. *topic name* is name of chapter/topic as is displayed in contents, *filename.htm* is HTML page name (relative to .hhp file) and *numeric_id* is optional - it is used only when you use *wxHtmlHelpController::Display(int)* (p. 725)

Items in the list may be nested - one `<li>` statement may contain a `<ul>` sub-statement:

```
<ul>

  <li> <object type="text/sitemap">
          <param name="Name" value="Top node">
          <param name="Local" value="top.htm">
      </object>
      <ul>
        <li> <object type="text/sitemap">
            <param name="Name" value="subnode in topnode">
            <param name="Local" value="subnode1.htm">
            </object>
   ...
      </ul>

  <li> <object type="text/sitemap">
          <param name="Name" value="Another Top">
          <param name="Local" value="top2.htm">
      </object>
   ...

</ul>
```

Index file (.hhk)

Index files have same format as contents file except that ID params are ignored and sublists are **not** allowed.

Input Filters

The wxHTML library provides a mechanism for reading and displaying files of many different file formats.

wxHtmlWindow::LoadPage (p. 753) can load not only HTML files but any known file. To make a file type known to wxHtmlWindow you must create a *wxHtmlFilter* (p. 721) filter and register it using *wxHtmlWindow::AddFilter* (p. 750).

Cells and Containers

This article describes mechanism used by *wxHtmlWinParser* (p. 758) and *wxHtmlWindow* (p. 749) to parse and display HTML documents.

Cells

You can divide any text (or HTML) into small fragments. Let's call these fragments **cells**. Cell is for example one word, horizontal line, image or any other part of document. Each cell has width and height (except special "magic" cells with zero dimensions - e.g. colour changers or font changers).

See *wxHtmlCell* (p. 705).

Containers

Container is kind of cell that may contain sub-cells. Its size depends on number and sizes of its sub-cells (and also depends on width of window).

See *wxHtmlContainerCell* (p. 711), *wxHtmlCell::Layout* (p. 709).

Using Containers in Tag Handler

wxHtmlWinParser (p. 758) provides a user-friendly way of managing containers. It is based on the idea of opening and closing containers.

Use *OpenContainer* (p. 762) to open new a container *within an already opened container*. This new container is a *sub-container* of the old one. (If you want to create a new container with the same depth level you can call `CloseContainer();` `OpenContainer();`.)

Use *CloseContainer* (p. 759) to close the container. This doesn't create a new container with same depth level but it returns "control" to the parent container.

It is clear there must be same number of calls to OpenContainer as to CloseContainer...

Example

This code creates a new paragraph (container at same depth level) with "Hello, world!":

```
m_WParser -> CloseContainer();
c = m_WParser -> OpenContainer();

m_WParser -> AddWord("Hello, ");
m_WParser -> AddWord("world!");

m_WParser -> CloseContainer();
m_WParser -> OpenContainer();
```

You can see that there was opened container before running the code. We closed it, created our own container, then closed our container and opened new container. The result was that we had *same depth level* after executing. This is general rule that should be followed by tag handlers: leave depth level of containers unmodified (in other words, number of OpenContainer and CloseContainer calls should be same within *HandleTag* (p. 747)'s body).

Tag Handlers

The wxHTML library provides architecture of pluggable *tag handlers*. Tag handler is class that understands particular HTML tag (or tags) and is able to interpret it.

wxHtmlWinParser (p. 758) has static table of **modules**. Each module contains one or more tag handlers. Each time a new wxHtmlWinParser object is constructed all modules are scanned and handlers are added to wxHtmlParser's list of available handlers (note: wxHtmlParser's list is non-static).

How it works

Common tag handler's *HandleTag* (p. 747) method works in four steps:

1. Save state of parent parser into local variables

2. Change parser state according to tag's params

3. Parse text between the tag and paired ending tag (if present)

4. Restore original parser state

See *wxHtmlWinParser* (p. 758) for methods for modifying parser's state. In general you can do things like opening/closing containers, changing colors, fonts etc.

Providing own tag handlers

You should create new .cpp file and place following lines into it:

```
#include <mod_templ.h>
#include <forcelink.h>
FORCE_LINK_ME(yourmodulefilenamewithoutcpp)
```

Then you must define handlers and one module.

Tag handlers

The handler is derived from *wxHtmlWinTagHandler* (p. 764)(or directly from *wxHtmlTagHandler* (p. 746))

You can use set of macros to define the handler (see src/html/m_*.cpp files for details). Handler definition must start with **TAG_HANDLER_BEGIN** macro and end with **TAG_HANDLER_END** macro. I strongly recommend to have a look at *include/wxhtml/mod_templ.h* file. Otherwise you won't understand the structure of macros. See macros reference:

TAG_HANDLER_BEGIN(*name*, *tags*)

Starts handler definition. *name* is handler identifier (in fact part of class name), *tags* is string containing list of tags supported by this handler (in uppercase). This macro derives new class from wxHtmlWinTagHandler and implements it is *GetSupportedTags* (p. 747) method.

Example: TAG_HANDLER_BEGIN(FONTS, "B,I,U,T")

TAG_HANDLER_VARS

This macro starts block of variables definitions. (Variables are identical to class attributes.) Example:

```
TAG_HANDLER_BEGIN(VARS_ONLY, "CRAZYTAG")
    TAG_HANDLER_VARS
        int my_int_var;
 wxString something_else;
TAG_HANDLER_END(VARS_ONLY)
```

This macro is used only in rare cases.

TAG_HANDLER_CONSTR(*name*)

This macro supplies object constructor. *name* is same name as the one from TAG_HANDLER_BEGIN macro. Body of constructor follow after this macro (you must use and). Example:

```
TAG_HANDLER_BEGIN(VARS2, "CRAZYTAG")
    TAG_HANDLER_VARS
        int my_int_var;
    TAG_HANDLER_CONSTR(vars2)
        { // !!!!!!
      my_int_var = 666;
  } // !!!!!!
TAG_HANDLER_END(VARS2)
```

Never used in wxHTML :-)

TAG_HANDLER_PROC(*varib*)

This is very important macro. It defines *HandleTag* (p. 747)method. *varib* is name of parameter passed to the method, usually*tag*. Body of method follows after this macro. Note than you must use and ! Example:

```
TAG_HANDLER_BEGIN(TITLE, "TITLE")
    TAG_HANDLER_PROC(tag)
        {
      printf("TITLE found...\n");
  }
TAG_HANDLER_END(TITLE)
```

TAG_HANDLER_END(*name*)

Ends definition of tag handler *name*.

Tags Modules

You can use set of 3 macros TAGS_MODULE_BEGIN, TAGS_MODULE_ADD and TAGS_MODULE_END to inherit new module from*wxHtmlTagsModule* (p. 748) and to create instance of it. See macros reference:

TAGS_MODULE_BEGIN(*modname*)

Begins module definition. *modname* is part of class name and must be unique.

TAGS_MODULE_ADD(*name*)

Adds the handler to this module. *name* is the identifier from TAG_HANDLER_BEGIN.

TAGS_MODULE_END(*modname*)

Ends the definition of module.

Example:

```
TAGS_MODULE_BEGIN(Examples)
    TAGS_MODULE_ADD(VARS_ONLY)
    TAGS_MODULE_ADD(VARS2)
    TAGS_MODULE_ADD(TITLE)
TAGS_MODULE_END(Examples)
```

Tags supported by wxHTML

wxHTML is not full implementation of HTML standard. Instead, it supports most common tags so that it is possible to display *simple* HTML documents with it. (For example it works fine with pages created in Netscape Composer or generated by tex2rtf).

Following tables list all tags known to wxHTML, together with supported parameters. A tag has general form of `<tagname param_1 param_2 ... param_n>` where param_i is either `paramname="paramvalue"` or `paramname=paramvalue` - these two are equivalent. Unless stated otherwise, wxHTML is case-insensitive.

Table of common parameter values

We will use these substitutions in tags descriptions:

```
[alignment]     CENTER
                LEFT
                RIGHT
                JUSTIFY

[v_alignment]   TOP
                BOTTOM
                CENTER

[color]         HTML 4.0-compliant colour specification

[fontsize]      -2
                -1
                +0
                +1
                +2
                +3
                +4
                 1
                 2
                 3
                 4
                 5
                 6
                 7

[pixels]        integer value that represents dimension in pixels

[percent]       i%
                where i is integer

[url]           an URL
```

```
[string]          text string

[coords]          c(1),c(2),c(3),...,c(n)
                  where c(i) is integer
```

List of supported tags

```
A                 NAME=[string]
                  HREF=[url]
                  TARGET=[target window spec]
ADDRESS
AREA              SHAPE=POLY
                  SHAPE=CIRCLE
                  SHAPE=RECT
                  COORDS=[coords]
                  HREF=[url]
B
BIG
BLOCKQUOTE
BODY              TEXT=[color]
                  LINK=[color]
                  BGCOLOR=[color]
BR                ALIGN=[alignment]
CENTER
CITE
CODE
DD
DIV               ALIGN=[alignment]
DL
DT
EM
FONT              COLOR=[color]
                  SIZE=[fontsize]
                  FACE=[comma-separated list of facenames]
HR                ALIGN=[alignment]
                  SIZE=[pixels]
                  WIDTH=[percent|pixels]
                  NOSHADE
H1
H2
H3
H4
H5
H6
I
IMG               SRC=[url]
                  WIDTH=[pixels]
                  HEIGHT=[pixels]
                  ALIGN=TEXTTOP
                  ALIGN=CENTER
                  ALIGN=ABSCENTER
                  ALIGN=BOTTOM
                  USEMAP=[url]
KBD
LI
MAP               NAME=[string]
META              HTTP-EQUIV="Content-Type"
                  CONTENT=[string]
OL
P                 ALIGN=[alignment]
PRE
SAMP
```

```
        SMALL
        STRIKE
        STRONG
        TABLE           ALIGN=[alignment]
                        WIDTH=[percent|pixels]
                        BORDER=[pixels]
                        VALIGN=[v_alignment]
                        BGCOLOR=[color]
                        CELLSPACING=[pixels]
                        CELLPADDING=[pixels]
        TD              ALIGN=[alignment]
                        VALIGN=[v_alignment]
                        BGCOLOR=[color]
                        WIDTH=[percent|pixels]
                        COLSPAN=[pixels]
                        ROWSPAN=[pixels]
                        NOWRAP
        TH              ALIGN=[alignment]
                        VALIGN=[v_alignment]
                        BGCOLOR=[color]
                        WIDTH=[percent|pixels]
                        COLSPAN=[pixels]
                        ROWSPAN=[pixels]
        TITLE
        TR              ALIGN=[alignment]
                        VALIGN=[v_alignment]
                        BGCOLOR=[color]
        TT
        U
        UL
```

Environment variables

This section describes all environment variables that affect execution of wxWidgets programs.

WXTRACE	(Debug build only.) This variable can be set to a comma-separated list of trace masks used in*wxLogTrace* (p. 1651) calls;*wxLog::AddTraceMask* (p. 892) is called for every mask in the list during wxWidgets initialization.
WXPREFIX	(Unix only.) Overrides installation prefix. Normally, the prefix is hard-coded and is the same as the value passed to `configure` via the `--prefix` switch when compiling the library (typically`/usr/local` or `/usr`). You can set WXPREFIX if you are for example distributing a binary version of an application and you don't know in advance where it will be installed.
WXMODE	(wxMGL only.) Sets MGL video mode. The value must be in form*widthxheight-depth*. The default is `640x480-16`.
WXSTDERR	(wxMGL only.) Redirects stderr output to a file.

wxPython overview

This topic was written by Robin Dunn, author of the wxPython wrapper.

What is wxPython?

wxPython is a blending of the wxWidgets GUI classes and thePython
(`http://www.python.org/`) programming language.

Python

So what is Python? Go tohttp://www.python.org (`http://www.python.org`) to learn
more, but in a nutshell Python is an interpreted, interactive, object-oriented programming
language. It is often compared to Tcl, Perl, Scheme or Java.

Python combines remarkable power with very clear syntax. It has modules, classes,
exceptions, very high level dynamic data types, and dynamic typing. There are
interfaces to many system calls and libraries, and new built-in modules are easily written
in C or C++. Python is also usable as an extension language for applications that need a
programmable interface.

Python is copyrighted but freely usable and distributable, even for commercial use.

wxPython

wxPython is a Python package that can be imported at runtime that includes a collection
of Python modules and an extension module (native code). It provides a series of Python
classes that mirror (or shadow) many of the wxWidgets GUI classes. This extension
module attempts to mirror the class hierarchy of wxWidgets as closely as possible. This
means that there is a wxFrame class in wxPython that looks, smells, tastes and acts
almost the same as the wxFrame class in the C++ version.

wxPython is very versatile. It can be used to create standalone GUI applications, or in
situations where Python is embedded in a C++ application as an internal scripting or
macro language.

Currently wxPython is available for Win32 platforms and the GTK toolkit (wxGTK) on
most Unix/X-windows platforms. See the wxPython website http://wxPython.org/
(`http://wxPython.org/`) for details about getting wxPython working for you.

Why use wxPython?

So why would you want to use wxPython over just C++ and wxWidgets? Personally I
prefer using Python for everything. I only use C++ when I absolutely have to eke more
performance out of an algorithm, and even then I usually code it as an extension module
and leave the majority of the program in Python.

Another good thing to use wxPython for is quick prototyping of your wxWidgets apps.
With C++ you have to continuously go though the edit-compile-link-run cycle, which can
be quite time consuming. With Python it is only an edit-run cycle. You can easily build an

application in a few hours with Python that would normally take a few days or longer with C++. Converting a wxPython app to a C++/wxWidgets app should be a straight forward task.

Other Python GUIs

There are other GUI solutions out there for Python.

Tkinter

Tkinter is the de facto standard GUI for Python. It is available on nearly every platform that Python and Tcl/TK are. Why Tcl/Tk? Well because Tkinter is just a wrapper around Tcl's GUI toolkit, Tk. This has its upsides and its downsides...

The upside is that Tk is a pretty versatile toolkit. It can be made to do a lot of things in a lot of different environments. It is fairly easy to create new widgets and use them interchangeably in your programs.

The downside is Tcl. When using Tkinter you actually have two separate language interpreters running, the Python interpreter and the Tcl interpreter for the GUI. Since the guts of Tcl is mostly about string processing, it is fairly slow as well. (Not too bad on a fast Pentium II, but you really notice the difference on slower machines.)

It wasn't until the latest version of Tcl/Tk that native Look and Feel was possible on non-Motif platforms. This is because Tk usually implements its own widgets (controls) even when there are native controls available.

Tkinter is a pretty low-level toolkit. You have to do a lot of work (verbose program code) to do things that would be much simpler with a higher level of abstraction.

PythonWin

PythonWin is an add-on package for Python for the Win32 platform. It includes wrappers for MFC as well as much of the Win32 API. Because of its foundation, it is very familiar for programmers who have experience with MFC and the Win32 API. It is obviously not compatible with other platforms and toolkits. PythonWin is organized as separate packages and modules so you can use the pieces you need without having to use the GUI portions.

Others

There are quite a few other GUI modules available for Python, some in active use, some that haven't been updated for ages. Most are simple wrappers around some C or C++ toolkit or another, and most are not cross-platform compatible. See this link (`http://www.python.org/download/Contributed.html#Graphics`)for a listing of a few of them.

Using wxPython

First things first...

I'm not going to try and teach the Python language here. You can do that at the Python Tutorial (`http://www.python.org/doc/tut/tut.html`). I'm also going to assume that you know a bit about wxWidgets already, enough to notice the similarities in the classes used.

Take a look at the following wxPython program. You can find a similar program in the `wxPython/demo` directory, named `DialogUnits.py`. If your Python and wxPython are properly installed, you should be able to run it by issuing this command:

`python` **DialogUnits.py**

```
001: ## import all of the wxPython GUI package
002: from wxPython.wx import *
003:
004: ## Create a new frame class, derived from the wxPython Frame.
005: class MyFrame(wxFrame):
006:
007:     def __init__(self, parent, id, title):
008:         # First, call the base class' __init__ method to create the
frame
009:         wxFrame.__init__(self, parent, id, title,
010:                         wxPoint(100, 100), wxSize(160, 100))
011:
012:         # Associate some events with methods of this class
013:         EVT_SIZE(self, self.OnSize)
014:         EVT_MOVE(self, self.OnMove)
015:
016:         # Add a panel and some controls to display the size and
position
017:         panel = wxPanel(self, -1)
018:         wxStaticText(panel, -1, "Size:",
019:                         wxDLG_PNT(panel, wxPoint(4, 4)),  wxDefaultSize)
020:         wxStaticText(panel, -1, "Pos:",
021:                         wxDLG_PNT(panel, wxPoint(4, 14)), wxDefaultSize)
022:         self.sizeCtrl = wxTextCtrl(panel, -1, "",
023:                                     wxDLG_PNT(panel, wxPoint(24, 4)),
024:                                     wxDLG_SZE(panel, wxSize(36, -1)),
025:                                     wxTE_READONLY)
026:         self.posCtrl = wxTextCtrl(panel, -1, "",
027:                                     wxDLG_PNT(panel, wxPoint(24, 14)),
028:                                     wxDLG_SZE(panel, wxSize(36, -1)),
029:                                     wxTE_READONLY)
030:
031:
032:     # This method is called automatically when the CLOSE event is
033:     # sent to this window
034:     def OnCloseWindow(self, event):
035:         # tell the window to kill itself
036:         self.Destroy()
037:
038:     # This method is called by the system when the window is resized,
039:     # because of the association above.
040:     def OnSize(self, event):
041:         size = event.GetSize()
042:         self.sizeCtrl.SetValue("%s, %s" % (size.width, size.height))
043:
044:         # tell the event system to continue looking for an event
handler,
045:         # so the default handler will get called.
```

```
046:            event.Skip()
047:
048:     # This method is called by the system when the window is moved,
049:     # because of the association above.
050:     def OnMove(self, event):
051:         pos = event.GetPosition()
052:         self.posCtrl.SetValue("%s, %s" % (pos.x, pos.y))
053:
054:
055: # Every wxWidgets application must have a class derived from wxApp
056: class MyApp(wxApp):
057:
058:     # wxWidgets calls this method to initialize the application
059:     def OnInit(self):
060:
061:         # Create an instance of our customized Frame class
062:         frame = MyFrame(NULL, -1, "This is a test")
063:         frame.Show(true)
064:
065:         # Tell wxWidgets that this is our main window
066:         self.SetTopWindow(frame)
067:
068:         # Return a success flag
069:         return true
070:
071:
072: app = MyApp(0)      # Create an instance of the application class
073: app.MainLoop()      # Tell it to start processing events
074:
```

Things to notice

1. At line 2 the wxPython classes, constants, and etc. are imported into the current module's namespace. If you prefer to reduce namespace pollution you can use "`from wxPython import wx`" and then access all the wxPython identifiers through the wx module, for example, "`wx.wxFrame`".

2. At line 13 the frame's sizing and moving events are connected to methods of the class. These helper functions are intended to be like the event table macros that wxWidgets employs. But since static event tables are impossible with wxPython, we use helpers that are named the same to dynamically build the table. The only real difference is that the first argument to the event helpers is always the window that the event table entry should be added to.

3. Notice the use of `wxDLG_PNT` and `wxDLG_SZE` in lines 19 - 29 to convert from dialog units to pixels. These helpers are unique to wxPython since Python can't do method overloading like C++.

4. There is an `OnCloseWindow` method at line 34 but no call to EVT_CLOSE to attach the event to the method. Does it really get called? The answer is, yes it does. This is because many of the *standard* events are attached to windows that have the associated *standard* method names. I have tried to follow the lead of the C++ classes in this area to determine what is *standard* but since that changes from time to time I can make no guarantees, nor will it be fully documented. When in doubt, use an EVT_*** function.

5. At lines 17 to 21 notice that there are no saved references to the panel or the static text items that are created. Those of you who know Python might be wondering what happens when Python deletes these objects when they go out of scope. Do they disappear from the GUI? They don't. Remember that in wxPython the Python objects are just shadows of the corresponding C++ objects. Once the C++ windows and controls are attached to their parents, the parents manage them and delete them when necessary. For this reason, most wxPython objects do not need to have a __del__ method that explicitly causes the C++ object to be deleted. If you ever have the need to forcibly delete a window, use the Destroy() method as shown on line 36.

6. Just like wxWidgets in C++, wxPython apps need to create a class derived from `wxApp` (line 56) that implements a method named`OnInit`, (line 59.) This method should create the application's main window (line 62) and use `wxApp.SetTopWindow()` (line 66) to inform wxWidgets about it.

7. And finally, at line 72 an instance of the application class is created. At this point wxPython finishes initializing itself, and calls the `OnInit` method to get things started. (The zero parameter here is a flag for functionality that isn't quite implemented yet. Just ignore it for now.) The call to `MainLoop` at line 73 starts the event loop which continues until the application terminates or all the top level windows are closed.

wxWidgets classes implemented in wxPython

The following classes are supported in wxPython. Most provide nearly full implementations of the public interfaces specified in the C++ documentation, others are less so. They will all be brought as close as possible to the C++ spec over time.

- *wxAcceleratorEntry* (p. 18)

- *wxAcceleratorTable* (p. 19)

- *wxActivateEvent* (p. 30)

- *wxBitmap* (p. 79)

- *wxBitmapButton* (p. 92)

- *wxBitmapDataObject* (p. 97)

- wxBMPHandler

- *wxBoxSizer* (p. 102)

- *wxBrush* (p. 103)

- *wxBusyInfo* (p. 115)

- *wxBusyCursor* (p. 115)

- *wxButton* (p. 117)

- *wxZipFSHandler* (p. 1749)

Where to go for help

Since wxPython is a blending of multiple technologies, help comes from multiple sources. Seehttp://wxpython.org/ (`http://wxpython.org/`) for details on various sources of help, but probably the best source is the wxPython-users mail list. You can view the archive or subscribe by going to

http://lists.wxwindows.org/mailman/listinfo/wxpython-users (`http://lists.wxwindows.org/mailman/listinfo/wxpython-users`)

Or you can send mail directly to the list using this address:

wxpython-users@lists.wxwindows.org

Syntax of the builtin regular expression library

A *regular expression* describes strings of characters. It's a pattern that matches certain strings and doesn't match others.

See also

wxRegEx (p. 1107)

Different Flavors of REs

Syntax of the builtin regular expression library (p. 1866)

Regular expressions ("RE''s), as defined by POSIX, come in two flavors: *extended* REs ("EREs") and *basic* REs ("BREs"). EREs are roughly those of the traditional *egrep*, while BREs are roughly those of the traditional *ed*. This implementation adds a third flavor, *advanced* REs ("AREs"), basically EREs with some significant extensions.

This manual page primarily describes AREs. BREs mostly exist for backward compatibility in some old programs; they will be discussed at the *end* (p. 1875). POSIX EREs are almost an exact subset of AREs. Features of AREs that are not present in EREs will be indicated.

Regular Expression Syntax

Syntax of the builtin regular expression library (p. 1866)

These regular expressions are implemented using the package written by Henry Spencer, based on the 1003.2 spec and some (not quite all) of the Perl5 extensions (thanks, Henry!). Much of the description of regular expressions below is copied verbatim from his manual entry.

An ARE is one or more *branches*, separated by '|', matching anything that matches any of the branches.

A branch is zero or more *constraints* or *quantified atoms*, concatenated. It matches a match for the first, followed by a match for the second, etc; an empty branch matches the empty string.

A quantified atom is an *atom* possibly followed by a single *quantifier*. Without a quantifier, it matches a match for the atom. The quantifiers, and what a so-quantified atom matches, are:

*	a sequence of 0 or more matches of the atom
+	a sequence of 1 or more matches of the atom
?	a sequence of 0 or 1 matches of the atom
{m}	a sequence of exactly *m* matches of the atom
{m,}	a sequence of *m* or more matches of the atom
{m,n}	a sequence of *m* through *n* (inclusive) matches of the atom; *m* may not exceed *n*
*? +? ?? {m}? {m,}? {m,n}?	*non-greedy* quantifiers, which match the same possibilities, but prefer the smallest number rather than the largest number of matches (see *Matching* (p. 1873))

The forms using **{** and **}** are known as *bound*s. The numbers *m* and *n* are unsigned decimal integers with permissible values from 0 to 255 inclusive. An atom is one of:

(re)	(where *re* is any regular expression) matches a match for *re*, with the match noted for possible reporting
(?:re)	as previous, but does no reporting (a "non-capturing" set of parentheses)
()	matches an empty string, noted for possible reporting
(?:)	matches an empty string, without reporting
[chars]	a *bracket expression*, matching any one of the *chars*(see *Bracket Expressions* (p. 1868) for more detail)
.	matches any single character
\k	(where *k* is a non-alphanumeric character) matches that character taken as an ordinary character, e.g. \\ matches a backslash character
\c	where *c* is alphanumeric (possibly followed by other characters), an *escape* (AREs only), see *Escapes* (p. 1870) below
{	when followed by a character other than a digit, matches the left-brace character '{'; when followed by a digit, it is the beginning of

a *bound* (see above)

x   where *x* is a single character with no other significance, matches
                               that character.

A *constraint* matches an empty string when specific conditions are met. A constraint may
not be followed by a quantifier. The simple constraints are as follows; some more
constraints are described later, under *Escapes* (p. 1870).

^   matches at the beginning of a line

$ matches at the end of a line

(?=re)   *positive lookahead*(AREs only), matches at any point where a
                               substring matching *re* begins

(?!re)   *negative lookahead* (AREs only), matches at any point where no
                               substring matching *re* begins

The lookahead constraints may not contain back references (see later), and all
parentheses within them are considered non-capturing.

An RE may not end with '\'.

Bracket Expressions

Syntax of the builtin regular expression library (p. 1866)

A *bracket expression* is a list of characters enclosed in '**[]**'. It normally matches any
single character from the list (but see below). If the list begins with '**^**', it matches any
single character (but see below) *not* from the rest of the list.

If two characters in the list are separated by '**-**', this is shorthand for the full *range* of
characters between those two (inclusive) in the collating sequence, e.g. **[0-9]** in ASCII
matches any decimal digit. Two ranges may not share an endpoint, so e.g. **a-c-e** is
illegal. Ranges are very collating-sequence-dependent, and portable programs should
avoid relying on them.

To include a literal **]** or **-** in the list, the simplest method is to enclose it in **[.** and **.]** to
make it a collating element (see below). Alternatively, make it the first character
(following a possible '**^**'), or (AREs only) precede it with '\'. Alternatively, for '**-**', make it
the last character, or the second endpoint of a range. To use a literal **-** as the first
endpoint of a range, make it a collating element or (AREs only) precede it with '\'. With
the exception of these, some combinations using **[** (see next paragraphs), and escapes,
all other special characters lose their special significance within a bracket expression.

Within a bracket expression, a collating element (a character, a multi-character
sequence that collates as if it were a single character, or a collating-sequence name for
either) enclosed in **[.** and **.]** stands for the sequence of characters of that collating
element.

wxWidgets: Currently no multi-character collating elements are defined. So in **[.X.]**, *X*

can either be a single character literal or the name of a character. For example, the following are both identical **[[.0.]-[.9.]]** and **[[.zero.]-[.nine.]]** and mean the same as **[0-9]**. See *Character Names* (p. 1875).

Within a bracket expression, a collating element enclosed in **[=** and **=]**is an equivalence class, standing for the sequences of characters of all collating elements equivalent to that one, including itself. An equivalence class may not be an endpoint of a range.

wxWidgets: Currently no equivalence classes are defined, so **[=X=]** stands for just the single character *X*. *X* can either be a single character literal or the name of a character, see *Character Names* (p. 1875).

Within a bracket expression, the name of a *character class* enclosed in **[:** and **:]** stands for the list of all characters (not all collating elements!) belonging to that class. Standard character classes are:

alpha	A letter.
upper	An upper-case letter.
lower	A lower-case letter.
digit	A decimal digit.
xdigit	A hexadecimal digit.
alnum	An alphanumeric (letter or digit).
print	An alphanumeric (same as alnum).
blank	A space or tab character.
space	A character producing white space in displayed text.
punct	A punctuation character.
graph	A character with a visible representation.
cntrl	A control character.

A character class may not be used as an endpoint of a range.

wxWidgets: In a non-Unicode build, these character classifications depend on the current locale, and correspond to the values return by the ANSI C 'is' functions: isalpha, isupper, etc. In Unicode mode they are based on Unicode classifications, and are not affected by the current locale.

There are two special cases of bracket expressions: the bracket expressions **[[:<:]]** and **[[:>:]]** are constraints, matching empty strings at the beginning and end of a word respectively. A word is defined as a sequence of word characters that is neither preceded nor followed by word characters. A word character is an *alnum* character or an underscore (_). These special bracket expressions are deprecated; users of AREs should use constraint escapes instead (see *Escapes* (p. 1870) below).

Escapes

Syntax of the builtin regular expression library (p. 1866)

Escapes (AREs only), which begin with a \ followed by an alphanumeric character, come in several varieties: character entry, class shorthands, constraint escapes, and back references. A \ followed by an alphanumeric character but not constituting a valid escape is illegal in AREs. In EREs, there are no escapes: outside a bracket expression, a \ followed by an alphanumeric character merely stands for that character as an ordinary character, and inside a bracket expression, \ is an ordinary character. (The latter is the one actual incompatibility between EREs and AREs.)

Character-entry escapes (AREs only) exist to make it easier to specify non-printing and otherwise inconvenient characters in REs:

\a   alert (bell) character, as in C

\b   backspace, as in C

\B   synonym for \ to help reduce backslash doubling in some applications where there are multiple levels of backslash processing

\c*X*   (where X is any character) the character whose low-order 5 bits are the same as those of *X*, and whose other bits are all zero

\e   the character whose collating-sequence name is '**ESC**', or failing that, the character with octal value 033

\f   formfeed, as in C

\n   newline, as in C

\r   carriage return, as in C

\t   horizontal tab, as in C

\u*wxyz*   (where *wxyz* is exactly four hexadecimal digits) the Unicode character **U+*wxyz*** in the local byte ordering

\U*stuvwxyz*   (where *stuvwxyz* is exactly eight hexadecimal digits) reserved for a somewhat-hypothetical Unicode extension to 32 bits

\v   vertical tab, as in C are all available.

\x*hhh*   (where *hhh* is any sequence of hexadecimal digits) the character whose hexadecimal value is **0x*hhh*** (a single character no matter how many hexadecimal digits are used).

\0   the character whose value is **0**

\xy   (where *xy* is exactly two octal digits, and is not a *back reference* (see below)) the character whose octal value is **0*xy***

\xyz   (where *xyz* is exactly three octal digits, and is not a back reference

(see below)) the character whose octal value is **0***xyz*

Hexadecimal digits are '**0**'-'**9**', '**a**'-'**f**', and '**A**'-'**F**'. Octal digits are '**0**'-'**7**'.

The character-entry escapes are always taken as ordinary characters. For example, **\135** is **]** in ASCII, but **\135** does not terminate a bracket expression. Beware, however, that some applications (e.g., C compilers) interpret such sequences themselves before the regular-expression package gets to see them, which may require doubling (quadrupling, etc.) the '****'.

Class-shorthand escapes (AREs only) provide shorthands for certain commonly-used character classes:

\d   **[[:digit:]]**

\s   **[[:space:]]**

\w   **[[:alnum:]_]** (note underscore)

\D   **[^[:digit:]]**

\S   **[^[:space:]]**

\W   **[^[:alnum:]_]** (note underscore)

Within bracket expressions, '**\d**', '**\s**', and '**\w**' lose their outer brackets, and '**\D**', '**\S**', and '**\W**' are illegal. (So, for example, **[a-c\d]** is equivalent to **[a-c[:digit:]]**. Also, **[a-c\D]**, which is equivalent to **[a-c^[:digit:]]**, is illegal.)

A constraint escape (AREs only) is a constraint, matching the empty string if specific conditions are met, written as an escape:

\A   matches only at the beginning of the string (see *Matching* (p. 1873), below, for how this differs from '**^**')

\m   matches only at the beginning of a word

\M   matches only at the end of a word

\y   matches only at the beginning or end of a word

\Y   matches only at a point that is not the beginning or end of a word

\Z   matches only at the end of the string (see *Matching* (p. 1873), below, for how this differs from '**$**')

m   (where *m* is a nonzero digit) a *back reference*, see below

mnn   (where *m* is a nonzero digit, and *nn* is some more digits, and the decimal value *mnn* is not greater than the number of closing capturing parentheses seen so far) a *back reference*, see below

A word is defined as in the specification of **[[:<:]]** and **[[:>:]]** above. Constraint escapes are illegal within bracket expressions.

A back reference (AREs only) matches the same string matched by the parenthesized subexpression specified by the number, so that (e.g.) **([bc])\1** matches **bb** or **cc** but not '**bc**'. The subexpression must entirely precede the back reference in the RE. Subexpressions are numbered in the order of their leading parentheses. Non-capturing parentheses do not define subexpressions.

There is an inherent historical ambiguity between octal character-entry escapes and back references, which is resolved by heuristics, as hinted at above. A leading zero always indicates an octal escape. A single non-zero digit, not followed by another digit, is always taken as a back reference. A multi-digit sequence not starting with a zero is taken as a back reference if it comes after a suitable subexpression (i.e. the number is in the legal range for a back reference), and otherwise is taken as octal.

Metasyntax

Syntax of the builtin regular expression library (p. 1866)

In addition to the main syntax described above, there are some special forms and miscellaneous syntactic facilities available.

Normally the flavor of RE being used is specified by application-dependent means. However, this can be overridden by a *director*. If an RE of any flavor begins with '*****:**', the rest of the RE is an ARE. If an RE of any flavor begins with '*****=**', the rest of the RE is taken to be a literal string, with all characters considered ordinary characters.

An ARE may begin with *embedded options*: a sequence **(?xyz)**(where *xyz* is one or more alphabetic characters) specifies options affecting the rest of the RE. These supplement, and can override, any options specified by the application. The available option letters are:

b	rest of RE is a BRE
c	case-sensitive matching (usual default)
e	rest of RE is an ERE
i	case-insensitive matching (see *Matching* (p. 1873), below)
m	historical synonym for **n**
n	newline-sensitive matching (see *Matching* (p. 1873), below)
p	partial newline-sensitive matching (see *Matching* (p. 1873), below)
q	rest of RE is a literal ("quoted") string, all ordinary characters
s	non-newline-sensitive matching (usual default)
t	tight syntax (usual default; see below)
w	inverse partial newline-sensitive ("weird") matching (see *Matching* (p. 1873), below)

x   expanded syntax (see below)

Embedded options take effect at the **)** terminating the sequence. They are available only at the start of an ARE, and may not be used later within it.

In addition to the usual (*tight*) RE syntax, in which all characters are significant, there is an *expanded* syntax, available in AREs with the embedded x option. In the expanded syntax, white-space characters are ignored and all characters between a **#** and the following newline (or the end of the RE) are ignored, permitting paragraphing and commenting a complex RE. There are three exceptions to that basic rule:

•white-space character or '**#**' preceded by '\' is retained

•space or '**#**' within a bracket expression is retained

•space and comments are illegal within multi-character symbols like the ARE '**(?:**' or
    the BRE '**\(**' Expanded-syntax white-space characters are blank, tab, newline,
    and any character that belongs to the *space* character class.

Finally, in an ARE, outside bracket expressions, the sequence '**(?#ttt)**' (where *ttt* is any text not containing a '**)**') is a comment, completely ignored. Again, this is not allowed between the characters of multi-character symbols like '**(?:**'. Such comments are more a historical artifact than a useful facility, and their use is deprecated; use the expanded syntax instead.

None of these metasyntax extensions is available if the application (or an initial *****=**director) has specified that the user's input be treated as a literal string rather than as an RE.

Matching

Syntax of the builtin regular expression library (p. 1866)

In the event that an RE could match more than one substring of a given string, the RE matches the one starting earliest in the string. If the RE could match more than one substring starting at that point, its choice is determined by its *preference*: either the longest substring, or the shortest.

Most atoms, and all constraints, have no preference. A parenthesized RE has the same preference (possibly none) as the RE. A quantified atom with quantifier **{m}** or **{m}?** has the same preference (possibly none) as the atom itself. A quantified atom with other normal quantifiers (including **{m,n}** with *m* equal to *n*) prefers longest match. A quantified atom with other non-greedy quantifiers (including **{m,n}?** with *m* equal to *n*) prefers shortest match. A branch has the same preference as the first quantified atom in it which has a preference. An RE consisting of two or more branches connected by the **|** operator prefers longest match.

Subject to the constraints imposed by the rules for matching the whole RE, subexpressions also match the longest or shortest possible substrings, based on their preferences, with subexpressions starting earlier in the RE taking priority over ones starting later. Note that outer subexpressions thus take priority over their component subexpressions.

Note that the quantifiers **{1,1}** and **{1,1}?** can be used to force longest and shortest preference, respectively, on a subexpression or a whole RE.

Match lengths are measured in characters, not collating elements. An empty string is considered longer than no match at all. For example, **bb*** matches the three middle characters of '**abbbc**', **(week|wee)(night|knights)**matches all ten characters of '**weeknights**', when **(.*).*** is matched against **abc** the parenthesized subexpression matches all three characters, and when **(a*)*** is matched against **bc** both the whole RE and the parenthesized subexpression match an empty string.

If case-independent matching is specified, the effect is much as if all case distinctions had vanished from the alphabet. When an alphabetic that exists in multiple cases appears as an ordinary character outside a bracket expression, it is effectively transformed into a bracket expression containing both cases, so that **x** becomes '**[xX]**'. When it appears inside a bracket expression, all case counterparts of it are added to the bracket expression, so that **[x]** becomes **[xX]** and **[^x]** becomes '**[^xX]**'.

If newline-sensitive matching is specified, **.** and bracket expressions using **^** will never match the newline character (so that matches will never cross newlines unless the RE explicitly arranges it) and **^** and **$** will match the empty string after and before a newline respectively, in addition to matching at beginning and end of string respectively. ARE **\A** and **\Z** continue to match beginning or end of string *only*.

If partial newline-sensitive matching is specified, this affects **.** and bracket expressions as with newline-sensitive matching, but not **^** and '**$**'.

If inverse partial newline-sensitive matching is specified, this affects **^** and **$** as with newline-sensitive matching, but not **.** and bracket expressions. This isn't very useful but is provided for symmetry.

Limits And Compatibility

Syntax of the builtin regular expression library (p. 1866)

No particular limit is imposed on the length of REs. Programs intended to be highly portable should not employ REs longer than 256 bytes, as a POSIX-compliant implementation can refuse to accept such REs.

The only feature of AREs that is actually incompatible with POSIX EREs is that \does not lose its special significance inside bracket expressions. All other ARE features use syntax which is illegal or has undefined or unspecified effects in POSIX EREs; the ***** syntax of directors likewise is outside the POSIX syntax for both BREs and EREs.

Many of the ARE extensions are borrowed from Perl, but some have been changed to clean them up, and a few Perl extensions are not present. Incompatibilities of note include '**\b**', '**\B**', the lack of special treatment for a trailing newline, the addition of complemented bracket expressions to the things affected by newline-sensitive matching, the restrictions on parentheses and back references in lookahead constraints, and the longest/shortest-match (rather than first-match) matching semantics.

The matching rules for REs containing both normal and non-greedy quantifiers have changed since early beta-test versions of this package. (The new rules are much simpler

and cleaner, but don't work as hard at guessing the user's real intentions.)

Henry Spencer's original 1986 *regexp* package, still in widespread use, implemented an early version of today's EREs. There are four incompatibilities between *regexp*'s near-EREs ('RREs' for short) and AREs. In roughly increasing order of significance:

- In AREs, \ followed by an alphanumeric character is either an escape or an error, while in RREs, it was just another way of writing the alphanumeric. This should not be a problem because there was no reason to write such a sequence in RREs.

- **{** followed by a digit in an ARE is the beginning of a bound, while in RREs, **{** was always an ordinary character. Such sequences should be rare, and will often result in an error because following characters will not look like a valid bound.

- In AREs, \ remains a special character within '**[]**', so a literal \ within **[]** must be written '\\'. \\ also gives a literal \ within **[]** in RREs, but only truly paranoid programmers routinely doubled the backslash.

- AREs report the longest/shortest match for the RE, rather than the first found in a specified search order. This may affect some RREs which were written in the expectation that the first match would be reported. (The careful crafting of RREs to optimize the search order for fast matching is obsolete (AREs examine all possible matches in parallel, and their performance is largely insensitive to their complexity) but cases where the search order was exploited to deliberately find a match which was *not* the longest/shortest will need rewriting.)

Basic Regular Expressions

Syntax of the builtin regular expression library (p. 1866)

BREs differ from EREs in several respects. '**|**', '**+**', and **?** are ordinary characters and there is no equivalent for their functionality. The delimiters for bounds are **\{** and '**\}**', with **{** and **}** by themselves ordinary characters. The parentheses for nested subexpressions are **\(** and '**\)**', with **(** and **)** by themselves ordinary characters. **^** is an ordinary character except at the beginning of the RE or the beginning of a parenthesized subexpression, **$** is an ordinary character except at the end of the RE or the end of a parenthesized subexpression, and ***** is an ordinary character if it appears at the beginning of the RE or the beginning of a parenthesized subexpression (after a possible leading '**^**'). Finally, single-digit back references are available, and **\<** and **\>** are synonyms for **[[:<:]]** and **[[:>:]]** respectively; no other escapes are available.

Regular Expression Character Names

Syntax of the builtin regular expression library (p. 1866)

Note that the character names are case sensitive.

NUL	'\0'
SOH	'\001'
STX	'\002'
ETX	'\003'
EOT	'\004'
ENQ	'\005'
ACK	'\006'
BEL	'\007'
alert	'\007'
BS	'\010'
backspace	'\b'
HT	'\011'
tab	'\t'
LF	'\012'
newline	'\n'
VT	'\013'
vertical-tab	'\v'
FF	'\014'
form-feed	'\f'
CR	'\015'
carriage-return	'\r'
SO	'\016'
SI	'\017'
DLE	'\020'
DC1	'\021'
DC2	'\022'
DC3	'\023'

DC4	'\024'
NAK	'\025'
SYN	'\026'
ETB	'\027'
CAN	'\030'
EM	'\031'
SUB	'\032'
ESC	'\033'
IS4	'\034'
FS	'\034'
IS3	'\035'
GS	'\035'
IS2	'\036'
RS	'\036'
IS1	'\037'
US	'\037'
space	' '
exclamation-mark	'!'
quotation-mark	'"'
number-sign	'#'
dollar-sign	'$'
percent-sign	'%'
ampersand	'&'
apostrophe	'\''
left-parenthesis	'('
right-parenthesis	')'
asterisk	'*'
plus-sign	'+'
comma	','

hyphen	'-'
hyphen-minus	'-'
period	'.'
full-stop	'.'
slash	'/'
solidus	'/'
zero	'0'
one	'1'
two	'2'
three	'3'
four	'4'
five	'5'
six	'6'
seven	'7'
eight	'8'
nine	'9'
colon	':'
semicolon	';'
less-than-sign	'<'
equals-sign	'='
greater-than-sign	'>'
question-mark	'?'
commercial-at	'@'
left-square-bracket	'['
backslash	'\'
reverse-solidus	'\'
right-square-bracket	']'
circumflex	'^'
circumflex-accent	'^'

underscore	'_'
low-line	'_'
grave-accent	'`'
left-brace	'{'
left-curly-bracket	'{'
vertical-line	'|'
right-brace	'}'
right-curly-bracket	'}'
tilde	'~'
DEL	'\177'

Archive formats such as zip

The archive classes handle archive formats such as zip, tar, rar and cab. Currently only the wxZip classes are included. wxTar classes are under development at wxCode (`http://wxcode.sf.net`).

For each archive type, there are the following classes (using zip here as an example):

wxZipInputStream (p. 1574)   Input stream

wxZipOutputStream (p. 1577)   Output stream

wxZipEntry (p. 1568)   Holds the meta-data for an entry (e.g. filename, timestamp, etc.)

There are also abstract wxArchive classes that can be used to write code that can handle any of the archive types, see '*Generic archive programming* (p. 1883)'. Also see *wxFileSystem* (p. 1749) for a higher level interface that can handle archive files in a generic way.

The classes are designed to handle archives on both seekable streams such as disk files, or non-seekable streams such as pipes and sockets (see '*Archives on non-seekable streams* (p. 1884)').

See also

wxFileSystem (p. 1749)

Creating an archive

Archive formats such as zip (p. 1879)

Call *PutNextEntry()* (p. 53) to create each new entry in the archive, then write the entry's

data. Another call to PutNextEntry() closes the current entry and begins the next.

For example:

```
wxFFileOutputStream out(_T("test.zip"));
wxZipOutputStream zip(out);
wxTextOutputStream txt(zip);

zip.PutNextEntry(_T("entry1.txt"));
txt << _T("Some text for entry1\n");

zip.PutNextEntry(_T("entry2.txt"));
txt << _T("Some text for entry2\n");
```

Extracting an archive

Archive formats such as zip (p. 1879)

GetNextEntry() (p. 47) returns a pointer to entry object containing the meta-data for the next entry in the archive (and gives away ownership). Reading from the input stream then returns the entry's data. Eof() becomes true after an attempt has been made to read past the end of the entry's data.

When there are no more entries, GetNextEntry() returns NULL and sets Eof().

```
// 'smart pointer' type created with wxDEFINE_SCOPED_PTR_TYPE
wxZipEntryPtr entry;

wxFFileInputStream in(_T("test.zip"));
wxZipInputStream zip(in);

while (entry.reset(zip.GetNextEntry()), entry.get() != NULL)
{
    // access meta-data
    wxString name = entry->GetName();
    // read 'zip' to access the entry's data
}
```

The *smart pointer* (p. 1139) type *wxZipEntryPtr*can be created like this:

```
#include <wx/ptr_scpd.h>
wxDEFINE_SCOPED_PTR_TYPE(wxZipEntry);
```

Modifying an archive

Archive formats such as zip (p. 1879)

To modify an existing archive, write a new copy of the archive to a new file, making any necessary changes along the way and transferring any unchanged entries using *CopyEntry()* (p. 52). For archive types which compress entry data, CopyEntry() is likely to be much more efficient than transferring the data using Read() and Write() since it will copy them without decompressing and recompressing them.

In general modifications are not possible without rewriting the archive, though it may be possible in some limited cases. Even then, rewriting the archive is usually a better choice since a failure can be handled without losing the whole archive. *wxTempFileOutputStream* (p. 1334) can be helpful to do this.

For example to delete all entries matching the pattern "*.txt":

```
wxFFileInputStreamPtr in(new wxFFileInputStream(_T("test.zip")));
wxTempFileOutputStream out(_T("test.zip"));

wxZipInputStream inzip(*in);
wxZipOutputStream outzip(out);

// 'smart pointer' type created with wxDEFINE_SCOPED_PTR_TYPE
wxZipEntryPtr entry;

// transfer any meta-data for the archive as a whole (the zip comment
// in the case of zip)
outzip.CopyArchiveMetaData(inzip);

// call CopyEntry for each entry except those matching the pattern
while (entry.reset(inzip.GetNextEntry()), entry.get() != NULL)
    if (!entry->GetName().Matches(_T("*.txt")))
        if (!outzip.CopyEntry(entry.release(), inzip))
            break;

// close the input stream by releasing the pointer to it, do this
// before closing the output stream so that the file can be replaced
in.reset();

// you can check for success as follows
bool success = inzip.Eof() && outzip.Close() && out.Commit();
```

The *smart pointer* (p. 1139) types *wxZipEntryPtr* and *wxFFileInputStreamPtr* can be created like this:

```
#include <wx/ptr_scpd.h>
wxDEFINE_SCOPED_PTR_TYPE(wxZipEntry);
wxDEFINE_SCOPED_PTR_TYPE(wxFFileInputStream);
```

Looking up an archive entry by name

Archive formats such as zip (p. 1879)

Also see *wxFileSystem* (p. 1749) for a higher level interface that is more convenient for accessing archive entries by name.

To open just one entry in an archive, the most efficient way is to simply search for it linearly by calling *GetNextEntry()* (p. 47) until the required entry is found. This works both for archives on seekable and non-seekable streams.

The format of filenames in the archive is likely to be different from the local filename format. For example zips and tars use unix style names, with forward slashes as the path separator, and absolute paths are not allowed. So if on Windows the file

"C:\MYDIR\MYFILE.TXT" is stored, then when reading the entry back *GetName()* (p. 45) will return "MYDIR\MYFILE.TXT". The conversion into the internal format and back has lost some information.

So to avoid ambiguity when searching for an entry matching a local name, it is better to convert the local name to the archive's internal format and search for that:

```
// 'smart pointer' type created with wxDEFINE_SCOPED_PTR_TYPE
wxZipEntryPtr entry;

// convert the local name we are looking for into the internal format
wxString name = wxZipEntry::GetInternalName(localname);

// open the zip
wxFFileInputStream in(_T("test.zip"));
wxZipInputStream zip(in);

// call GetNextEntry() until the required internal name is found
do {
    entry.reset(zip.GetNextEntry());
}
while (entry.get() != NULL && entry->GetInternalName() != name);

if (entry.get() != NULL) {
    // read the entry's data...
}
```

To access several entries randomly, it is most efficient to transfer the entire catalogue of entries to a container such as a std::map or a *wxHashMap* (p. 685) then entries looked up by name can be opened using the *OpenEntry()* (p. 47) method.

```
WX_DECLARE_STRING_HASH_MAP(wxZipEntry*, ZipCatalog);
ZipCatalog::iterator it;
wxZipEntry *entry;
ZipCatalog cat;

// open the zip
wxFFileInputStream in(_T("test.zip"));
wxZipInputStream zip(in);

// load the zip catalog
while ((entry = zip.GetNextEntry()) != NULL) {
    wxZipEntry*& current = cat[entry->GetInternalName()];
    // some archive formats can have multiple entries with the same
name
    // (e.g. tar) though it is an error in the case of zip
    delete current;
    current = entry;
}

// open an entry by name
if ((it = cat.find(wxZipEntry::GetInternalName(localname))) !=
cat.end()) {
    zip.OpenEntry(*it->second);
    // ... now read entry's data
}
```

To open more than one entry simultaneously you need more than one underlying stream

on the same archive:

```
// opening another entry without closing the first requires another
// input stream for the same file
wxFFileInputStream in2(_T("test.zip"));
wxZipInputStream zip2(in2);
if ((it = cat.find(wxZipEntry::GetInternalName(local2))) != cat.end())
    zip2.OpenEntry(*it->second);
```

Generic archive programming

Archive formats such as zip (p. 1879)

Also see *wxFileSystem* (p. 1749) for a higher level interface that can handle archive files in a generic way.

The specific archive classes, such as the wxZip classes, inherit from the following abstract classes which can be used to write code that can handle any of the archive types:

wxArchiveInputStream (p. 47)   Input stream

wxArchiveOutputStream (p. 51)   Output stream

wxArchiveEntry (p. 44) Holds the meta-data for an entry (e.g. filename)

In order to able to write generic code it's necessary to be able to create instances of the classes without knowing which archive type is being used. So there is a class factory for each archive type, derived from *wxArchiveClassFactory* (p. 42), which can create the other classes.

For example, given *wxArchiveClassFactory* factory*, streams and entries can be created like this:

```
// create streams without knowing their type
wxArchiveInputStreamPtr  inarc(factory->NewStream(in));
wxArchiveOutputStreamPtr outarc(factory->NewStream(out));

// create an empty entry object
wxArchiveEntryPtr        entry(factory->NewEntry());
```

The *smart pointer* (p. 1139) types *wxArchiveInputStreamPtr*,*wxArchiveOutputStreamPtr* and *wxArchiveEntryPtr* would need to have already have been defined, which could be done like this:

```
#include <wx/ptr_scpd.h>
wxDEFINE_SCOPED_PTR_TYPE(wxArchiveInputStream);
wxDEFINE_SCOPED_PTR_TYPE(wxArchiveOutputStream);
wxDEFINE_SCOPED_PTR_TYPE(wxArchiveEntry);
```

The class factory itself can either be created explicitly:

```
wxArchiveClassFactory *factory = new wxZipClassFactory;
```

or using wxWidgets' *RTTI* (p. 1719):

```
wxArchiveClassFactory *MakeFactory(const wxString& type)
{
    wxString name = _T("wx") + type.Left(1).Upper() +
                    type.Mid(1).Lower() + _T("ClassFactory");

    wxObject *pObj = wxCreateDynamicObject(name);
    wxArchiveClassFactory *pcf = wxDynamicCast(pObj,
wxArchiveClassFactory);

    if (!pcf) {
        wxLogError(_T("can't handle '%s' archives"), type.c_str());
        delete pObj;
    }

    return pcf;
}
```

Archives on non-seekable streams

Archive formats such as zip (p. 1879)

In general, handling archives on non-seekable streams is done in the same way as for seekable streams, with a few caveats.

The main limitation is that accessing entries randomly using *OpenEntry()* (p. 47) is not possible, the entries can only be accessed sequentially in the order they are stored within the archive.

For each archive type, there will also be other limitations which will depend on the order the entries' meta-data is stored within the archive. These are not too difficult to deal with, and are outlined below.

PutNextEntry and the entry size

When writing archives, some archive formats store the entry size before the entry's data (tar has this limitation, zip doesn't). In this case the entry's size must be passed to *PutNextEntry()* (p. 53) or an error occurs.

This is only an issue on non-seekable streams, since otherwise the archive output stream can seek back and fix up the header once the size of the entry is known.

For generic programming, one way to handle this is to supply the size whenever it is known, and rely on the error message from the output stream when the operation is not supported.

GetNextEntry and the weak reference mechanism

Some archive formats do not store all an entry's meta-data before the entry's data (zip is an example). In this case, when reading from a non-seekable stream, *GetNextEntry()* (p. 47) can only return a partially populated *wxArchiveEntry* (p. 44)object - not all the fields

are set.

The input stream then keeps a weak reference to the entry object and updates it when more meta-data becomes available. A weak reference being one that does not prevent you from deleting the wxArchiveEntry object - the input stream only attempts to update it if it is still around.

The documentation for each archive entry type gives the details of what meta-data becomes available and when. For generic programming, when the worst case must be assumed, you can rely on all the fields of wxArchiveEntry being fully populated when GetNextEntry() returns, with the the following exceptions:

GetSize() (p. 46) Guaranteed to be available after the entry has been read to *Eof()* (p. 815), or *CloseEntry()* (p. 47) has been called

IsReadOnly() (p. 46)   Guaranteed to be available after the end of the archive has been reached, i.e. after GetNextEntry() returns NULL and Eof() is true

This mechanism allows *CopyEntry()* (p. 52)to always fully preserve entries' meta-data. No matter what order order the meta-data occurs within the archive, the input stream will always have read it before the output stream must write it.

wxArchiveNotifier

Notifier objects can be used to get a notification whenever an input stream updates a *wxArchiveEntry* (p. 44) object's data via the weak reference mechanism.

Consider the following code which renames an entry in an archive. This is the usual way to modify an entry's meta-data, simply set the required field before writing it with *CopyEntry()* (p. 52):

```
wxArchiveInputStreamPtr  arc(factory->NewStream(in));
wxArchiveOutputStreamPtr outarc(factory->NewStream(out));
wxArchiveEntryPtr        entry;

outarc->CopyArchiveMetaData(*arc);

while (entry.reset(arc->GetNextEntry()), entry.get() != NULL) {
    if (entry->GetName() == from)
        entry->SetName(to);
    if (!outarc->CopyEntry(entry.release(), *arc))
        break;
}

bool success = arc->Eof() && outarc->Close();
```

However, for non-seekable streams, this technique cannot be used for fields such as *IsReadOnly()* (p. 46), which are not necessarily set when *GetNextEntry()* (p. 47) returns. In this case a *wxArchiveNotifier* (p. 50) can be used:

```
class MyNotifier : public wxArchiveNotifier
{
public:
    void OnEntryUpdated(wxArchiveEntry& entry) {
entry.SetIsReadOnly(false); }
};
```

The meta-data changes are done in your notifier's *OnEntryUpdated()* (p. 51) method, then *SetNotifier()* (p. 46) is called before CopyEntry():

```
wxArchiveInputStreamPtr  arc(factory->NewStream(in));
wxArchiveOutputStreamPtr outarc(factory->NewStream(out));
wxArchiveEntryPtr        entry;
MyNotifier               notifier;

outarc->CopyArchiveMetaData(*arc);

while (entry.reset(arc->GetNextEntry()), entry.get() != NULL) {
    entry->SetNotifier(notifier);
    if (!outarc->CopyEntry(entry.release(), *arc))
        break;
}

bool success = arc->Eof() && outarc->Close();
```

SetNotifier() calls OnEntryUpdated() immediately, then the input stream calls it again whenever it sets more fields in the entry. Since OnEntryUpdated() will be called at least once, this technique always works even when it is not strictly necessary to use it. For example, changing the entry name can be done this way too and it works on seekable streams as well as non-seekable.

Chapter 12 Platform details

wxWidgets defines a common API across platforms, but uses the native graphical user interface (GUI) on each platform, so your program will take on the native look and feel that users are familiar with. Unfortunately native toolkits and hardware do not always support the functionality that the wxWidgets API requires. This chapter collects notes about differences among supported platforms and ports.

wxGTK port

wxGTK is a port of wxWidgets using the GTK+ library available from www.gtk.org. It makes use of GTK+'s native widgets wherever possible and uses wxWidgets' generic controls when needed. GTK+ itself has been ported to a number of systems, but so far only the original X11 version is supported. Support for the recently released GTK+ 2.0 including Unicode support is work in progress.

You will need GTK+ 1.2.3 or higher which is available from:

http://www.gtk.org (`http://www.gtk.org`)

In order to configure wxWidgets to compile wxGTK you will need to type:

```
configure --with-gtk
```

For further information, please see the files in docs/gtk in the distribution.

wxMSW port

wxMSW is a port of wxWidgets for the Windows platforms including Windows 95, 98, ME, 2000, NT, XP in ANSI and Unicode mode (for Windows 95 through the MSLU extension library). wxMSW ensures native look and feel for XP as well when using wxWidgets version 2.3.3 or higher. wxMSW can be compile with a great variety of compilers including MS VC++, Borland 5.5, MinGW32, Cygwin and Watcom as well as cross-compilation with a Linux hosted MinGW32 tool chain.

For further information, please see the files in docs/msw in the distribution.

wxWinCE

wxWinCE is the name given to wxMSW when compiled on Windows CE devices; most of wxMSW is common to Win32 and Windows CE but there are some simplifications, enhancements, and differences in behaviour.

For building instructions, see docs/msw/wince in the distribution. The rest of this section

documents issues you need to be aware of when programming for Windows CE devices.

General issues for wxWinCE programming

Mobile applications generally have fewer features and simpler user interfaces. Simply omit whole sizers, static lines and controls in your dialogs, and use comboboxes instead of listboxes where appropriate. You also need to reduce the amount of spacing used by sizers, for which you can use a macro such as this:

```
#if defined(__WXWINCE__)
    #define wxLARGESMALL(large,small) small
#else
    #define wxLARGESMALL(large,small) large
#endif

// Usage
topsizer->Add( CreateTextSizer( message ), 0, wxALL, wxLARGESMALL(10,0) );
```

There is only ever one instance of a Windows CE application running, and wxWidgets will take care of showing the current instance and shutting down the second instance if necessary.

You can test the return value of wxSystemSettings::GetScreenType() for a qualitative assessment of what kind of display is available, or use wxGetDisplaySize() if you need more information.

You can also use wxGetOsVersion to test for a version of Windows CE at run-time (see the next section). However, because different builds are currently required to target different kinds of device, these values are hard-wired according to the build, and you cannot dynamically adapt the same executable for different major Windows CE platforms. This would require a different approach to the way wxWidgets adapts its behaviour (such as for menubars) to suit the style of device.

See the "Life!" example (demos/life) for an example of an application that has been tailored for PocketPC and Smartphone use.

Note: don't forget to have this line in your .rc file, as for desktop Windows applications:

```
#include "wx/msw/wx.rc"
```

Testing for WinCE SDKs

Use these preprocessor symbols to test for the different types of device or SDK:

__SMARTPHONE__   Generic mobile devices with phone buttons and a small display

__PDA__   Generic mobile devices with no phone

__HANDHELDPC__   Generic mobile device with a keyboard

__WXWINCE__   Microsoft-powered Windows CE devices, whether PocketPC, Smartphone or Standard SDK

WIN32_PLATFORM_WFSP Microsoft-powered smartphone

__POCKETPC__Microsoft-powered PocketPC devices with touch-screen

__WINCE_STANDARDSDK__   Microsoft-powered Windows CE devices, for generic Windows CE applications

__WINCE_NET__   Microsoft-powered Windows CE .NET devices (_WIN32_WCE is 400 or greater)

wxGetOsVersion will return these values:

wxWINDOWS_POCKETPC   The application is running under PocketPC.

wxWINDOWS_SMARTPHONE   The application is running under Smartphone.

wxWINDOWS_CE   The application is running under Windows CE (built with the Standard SDK).

Window sizing in wxWinCE

When creating frames and dialogs, create them with wxDefaultPosition and wxDefaultSize, which will tell WinCE to create them full-screen.

Don't call Fit() and Centre(), so the content sizes to the window rather than fitting the window to the content. (We really need a single API call that will do the right thing on each platform.)

If the screen orientation changes, the windows will automatically be resized so no further action needs to be taken (unless you want to change the layout according to the orientation, which you could detect in idle time, for example). However, if the input panel (SIP) is shown, windows do not yet resize accordingly. This will be implemented soon.

Closing top-level windows in wxWinCE

You won't get a wxCloseEvent when the user clicks on the X in the titlebar on Smartphone and PocketPC; the window is simply hidden instead. However the system may send the event to force the application to close down.

Hibernation in wxWinCE

Smartphone and PocketPC will send a wxEVT_HIBERNATE to the application object in low memory conditions. Your application should release memory and close dialogs, and wake up again when the next wxEVT_ACTIVATE or wxEVT_ACTIVATE_APP message is received. (wxEVT_ACTIVATE_APP is generated whenever a wxEVT_ACTIVATE event is received in Smartphone and PocketPC, since these platforms do not support WM_ACTIVATEAPP.)

Hardware buttons in wxWinCE

Special hardware buttons are sent to a window via the wxEVT_HOTKEY event under Smartphone and PocketPC. You should first register each required button with *wxWindow::RegisterHotKey* (p. 1523), and unregister the button when you're done with it. For example:

```
win->RegisterHotKey(0, wxMOD_WIN, WXK_SPECIAL1);
```

```
        win->UnregisterHotKey(0);
```

You may have to register the buttons in a wxEVT_ACTIVATE event handler since other applications will grab the buttons.

There is currently no method of finding out the names of the special buttons or how many there are.

Dialogs in wxWinCE

PocketPC dialogs have an OK button on the caption, and so you should generally not repeat an OK button on the dialog. You can add a Cancel button if necessary, but some dialogs simply don't offer you the choice (the guidelines recommend you offer an Undo facility to make up for it). When the user clicks on the OK button, your dialog will receive a wxID_OK event by default. If you wish to change this, call wxDialog::SetAffirmativeId with the required identifier to be used. Or, override wxDialog::DoOK (return false to have wxWidgets simply call Close to dismiss the dialog).

Smartphone dialogs do *not* have an OK button on the caption, and are closed using one of the two menu buttons. You need to assign these using wxTopLevelWindow::SetLeftMenu and wxTopLevelWindow::SetRightMenu, for example:

```
#ifdef __SMARTPHONE__
    SetLeftMenu(wxID_OK);
    SetRightMenu(wxID_CANCEL, _("Cancel"));
#elif defined(__POCKETPC__)
    // No OK/Cancel buttons on PocketPC, OK on caption will close
#else
    topsizer->Add( CreateButtonSizer( wxOK|wxCANCEL ), 0, wxEXPAND |
wxALL, 10 );
#endif
```

For implementing property sheets (flat tabs), use a wxNotebook with wxNB_FLAT|wxNB_BOTTOM and have the notebook left, top and right sides overlap the dialog by about 3 pixels to eliminate spurious borders. You can do this by using a negative spacing in your sizer Add() call. The cross-platform property sheet dialog *wxPropertySheetDialog* (p. 1081) is provided, to show settings in the correct style on PocketPC and on other platforms.

Notifications (bubble HTML text with optional buttons and links) will also be implemented in the future for PocketPC.

Modeless dialogs probably don't make sense for PocketPC and Smartphone, since frames and dialogs are normally full-screen, and a modeless dialog is normally intended to co-exist with the main application frame.

Menubars and toolbars in wxWinCE

Menubars and toolbars in PocketPC

On PocketPC, a frame must always have a menubar, even if it's empty. An empty menubar/toolbar is automatically provided for dialogs, to hide any existing menubar for the duration of the dialog.

Menubars and toolbars are implemented using a combined control, but you can use

essentially the usual wxWidgets API; wxWidgets will combine the menubar and toolbar. However, there are some restrictions:

- You must create the frame's primary toolbar with wxFrame::CreateToolBar, because this uses the special wxToolMenuBar class (derived from wxToolBar) to implement the combined toolbar and menubar. Otherwise, you can create and manage toolbars using the wxToolBar class as usual, for example to implement an optional formatting toolbar above the menubar as Pocket Word does. But don't assign a wxToolBar to a frame using SetToolBar - you should always use CreateToolBar for the main frame toolbar.

- Deleting and adding tools to wxToolMenuBar after Realize is called is not supported.

- For speed, colours are not remapped to the system colours as they are in wxMSW. Provide the tool bitmaps either with the correct system button background, or with transparency (for example, using XPMs).

- Adding controls to wxToolMenuBar is not supported. However, wxToolBar supports controls.

Unlike in all other ports, a wxDialog has a wxToolBar, automatically created for you. You may either leave it blank, or access it with wxDialog::GetToolBar and add buttons, then calling wxToolBar::Realize. You cannot set or recreate the toolbar.

Menubars and toolbars in Smartphone

On Smartphone, there are only two menu buttons, so a menubar is simulated using a nested menu on the right menu button. Any toolbars are simply ignored on Smartphone.

Closing windows in wxWinCE

The guidelines state that applications should not have a Quit menu item, since the user should not have to know whether an application is in memory or not. The close button on a window does not call the window's close handler; it simply hides the window. However, the guidelines say that the Ctrl+Q accelerator can be used to quit the application, so wxWidgets defines this accelerator by default and if your application handles wxID_EXIT, it will do the right thing.

Control differences on wxWinCE

These controls and styles are specific to wxWinCE:

- **wxTextCtrl** The wxTE_CAPITALIZE style causes a CAPEDIT control to be created, which capitalizes the first letter.

These controls are missing from wxWinCE:

- **wxCheckListBox** This can be implemented using a wxListCtrl in report mode with checked/unchecked images.

- **MDI classes** MDI is not supported under Windows CE.

- **wxMiniFrame** Not supported under Windows CE.

Tooltips are not currently supported for controls, since on PocketPC controls with tooltips are distinct controls, and it will be hard to add dynamic tooltip support.

Control borders on PocketPC and Smartphone should normally be specified with wxSIMPLE_BORDER instead of wxSUNKEN_BORDER. Controls will usually adapt appropriately by virtue of their GetDefaultBorder() function, but if you wish to specify a style explicitly you can use wxDEFAULT_CONTROL_BORDER which will give a simple border on PocketPC and Smartphone, and the sunken border on other platforms.

Online help in wxWinCE

You can use the help controller wxWinceHelpController which controls simple `.htm` files, usually installed in the Windows directory. See the Windows CE reference for how to format the HTML files.

Installing your PocketPC and Smartphone applications

To install your application, you need to build a CAB file using the parameters defined in a special .inf file. The CabWiz program in your SDK will compile the CAB file from the .inf file and files that it specifies.

For delivery, you can simply ask the user to copy the CAB file to the device and execute the CAB file using File Explorer. Or, you can write a program for the desktop PC that will find the ActiveSync Application Manager and install the CAB file on the device, which is obviously much easier for the user.

Here are some links that may help.

- A setup builder that takes CABs and builds a setup program is at http://www.eskimo.com/~scottlu/win/index.html (`http://www.eskimo.com/~scottlu/win/index.html`).

- Sample installation files can be found in `Windows CE Tools/wce420/POCKET PC 2003/Samples/Win32/AppInst`.

- An installer generator using wxPython can be found at http://ppcquicksoft.iespana.es/ppcquicksoft/myinstall.html (`http://ppcquicksoft.iespana.es/ppcquicksoft/myinstall.html`).

- Miscellaneous Windows CE resources can be found at http://www.orbworks.com/pcce/resources.html (`http://www.orbworks.com/pcce/resources.html`).

- Installer creation instructions with a setup.exe for installing to PPC can be found at http://www.pocketpcdn.com/articles/creatingsetup.html (`http://www.pocketpcdn.com/articles/creatingsetup.html`).

- Microsoft instructions are at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnce30/html/appinstall30.asp?frame=true&hidetoc=true (`http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnce30/html/appinstall30.asp?frame=true&hidetoc=true`).

- Troubleshooting WinCE application installations: http://support.microsoft.com/default.aspx?scid=KB;en-us;q181007 (`http://support.microsoft.com/default.aspx?scid=KB;en-us;q181007`)

You may also check out `demos/life/setup/wince` which contains scripts to create a PocketPC installation for ARM-based devices. In particular, `build.bat` builds the distribution and copies it to a directory called `Deliver`.

wxFileDialog in PocketPC

Allowing the user to access files on memory cards, or on arbitrary parts of the filesystem, is a pain; the standard file dialog only shows folders under My Documents or folders on memory cards (not the system or card root directory, for example). This is a known problem for PocketPC developers, and a wxFileDialog replacement will need to be written.

Remaining issues

These are some of the remaining problems to be sorted out, and features to be supported.

- **Font dialog.** The generic font dialog is currently used, which needs to be simplified (and speeded up).

- **Sizer speed.** Particularly for dialogs containing notebooks, layout seems slow. Some analysis is required.

- **Notification boxes.** The balloon-like notification messages, and their icons, should be implemented. This will be quite straightforward.

- **SIP size.** We need to be able to get the area taken up by the SIP (input panel), and the remaining area, by calling SHSipInfo. We also may need to be able to show and hide the SIP programmatically, with SHSipPreference. See also the *Input Dialogs* topic in the *Programming Windows CE* guide for more on this, and how to have dialogs show the SIP automatically using the WC_SIPREF control.

- **wxStaticBitmap.** The About box in the "Life!" demo shows a bitmap that is the correct size on the emulator, but too small on a VGA Pocket Loox device.

- **wxStaticLine.** Lines don't show up, and the documentation suggests that missing styles are implemented with WM_PAINT.

- **wxCheckListBox.** This class needs to be implemented in terms of a wxListCtrl in report mode, using icons for checkbox states. This is necessary because owner-draw listboxes are not supported on Windows CE.

- **wxFileDialog.** A more flexible dialog needs to be written (probably using wxGenericFileDialog) that can access arbitrary locations.

- **HTML control.** PocketPC has its own HTML control which can be used for showing local pages or navigating the web. We should create a version of wxHtmlWindow that uses this control, or have a separately-named control (wxHtmlCtrl), with a syntax as close as possible to wxHtmlWindow.

- **Tooltip control.** PocketPC uses special TTBUTTON and TTSTATIC controls for adding tooltips, with the tooltip separated from the label with a double tilde. We need to support this using SetToolTip. (Unfortunately it does not seem possible to dynamically remove the tooltip, so an extra style may be required.)

- **Focus.** In the wxPropertySheetDialog demo on Smartphone, it's not possible to navigate between controls. The focus handling in wxWidgets needs investigation. See in particular src/common/containr.cpp, and note that the default OnActivate handler in src/msw/toplevel.cpp sets the focus to the first child of the dialog.

- **OK button.** We should allow the OK button on a dialog to be optional, perhaps by using wxCLOSE_BOX to indicate when the OK button should be displayed.

- **Dynamic adaptation.** We should probably be using run-time tests more than preprocessor tests, so that the same WinCE application can run on different versions of the operating system.

- **Modeless dialogs.** When a modeless dialog is hidden with the OK button, it doesn't restore the frame's menubar. See for example the find dialog in the dialogs sample. However, the menubar is restored if pressing Cancel (the window is closed). This reflects the fact that modeless dialogs are not very useful on Windows CE; however, we could perhaps destroy/restore a modeless dialog's menubar on deactivation and activation.

- **Home screen plugins.** Figure out how to make home screen plugins for use with wxWidgets applications (see `http://www.codeproject.com/ce/CTodayWindow.asp` for inspiration). Although we can't use wxWidgets to create the plugin (too large), we could perhaps write a generic plugin that takes registry information from a given application, with options to display information in a particular way using icons and text from a specified location.

- **Further abstraction.** We should be able to abstract away more of the differences between desktop and mobile applications, in particular for sizer layout.

wxMac port

wxMac is a port of wxWidgets for the Macintosh OS platform. Currently MacOS 8.6 or higher, MacOS 9.0 or higher and MacOS X 10.0 or higher are supported, although most development effort goes into MacOS X support. wxMac can be compiled both using Apple's developer tools and MetroWerks CodeWarrior in different versions. Support for MacOS 8.X and MacOS 9.X is only available through CodeWarrior. wxMac uses the Carbon API (and optionally the Classic API under MacOS 8.X). You will need wxWidgets version 2.3.3 or higher for a stable version of wxMac.

For further information, please see the files in docs/mac in the distribution.

wxPalmOS port

wxPalmOS is a port of wxWidgets for the Palm OS 6 (Cobalt). It ensures native look and feel for Palm devices when using wxWidgets version 2.5.4 or higher. wxPalmOS can be compiled with freely distributable Palm OS Developer Studio.

For further information, please see the files in docs/palmos in the distribution.

wxOS2 port

wxOS2 is a port of wxWidgets for the IBM OS/2 platform. It is currently under construction.

wxMGL port

wxMGL is a port of wxWidgets using the MGL library available from SciTech as the underlying graphics backend. wxMGL draws its widgets using the wxUniversal widget set which is now part of wxWidgets. MGL itself runs on a variety of platforms including DOS, Linux hardware (similar to the Linux framebuffer) and various graphics systems such as Win32, X11 and OS/2. Note that currently MGL for Linux runs only on x86-based systems.

You will need wxWidgets 2.3.3 or higher and MGL 5.0 or higher. The latter is available from

http://www.scitechsoft.com/products/product_download.html (http://www.scitechsoft.com/products/product_download.html)

In order to configure wxWidgets to compile wxMGL you will need to type:

```
configure --with-mgl --with-universal
```

Under DOS, wxMGL uses a dmake based make system.

For further information, please see the files in docs/mgl in the distribution.

wxX11 port

wxX11 is a port of wxWidgets using X11 (The X Window System) as the underlying graphics backend. wxX11 draws its widgets using the wxUniversal widget set which is now part of wxWidgets. wxX11 is well-suited for a number of special applications such as those running on systems with few resources (PDAs) or for applications which need to use a special themed look. You will need wxWidgets 2.3.2 or higher.

In order to configure wxWidgets to compile wxX11 you will need to type:

```
configure --with-x11 --with-universal
```

For further information, please see the files in docs/x11 in the distribution. There is also a page on the use of wxWidgets for embedded applications on the wxWidgets web site.

Chapter 13 Index

—K—

—L—

—M—

—X—

—Y—