

wxWindows 2.2: A portable C++ and Python GUI toolkit

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

November 12th 2000

Contents

Copyright notice	xiv
Introduction	1
What is wxWindows?	1
Why another cross-platform development tool?	1
Changes from version 1.xx	3
Changes from version 2.0.....	4
wxWindows requirements.....	4
Availability and location of wxWindows	5
Acknowledgments.....	5
Multi-platform development with wxWindows	7
Include files	7
Libraries	7
Configuration	8
Makefiles.....	8
Windows-specific files.....	9
Allocating and deleting wxWindows objects	9
Architecture dependency	10
Conditional compilation.....	10
C++ issues	11
File handling	12
Programming strategies	13
Strategies for reducing programming errors.....	13
Strategies for portability	13
Strategies for debugging.....	14
Alphabetical class reference	16
wxAcceleratorEntry	16
wxAcceleratorTable	17
wxActivateEvent	20
wxApp	21
wxArray	32
wxArrayString	44
wxAutomationObject.....	49
wxBitmap	54
wxBitmapHandler.....	66

wxBitmapButton	70
wxBitmapDataObject	75
wxBoolFormValidator	76
wxBoolListValidator	77
wxBoxSizer	77
wxBrush	80
wxBrushList	85
wxBusyCursor	87
wxBusyInfo	88
wxButton	89
wxBufferedInputStream	92
wxBufferedOutputStream	93
wxCalculateLayoutEvent	94
wxCalendarCtrl	95
wxCalendarDateAttr	101
wxCalendarEvent	104
wxCaret	105
wxCheckBox	108
wxCheckListBox	111
wxChoice	113
wxClassInfo	119
wxClientDC	120
wxClipboard	121
wxCloseEvent	124
wxCmdLineParser	126
wxColour	135
wxColourData	138
wxColourDatabase	140
wxColourDialog	142
wxComboBox	143
wxCommand	151
wxCommandEvent	152
wxCommandProcessor	158
wxCondition	160
wxConfigBase	162
wxControl	176
wxCountingOutputStream	177
wxCriticalSection	178
wxCriticalSectionLocker	179
wxCSCConv	180

wxCustomDataObject	181
wxCursor	184
wxDatabase	188
wxDataFormat	194
wxDataObject	196
wxDataObjectComposite	200
wxDataObjectSimple	201
wxDataInputStream	203
wxDataOutputStream	205
wxDate	206
wxDateSpan	214
wxDateTime	215
wxDateTimeHolidayAuthority	242
wxDateTimeWorkDays	242
wxDb	242
wxDbColInf	265
wxDbColFor	265
wxDbInf	266
wxDbTable	266
wxDbTableInf	280
wxDC	280
wxDDEClient	298
wxDDEConnection	299
wxDDEServer	303
wxDebugContext	304
wxDebugStreamBuf	310
wxDialog	310
wxDialUpEvent	318
wxDialUpManager	319
wxDir	323
wxDirDialog	325
wxDllLoader	327
wxDocChildFrame	330
wxDocManager	332
wxDocMDIChildFrame	341
wxDocMDIParentFrame	343
wxDocParentFrame	344
wxDocTemplate	345
wxDocument	351
wxDragImage	359

wxDropFilesEvent	364
wxDropSource	365
wxDropTarget	368
wxEncodingConverter.....	371
wxEraseEvent	374
wxEvent	375
wxEvtHandler.....	378
wxExpr	385
wxExprDatabase.....	392
wxFile.....	395
wxFFile.....	402
wxFileDataObject.....	406
wxFileDialog	407
wxFileDropTarget	412
wxFileHistory	413
wxFileInputStream	416
wxFileOutputStream	417
wxFileStream	418
wxFFileInputStream.....	419
wxFFileOutputStream	420
wxFFileStream.....	421
wxFilenameListValidator.....	422
wxFileSystem.....	422
wxFileSystemHandler	424
wxFileType.....	427
wxFlexGridSizer.....	431
wxFilterInputStream.....	432
wxFilterOutputStream.....	432
wxFocusEvent	433
wxFont	434
wxFontData.....	441
wxFontDialog	444
wxFontEnumerator	445
wxFontList.....	447
wxFontMapper	448
wxFrame	452
wxFSFile	464
wxFTP.....	466
wxGauge.....	470
wxGDIObject.....	474

wxGLCanvas	475
wxGenericValidator.....	477
wxGrid.....	479
wxGridSizer	492
wxHashTable	493
wxHelpController	495
wxHtmlCell.....	501
wxHtmlColourCell	506
wxHtmlContainerCell	507
wxHtmlDCRenderer	512
wxHtmlEasyPrinting.....	515
wxHtmlFilter	518
wxHtmlHelpController	519
wxHtmlHelpData	523
wxHtmlHelpFrame	525
wxHtmlLinkInfo	529
wxHtmlParser	530
wxHtmlPrintout.....	534
wxHtmlTag.....	536
wxHtmlTagHandler	539
wxHtmlTagsModule	541
wxHtmlWidgetCell.....	542
wxHtmlWindow	542
wxHtmlWinParser	548
wxHtmlWinTagHandler	555
wxHTTP	555
wxIdleEvent	557
wxIcon.....	558
wxImage.....	565
wxImageHandler.....	580
wxImageList.....	584
wxIndividualLayoutConstraint	588
wxInitDialogEvent	591
wxInputStream.....	592
wxIntegerFormValidator.....	594
wxIntegerListValidator	595
wxIPv4address.....	595
wxJoystick.....	597
wxJoystickEvent	604
wxKeyEvent	607

wxLayoutAlgorithm	610
wxLayoutConstraints	613
wxList	615
wxListBox	621
wxListCtrl	630
wxListEvent	644
wxListOfStringsListValidator	647
wxLocale	648
wxLog	651
wxLongLong	656
wxMask	659
wxMBCConv	661
wxMBCConvFile	664
wxMBCConvUTF7	665
wxMBCConvUTF8	665
wxMDIChildFrame	666
wxMDIClientWindow	670
wxMDIParentFrame	671
wxMemoryDC	678
wxMemoryFSHandler	680
wxMemoryInputStream	681
wxMemoryOutputStream	682
wxMenu	683
wxMenuBar	693
wxMenuItem	702
wxMenuEvent	707
wxMessageDialog	709
wxMetafile	710
wxMetafileDC	712
wxMimeTypesManager	713
wxMiniFrame	716
wxModule	719
wxMouseEvent	721
wxMoveEvent	729
wxMultipleChoiceDialog	730
wxMutex	730
wxMutexLocker	733
wxNotebookSizer	734
wxNodeBase	735
wxNotebook	736

wxNotebookEvent	743
wxNotifyEvent	745
wxObject	746
wxObjectRefData	750
wxOutputStream	751
wxPageSetupDialogData	752
wxPageSetupDialog	758
wxPaintDC	759
wxPaintEvent	760
wxPalette	761
wxPanel	764
wxPanelTabView	767
wxPathList	769
wxPen	771
wxPenList	778
wxPlotCurve	780
wxPlotWindow	782
wxPoint	785
wxPostScriptDC	786
wxPreviewCanvas	787
wxPreviewControlBar	788
wxPreviewFrame	790
wxPrintData	792
wxPrintDialog	797
wxPrintDialogData	799
wxPrinter	803
wxPrinterDC	806
wxPrintout	807
wxPrintPreview	810
wxPrivateDropTarget	814
wxProcess	815
wxProgressDialog	818
wxProcessEvent	820
wxProperty	821
wxPropertyFormDialog	824
wxPropertyFormFrame	824
wxPropertyFormPanel	825
wxPropertyFormValidator	826
wxPropertyFormView	827
wxPropertyListDialog	830

wxPropertyListFrame	830
wxPropertyListPanel	831
wxPropertyListValidator	832
wxPropertyListView	834
wxPropertySheet	837
wxPropertyValidator	839
wxPropertyValidatorRegistry	840
wxPropertyValue	841
wxPropertyView	846
wxProtocol	849
wxQueryCol	851
wxQueryField	854
wxQueryLayoutInfoEvent	856
wxRadioButton	858
wxRadioButton	864
wxRealFormValidator	867
wxRealListValidator	867
wxRealPoint	868
wxRect	868
wxRecordSet	872
wxRegion	885
wxRegionIterator	889
wxSashEvent	892
wxSashLayoutWindow	894
wxSashWindow	897
wxScreenDC	901
wxScrollBar	903
wxScrollWinEvent	908
wxScrollEvent	909
wxScrolledWindow	911
wxSingleChoiceDialog	919
wxSize	922
wxSizeEvent	923
wxSizer	924
wxSlider	929
wxSocketAddress	937
wxSocketBase	938
wxSocketClient	956
wxSocketEvent	958
wxSocketServer	959

wxSocketInputStream.....	962
wxSocketOutputStream.....	962
wxSpinButton.....	963
wxSpinCtrl.....	966
wxSpinEvent.....	969
wxSplitterEvent.....	970
wxSplitterWindow.....	973
wxStaticBitmap.....	982
wxStaticBox.....	985
wxStaticBoxSizer.....	986
wxStaticLine.....	987
wxStaticText.....	989
wxStatusBar.....	991
wxStopWatch.....	997
wxStreamBase.....	998
wxStreamBuffer.....	1000
wxString.....	1007
wxStringFormValidator.....	1029
wxStringList.....	1029
wxStringListValidator.....	1031
wxStringTokenizer.....	1032
wxSysColourChangedEvent.....	1034
wxSystemSettings.....	1035
wxTabbedDialog.....	1038
wxTabbedPanel.....	1040
wxTabControl.....	1041
wxTabView.....	1044
wxTabCtrl.....	1052
wxTabEvent.....	1058
wxTaskBarIcon.....	1059
wxTCPClient.....	1061
wxTCPConnection.....	1063
wxTCPServer.....	1067
wxTempFile.....	1068
wxTextCtrl.....	1070
wxTextDataObject.....	1083
wxTextInputStream.....	1084
wxTextOutputStream.....	1087
wxTextEntryDialog.....	1089
wxTextDropTarget.....	1090

wxTimeSpan	1092
wxTextValidator	1092
wxTextFile	1095
wxThread	1101
wxTime	1108
wxTimer	1113
wxTimerEvent	1115
wxTipProvider	1116
wxToolBar	1117
wxToolTip	1133
wxTreeCtrl	1134
wxTreeItemData	1149
wxTreeEvent	1151
wxTreeLayout	1152
wxTreeLayoutStored	1159
wxUpdateUIEvent	1160
wxURL	1164
wxValidator	1166
wxVariant	1169
wxVariantData	1177
wxView	1179
wxWave	1183
wxWindow	1184
wxWindowDC	1234
wxWindowDisabler	1235
wxWizard	1235
wxWizardEvent	1238
wxWizardPage	1239
wxWizardPageSimple	1241
wxZipInputStream	1242
wxZlibInputStream	1243
wxZlibOutputStream	1243
Functions	1245
Version macros	1245
Thread functions	1246
File functions	1247
Network functions	1252
User identification	1253
String functions	1254

Dialog functions	1256
GDI functions	1262
Printer settings	1263
Clipboard functions	1266
Miscellaneous functions	1268
Macros	1285
wxWindows resource functions	1293
Log functions	1297
Time functions	1300
Debugging macros and functions	1302
Keycodes	1304
Classes by category	1306
Topic overviews	1316
Notes on using the reference	1316
Writing a wxWindows application: a rough guide	1316
wxWindows "Hello World"	1317
wxWindows samples	1320
wxApp overview	1328
Run time class information overview	1329
wxString overview	1331
Date and time classes overview	1336
Unicode support in wxWindows	1340
wxMBCnv classes overview	1343
Internationalization	1346
Writing non-English applications	1347
Container classes overview	1349
File classes and functions overview	1350
wxStreams overview	1351
wxLog classes overview	1353
Debugging overview	1356
wxConfig classes overview	1358
wxExpr overview	1359
wxFileSystem	1362
Event handling overview	1364
Window styles	1371
Window deletion overview	1371
wxDialog overview	1373
wxValidator overview	1374
Constraints overview	1376

The wxWindows resource system	1379
Scrolling overview	1386
Bitmaps and icons overview	1388
Device context overview	1391
wxFont overview	1392
Font encoding overview	1393
wxSplitterWindow overview	1394
wxTreeCtrl overview	1396
wxListCtrl overview	1397
wxImageList overview	1398
Common dialogs overview	1398
Document/view overview	1402
wxTab classes overview	1408
wxTabView overview	1412
Toolbar overview	1412
wxGrid classes overview	1417
wxTipProvider overview	1418
Printing overview	1419
Multithreading overview	1420
Drag and drop overview	1420
wxDataObject overview	1422
Database classes overview	1423
Interprocess communication overview	1428
wxHTML Notes	1432
wxHTML quick start	1432
HTML Printing	1433
Help Files Format	1433
Input Filters	1435
Cells and Containers	1435
Tag Handlers	1436
Tags supported by wxHTML	1439
Property sheet classes	1443
Introduction	1443
Headers	1445
Topic overviews	1445
Classes by category	1453
wxPython Notes	1455
What is wxPython?	1455

Why use wxPython?	1455
Other Python GUIs	1456
Using wxPython	1457
wxWindows classes implemented in wxPython.....	1459
Where to go for help	1463
Porting from wxWindows 1.xx	1465
Preparing for version 2.0	1465
The new event system.....	1467
Class hierarchy	1468
GDI objects	1468
Dialogs and controls	1468
Device contexts and painting.....	1470
Miscellaneous	1470
Backward compatibility	1471
Quick reference	1471
References.....	1476
Index.....	1478

Chapter 1 Copyright notice

(c) 1999 Julian Smart, Robert Roebing, Vadim Zeitlin and other members of the
wxWindows team
Portions (c) 1996 Artificial Intelligence Applications Institute

Please also see the wxWindows license files (preamble.txt, lgpl.txt, gpl.txt, license.txt, licendoc.txt) for conditions of software and documentation use.

wxWindows Library License, Version 3

Copyright (C) 1998 Julian Smart, Robert Roebing, 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.

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

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.

END OF TERMS AND CONDITIONS

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 wxWindows?

wxWindows is a C++ framework providing GUI (Graphical User Interface) and other facilities on more than one platform. Version 2 currently supports MS Windows (16-bit, Windows 95 and Windows NT), Unix with GTK+, Unix with Motif, and Mac. An OS/2 port is in progress.

wxWindows was originally developed at the Artificial Intelligence Applications Institute, University of Edinburgh, for internal use, and was first made publicly available in 1993. Version 2 is a vastly improved version written and maintained by Julian Smart, Robert Roebing, Vadim Zeitlin and many others.

This manual discusses wxWindows in the context of multi-platform development.

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?

wxWindows 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 wxWindows 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 wxWindows has.

As open source software, wxWindows has benefited from comments, ideas, bug fixes, enhancements and the sheer enthusiasm of users. This gives wxWindows 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. wxWindows helps to insulate the programmer from these winds of change. Although wxWindows 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 wxWindows even if they are developing on only one platform.

It is impossible to sum up the functionality of wxWindows 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).
- Dialog Editor for building dialogs.
- 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).

Changes from version 1.xx

These are a few of the major differences between versions 1.xx and 2.0.

Removals:

- XView is no longer supported;
- all controls (panel items) no longer have labels attached to them;
- wxForm has been removed;
- wxCanvasDC, wxPanelDC removed (replaced by wxClientDC, wxWindowDC, wxPaintDC which can be used for any window);
- wxMultiText, wxTextWindow, wxText removed and replaced by wxTextCtrl;
- classes no longer divided into generic and platform-specific parts, for efficiency.

Additions and changes:

- class hierarchy changed, and restrictions about subwindow nesting lifted;
- header files reorganized to conform to normal C++ standards;
- classes less dependent on each another, to reduce executable size;
- wxString used instead of char* wherever possible;
- the number of separate but mandatory utilities reduced;
- the event system has been overhauled, with virtual functions and callbacks being replaced with MFC-like event tables;
- new controls, such as wxTreeCtrl, wxListCtrl, wxSpinButton;
- less inconsistency about what events can be handled, so for example mouse clicks or key presses on controls can now be intercepted;
- the status bar is now a separate class, wxStatusBar, and is implemented in generic wxWindows code;
- some renaming of controls for greater consistency;
- wxBitmap has the notion of bitmap handlers to allow for extension to new formats without `ifdefing`;
- new dialogs: wxPageSetupDialog, wxFileDialog, wxDirDialog, wxMessageDialog, wxSingleChoiceDialog, wxTextEntryDialog;
- GDI objects are reference-counted and are now passed to most functions by reference, making memory management far easier;
- wxSystemSettings class allows querying for various system-wide properties such as dialog font, colours, user interface element sizes, and so on;
- better platform look and feel conformance;
- toolbar functionality now separated out into a family of classes with the same API;
- device contexts are no longer accessed using `wxWindow::GetDC` - they are created temporarily with the window as an argument;
- events from sliders and scrollbars can be handled more flexibly;
- the handling of window close events has been changed in line with the new event system;
- the concept of *validator* has been added to allow much easier coding of the relationship between controls and application data;

- the documentation has been revised, with more cross-referencing.

Platform-specific changes:

- The Windows header file (`windows.h`) is no longer included by `wxWindows` headers;
- `wx.dll` supported under Visual C++;
- the full range of Windows 95 window decorations are supported, such as modal frame borders;
- MDI classes brought out of `wxFrame` into separate classes, and made more flexible.

Changes from version 2.0

These are a few of the differences between versions 2.0 and 2.2.

Removals:

- GTK 1.0 no longer supported.

Additions and changes:

- Corrected many classes to conform better to documented behaviour.
- Added handlers for more image formats (Now GIF, JPEG, PCX, BMP, XPM, PNG, PNM).
- Improved support for socket and network functions.
- Support for different national font encodings.
- Sizer based layout system.
- HTML widget and help system.
- Added some controls (e.g. `wxSpinCtrl`) and supplemented many.
- Many optical improvements to GTK port.
- Support for menu accelerators in GTK port.
- Enhanced and improved support for scrolling, including child windows.
- Complete rewrite of clipboard and drag and drop classes.
- Improved support for ODBC databases.
- Improved tab traversal in dialogs.

wxWindows requirements

To make use of `wxWindows`, you currently need one or both of the following setups.

(a) PC:

1. A 486 or higher PC running MS Windows.
2. A Windows compiler: most are supported, but please see `install.txt` for details. Supported compilers include Microsoft Visual C++ 4.0 or higher, Borland C++, Cygwin, Metrowerks CodeWarrior.
3. At least 60 MB of disk space.

(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, Motif 1.2 or higher, Lesstif.
3. At least 60 MB of disk space.

Availability and location of wxWindows

wxWindows is available by anonymous FTP and World Wide Web from <ftp://www.remstar.com/pub/wxwin> (<ftp://www.remstar.com/pub/wxwin>) and/or <http://www.wxwindows.org> (<http://www.wxwindows.org>).

You can also buy a CD-ROM using the form on the Web site, or by contacting:

Julian Smart
12 North Street West
Uppingham
Rutland
LE15 9SG
julian.smart@ukonline.co.uk

Acknowledgments

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

We would particularly like to thank the following for their contributions to wxWindows, 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, Michael Bedward, Kai Bendorf, Yura Bidus, Keith Gary Boyce, Chris Breeze, Pete Britton, Ian Brown, C. Buckley, Dmitri Chubraev, Robin Corbet, Cecil Coupe, Andrew Davison, Neil Dudman, Robin Dunn, Hermann Dunkel, Jos van Eijndhoven, 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, 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 Roebing, Dino Scaringella, Jobst Schmalenbach, Arthur Seaton, Paul Shirley, Vaclav Slavik, Stein Somers, Petr Smilauer, Neil Smith, Kari Systä, Arthur Tetzlaff-Deas, Jonathan Tonberg, Jyrki Tuomi, David Webster, Janos Vegh, Andrea Venturoli, Vadim Zeitlin, Xiaokun Zhu, Edward Zimmermann.

'Graphplace', the basis for the wxGraphLayout library, is copyright Dr. Jos T.J. van Eindhoven 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 wxWindows

This chapter describes the practical details of using wxWindows. 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 wxWindows.

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 (those tested are Microsoft Visual C++, Borland C++ and Watcom C++).

Borland precompilation is largely automatic. Visual C++ requires specification of `"wx/wxprec.h"` as the file to use for precompilation. Watcom C++ is automatic apart from the specification of the `.pch` file. Watcom C++ is strange in requiring the precompiled header to be used only for object files compiled in the same directory as that in which the precompiled header was created. Therefore, the wxWindows Watcom C++ makefiles go through hoops deleting and recreating a single precompiled header file for each module, thus preventing an accumulation of many multi-megabyte `.pch` files.

Libraries

The GTK and Motif ports of wxWindow can create either a static library or a shared library on most Unix or Unix-like systems. The static library is called `libwx_gtk.a` and `libwx_motif.a` whereas the name of the shared library is dependent on the system it is created on and the version you are using. The library name for the GTK version of wxWindows 2.2 on Linux and Solaris will be `libwx_gtk-2.2.so.0.0.0`, on HP-UX, it will be `libwx_gtk-2.2.sl`, on AIX just `libwx_gtk.a` etc.

Under Windows, use the library `wx.lib` (release) or `wxd.lib` (debug) for stand-alone Windows applications, or `wxdll.lib` (`wxdlld.lib`) for creating DLLs.

Configuration

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.

Under Unix (GTK and Motif) the corresponding `setup.h` files are generated automatically when configuring the wxWindows using the "configure" script. When using the RPM packages for installing wxWindows on Linux, a correct `setup.h` is shipped in the package and this must not be changed.

Makefiles

At the moment there is no attempt to make Unix makefiles and PC makefiles compatible, i.e. one makefile is required for each environment. The Unix ports use a sophisticated system based on the GNU autoconf tool and this system will create the makefiles as required on the respective platform. Although the makefiles are not identical in Windows, Mac and Unix, care has been taken to make them relatively similar so that moving from one platform to another will be painless.

Sample makefiles for Unix (suffix `.unx`), MS C++ (suffix `.DOS` and `.NT`), Borland C++ (`.BCC` and `.B32`) and Symantec C++ (`.SC`) are included for the library, demos and utilities.

The controlling makefile for wxWindows is in the MS-Windows directory `src/msw` for the different Windows compiler and in the build directory when using the Unix ports. The build directory can be chosen by the user. It is the directory in which the "configure" script is run. This can be the normal base directory (by running `./configure` there) or any other directory (e.g. `../configure` after creating a build-directory in the directory level above the base directory).

Please see the platform-specific `install.txt` file for further details.

Windows-specific files

wxWindows 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:

```
rcinclude "wx/msw/wx.rc"
```

which includes essential internal wxWindows 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.

Module definition file

A module definition file (extension DEF) is required for 16-bit applications, and looks like the following:

```
NAME           Hello
DESCRIPTION    'Hello'
EXETYPE        WINDOWS
STUB           'WINSTUB.EXE'
CODE           PRELOAD MOVEABLE DISCARDABLE
DATA           PRELOAD MOVEABLE MULTIPLE
HEAPSIZE       1024
STACKSIZE      8192
```

The only lines which will usually have to be changed per application are NAME and DESCRIPTION.

Allocating and deleting wxWindows 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 `wxWindows` 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 `wxWindows`, make sure you delete the array explicitly before `wxWindows` 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). `wxWindows` 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 *Macros* (p. 1285) section.

Conditional compilation

One of the purposes of wxWindows 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

wxWindows does not use templates since it is a notoriously unportable feature.

RTTI

wxWindows does not use run-time type information since wxWindows 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 wxWindows 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 wxWindows itself and any wxWindows 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 wxWindows, 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 Programming strategies

This chapter is intended to list strategies that may be useful when writing and debugging wxWindows 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 wxWindows, 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 wxWindows 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 (wxWindows resource files) on different platforms, with slightly different dimensions in each. Or space your panel items out to avoid problems.

Use wxWindows resource files

Use .wrc (wxWindows resource files) where possible, because they can be easily changed independently of source code. Bitmap resources can be set up to load different kinds of bitmap depending on platform (see the section on resource files).

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. 1297).

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 wxWindows debugging facilities

You can use `wxDebugContext` to check for memory leaks and corrupt memory: in fact in debugging mode, wxWindows will automatically check for memory leaks at the end of the program if wxWindows 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. 1302) as part of a 'defensive programming' strategy, scattering `wxASSERT`s 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. 1356) for further information.

Check Windows debug messages

Under Windows, it is worth running your program with `DbgView` (<http://www.sysinternals.com>) running or some other program that shows Windows-generated debug messages. It is possible it will show invalid handles being used. You may have fun seeing what commercial programs cause these normally hidden errors! Microsoft recommend using the debugging version of Windows, which shows up even more problems. However, I doubt it is worth the hassle for most applications. wxWindows is designed to minimize the possibility of such errors, but they can still happen occasionally, slipping through unnoticed because they are not severe enough to cause a crash.

Genetic mutation

If we had sophisticated genetic algorithm tools that could be applied to programming, we could use them. Until then, a common -- if rather irrational -- technique is to just make arbitrary changes to the code until something different happens. You may have an intuition why a change will make a difference; otherwise, just try altering the order of code, comment lines out, anything to get over an impasse. Obviously, this is usually a last resort.

Chapter 5 Alphabetical class reference

wxAcceleratorEntry

An object used by an application wishing to create an *accelerator table* (p. 17).

Derived from

None

Include files

<wx/accel.h>

See also

wxAcceleratorTable (p. 17), *wxWindow::SetAcceleratorTable* (p. 1221)

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. 1304) 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. 1304) 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. 746)

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. 16), *wxWindow::SetAcceleratorTable* (p. 1221)

wxAcceleratorTable::wxAcceleratorTable

wxAcceleratorTable()

Default constructor.

wxAcceleratorTable(const wxAcceleratorTable& *bitmap*)

Copy constructor.

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

Creates from an array of *wxAcceleratorEntry* (p. 16) 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.

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& *accl*)

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

Parameters

accl
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.

wxActivateEvent

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

Derived from

wxEvent (p. 375)
wxObject (p. 746)

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 <code>wxEVT_ACTIVATE</code> event.
EVT_ACTIVATE_APP(func)	Process a <code>wxEVT_ACTIVATE_APP</code> 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)

See also

`wxWindow::OnActivate` (p. 1205), `wxApp::OnActivate` (p. 25), *Event handling overview* (p. 1364)

wxActivateEvent::wxActivateEvent

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

Constructor.

wxActivateEvent::m_active

bool m_active

TRUE if the window or application was activated.

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. 28);
- 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 `wxWindows` 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. 378)
wxObject (p. 746)

Include files

<wx/app.h>

See also

wxApp overview (p. 1328)

wxApp::wxApp

void wxApp()

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

wxApp::~~wxApp

void ~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

char ** 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. 651)

wxApp::Dispatch

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. 29)

wxApp::GetAppName

wxString GetAppName() const

Returns the application name.

Remarks

wxWindows sets this to a reasonable default before calling *wxApp::OnInit* (p. 28), but the application can reset it at will.

wxApp::GetAuto3D

bool GetAuto3D() const

Returns TRUE if 3D control mode is on, FALSE otherwise.

See also

wxApp::SetAuto3D (p. 30)

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. 30)

wxApp::GetExitOnFrameDelete

bool GetExitFrameOnDelete() const

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

See also

wxApp::SetExitOnFrameDelete (p. 31)

wxApp::GetTopWindow

wxWindow * GetTopWindow() const

Returns a pointer to the top window.

Remarks

If the top window hasn't been set using *wxApp::SetTopWindow* (p. 31), this function will find the first top-level window (frame or dialog) and return that.

See also

SetTopWindow (p. 31)

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. 32)

wxApp::GetVendorName

wxString GetVendorName() const

Returns the application's vendor name.

wxApp::ExitMainLoop

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::Initialized

bool Initialized()

Returns TRUE if the application has been initialized (i.e. if *wxApp::OnInit* (p. 28) has returned successfully). This can be useful for error message routines to determine which method of output is best for the current state of the program (some windowing systems may not like dialogs to pop up before the main loop has been entered).

wxApp::MainLoop

int MainLoop()

Called by *wxWindows* 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::OnActivate

void OnActivate(wxActivateEvent& event)

Provide this member function to know whether the application is being activated or deactivated (Windows only).

See also

wxWindow::OnActivate (p. 1205), *wxActivateEvent* (p. 20)

wxApp::OnExit

int OnExit()

Provide this member function for any processing which needs to be done as the application is about to exit.

wxApp::OnCharHook

void OnCharHook(wxKeyEvent& event)

This event handler function is called (under Windows only) to allow the window to intercept keyboard events before they are processed by child windows.

Parameters

event

The keypress event.

Remarks

Use the `wxEVT_CHAR_HOOK` macro in your event table.

If you use this member, you can selectively consume keypress events by calling *wxEvent::Skip* (p. 378) for characters the application is not interested in.

See also

wxKeyEvent (p. 607), *wxWindow::OnChar* (p. 1205), *wxWindow::OnCharHook* (p. 1206), *wxDialog::OnCharHook* (p. 314)

wxApp::OnFatalException

void OnFatalException()

This function may be called if something fatal happens: an unhandled exception under Win32 or a fatal signal under Unix, for example. However, this will not happen by

default: you have to explicitly call *wxHandleFatalExceptions* (p. 1279) 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

wxHandleFatalExcetions (p. 1279)

wxApp::OnIdle

void OnIdle(wxIdleEvent& event)

Override this member function for any processing which needs to be done when the application is idle. You should call *wxApp::OnIdle* from your own function, since this forwards *OnIdle* events to windows and also performs garbage collection for windows whose destruction has been delayed.

wxWindows' strategy for *OnIdle* processing is as follows. After pending user interface events for an application have all been processed, *wxWindows* sends an *OnIdle* event to the application object. *wxApp::OnIdle* itself sends an *OnIdle* event to each application window, allowing windows to do idle processing such as updating their appearance. If either *wxApp::OnIdle* or a window *OnIdle* function requested more time, by calling *wxIdleEvent::RequestMore* (p. 558), *wxWindows* will send another *OnIdle* event to the application object. This will occur in a loop until either a user event is found to be pending, or *OnIdle* requests no more time. Then all pending user events are processed until the system goes idle again, when *OnIdle* is called, and so on.

See also

wxWindow::OnIdle (p. 1211), *wxIdleEvent* (p. 557), *wxWindow::SendIdleEvents* (p. 29)

wxApp::OnEndSession

void OnEndSession(wxCloseEvent& event)

This is an event handler function called when the operating system or GUI session is about to close down. The application has a chance to silently save information, and can optionally close itself.

Use the `EVT_END_SESSION` event table macro to handle query end session events.

The default handler calls *wxWindow::Close* (p. 1190) with a `TRUE` argument (forcing the application to close itself silently).

Remarks

Under X, `OnEndSession` is called in response to the 'die' event.

Under Windows, `OnEndSession` is called in response to the `WM_ENDSESSION` message.

See also

wxWindow::Close (p. 1190), *wxWindow::OnCloseWindow* (p. 1208), *wxCloseEvent* (p. 124), *wxApp::OnQueryEndSession* (p. 28)

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. 31).

Return TRUE to continue processing, FALSE to exit the application.

wxApp::OnQueryEndSession

void OnQueryEndSession(wxCloseEvent& event)

This is an event handler function called when the operating system or GUI session is about to close down. Typically, an application will try to save unsaved documents at this point.

If *wxCloseEvent::CanVeto* (p. 125) returns TRUE, the application is allowed to veto the shutdown by calling *wxCloseEvent::Veto* (p. 126). The application might veto the shutdown after prompting for documents to be saved, and the user has cancelled the save.

Use the `EVT_QUERY_END_SESSION` event table macro to handle query end session events.

You should check whether the application is forcing the deletion of the window using *wxCloseEvent::GetForce* (p. 126). If this is TRUE, destroy the window using *wxWindow::Destroy* (p. 1192). If not, it is up to you whether you respond by destroying the window.

The default handler calls *wxWindow::Close* (p. 1190) on the top-level window, and vetoes the shutdown if `Close` returns FALSE. This will be sufficient for many applications.

Remarks

Under X, `OnQueryEndSession` is called in response to the 'save session' event.

Under Windows, `OnQueryEndSession` is called in response to the `WM_QUERYENDSESSION` message.

See also

`wxWindow::Close` (p. 1190), `wxWindow::OnCloseWindow` (p. 1208), `wxCloseEvent` (p. 124), `wxApp::OnEndSession` (p. 27)

wxApp::ProcessMessage

bool ProcessMessage(MSG *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 `wxWindows` with another class library with its own message loop, you should make sure that this function is called to allow `wxWindows` to receive messages. For example, to allow co-existence with the Microsoft Foundation Classes, override the `PreTranslateMessage` function:

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

wxApp::Pending

bool Pending()

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

See also

`wxApp::Dispatch` (p. 23)

wxApp::SendIdleEvents

bool SendIdleEvents()

Sends idle events to all top-level windows.

bool SendIdleEvents(wxWindow* win)

Sends idle events to a window and its children.

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

wxApp::OnIdle (p. 27), *wxWindow::OnIdle* (p. 1211), *wxIdleEvent* (p. 557)

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 wxWindows.

See also

wxApp::GetAppName (p. 23)

wxApp::SetAuto3D

void SetAuto3D(const bool auto3D)

Switches automatic 3D controls on or off.

Parameters

auto3D

If TRUE, all controls will be created with 3D appearances unless overridden for a control or dialog. The default is TRUE

Remarks

This has an effect on Windows only.

See also

wxApp::GetAuto3D (p. 23)

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. 24)

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.

wxApp::SetTopWindow

void SetTopWindow(wxWindow* *window*)

Sets the 'top' window. You can call this from within *wxApp::OnInit* (p. 28) to let wxWindows 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, wxWindows 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. 24), *wxApp::OnInit* (p. 28)

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 wxWindows.

See also

wxApp::GetVendorName (p. 25)

wxApp::GetStdIcon

virtual wxIcon GetStdIcon(int *which*) const

Returns the icons used by wxWindows internally, e.g. the ones used for message boxes. This function is used internally and can be overridden by the user to change the default icons.

Parameters

which

One of the wxICON_XXX specifies which icon to return.

See *wxMessageBox* (p. 1261) for a list of icon identifiers.

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.

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. 1356) 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. 615)). 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. 35) 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*.

wxWindows 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 *wxWindows* 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 *wxWindows*).

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, its `Index()` (p. 41) function execution time is $O(\log(N))$ instead of $O(N)$ for the usual arrays but the `Add()` (p. 40) 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 linkin 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(MyDirectory *, ArrayOfDirectories);
WX_DEFINE_SORTED_ARRAY(MyFile *, ArrayOfFiles);
```

See also:

Container classes overview (p. 1349), *wxList* (p. 615)

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. 36)

WX_DEFINE_EXPORTED_ARRAY (p. 36)

WX_DEFINE_SORTED_ARRAY (p. 37)

WX_DEFINE_SORTED_EXPORTED_ARRAY (p. 37)

WX_DECLARE_EXPORTED_OBJARRAY (p. 37)

WX_DEFINE_OBJARRAY (p. 38)

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. 39)

wxArray copy constructors and assignment operators (p. 39)

~wxArray (p. 39)

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. 43) function is provided to unallocate the extra memory. The *Alloc()* (p. 40) 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. 40)
Shrink (p. 43)

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. 42) method.

Count (p. 40)
GetCount (p. 41)
IsEmpty (p. 42)
Item (p. 42)
Last (p. 42)

Adding items

Add (p. 40)
Insert (p. 41)
WX_APPEND_ARRAY (p. 38)

Removing items

WX_CLEAR_ARRAY (p. 38)
Empty (p. 41)
Clear (p. 40)
RemoveAt (p. 43)
Remove (p. 42)

Searching and sorting

Index (p. 41)
Sort (p. 43)

WX_DEFINE_ARRAY

WX_DEFINE_ARRAY(*T*, *name*)

WX_DEFINE_EXPORTED_ARRAY(*T*, *name*)

This macro defines a new array class named *name* and containing the elements of type *T*. The second form is used when compiling DLL under Windows and array needs to be visible outside the DLL. Example:

```
WX_DEFINE_ARRAY(int, wxArrayInt);

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

Note that wxWindows 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*)

This macro defines a new sorted array class named *name* and containing the elements of type *T*. The second form is used when compiling DLL under Windows and array needs to be visible outside the 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*)

This macro declares a new object array class named *name* and containing the elements of type *T*. The second form is used when compiling DLL under Windows and array needs to be visible outside the DLL.

Example:

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

You must use `WX_DEFINE_OBJARRAY()` (p. 38) macro to define the array class - otherwise you would get link errors.

WX_DEFINE_OBJARRAY

WX_DEFINE_OBJARRAY(*name*)

This macro defines the methods of the array class *name* not defined by the `WX_DECLARE_OBJARRAY()` (p. 37) 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.

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. 38) macro

for this.

wxArray::Add

void Add(T *item*)

void Add(T **item*)

void Add(T &*item*)

Appends a new element to the array (where *T* is the type of the array elements.)

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.

You may also use `WX_APPEND_ARRAY` (p. 38) macro to append all elements of one array to another one.

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. 41) and additionally frees the memory allocated to the array.

wxArray::Count

size_t Count() const

Same as `GetCount()` (p. 41). 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. 42) 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. 40) 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)

void Insert(T *item, size_t n)

void Insert(T &item, size_t n)

Insert a new item into the array before the 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. 40) 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. 41) 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. 38) macro which deletes all elements of a `wxArray` (supposed to contain pointers).

wxArray::RemoveAt

RemoveAt(size_t index)

Removes an element from the array by index. When an element is removed from `wxObjArray` it is deleted by the array - use `Detach()` (p. 41) 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. 38) macro which deletes all elements of a `wxArray` (supposed to contain pointers).

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. 1007) objects. It has the same features as all `wxArray` (p. 32) 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. 32), 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. 48), *Last* (p. 48) or *operator[]* (p. 45) 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. 47) 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. 32) class for the `wxString` member data: it is not implemented like this, but it does have all of the `wxArray` functions.

Include files

```
<wx/string.h>
```

See also

wxArray (p. 32), *wxString* (p. 1007), *wxString overview* (p. 1331)

wxArrayString::wxArrayString

wxArrayString()

wxArrayString(const wxArrayString& array)

Default and copy constructors.

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

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. 48) method.

wxArrayString::Add

size_t Add(const wxString& str)

Appends a new item to the array and return the index of th new item in the array.

Warning: For sorted arrays, the index of the inserted item will not be, in general, equal to *GetCount()* (p. 47) - 1 because the item is inserted at the correct position to keep the array sorted and not appended.

See also: *Insert* (p. 47)

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. 35)

wxArrayString::Clear

void Clear()

Clears the array contents and frees memory.

See also: *Empty* (p. 47)

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. 47) instead.

wxArrayString::Empty

void Empty()

Empties the array: after a call to this function *GetCount* (p. 47) 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. 46) 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)

Insert a 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()* + 1 this function behaves as *Add* (p. 46).

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. 47) would then not work!

wxArrayString::IsEmpty

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. 45) for the operator version.

wxArrayString::Last

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. 47)

void Remove(size_t nIndex)

Removes the item at given position.

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. 46), *Dynamic array memory management* (p. 35)

wxArrayString::Sort

void Sort(bool *reverseOrder* = FALSE)

Sorts the array in alphabetical order or in reverse alphabetical order if *reverseOrder* is TRUE.

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. 47) 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. 47) would then not work!

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. 1169) 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 works under Windows only, and currently only for Visual C++.

Derived from

wxObject (p. 746)

Include files

<wx/msw/ole/automtn.h>

See also

wxVariant (p. 1169)

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. 52) 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. 52)

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.

Derived from

wxGDIObject (p. 474)

wxObject (p. 746)

Include files

<wx/bitmap.h>

Predefined objects

Objects:

wxNullBitmap

See also

wxBitmap overview (p. 1388), *supported bitmap file formats* (p. 1389), *wxDC::Blit* (p. 281), *wxIcon* (p. 558), *wxCursor* (p. 184), *wxBitmap* (p. 54), *wxMemoryDC* (p. 678)

wxBitmap::wxBitmap

wxBitmap()

Default constructor.

wxBitmap(const wxBitmap& *bitmap*)

Copy constructor.

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. 565) 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.

wxBitmap(const char bits)**

Creates a bitmap from XPM data.

wxBitmap(const wxString& name, long type)

Loads a bitmap from a file or resource.

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_RESOURCE Load a Windows bitmap 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.
wxBITMAP_TYPE_RESOURCE	Load a Windows resource name.

The validity of these flags depends on the platform and wxWindows configuration. If all possible wxWindows 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. 565) 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.

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 wxWindows 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. 61)

wxPython note: Constructors supported by wxPython are:

- wxBitmap(name, flag)** Loads a bitmap from a file
- wxBitmapFromData(data, type, width, height, depth=1)** Creates a bitmap from the given data, which can be of arbitrary type.
- wxNoRefBitmap(name, flag)** This one won't own the reference, so Python won't call the destructor, this is good for toolbars and such where the parent will manage the bitmap.
- wxEmptyBitmap(width, height, depth = -1)** Creates an empty bitmap with the given specifications

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 `wxWindows` 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. 66)

wxBitmap::CleanUpHandlers

static void CleanUpHandlers()

Deletes all bitmap handlers.

This function is called by wxWindows on exit.

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. 54) 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. 54)

wxBitmap::FindHandler

static wxBitmapHandler* FindHandler(const wxString& name)

Finds the handler with the given name.

static wxBitmapHandler* FindHandler(const wxString& extension, long bitmapType)

Finds the handler associated with the given extension and type.

static wxBitmapHandler* FindHandler(long 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. 66)

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. 66)

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. 761)

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. 64), *wxMask* (p. 659)

wxBitmap::GetWidth

int GetWidth() const

Gets the width of the bitmap in pixels.

See also

wxBitmap::GetHeight (p. 60)

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 wxWindows configuration, can be handlers for Windows bitmap, Windows bitmap resource, and XPM.

This function is called by wxWindows on startup.

See also

wxBitmapHandler (p. 66)

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. 66)

wxBitmap::LoadFile

bool LoadFile(const wxString& name, long 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 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 wxWindows configuration.

In addition, wxBitmap can read all formats that *wxImage* (p. 565) 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. 60) member.

See also

wxBitmap::SaveFile (p. 63)

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. 66)

wxBitmap::SaveFile

bool SaveFile(const wxString& name, int 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 wxWindows configuration.

In addition, wxBitmap can save all formats that *wxImage* (p. 565) 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 wxWindows has been configured, not all formats may be available.

See also

wxBitmap::LoadFile (p. 61)

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. 60), *wxMask* (p. 659)

wxBitmap::SetOk

void SetOk(int *isOk*)

Sets the validity member (does not affect the bitmap data).

Parameters

isOk
Validity flag.

wxBitmap::SetPalette

void SetPalette(const wxPalette& *palette*)

Sets the associated palette.

Parameters

palette

The palette to set.

See also

wxPalette (p. 761)

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.

wxBitmapHandler

Overview (p. 1388)

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. 57) in your application initialisation.

Derived from

wxObject (p. 746)

Include files

<wx/bitmap.h>

See also

wxBitmap (p. 54), *wxIcon* (p. 558), *wxCursor* (p. 184)

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. 54) 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. 54) for values this can take.

Return value

TRUE if the operation succeeded, FALSE otherwise.

See also

wxBitmap::LoadFile (p. 61)
wxBitmap::SaveFile (p. 63)
wxBitmapHandler::SaveFile (p. 69)

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. 54) 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. 61)
wxBitmap::SaveFile (p. 63)
wxBitmapHandler::LoadFile (p. 68)

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.

wxBitmapButton

A bitmap button is a control that contains a bitmap. It may be placed on a *dialog box* (p. 310) or *panel* (p. 764), or indeed almost any other window.

Derived from

wxButton (p. 89)

wxControl (p. 176)

wxWindow (p. 1184)

wxEvtHandler (p. 378)

wxObject (p. 746)

Include files

<wx/bmpbuttn.h>

Remarks

A bitmap button can be supplied with a single bitmap, and wxWindows 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.

See also *window styles overview* (p. 1371).

Event handling

EVT_BUTTON(id, func)	Process a <code>wxEVT_COMMAND_BUTTON_CLICKED</code> event, when the button is clicked.
-----------------------------	--

See also

wxButton (p. 89)

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. 70).

validator

Window validator.

name

Window name.

Remarks

The *bitmap* parameter is normally the only bitmap you need to provide, and *wxWindows* will draw the button correctly in its different states. If you want more control, call any of the functions *wxBitmapButton::SetBitmapSelected* (p. 75), *wxBitmapButton::SetBitmapFocus* (p. 74), *wxBitmapButton::SetBitmapDisabled* (p. 74).

Note that the bitmap passed is smaller than the actual button created.

See also

wxBitmapButton::Create (p. 72), *wxValidator* (p. 1166)

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. 71).

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. 74)

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. 74)

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. 74)

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. 75)

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. 73), *wxBitmapButton::SetBitmapLabel* (p. 74), *wxBitmapButton::SetBitmapSelected* (p. 75), *wxBitmapButton::SetBitmapFocus* (p. 74)

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. 73), *wxBitmapButton::SetBitmapLabel* (p. 74), *wxBitmapButton::SetBitmapSelected* (p. 75), *wxBitmapButton::SetBitmapDisabled* (p. 74)

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. 73)

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. 73), *wxBitmapButton::SetBitmapLabel* (p. 74), *wxBitmapButton::SetBitmapFocus* (p. 74), *wxBitmapButton::SetBitmapDisabled* (p. 74)

wxBitmapDataObject

`wxBitmapDataObject` is a specialization of `wxDataObject` for bitmap data. It can be used without change to paste data into the *wxClipboard* (p. 121) or a *wxDropSource* (p. 365). 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. 76) may be overridden to increase

efficiency.

Derived from

wxDataObjectSimple (p. 201)
wxDataObject (p. 196)

Include files

<wx/dataobj.h>

See also

Clipboard and drag and drop overview (p. 1420), *wxDataObject* (p. 196),
wxDataObjectSimple (p. 201), *wxFileDataObject* (p. 406), *wxTextDataObject* (p. 1083),
wxDataObject (p. 196)

wxBitmapDataObject(const wxBitmap& bitmap = wxNullBitmap)

Constructor, optionally passing a bitmap (otherwise use *SetBitmap* (p. 76) 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 wxWindows' internals. Use this method to get data in bitmap form from the *wxClipboard* (p. 121).

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.

wxBoolFormValidator

This class validates a boolean value for a *form view* (p. 827). The associated control must be a *wxCheckBox*.

See also

Property validator classes (p. 1453)

wxBoolFormValidator::wxBoolFormValidator

```
void wxBoolFormValidator(long flags=0)
```

Constructor.

wxBoolListValidator

This class validates a boolean value for a *property list view* (p. 834).

See also

Validator classes (p. 1453)

wxBoolListValidator::wxBoolListValidator

```
void wxBoolListValidator(long flags=0)
```

Constructor.

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.

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 *option* 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 its original size, `wxGROW` flag (same as `wxEXPAND`) forces the window to grow with the sizer, and `wxSHAPED` flag tells the window to change its 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,
wxDIALOG_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

    SetAutoLayout( TRUE ); // tell dialog to use sizer
    SetSizer( topsizer ); // actually set the sizer

    topsizer->Fit( this ); // set size to minimum size as
    calculated by the sizer
    topsizer->SetSizeHints( this ); // set size hints to honour minimum
    size
}

```

Derived from

[wxSizer](#) (p. 924)
[wxObject](#) (p. 746)

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. 1228) 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. 474)

wxObject (p. 746)

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, wxWindows 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. 28) 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. 85), `wxDC` (p. 280), `wxDC::SetBrush` (p. 295)

wxBrush::wxBrush

wxBrush()

Default constructor. The brush will be uninitialised, and `wxBrush::Ok` (p. 83) will return `FALSE`.

wxBrush(const wxColour& colour, int style)

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.
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. 85), *wxColour* (p. 135), *wxColourDatabase* (p. 140)

wxBrush::~~wxBrush

void ~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 wxWindows 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. 83)

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. 62) returns FALSE).

See also

wxBrush::SetStipple (p. 84)

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. 84), *wxBrush::SetColour* (p. 83), *wxBrush::SetStipple* (p. 84)

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. 82)

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. 54)

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. 83)

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. 615)
wxObject (p. 746)

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 wxWindows 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 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 brush, because the referencing 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 wxWindows to keep track of brushes in order to clean them up on exit. It is also kept for backward compatibility with earlier versions of wxWindows.

See also

wxBrush (p. 80)

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 wxWindows to add a brush to the list.

wxBrushList::FindOrCreateBrush

wxBrush * FindOrCreateBrush(const wxColour& colour, int style)

Finds a brush with the specified attributes and returns it, else creates a new brush, adds it to the brush list, and returns it.

wxBrush * FindOrCreateBrush(const wxString& colourName, int style)

Finds a brush with the specified attributes and returns it, else creates a new brush, adds it to the brush list, and returns it.

Finds a brush of the given specification, or creates one and adds it to the list.

Parameters

colour

Colour object.

colourName

Colour name, which should be in the colour database.

style

Brush style. See *wxBrush::SetStyle* (p. 84) for a list of styles.

wxBrushList::RemoveBrush

void RemoveBrush(wxBrush *brush)

Used by wxWindows to remove a brush from the list.

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. 1269) in the constructor, and *wxEndBusyCursor* (p. 1272) in the destructor.

Derived from

None

Include files

<wx/utils.h>

See also

wxBeginBusyCursor (p. 1269), *wxEndBusyCursor* (p. 1272), *wxWindowDisabler* (p. 1235)

wxBusyCursor::wxBusyCursor

```
wxBusyCursor(wxCursor* cursor = wxHOURLASS_CURSOR)
```

Constructs a busy cursor object, calling *wxBeginBusyCursor* (p. 1269).

wxBusyCursor::~~wxBusyCursor

```
~wxBusyCursor()
```

Destroys the busy cursor object, calling *wxEndBusyCursor* (p. 1272).

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.

Derived from

None

Include files

<wx/busyinfo.h>

wxBusyInfo::wxBusyInfo

wxBusyInfo(const wxString& msg)

Constructs a busy info object, displays *msg*.

wxButton

A button is a control that contains a text string, and is one of the commonest elements of a GUI. It may be placed on a *dialog box* (p. 310) or *panel* (p. 764), or indeed almost any other window.

Derived from

wxControl (p. 176)

wxWindow (p. 1184)

wxEvtHandler (p. 378)

wxObject (p. 746)

Include files

<wx/button.h>

Window styles

wxBU_LEFT

Left-justifies the label. WIN32 only.

wxBU_TOP

Aligns the label to the top of the button. WIN32 only.

wxBU_RIGHT

Right-justifies the bitmap label. WIN32 only.

wxBU_BOTTOM

Aligns the label to the bottom of the button. WIN32 only.

See also *window styles overview* (p. 1371).

Event handling

EVT_BUTTON(id, func)

Process a `wxEVT_COMMAND_BUTTON_CLICKED` event, when the button is clicked.

See also

`wxBitmapButton` (p. 70)

wxButton::wxButton

wxButton()

Default constructor.

wxButton(wxWindow* parent, wxWindowID id, const wxString& label, const wxPoint& pos, const wxSize& size = wxDefaultSize, long style = 0, const wxValidator& validator, 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.

label

Text to be displayed on the button.

pos

Button position.

size

Button size. If the default size (-1, -1) is specified then the button is sized appropriately for the text.

style

Window style. See `wxButton` (p. 89).

validator

Window validator.

name

Window name.

See also

wxButton::Create (p. 91), *wxValidator* (p. 1166)

wxButton::~~wxButton

~wxButton()

Destructor, destroying the button.

wxButton::Create

bool Create(wxWindow* parent, wxWindowID id, const wxString& label, 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 *wxButton::wxButton* (p. 90).

wxButton::GetLabel

wxString GetLabel() const

Returns the string label for the button.

Return value

The button's label.

See also

wxButton::SetLabel (p. 92)

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. 1225) which sets the keyboard focus for windows and text panel items, and *wxPanel::SetDefaultItem* (p. 767).

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: wxWindows 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. 91)

wxBufferedInputStream

This stream acts as a cache. It caches the bytes read from the specified input stream (See *wxFilterInputStream* (p. 432)). 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. 432)

Include files

<wx/stream.h>

See also

wxStreamBuffer (p. 1000), *wxInputStream* (p. 592), *wxBufferedOutputStream* (p. 93)

wxBufferedOutputStream

This stream acts as a cache. It caches the bytes to be written to the specified output stream (See *wxFilterOutputStream* (p. 432)). 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. 432)

Include files

<wx/stream.h>

See also

wxStreamBuffer (p. 1000), *wxOutputStream* (p. 751)

wxBufferedOutputStream::wxBufferedOutputStream

wxBufferedOutputStream(const **wxOutputStream**& *parent*)

Creates a buffered stream using a buffer of a default size of 1024 bytes for caching 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.

wxCalculateLayoutEvent

This event is sent by *wxLayoutAlgorithm* (p. 610) to calculate the amount of the remaining client area that the window should occupy.

Derived from

wxEvent (p. 375)

wxObject (p. 746)

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. 856), *wxSashLayoutWindow* (p. 894), *wxLayoutAlgorithm* (p. 610).

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 interactively. For this, it displays a window containing several parts: the control to pick the month and the year at the top (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. 101) class.

An item without custom attributes is drawn with the default colours and font and without border, but setting custom attributes with `SetAttr` (p. 100) 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. 1340) using *SetHoliday* (p. 103) 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. 176)
wxWindow (p. 1184)
wxEvtHandler (p. 378)
wxObject (p. 746)

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

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. 104) 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. 101) 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. 1321)
wxCalendarDateAttr (p. 101)
wxCalendarEvent (p. 104)

wxCalendarCtrl::wxCalendarCtrl

wxCalendarCtrl()

Default constructor, use *Create* (p. 97) after it.

wxCalendarCtrl::wxCalendarCtrl

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. 97) 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. 1186) 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. 99)

wxCalendarCtrl::GetHeaderColourBg

const wxColour& GetHeaderColourBg() const

Gets the background colour of the header part of the calendar window.

See also

SetHeaderColours (p. 99)

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. 99)

wxCalendarCtrl::GetHighlightColourBg

const wxColour& GetHighlightColourBg() const

Gets the background highlight colour.

See also

SetHighlightColours (p. 99)

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. 100)

wxCalendarCtrl::GetHolidayColourBg

const wxColour& GetHolidayColourBg() const

Return the background colour currently used for holiday highlighting.

See also

SetHolidayColours (p. 100)

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 range 1...31).

wxCalendarCtrl::HitTest

```
wxCalendarHitTestResult HitTest(const wxPoint& pos, wxDateTime* date = NULL,  
wxDateTime::WeekDay* wd = NULL)
```

Returns one of `wxCAL_HITTEST_XXX constants` (p. 95) 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. 95).

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. 95)

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. 101)

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. 101) to use for the item with this attribute.

wxCalendarEvent

The `wxCalendarEvent` class is used together with `wxCalendarCtrl` (p. 95).

See also

`wxCalendarCtrl` (p. 95)

wxCalendarEvent::GetDate

wxcalendareventgetdate

const wxDateTime& GetDate() const

Returns the date. This function may be called for all event types except `EVT_CALEDAR_WEEKDAY_CLICKED` one for which it doesn't make sense.

wxCalendarEvent::GetWeekDay

wxcalendareventgetweekday

wxDateTime::WeekDay GetWeekDay() const

Returns the week day on which the user clicked in `EVT_CALEDAR_WEEKDAY_CLICKED` handler. It doesn't make sense to call this function in other handlers.

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. 1194). 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).

wxCaret::GetSize

void GetSize(int* width, int* height) const

wxSize GetSize() const

Get the caret size.

wxCaret::GetWindow

wxWindow* GetWindow() const

Get the window the caret is associated with.

wxCaret::Hide

void Hide()

Same as *wxCaret::Show(FALSE)* (p. 108).

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 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. 106)

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 is either on (checkmark is visible) or off (no checkmark).

Derived from

wxControl (p. 176)

wxWindow (p. 1184)

wxEvtHandler (p. 378)

wxObject (p. 746)

Include files

<wx/checkbox.h>

Window styles

There are no special styles for `wxCheckBox`.

See also *window styles overview* (p. 1371).

Event handling

EVT_CHECKBOX(id, func) Process a `wxEVT_COMMAND_CHECKBOX_CLICKED` event, when the checkbox is clicked.

See also

wxRadioButton (p. 864), *wxCommandEvent* (p. 152)

`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. 108).

validator

Window validator.

name

Window name.

See also

wxCheckBox::Create (p. 110), *wxValidator* (p. 1166)

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. 109) for details.

wxCheckBox::GetValue

bool GetValue() const

Gets the state of the checkbox.

Return value

Returns TRUE if it is checked, FALSE otherwise.

wxCheckBox::SetValue

void SetValue(const 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.

wxCheckListBox

A checkListBox is like a listBox, but allows items to be checked or unchecked.

This class is currently implemented under Windows and GTK. When using this class under Windows wxWindows must be compiled with USE_OWNER_DRAWN set to 1.

Only the new functions for this class are documented; see also *wxListBox* (p. 621).

Derived from

wxListBox (p. 621)
wxControl (p. 176)
wxWindow (p. 1184)
wxEvtHandler (p. 378)
wxObject (p. 746)

Include files

<wx/checklst.h>

Window styles

See *wxListBox* (p. 621).

Event handling

EVT_CHECKLISTBOX(id, func)	Process a wxEVT_COMMAND_CHECKLISTBOX_TOGGLE event, when an item in the check list box is checked or unchecked.
-----------------------------------	--

See also

wxListBox (p. 621), *wxChoice* (p. 113), *wxComboBox* (p. 143), *wxListCtrl* (p. 630), *wxCommandEvent* (p. 152)

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")
```

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. 111).

validator

Window validator.

name

Window name.

wxPython note: The *wxCheckListBox* constructor in wxPython reduces the *n* and *choices* arguments to a single argument, which is a list of strings.

wxCheckListBox::~wxCheckListBox

```
void ~wxCheckListBox()
```

Destructor, destroying the list box.

wxCheckListBox::Check

void Check(int *item*, bool *check* = *TRUE*)

Checks the given item.

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

wxControl (p. 176)

wxWindow (p. 1184)

wxEvtHandler (p. 378)

wxObject (p. 746)

Include files

<wx/choice.h>

Window styles

There are no special styles for wxChoice.

See also *window styles overview* (p. 1371).

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. 621), `wxComboBox` (p. 143), `wxCommandEvent` (p. 152)

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")

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. 113).

validator

Window validator.

name

Window name.

See also

wxChoice::Create (p. 115), *wxValidator* (p. 1166)

wxPython note: The *wxChoice* constructor in *wxPython* reduces the `nand choices` arguments to a single argument, which is a list of strings.

wxChoice::~~wxChoice

~wxChoice()

Destructor, destroying the choice item.

wxChoice::Append

void Append(const wxString& item)

Adds the item to the end of the choice control.

void Append(const wxString& item, void* clientData)

Adds the item to the end of the combobox, associating the given data with the item.

Parameters

item

String to add.

clientData

Client data to associate with the item.

wxChoice::Clear

void Clear()

Clears the strings from 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 wxString& name = "choice")

Creates the choice for two-step construction. See *wxChoice::wxChoice* (p. 114).

wxChoice::FindString

int FindString(const wxString& string) const

Finds a choice matching the given string.

Parameters

string
String to find.

Return value

Returns the position if found, or -1 if not found.

wxChoice::GetColumns

int GetColumns() const

Gets the number of columns in this choice item.

Remarks

This is implemented for Motif only.

wxChoice::GetClientData

void* GetClientData(int n) const

Returns a pointer to the client data associated with the given item (if any).

Parameters

n
An item, starting from zero.

Return value

A pointer to the client data, or NULL if the item was not found.

wxChoice::GetSelection

int GetSelection() const

Gets the id (position) of the selected string, or -1 if there is no selection.

wxChoice::GetString

wxString GetString(int *n*) const

Returns the string at the given position.

Parameters

n
The zero-based position.

Return value

The string at the given position, or the empty string if *n* is invalid.

wxChoice::GetStringSelection

wxString GetStringSelection() const

Gets the selected string, or the empty string if no string is selected.

wxChoice::Number

int Number() const

Returns the number of strings in the choice control.

wxChoice::SetClientData

void SetClientData(int *n*, void* *data*)

Associates the given client data pointer with the given item.

Parameters

n
The zero-based item.

data
The client data.

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.

wxChoice::SetSelection

void SetSelection(int *n*)

Sets the choice by passing the desired string position. This does not cause a `wxEVT_COMMAND_CHOICE_SELECTED` event to get emitted.

Parameters

n
The string position to select, starting from zero.

See also

wxChoice::SetStringSelection (p. 118)

wxChoice::SetStringSelection

void SetStringSelection(const wxString& *string*)

Sets the choice by passing the desired string. This does not cause a `wxEVT_COMMAND_CHOICE_SELECTED` event to get emitted.

Parameters

string
The string to select.

See also

wxChoice::SetSelection (p. 118)

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. 1330), *wxObject* (p. 746)

wxClassInfo::wxClassInfo

wxClassInfo(char* *className*, char* *baseClass1*, char* *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(char* *name*)

Finds the wxClassInfo object for a class of the given string name.

wxClassInfo::GetBaseClassName1

char* GetBaseClassName1() const

Returns the name of the first base class (NULL if none).

wxClassInfo::GetBaseClassName2

char* GetBaseClassName2() const

Returns the name of the second base class (NULL if none).

wxClassInfo::GetClassName

char * 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 wxWindows library initialization.

wxClassInfo::IsKindOf

bool IsKindOf(wxClassInfo* info)

Returns TRUE if this class is a kind of (inherits from) the given class.

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. 759) object.

To draw on the whole window including decorations, construct a *wxWindowDC* (p. 1234) object (Windows only).

Derived from

wxWindowDC (p. 1234)
wxDC (p. 280)

Include files

<wx/dcclient.h>

See also

wxDC (p. 280), *wxMemoryDC* (p. 678), *wxPaintDC* (p. 759), *wxWindowDC* (p. 1234), *wxScreenDC* (p. 901)

wxClientDC::wxClientDC

wxClientDC(*wxWindow** window)

Constructor. Pass a pointer to the window on which you wish to paint.

wxClipboard

A class for manipulating the clipboard. Note that this is not compatible with the clipboard class from wxWindows 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. 1422) for further information.

Call *wxClipboard::Open* (p. 123) to get ownership of the clipboard. If this operation returns TRUE, you now own the clipboard. Call *wxClipboard::SetData* (p. 124) to put data on the clipboard, or *wxClipboard::GetData* (p. 123) to retrieve data from the clipboard. Call *wxClipboard::Close* (p. 123) 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. 746)**Include files**

<wx/clipbrd.h>

See also*Drag and drop overview* (p. 1420), *wxDataObject* (p. 196)**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. 123).

After this function has been called, the clipboard owns the data, so do not delete the data explicitly.

See also

wxClipboard::SetData (p. 124)

wxClipboard::Clear

void Clear()

Clears the global clipboard object and the system's clipboard if possible.

wxClipboard::Close

bool Close()

Call this function to close the clipboard, having opened it with *wxClipboard::Open* (p. 123).

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 the format of the given data object is available on the clipboard.

wxClipboard::Open

bool Open()

Call this function to open the clipboard before calling `wxClipboard::SetData` (p. 124) and `wxClipboard::GetData` (p. 123).

Call `wxClipboard::Close` (p. 123) 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. 122)

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.

Derived from

`wxEvent` (p. 375)

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 <code>wxFrame</code> and <code>wxDialog</code> classes.
EVT_QUERY_END_SESSION(func)	Process a query end session event, supplying the member function. This event applies to <code>wxApp</code> only.
EVT_END_SESSION(func)	Process an end session event, supplying the member function. This event applies to <code>wxApp</code> only.

See also

`wxWindow::OnCloseWindow` (p. 1208), `wxWindow::Close` (p. 1190), `wxApp::OnQueryEndSession` (p. 28), `wxApp::OnEndSession` (p. 27), *Window deletion overview* (p. 1371)

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::GetSessionEnding

bool GetSessionEnding() const

Returns TRUE if the session is ending.

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. 125) returns TRUE.

wxCmdLineParser

wxCmdLineParser is a class for parsing 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. 129) 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, <code>-v</code> 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, <code>-o:filename</code> 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. 132). 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. 134) 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. 133) 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 program 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. 135) to retrieve the number of parameters effectively specified after calling *Parse* (p. 134).

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. 22) and `wxApp::argv` (p. 23)
console sample

Construction

Before `Parse` (p. 134) 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. 134).

To specify the command line to parse you may use either one of constructors accepting it (`wxCmdLineParser(argc, argv)` (p. 130) or `wxCmdLineParser` (p. 131) usually) or, if you use *the default constructor* (p. 130), you can do it later by calling `SetCmdLine` (p. 131).

The same holds for command line description: it can be specified either in the constructor (*without command line* (p. 131) or *together with it* (p. 131)) or constructed later using either `SetDesc` (p. 132) or combination of `AddSwitch` (p. 133), `AddOption` (p. 133) and `AddParam` (p. 133) methods.

Using constructors or `SetDesc` (p. 132) 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. 134).

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. 132).

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. 132) method.

Finally, *SetLogo* (p. 132) can be used to show some application-specific text before the explanation given by *Usage* (p. 134) function.

Parsing command line

After the command line description was constructed and the desired options were set, you can finally call *Parse* (p. 134) 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 *wxWindows* logging functions.

Getting results

After calling *Parse* (p. 134) (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. 134) 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. 131) or *SetCmdLineLater* (p. 131) later.

wxCmdLineParser::wxCmdLineParser

wxCmdLineParser(int argc, char 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.

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. 130), but also specifies the *command line description* (p. 132).

wxCmdLineParser::wxCmdLineParser

wxCmdLineParser(const wxCmdLineEntryDesc* desc, int argc, char argv)**

Same as *wxCmdLineParser* (p. 130), but also specifies the *command line description* (p. 132).

wxCmdLineParser::wxCmdLineParser

wxCmdLineParser(const wxCmdLineEntryDesc* desc, const wxString& cmdline)

Same as *wxCmdLineParser* (p. 130), but also specifies the *command line description* (p. 132).

wxCmdLineParser::SetCmdLine

void SetCmdLine(int argc, char argv)**

Set command line to parse after using one of the constructors which don't do it.

See also

wxCmdLineParser (p. 130)

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. 130)

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. 129)

wxCmdLineParser::DisableLongOptions

void DisableLongOptions()

Identical to *EnableLongOptions(FALSE)* (p. 132).

wxCmdLineParser::SetLogo

void SetLogo(const wxString& *logo*)

logo is some extra text which will be shown by *Usage* (p. 134) 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()

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.

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. 132)

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. 135)

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. 140) 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.

Derived from

wxObject (p. 746)

Include files

<wx/colour.h>

Predefined objects

Objects:

wxNullColour

Pointers:

wxBLACK
wxWHITE
wxRED
wxBLUE
wxGREEN
wxCYAN
wxLIGHT_GREY

See also

wxColourDatabase (p. 140), *wxPen* (p. 771), *wxBrush* (p. 80), *wxColourDialog* (p. 142)

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& *colourName*)

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. 140)

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. 140)

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. 746)

Include files

<wx/cmndata.h>

See also

wxColour (p. 135), *wxColourDialog* (p. 142), *wxColourDialog overview* (p. 1398)

wxColourData::wxColourData

wxColourData()

Constructor. Initializes the custom colours to white, 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. 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 all white.

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 all white.

wxColourData::operator =

void operator =(const wxColourData& *data*)

Assignment operator for the colour data.

wxColourDatabase

wxWindows 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 *Append*. There is only one instance of this class:

wxTheColourDatabase.

Derived from

wxList (p. 615)
wXObject (p. 746)

Include files

<wx/gdicmn.h>

Remarks

The colours in the standard database are as follows:

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. 135)

wxColourDatabase::wxColourDatabase

wxColourDatabase()

Constructs the colour database.

wxColourDatabase::FindColour

wxColour* FindColour(const wxString& colourName)

Finds a colour given the name. Returns NULL if not found.

wxColourDatabase::FindName

wxString FindName(const wxColour& colour) const

Finds a colour name given the colour. Returns NULL if not found.

wxColourDatabase::Initialize

void Initialize()

Initializes the database with a number of stock colours. Called by wxWindows on start-up.

wxColourDialog

This class represents the colour chooser dialog.

Derived from

wxDialog (p. 310)
wxWindow (p. 1184)
wxEvtHandler (p. 378)
wxObject (p. 746)

Include files

<wx/colordlg.h>

See also

wxColourDialog Overview (p. 1398), *wxColour* (p. 135), *wxColourData* (p. 138)

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.

See also

wxColourData (p. 138)

wxColourDialog::~~wxColourDialog

~wxColourDialog()

Destructor.

wxColourDialog::Create

bool Create(wxWindow* parent, wxColourData* data = NULL)

Same as *constructor* (p. 142).

wxColourDialog::GetColourData

wxColourData& GetColourData()

Returns the *colour data* (p. 138) associated with the colour dialog.

wxColourDialog::ShowModal

int ShowModal()

Shows the dialog, returning wxID_OK if the user pressed OK, and wxOK_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

wxChoice (p. 113)

wxControl (p. 176)

wxWindow (p. 1184)

wxEvtHandler (p. 378)

wxObject (p. 746)

Include files

<wx/combobox.h>

Window styles

wxCB_SIMPLE	Creates a combobox with a permanently displayed list. Windows only.
wxCB_DROPDOWN wxCB_READONLY	Creates a combobox with a drop-down list. 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. 1371).

Event handling

EVT_COMBOBOX(id, func)	Process a wxEVT_COMMAND_COMBOBOX_SELECTED event, when an item on the list is selected.
EVT_TEXT(id, func)	Process a wxEVT_COMMAND_TEXT_UPDATED event, when the combobox text changes.

See also

wxListBox (p. 621), *wxTextCtrl* (p. 1070), *wxChoice* (p. 113), *wxCommandEvent* (p. 152)

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")

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.

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. 143).

validator

Window validator.

name

Window name.

See also

wxComboBox::Create (p. 146), *wxValidator* (p. 1166)

wxPython note: The *wxComboBox* constructor in wxPython reduces the *n* and *choices* arguments to a single argument, which is a list of strings.

wxComboBox::~~wxComboBox

~wxComboBox()

Destructor, destroying the combobox.

wxComboBox::Append

void Append(const wxString& item)

Adds the item to the end of the combobox.

void Append(const wxString& item, void* clientData)

Adds the item to the end of the combobox, associating the given data with the item.

Parameters

item

The string to add.

clientData

Client data to associate with the item.

wxComboBox::Clear

void Clear()

Clears all strings from 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")

Creates the combobox for two-step construction. Derived classes should call or replace this function. See *wxComboBox::wxComboBox* (p. 144) for further details.

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::Delete

void Delete(int n)

Deletes an item from the combobox.

Parameters

n

The item to delete, starting from zero.

wxComboBox::FindString

int FindString(const wxString& *string*)

Finds a choice matching the given string.

Parameters*string*

The item to find.

Return value

The position if found, or -1 if not found.

wxComboBox::GetClientData

void* GetClientData(int *n*) const

Returns a pointer to the client data associated with the given item (if any).

Parameters*n*

An item, starting from zero.

Return value

A pointer to the client data, or NULL if the item was not found.

wxComboBox::GetInsertionPoint

long GetInsertionPoint() const

Returns the insertion point for the combobox's text field.

wxComboBox::GetLastPosition

long GetLastPosition() const

Returns the last position in the combobox text field.

wxComboBox::GetSelection

int GetSelection() const

Gets the position of the selected string, or -1 if there is no selection.

wxComboBox::GetString

wxString GetString(int *n*) const

Returns the string at position *n*.

Parameters

n
The item position, starting from zero.

Return value

The string if the item is found, otherwise the empty string.

wxComboBox::GetStringSelection

wxString GetStringSelection() const

Gets the selected string.

wxComboBox::GetValue

wxString GetValue() const

Returns the current value in the combobox text field.

wxComboBox::Number

int Number() const

Returns the number of items in the combobox list.

wxComboBox::Paste

void Paste()

Pastes text from the clipboard to the text field.

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::SetClientData

void SetClientData(int *n*, void* *data*)

Associates the given client data pointer with the given item.

Parameters

n
The zero-based item.

data
The client data.

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(int *n*)

Selects the given item in the combobox list. This does not cause a `wxEVT_COMMAND_COMBOBOX_SELECTED` event to get emitted.

void SetSelection(long *from*, long *to*)

Selects the text between the two positions, in the combobox text field.

Parameters

n
The zero-based item to select.

from
The first position.

to
The second position.

wxPython note: The second form of this method is called `SetMark` in wxPython.

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.

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. 746)

Include files

<wx/docview.h>

See also

Overview (p. 1406)

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-dependant).

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. 1134), have separate command event classes.

Derived from

wxEvent (p. 375)

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 <i>wxEVT_COMMAND_BUTTON_CLICKED</i> command, which is generated by a <i>wxButton</i> control.
EVT_CHECKBOX(id, func)	Process a <i>wxEVT_COMMAND_CHECKBOX_CLICKED</i> command, which is generated by a <i>wxCheckBox</i> control.
EVT_CHOICE(id, func)	Process a <i>wxEVT_COMMAND_CHOICE_SELECTED</i> command, which is generated by a <i>wxChoice</i> control.
EVT_LISTBOX(id, func)	Process a <i>wxEVT_COMMAND_LISTBOX_SELECTED</i> command, which is generated by a <i>wxListBox</i> control.
EVT_LISTBOX_DCLICK(id, func)	Process a <i>wxEVT_COMMAND_LISTBOX_DOUBLECLICKED</i> command, which is generated by a <i>wxListBox</i> control.
EVT_TEXT(id, func)	Process a <i>wxEVT_COMMAND_TEXT_UPDATED</i> command, which is generated by a <i>wxTextCtrl</i> control.
EVT_TEXT_ENTER(id, func)	Process a <i>wxEVT_COMMAND_TEXT_ENTER</i> command, which is generated by a <i>wxTextCtrl</i> control.

	control. Note that you must use <code>wxTE_PROCESS_ENTER</code> flag when creating the control if you want it to generate such events.
EVT_MENU(id, func)	Process a <code>wxEVT_COMMAND_MENU_SELECTED</code> command, which is generated by a menu item.
EVT_MENU_RANGE(id1, id2, func)	Process a <code>wxEVT_COMMAND_MENU_RANGE</code> command, which is generated by a range of menu items.
EVT_SLIDER(id, func)	Process a <code>wxEVT_COMMAND_SLIDER_UPDATED</code> command, which is generated by a <code>wxSlider</code> control.
EVT_RADIOBOX(id, func)	Process a <code>wxEVT_COMMAND_RADIOBOX_SELECTED</code> command, which is generated by a <code>wxRadioButton</code> control.
EVT_RADIOBUTTON(id, func)	Process a <code>wxEVT_COMMAND_RADIOBUTTON_SELECTED</code> command, which is generated by a <code>wxRadioButton</code> control.
EVT_SCROLLBAR(id, func)	Process a <code>wxEVT_COMMAND_SCROLLBAR_UPDATED</code> command, which is generated by a <code>wxScrollBar</code> control. This is provided for compatibility only; more specific scrollbar event macros should be used instead (see <i>wxCommandEvent</i> (p. 909)).
EVT_COMBOBOX(id, func)	Process a <code>wxEVT_COMMAND_COMBOBOX_SELECTED</code> command, which is generated by a <code>wxComboBox</code> control.
EVT_TOOL(id, func)	Process a <code>wxEVT_COMMAND_TOOL_CLICKED</code> event (a synonym for <code>wxEVT_COMMAND_MENU_SELECTED</code>). Pass the id of the tool.
EVT_TOOL_RANGE(id1, id2, func)	Process a <code>wxEVT_COMMAND_TOOL_CLICKED</code> event for a range id identifiers. Pass the ids of the tools.
EVT_TOOL_RCLICKED(id, func)	Process a <code>wxEVT_COMMAND_TOOL_RCLICKED</code> event. Pass the id of the tool.
EVT_TOOL_RCLICKED_RANGE(id1, id2, func)	Process a <code>wxEVT_COMMAND_TOOL_RCLICKED</code> event for a range of ids. Pass the ids of the tools.
EVT_TOOL_ENTER(id, func)	Process a <code>wxEVT_COMMAND_TOOL_ENTER</code> event. Pass the id of the toolbar itself. The value of <code>wxCommandEvent::GetSelection</code> 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::m_clientData

void* m_clientData

Contains a pointer to client data for listboxes and choices, if the event was a selection. Beware, this is not implemented anyway...

wxCommandEvent::m_commandInt

int m_commandInt

Contains an 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::m_commandString

char* m_commandString

Contains a string corresponding to a listbox or choice selection.

wxCommandEvent::m_extraLong

long m_extraLong

Extra information. 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::wxCommandEvent

wxCommandEvent(WXTYPE *commandEventType* = 0, int *id* = 0)

Constructor.

wxCommandEvent::Checked

bool Checked() const

Deprecated, use *IsChecked* (p. 157) instead.

wxCommandEvent::GetClientData

void* GetClientData()

Returns client data pointer for a listbox or choice selection event (not valid for a deselection).

wxCommandEvent::GetExtraLong

long GetExtraLong()

Returns the **m_extraLong** member.

wxCommandEvent::GetInt

int GetInt()

Returns the **m_commandInt** member.

wxCommandEvent::GetSelection

int GetSelection()

Returns item index for a listbox or choice selection event (not valid for a deselection).

wxCommandEvent::GetString

char* 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 choice 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::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(char* 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. 746)

Include files

<wx/docview.h>

See also

wxCommandProcessor overview (p. 1406), *wxCommand* (p. 151)

wxCommandProcessor::wxCommandProcessor

wxCommandProcessor(int maxCommands = 100)

Constructor.

`maxCommands` defaults to a rather arbitrary 100, but can be set from 1 to any integer. If your `wxCommand` classes store a lot of data, you may wish to limit the number of commands stored to a smaller number.

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::Do

virtual bool Do()

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::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::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::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. 1107) for the worker thread, but if there are several worker threads it already makes much more sense).

Once the thread(s) are signaled, the condition then resets to the not signaled state, ready to fire again.

Derived from

None.

Include files

<wx/thread.h>

See also

wxThread (p. 1101), *wxMutex* (p. 730)

wxCondition::wxCondition

wxCondition()

Default constructor.

wxCondition::~~wxCondition

~wxCondition()

Destroys the wxCondition object.

wxCondition::Broadcast

void Broadcast()

Broadcasts to all waiting objects.

wxCondition::Signal

void Signal()

Signals the object.

wxCondition::Wait

void Wait()

Waits indefinitely.

bool Wait(unsigned long sec, unsigned long nsec)

Waits until a signal is raised or the timeout has elapsed.

Parameters

sec

Timeout in seconds

nsec

Timeout nanoseconds component (added to *sec*).

Return value

The second form returns if the signal was raised, or FALSE if there was a timeout.

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: wxIniConfig, wxFileConfig, 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, wxWindows provides a typedef wxConfig which is mapped onto the native wxConfigBase implementation on the given platform: i.e. wxRegConfig under Win32, wxIniConfig under Win16 and wxFileConfig otherwise.

See *config overview* (p. 1358) 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. 163)

Derived from

No base class

Include files

<wx/config.h> (to let wxWindows choose a wxConfig class for your platform)
 <wx/confbase.h> (base config class)
 <wx/fileconf.h> (wxFileconfig class)
 <wx/msw/regconf.h> (wxRegConfig class)
 <wx/msw/iniconf.h> (wxIniConfig 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", -1);
...
...
...
// 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. Of course, you should delete it on the program termination (otherwise, not only a memory leak will

result, but even more importantly the changes won't be written back!).

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 wxWindows library itself would take advantage of it and could save various information in it. For example *wxFontMapper* (p. 448) or Unix version of *wxFileDialog* (p. 407) have ability to use wxConfig class.

Set (p. 174)

Get (p. 170)

Create (p. 169)

DontCreateOnDemand (p. 169)

Constructor and destructor

wxConfigBase (p. 168)

~wxConfigBase (p. 169)

Path management

As explained in *config overview* (p. 1358), 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. 175)

GetPath (p. 172)

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 surprizes 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:

```
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. 171)
GetNextGroup (p. 171)
GetFirstEntry (p. 171)
GetNextEntry (p. 171)
GetNumberOfEntries (p. 172)
GetNumberOfGroups (p. 172)

Tests of existence

HasGroup (p. 172)
HasEntry (p. 172)
Exists (p. 170)
GetEntryType (p. 170)

Miscellaneous functions

GetAppName (p. 170)
GetVendorName (p. 172)
SetUmask (p. 175)

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.

Read (p. 173)

Write (p. 175)

Flush (p. 170)

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.

RenameEntry (p. 174)

RenameGroup (p. 174)

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.

DeleteEntry (p. 169)

DeleteGroup (p. 170)

DeleteAll (p. 169)

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 valid 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 contains '\$' 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:

IsExpandingEnvVars (p. 172)

SetExpandEnvVars (p. 175)

SetRecordDefaults (p. 175)

IsRecordingDefaults (p. 172)

wxConfigBase::wxConfigBase

wxConfigBase(const wxString& appName = wxEmptyString, const wxString& vendorName = wxEmptyString, const wxString& localFilename = wxEmptyString, const wxString& globalFilename = wxEmptyString, long style = 0)

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. 23) 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.

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 new config object "accidental" creation.

`wxConfigBase::DeleteAll`

`bool DeleteAll()`

Delete the whole underlying object (disk file, registry key, ...). Primarily for use by desinstallation 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

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
{
    Unknown,
    String,
    Boolean,
    Integer,
    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.

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.

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.

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.

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.

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 the *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 the *oldName* doesn't exist or if *newName* already exists.

wxConfigBase::Set

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 the strPath doesn't exist it is created.

wxConfigBase::SetRecordDefaults

void SetRecordDefaults(bool bDoIt = TRUE)

Sets whether defaults are written back to the config file.

If on (default is off) all default values 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::SetUmask

void SetUmask(int mode)

NB: this function is not in the base wxConfigBase class but is only implemented in wxFileConfig. Moreover, this function is Unix-specific and doesn't do anything on other platforms.

SetUmask() 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 should do `SetUmask(0077)`.

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.

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. 1184)
wxEvtHandler (p. 378)
wxObject (p. 746)

Include files

<wx/control.h>

See also

wxValidator (p. 1166)

wxControl::Command

void Command(wxCommandEvent& event)

Simulates the effect of the user issuing a command to the item. See *wxCommandEvent* (p. 152).

wxControl::GetLabel

wxString& GetLabel()

Returns the control's text.

wxControl::SetLabel

void SetLabel(const wxString& label)

Sets the item's text.

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. 751) *wxStreamBase* (p. 998)**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 the same exactly purpose as *mutexes* (p. 730). 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. 179) class whenever possible instead of directly using *wxCriticalSection* for the same reasons *wxMutexLocker* (p. 733) is preferable to *wxMutex* (p. 730) - please see *wxMutex* for an example.

Derived from

None.

Include files

<wx/thread.h>

See also

wxThread (p. 1101), *wxCondition* (p. 160), *wxMutexLocker* (p. 733), *wxCriticalSection* (p. 178)

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. 178) 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 SetFoo()
{
    // 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. 178), *wxMutexLocker* (p. 733)

wxCriticalSectionLocker::wxCriticalSectionLocker

wxCriticalSectionLocker(wxCriticalSection& *criticalsection*)

Constructs a `wxCriticalSectionLocker` object associated with given *criticalsection* and enters it.

wxCriticalSectionLocker::~~wxCriticalSectionLocker

~wxCriticalSectionLocker()

Destuctor leaves the critical section.

wxCSCConv

This class converts between any character sets and Unicode. It has one predefined instance, **wxCSCConvLocal**, for the default user character set.

Derived from

wxMBConv (p. 661)

Include files

<wx/strconv.h>

See also

wxMBCConv (p. 661), *wxEncodingConverter* (p. 371), *wxMBCConv classes overview* (p. 1343)

wxCSCConv::wxCSCConv

wxCSCConv(const wxChar* charset)

Constructor. Specify the name of the character set you want to convert from/to.

wxCSCConv::~~wxCSCConv

~wxCSCConv()

Destructor.

wxCSCConv::LoadNow

void LoadNow()

If the conversion tables needs to be loaded from disk, this method will do so. Otherwise, they will be loaded when any of the conversion methods are called.

wxCSCConv::MB2WC

size_t MB2WC(wchar_t* buf, const char* psz, size_t n) const

Converts from the selected character set to Unicode. Returns the size of the destination buffer.

wxCSCConv::WC2MB

size_t WC2MB(char* buf, const wchar_t* psz, size_t n) const

Converts from Unicode to the selected character set. Returns the size of the destination buffer.

wxCustomDataObject

wxCustomDataObject is a specialization of *wxDataObjectSimple* (p. 201) 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. 183) or `TakeData` (p. 183) 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. 183), `GetData` (p. 183) and `SetData` (p. 183) (or may be only the first two or only the last one if you only allow reading/writing the data)

Derived from

`wxDataObjectSimple` (p. 201)
`wxDataObject` (p. 196)

Include files

```
<wx/dataobj.h>
```

See also

`wxDataObject` (p. 196)

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. 202) 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. 183) 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

wxPython note: This method expects a string in wxPython. You can pass nearly any object by pickling it first.

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.

wxCustomDataObject::TakeData

virtual void TakeData(size_t size, const void *data)

Like *SetData* (p. 183), but doesn't copy the data - instead the object takes ownership of the pointer.

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. 558) for an example).

A single cursor object may be used in many windows (any subwindow type). The wxWindows 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. 1263) is also available for MS Windows use.

Derived from

wxBitmap (p. 54)
wxGDIObject (p. 474)
wxObject (p. 746)

Include files

<wx/cursor.h>

Predefined objects

Objects:

wxNullCursor

Pointers:

wxSTANDARD_CURSOR
wxHOURLASS_CURSOR
wxCROSS_CURSOR

See also

wxBitmap (p. 54), *wxIcon* (p. 558), *wxWindow::SetCursor* (p. 1223), `::wxSetCursor` (p. 1263)

wxCursor::wxCursor

wxCursor()

Default constructor.

wxCursor(const char bits[], int width, int height, int hotSpotX=-1, int hotSpotY=-1, const char maskBits[]=NULL)

Constructs a cursor by passing an array of bits (Motif and Xt only). *maskBits* is used only under Motif.

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.

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 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 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_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.

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

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. wxWindows 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.

wxDatabase

Every database object represents an ODBC connection. The connection may be closed and reopened.

Derived from

wxObject (p. 746)

Include files

<wx/odbc.h>

See also

wxDatabase overview (p. 1425), *wxRecordSet* (p. 872)

A much more robust and feature-rich set of ODBC classes is now available and recommended for use in place of the *wxDatabase* class.

See details of these classes in: *wxDB* (p. 242), *wxDBTable* (p. 266)

wxDatabase::wxDatabase

wxDatabase()

Constructor. The constructor of the first *wxDatabase* instance of an application initializes the ODBC manager.

wxDatabase::~~wxDatabase

~wxDatabase()

Destructor. Resets and destroys any associated *wxRecordSet* instances.

The destructor of the last *wxDatabase* instance will deinitialize the ODBC manager.

wxDatabase::BeginTrans

bool BeginTrans()

Not implemented.

wxDatabase::Cancel

void Cancel()

Not implemented.

wxDatabase::CanTransact

bool CanTransact()

Not implemented.

wxDatabase::CanUpdate

bool CanUpdate()

Not implemented.

wxDatabase::Close

bool Close()

Resets the statement handles of any associated wxRecordSet objects, and disconnects from the current data source.

wxDatabase::CommitTrans

bool CommitTrans()

Commits previous transactions. Not implemented.

wxDatabase::ErrorOccured

bool ErrorOccured()

Returns TRUE if the last action caused an error.

wxDatabase::ErrorSnapshot

void ErrorSnapshot(HSTMT statement = SQL_NULL_HSTMT)

This function will be called whenever an ODBC error occurred. It stores the error related information returned by ODBC. If a statement handle of the concerning ODBC action is available it should be passed to the function.

wxDatabase::GetDatabaseName

wxString GetDatabaseName()

Returns the name of the database associated with the current connection.

wxDatabase::GetDataSource

wxString GetDataSource()

Returns the name of the connected data source.

wxDatabase::GetErrorClass

wxString GetErrorClass()

Returns the error class of the last error. The error class consists of five characters where the first two characters contain the class and the other three characters contain the subclass of the ODBC error. See ODBC documentation for further details.

wxDatabase::GetErrorCode

wxRETCODE GetErrorCode()

Returns the error code of the last ODBC function call. This will be one of:

SQL_ERROR	General error.
SQL_INVALID_HANDLE	An invalid handle was passed to an ODBC function.
SQL_NEED_DATA	ODBC expected some data.
SQL_NO_DATA_FOUND	No data was found by this ODBC call.
SQL_SUCCESS	The call was successful.
SQL_SUCCESS_WITH_INFO	The call was successful, but further information can be obtained from the ODBC manager.

wxDatabase::GetErrorMessage

wxString GetErrorMessage()

Returns the last error message returned by the ODBC manager.

wxDatabase::GetErrorNumber

long GetErrorNumber()

Returns the last native error. A native error is an ODBC driver dependent error number.

wxDatabase::GetHDBC

HDBC GetHDBC()

Returns the current ODBC database handle.

wxDatabase::GetHENV

HENV GetHENV()

Returns the ODBC environment handle.

wxDatabase::GetInfo

bool GetInfo(long *infoType*, long **buf*)

bool GetInfo(long *infoType*, const wxString& *buf*, int *bufSize*=-1)

Returns requested information. The return value is TRUE if successful, FALSE otherwise.

infoType is an ODBC identifier specifying the type of information to be returned.

buf is a character or long integer pointer to storage which must be allocated by the application, and which will contain the information if the function is successful.

bufSize is the size of the character buffer. A value of -1 indicates that the size should be computed by the GetInfo function.

wxDatabase::GetPassword

wxString GetPassword()

Returns the password of the current user.

wxDatabase::GetUsername

wxString GetUsername()

Returns the current username.

wxDatabase::GetODBCVersionFloat

float GetODBCVersionFloat(bool implementation=TRUE)

Returns the version of ODBC in floating point format, e.g. 2.50.

implementation should be TRUE to get the DLL version, or FALSE to get the version defined in the `sql.h` header file.

This function can return the value 0.0 if the header version number is not defined (for early versions of ODBC).

wxDatabase::GetODBCVersionString

wxString GetODBCVersionString(bool implementation=TRUE)

Returns the version of ODBC in string format, e.g. "02.50".

implementation should be TRUE to get the DLL version, or FALSE to get the version defined in the `sql.h` header file.

This function can return the value "00.00" if the header version number is not defined (for early versions of ODBC).

wxDatabase::InWaitForDataSource

bool InWaitForDataSource()

Not implemented.

wxDatabase::IsOpen

bool IsOpen()

Returns TRUE if a connection is open.

wxDatabase::Open

```
bool Open(const wxString& datasource, bool exclusive = FALSE, bool readOnly = TRUE, const wxString& username = "ODBC", const wxString& password = "")
```

Connect to a data source. *datasource* contains the name of the ODBC data source. The parameters *exclusive* and *readOnly* are not used.

wxDatabase::OnSetOptions

```
void OnSetOptions(wxRecordSet *recordSet)
```

Not implemented.

wxDatabase::OnWaitForDataSource

```
void OnWaitForDataSource(bool stillExecuting)
```

Not implemented.

wxDatabase::RollbackTrans

```
bool RollbackTrans()
```

Sends a rollback to the ODBC driver. Not implemented.

wxDatabase::SetDataSource

```
void SetDataSource(const wxString& s)
```

Sets the name of the data source. Not implemented.

wxDatabase::SetLoginTimeout

```
void SetLoginTimeout(long seconds)
```

Sets the time to wait for an user login. Not implemented.

wxDatabase::SetPassword

```
void SetPassword(const wxString& s)
```

Sets the password of the current user. Not implemented.

wxDatabase::SetSynchronousMode

void SetSynchronousMode(bool *synchronous*)

Toggles between synchronous and asynchronous mode. Currently only synchronous mode is supported, so this function has no effect.

wxDatabase::SetQueryTimeout

void SetQueryTimeout(long *seconds*)

Sets the time to wait for a response to a query. Not implemented.

wxDatabase::SetUsername

void SetUsername(const wxString& *s*)

Sets the name of the current user. Not implemented.

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 (`CLIPFORMAT` under 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

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. 1420), *DnD sample* (p. 1323), *wxDataObject* (p. 196)

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. 196) or *SetId* (p. 196) later in this case)

wxDataFormat::wxDataFormat

wxDataFormat(const wxChar *format)

Constructs a data format object for a custom format identified by its name *format*.

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.

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 usual '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 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 allows to distinguish between them. See *wxDataFormat* (p. 194) 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, `wxWindows` has two predefined classes which derive from `wxDataObject`: *wxDataObjectSimple* (p. 201) and *wxDataObjectComposite* (p. 200). *wxDataObjectSimple* (p. 201) is the simplest `wxDataObject` possible and only holds data in a single format (such as HTML or text) and *wxDataObjectComposite* (p. 200) is the simplest way to implement `wxDataObject` which 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 the clipboard and drag and drop, nor the 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. 406), `wxTextDataObject` (p. 1083) and `wxBitmapDataObject` (p. 75) which can be used without change.

You may also derive your own data object classes from `wxCustomDataObject` (p. 181) for user-defined types. The format of user-defined data is given as 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. 201) 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. 199).

Derived from

None

Include files

<wx/dataobj.h>

See also

Clipboard and drag and drop overview (p. 1420), *DnD sample* (p. 1323), *wxFileDataObject* (p. 406), *wxTextDataObject* (p. 1083), *wxBitmapDataObject* (p. 75), *wxCustomDataObject* (p. 181), *wxDropTarget* (p. 368), *wxDropSource* (p. 365), *wxTextDropTarget* (p. 1090), *wxFileDropTarget* (p. 412)

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.

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. 196) derivation which may be used to support multiple formats. It contains several `wxDataObjectSimple` (p. 201) 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. 201)) and its format determines the preferred format of the composite data object as well.

See `wxDataObject` (p. 196) 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. 196)

Include files

<wx/dataobj.h>

See also

Clipboard and drag and drop overview (p. 1420), `wxDataObject` (p. 196), `wxDataObjectSimple` (p. 201), `wxFileDataObject` (p. 406), `wxTextDataObject` (p. 1083), `wxBitmapDataObject` (p. 75)

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. 196) 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.

Virtual functions to override

The objects supporting rendering the data must override *GetDataSize* (p. 202) and *GetDataHere* (p. 202) while the objects which may be set must override *SetData* (p. 202). Of course, the objects supporting both operations must override all three methods.

Derived from

wxDataObject (p. 196)

Include files

<wx/dataobj.h>

See also

Clipboard and drag and drop overview (p. 1420), *DnD sample* (p. 1323), *wxFileDataObject* (p. 406), *wxTextDataObject* (p. 1083), *wxBitmapDataObject* (p. 75)

wxDataObjectSimple::wxDataObjectSimple

```
wxDataObjectSimple(const wxDataFormat& format = wxFormatInvalid)
```

Constructor accepts the supported format (none by default) which may also be set later with *SetFormat* (p. 202).

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.

wxDatInputStream

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. 1084) 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. 205).

Derived from

None

Include files

<wx/datstrm.h>

wxDatInputStream::wxDatInputStream

wxDatInputStream(wxInputStream& stream)

Constructs a datastream object from an input stream. Only read methods will be available.

Parameters

stream

The input stream.

wxDatInputStream::~~wxDatInputStream

~wxDatInputStream()

Destroys the wxDatInputStream object.

wxDatInputStream::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).

wxDatInputStream::Read8

wxUInt8 Read8()

Reads a single byte from the stream.

wxDatInputStream::Read16

wxUInt16 Read16()

Reads a 16 bit integer from the stream.

wxDatInputStream::Read32

wxUInt32 Read32()

Reads a 32 bit integer from the stream.

wxDatInputStream::ReadDouble

double ReadDouble()

Reads a double (IEEE encoded) from the stream.

wxDatInputStream::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.

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. 1087) instead.

The << operator is overloaded and you can use this class like a standard C++ iostream. See *wxDataInputStream* (p. 203) for its usage and caveats.

See also *wxDataInputStream* (p. 203).

Derived from

None

wxDataOutputStream::wxDataOutputStream

wxDataOutputStream(wxOutputStream& stream)

Constructs a datastream object from an output stream. Only write methods will be available.

Parameters

stream

The output stream.

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.

wxDataOutputStream::Write16

void Write16(wxUint16 i16)

Writes the 16 bit integer *i16* to the stream.

wxDataOutputStream::Write32

void Write32(wxUint32 i32)

Writes the 32 bit integer *i32* to the stream.

wxDataOutputStream::WriteDouble

void WriteDouble(double f)

Writes the double *f* to the stream using the IEEE format.

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.

wxDate

A class for manipulating dates.

NOTE: this class is retained only for compatibility, and has been replaced by *wxDateTime* (p. 215). *wxDate* may be withdrawn in future versions of *wxWindows*.

Derived from

wxObject (p. 746)

Include files

<wx/date.h>

See also

wxTime (p. 1108)

wxDate::wxDate

wxDate()

Default constructor.

wxDate(const wxDate& date)

Copy constructor.

wxDate(int month, int day, int year)

Constructor taking month, day and year.

wxDate(long julian)

Constructor taking an integer representing the Julian date. This is the number of days since 1st January 4713 B.C., so to convert from the number of days since 1st January 1901, construct a date for 1/1/1901, and add the number of days.

wxDate(const wxString& dateString)

Constructor taking a string representing a date. This must be either the string TODAY, or of the form MM/DD/YYYY or MM-DD-YYYY. For example:

```
wxDate date("11/26/1966");
```

Parameters

date

Date to copy.

month

Month: a number between 1 and 12.

day

Day: a number between 1 and 31.

year

Year, such as 1995, 2005.

wxDate::~~wxDate

void ~wxDate()

Destructor.

wxDate::AddMonths

wxDate& AddMonths(int months=1)

Adds the given number of months to the date, returning a reference to 'this'.

wxDate::AddWeeks

wxDate& AddWeeks(int weeks=1)

Adds the given number of weeks to the date, returning a reference to 'this'.

wxDate::AddYears

wxDate& AddYears(int years=1)

Adds the given number of months to the date, returning a reference to 'this'.

wxDate::FormatDate

wxString FormatDate(int type=-1) const

Formats the date according to *type* if not -1, or according to the current display type if -1.

Parameters

type

-1 or one of:

wxDAY	Format day only.
wxMONTH	Format month only.
wxMDY	Format MONTH, DAY, YEAR.
wxFULL	Format day, month and year in US style: DAYOFWEEK, MONTH, DAY, YEAR.
wxEUROPEAN	Format day, month and year in European style: DAY,

MONTH, YEAR.

wxDate::GetDay

int GetDay() const

Returns the numeric day (in the range 1 to 31).

wxDate::GetDayOfWeek

int GetDayOfWeek() const

Returns the integer day of the week (in the range 1 to 7).

wxDate::GetDayOfWeekName

wxString GetDayOfWeekName() const

Returns the name of the day of week.

wxDate::GetDayOfYear

long GetDayOfYear() const

Returns the day of the year (from 1 to 365).

wxDate::GetDaysInMonth

int GetDaysInMonth() const

Returns the number of days in the month (in the range 1 to 31).

wxDate::GetFirstDayOfMonth

int GetFirstDayOfMonth() const

Returns the day of week that is first in the month (in the range 1 to 7).

wxDate::GetJulianDate

long GetJulianDate() const

Returns the Julian date.

wxDate::GetMonth

int GetMonth() const

Returns the month number (in the range 1 to 12).

wxDate::GetMonthEnd

wxDate GetMonthEnd()

Returns the date representing the last day of the month.

wxDate::GetMonthName

wxString GetMonthName() const

Returns the name of the month. Do not delete the returned storage.

wxDate::GetMonthStart

wxDate GetMonthStart() const

Returns the date representing the first day of the month.

wxDate::GetWeekOfMonth

int GetWeekOfMonth() const

Returns the week of month (in the range 1 to 6).

wxDate::GetWeekOfYear

int GetWeekOfYear() const

Returns the week of year (in the range 1 to 52).

wxDate::GetYear

int GetYear() const

Returns the year as an integer (such as '1995').

wxDate::GetYearEnd

wxDate GetYearEnd() const

Returns the date representing the last day of the year.

wxDate::GetYearStart

wxDate GetYearStart() const

Returns the date representing the first day of the year.

wxDate::IsLeapYear

bool IsLeapYear() const

Returns TRUE if the year of this date is a leap year.

wxDate::Set

wxDate& Set()

Sets the date to current system date, returning a reference to 'this'.

wxDate& Set(long julian)

Sets the date to the given Julian date, returning a reference to 'this'.

wxDate& Set(int month, int day, int year)

Sets the date to the given date, returning a reference to 'this'.

month is a number from 1 to 12.

day is a number from 1 to 31.

year is a year, such as 1995, 2005.

wxDate::SetFormat

void SetFormat(int format)

Sets the current format type.

Parameters

format

-1 or one of:

wxDAY	Format day only.
wxMONTH	Format month only.
wxMDY	Format MONTH, DAY, YEAR.
wxFULL	Format day, month and year in US style: DAYOFWEEK, MONTH, DAY, YEAR.
wxEUROPEAN	Format day, month and year in European style: DAY, MONTH, YEAR.

wxDate::SetOption

int SetOption(int *option*, const bool *enable=TRUE*)

Enables or disables an option for formatting.

Parameters

option

May be one of:

wxNO_CENTURY	The century is not formatted.
wxDATE_ABBR	Month and day names are abbreviated to 3 characters when formatting.

wxDate::operator wxString

operator wxString()

Conversion operator, to convert wxDate to wxString by calling FormatDate.

wxDate::operator +

wxDate operator +(long *i*)

wxDate operator +(int *i*)

Adds an integer number of days to the date, returning a date.

wxDate::operator -

wxDate operator -(long *i*)**wxDate operator -(int *i*)**

Subtracts an integer number of days from the date, returning a date.

long operator -(const wxDate& *date*)

Subtracts one date from another, return the number of intervening days.

wxDate::operator +=

wxDate& operator +=(long *i*)

Postfix operator: adds an integer number of days to the date, returning a reference to 'this' date.

wxDate::operator -=

wxDate& operator -=(long *i*)

Postfix operator: subtracts an integer number of days from the date, returning a reference to 'this' date.

wxDate::operator ++

wxDate& operator ++()

Increments the date (postfix or prefix).

wxDate::operator --

wxDate& operator --()

Decrements the date (postfix or prefix).

wxDate::operator <

friend bool operator <(const wxDate& *date1*, const wxDate& *date2*)

Function to compare two dates, returning TRUE if *date1* is earlier than *date2*.

wxDate::operator <=

friend bool operator <=(const wxDate& date1, const wxDate& date2)

Function to compare two dates, returning TRUE if *date1* is earlier than or equal to *date2*.

wxDate::operator >

friend bool operator >(const wxDate& date1, const wxDate& date2)

Function to compare two dates, returning TRUE if *date1* is later than *date2*.

wxDate::operator >=

friend bool operator >=(const wxDate& date1, const wxDate& date2)

Function to compare two dates, returning TRUE if *date1* is later than or equal to *date2*.

wxDate::operator ==

friend bool operator ==(const wxDate& date1, const wxDate& date2)

Function to compare two dates, returning TRUE if *date1* is equal to *date2*.

wxDate::operator !=

friend bool operator !=(const wxDate& date1, const wxDate& date2)

Function to compare two dates, returning TRUE if *date1* is not equal to *date2*.

wxDate::operator <<

friend ostream& operator <<(ostream& os, const wxDate& date)

Function to output a wxDate to an ostream.

wxDateSpan

The documentation for this section has not yet been written.

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 (= Greenwich Mean Time): they're guaranteed
    // consequent numbers, so writing something like `GMT0 + offset'
    // 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::Monthenum`.

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. 223) and `GetWeekDayName` (p. 224) functions use the following 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 parameter 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 which conventions should be used using `SetCountry` (p. 225). 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. 231) and `GetWeekOfMonth` (p. 232)).

The desired behaviour 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. 1336), `wxTimeSpan`, `wxDateSpan`, `wxCalendarCtrl` (p. 95)

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. 225)
GetCountry (p. 223)
IsWestEuropeanCountry (p. 224)
GetCurrentYear (p. 223)
ConvertYearToBC (p. 222)
GetCurrentMonth (p. 223)
IsLeapYear (p. 224)
GetCentury (p. 223)
GetNumberOfDays (p. 224)
GetNumberOfDays (p. 224)
GetMonthName (p. 223)
GetWeekDayName (p. 224)
GetAmPmStrings (p. 222)
IsDSTApplicable (p. 225)
GetBeginDST (p. 222)
GetEndDST (p. 223)
Now (p. 225)
UNow (p. 226)
Today (p. 225)

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. 230) method to check that the values were correct as constructors can not return an error code.

wxDateTime() (p. 226)
wxDateTime(time_t) (p. 226)
wxDateTime(struct tm) (p. 226)
wxDateTime(double jdn) (p. 226)
wxDateTime(h, m, s, ms) (p. 227)
wxDateTime(day, mon, year, h, m, s, ms) (p. 227)
SetToCurrent (p. 227)
Set(time_t) (p. 227)
Set(struct tm) (p. 227)
Set(double jdn) (p. 228)
Set(h, m, s, ms) (p. 228)
Set(day, mon, year, h, m, s, ms) (p. 228)

ResetTime (p. 228)
SetYear (p. 228)
SetMonth (p. 228)
SetDay (p. 228)
SetHour (p. 229)
SetMinute (p. 229)
SetSecond (p. 229)
SetMillisecond (p. 229)
operator=(time_t) (p. 229)
operator=(struct tm) (p. 229)

Accessors

Here are the trivial accessors. Other functions, which might have to perform some more complicated calculations to find the answer are under *Calendar calculations* (p. 221) section.

IsValid (p. 230)
GetTicks (p. 230)
GetYear (p. 230)
GetMonth (p. 230)
GetDay (p. 230)
GetWeekDay (p. 231)
GetHour (p. 231)
GetMinute (p. 231)
GetSecond (p. 231)
GetMillisecond (p. 231)
GetDayOfYear (p. 231)
GetWeekOfYear (p. 231)
GetWeekOfMonth (p. 232)
GetYearDay (p. 240)
IsWorkDay (p. 232)
IsGregorianDate (p. 232)

Date comparison

There are several function to allow date comparison. To supplement them, a few global operators $>$, $<$ etc taking `wxDateTime` are defined.

IsEqualTo (p. 232)
IsEarlierThan (p. 232)
IsLaterThan (p. 232)
IsStrictlyBetween (p. 233)
IsBetween (p. 233)
IsSameDate (p. 233)
IsSameTime (p. 233)
IsEqualUpTo (p. 233)

Date arithmetics

These functions carry out *arithmetics* (p. 1338) 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.

Add(wxTimeSpan) (p. 234)
Add(wxDateSpan) (p. 234)
Subtract(wxTimeSpan) (p. 234)
Subtract(wxDateSpan) (p. 234)
Subtract(wxDateTime) (p. 235)
operator+=(wxTimeSpan) (p. 234)
operator+=(wxDateSpan) (p. 234)
operator-=(wxTimeSpan) (p. 234)
operator-=(wxDateSpan) (p. 234)

Parsing and formatting dates

These functions perform 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. 237) and *FormatTime* (p. 237)), using the international standard representation defined by ISO 8601 (*FormatISODate* (p. 237) and *FormatISOTime* (p. 237)) or by specifying any format at all and using *Format* (p. 236) directly.

The conversions from text are more interesting, as there are much more possibilities to care about. The simple cases can be taken care of with *ParseFormat* (p. 235) which can parse any date in the given (rigid) format. *ParseRfc822Date* (p. 235) 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. 236), hence the need for a separate function.

But the most interesting functions are *ParseDateTime* (p. 236) and *ParseDate* (p. 236) and *ParseTime* (p. 236). They try to parse the date and 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. 236) can parse the strings such as "tomorrow", "March first", "next Sunday".

ParseRfc822Date (p. 235)
ParseFormat (p. 235)
ParseDateTime (p. 236)
ParseDate (p. 236)
ParseTime (p. 236)
Format (p. 236)

FormatDate (p. 237)
FormatTime (p. 237)
FormatISODate (p. 237)
FormatISOTime (p. 237)

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. 237)
GetWeekDayInSameWeek (p. 237)
SetToNextWeekDay (p. 238)
GetNextWeekDay (p. 238)
SetToPrevWeekDay (p. 238)
GetPrevWeekDay (p. 238)
SetToWeekDay (p. 238)
GetWeekDay (p. 239)
SetToLastWeekDay (p. 239)
GetLastWeekDay (p. 239)
SetToTheWeek (p. 239)
GetWeek (p. 239)
SetToLastMonthDay (p. 239)
GetLastMonthDay (p. 240)
SetToYearDay (p. 240)
GetYearDay (p. 240)

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. 228) and you may also get its *JDN*, *MJD* (p. 241) or *Rata Die number* (p. 241) from it.

wxDateTime(double jdn) (p. 226)
Set(double jdn) (p. 228)
GetJulianDayNumber (p. 240)
GetJDN (p. 240)
GetModifiedJulianDayNumber (p. 241)
GetMJD (p. 241)
GetRataDie (p. 241)

Time zone and DST support

Please see the *time zone overview* (p. 1339) for more information about time zones. Normally, these functions should be rarely used.

ToTimezone (p. 241)
MakeTimezone (p. 241)
ToGMT (p. 241)
MakeGMT (p. 242)
GetBeginDST (p. 222)
GetEndDST (p. 223)
IsDST (p. 242)

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. 1339).

See also

GetEndDST (p. 223)

wxDatetime::GetCountry

static Country GetCountry()

Returns the current default country. The default country is used for DST calculations, for example.

See also

SetCountry (p. 225)

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. 222)

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. 224)

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::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. 223)

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 in 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. 226) should be used for better precision (but it is less efficient and might not be available on all platforms).

See also

Today (p. 225)

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. 215).

See also

GetCountry (p. 223)

wxDatetime::Today

static wxDateTime Today()

Returns the object corresponding to the midnight of the current day (i.e. the same as `Now()` (p. 225), but the time part is set to 0).

See also

`Now` (p. 225)

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. 225)

wxDateTime::wxDateTime

wxDateTime()

Default constructor. Use one of `Set ()` functions to initialize the object later.

wxDateTime::wxDateTime

wxDateTime& wxDateTime(time_t *time_t*)

Same as `Set` (p. 226).

wxPython note: This constructor is named `wxDateTimeFromTimeT` in wxPython.

wxDateTime::wxDateTime

wxDateTime& wxDateTime(const struct tm& *tm*)

Same as `Set` (p. 226)

wxPython note: Unsupported.

wxDateTime::wxDateTime

wxDateTime& wxDateTime(double *jdn*)

Same as *Set* (p. 226)

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. 227)

wxPython note: This constructor is named `wxDateTimeFromHMS` in wxPython.

wxDateTime::wxDateTime

wxDateTime& wxDateTime(`wxDateTime_t day`, `Month month = Inv_Month`,
`wxDateTime_t hour = 0`, `wxDateTime_t minute = 0`, `wxDateTime_t second = 0`,
`wxDateTime_t millisec = 0`)

Same as *Set* (p. 228)

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. 225) 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. 225) 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, 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 time_t)

Same as *Set* (p. 227).

wxDatetime::operator=

wxDatetime& operator(const struct tm& tm)

Same as *Set* (p. 227).

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. The week number is in 1...53 range (52 for non leap years).

The function depends on the *week start* (p. 215) convention specified by the *flags* argument.

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. 231), this function supports both conventions for the week start. See the description of these *week start* (p. 215) conventions.

wxDateTime::IsWorkDay

bool IsWorkDay(Country country = Country_Default) const

Returns `TRUE` if 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::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. 233)

wxDateTime::IsBetween

bool IsBetween(const wxDateTime&t1, const wxDateTime&t2) const

Returns `TRUE` if *IsStrictlyBetween* (p. 233) is `TRUE` or if the date is equal to one of the limit values.

See also

IsStrictlyBetween (p. 233)

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::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::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 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 `NULL` 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. 236) or *ParseDate* (p. 236) instead.

wxDateTime::ParseFormat

const wxChar * ParseFormat(const wxChar *date, const wxChar *format = "%c", 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 be not perfect). This function does handle the month and weekday names in the current locale on all platforms, however.

Please the description of ANSI C function `strptime(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 day of the month), the month and the year are taken from *dateDef*. If it is not specified, *Today* (p. 225) 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. 235), 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.

wxDateTime::ParseDate

const wxChar * ParseDate(const wxChar *date)

This function is like *ParseDateTime* (p. 236), but it only allows the date to be specified. It is thus flexible than *ParseDateTime* (p. 236), 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. 236), 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 = "%c", 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 wxWindows-specific extensions: you can optionally specify the width of the field to follow using `printf(3)`-like syntax and the format specifier `%l` can be used to get the number of milliseconds.

See also

ParseFormat (p. 235)

wxDatetime::FormatDate

wxString FormatDate() const

Identical to calling *Format()* (p. 236) with "%x" argument (which means 'preferred date representation for the current locale').

wxDatetime::FormatTime

wxString FormatTime() const

Identical to calling *Format()* (p. 236) 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)

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) const

Returns the copy of this object to which `SetToWeekDayInSameWeek` (p. 237) 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. 238) 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. 238) 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. 238) 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. 239) was applied.

wxDatetime::SetToTheWeek

bool SetToTheWeek(wxDatetime_t numWeek, WeekDay weekday = Mon)

Set the date to the given *weekday* in the week with given number *numWeek*. The number should be in range 1...53 and `FALSE` will be returned if the specified date doesn't exist. `TRUE` is returned if the date was changed successfully.

wxDatetime::GetWeek

wxDatetime GetWeek(wxDatetime_t numWeek, WeekDay weekday = Mon) const

Returns the copy of this object to which *SetToTheWeek* (p. 239) was applied.

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. 239) 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. 240) was applied.

wxDatetime::GetJulianDayNumber

double GetJulianDayNumber() **const**

Returns the *JDN* (p. 228) corresponding to this date. Beware of rounding errors!

See also

GetModifiedJulianDayNumber (p. 241)

wxDatetime::GetJDN

double GetJDN() **const**

Synonym for *GetJulianDayNumber* (p. 240).

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. 241).

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. 241) with the argument GMT0.

wxDatetime::MakeGMT

wxDatetime& MakeGMT(bool *noDST* = FALSE)

This is the same as calling *MakeTimezone* (p. 241) 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. 222) and *GetEndDST* (p. 223)

wxDatetimeHolidayAuthority

TODO

wxDatetimeWorkDays

TODO

wxDdb

A *wxDdb* instance is a connection to an ODBC data source which may be opened, closed, and re-opened an unlimited number of times. A database connection allows function to be performed directly on the data source, as well as allowing access to any tables/views defined in the data source to which the user has sufficient privileges.

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. 265)
- `wxDbCollnf` (p. 265)
- `wxDbTableInf` (p. 280)
- `wxDbInf` (p. 266)

Constants

NOTE: In a future release, all ODBC class constants will be prefaced with 'wx'

<code>wxDB_PATH_MAX</code> the ODBC located.	Maximum path length allowed to be passed to driver to indicate where the data file is located.
<code>DB_MAX_COLUMN_NAME_LEN</code> column	Maximum supported length for the name of a column
<code>DB_MAX_ERROR_HISTORY</code> the errors.	Maximum number of error messages retained in queue before being overwritten by new errors.
<code>DB_MAX_ERROR_MSG_LEN</code> returned	Maximum supported length of an error message by the ODBC classes
<code>DB_MAX_STATEMENT_LEN</code> statement	Maximum supported length for a complete SQL to be passed to the ODBC driver
<code>DB_MAX_TABLE_NAME_LEN</code> table	Maximum supported length for the name of a table
<code>DB_MAX_WHERE_CLAUSE_LEN</code> can be	Maximum supported WHERE clause length that passed to the ODBC driver
<code>DB_TYPE_NAME_LEN</code> data type	Maximum length of the name of a column's data type

Enumerated types

enum **wxDbSqlLogState**
sqlLogOFF, sqlLogON

enum **wxDBMS**

These are the databases currently tested and working with the ODBC classes. A call to `wxDb::Dbms` (p. 263) will return one of these enumerated values listed below.

```
dbmsUNIDENTIFIED,  
dbmsORACLE,
```

```

dbmsSYBASE_ASA,      // Adaptive Server Anywhere
dbmsSYBASE_ASE,      // Adaptive Server Enterprise
dbmsMS_SQL_SERVER,
dbmsMY_SQL,
dbmsPOSTGRES,
dbmsACCESS,
dbmsDBASE,
dbmsINFORMIX

```

See the remarks in *wxDb::Dbms* (p. 263) 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. 258). Contains the count of bytes in the *wxDb::errorMsg* string.

int wxDb::DB_STATUS

The last ODBC error 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 data source is opened, all of the information contained in the dbInf structure is queried from the data source. This information is used almost exclusively within the ODBC class library. Where there is a need for this information outside of the class library a member function such as wxDbTable::IsCursorClosedOnCommit() has been added for ease of use.

```

char dbmsName[40] - Name of the dbms product
char dbmsVer[64] - Version # of the dbms product
char driverName[40] - Driver name

```

```

char   odbcVer[60]           - ODBC version of the driver
char   drvMgrOdbcVer[60]    - ODBC version of the driver manager
char   driverVer[60]        - Driver version
char   serverName[80]       - Server Name, typically a connect string
char   databaseName[128]    - Database filename
char   outerJoins[2]        - Does datasource support outer joins
char   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
  char   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

```

char wxDb::errorList[DB_MAX_ERROR_HISTORY][DB_MAX_ERROR_MSG_LEN]
 The last n ODBC errors that have occurred on this database connection.

char wxDb::errorMsg[SQL_MAX_MESSAGE_LENGTH]
 This member variable is populated as a result of calling `wxD::GetNextError` (p. 258). It contains the ODBC error message text.

SDWORD wxDb::nativeError

Set by `wxD::DispAllErrors`, `wxD::GetNextError`, and `wxD::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 `wxD::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 wxWindows library is built. This behavior can be overridden when an instance of a wxDb is created (see *wxDb constructor* (p. 247)).

See also

wxDbColFor (p. 265), *wxDbCollInf* (p. 265), *wxDbTable* (p. 266), *wxDbTableInf* (p. 280), *wxDbInf* (p. 266)

Associated non-class functions

The following functions are used in conjunction with the wxDb class.

wxDb * wxDbGetConnection(wxDbConnectInf *pDbConfig, bool FwdOnlyCursors=(bool)wxODBC_FWD_ONLY_CURSORS)

bool wxDbFreeConnection(wxDb *pDb)

void wxDbCloseConnections()

int wxDbConnectionsInUse()

bool wxDbSqlLog(wxDbSqlLogState state, const wxChar *filename = SQL_LOG_FILENAME)

bool wxDbGetDataSource(HENV henv, char *Dsn, SWORD DsnMax, char *DsDesc, SWORD DsDescMax, UWORD direction = SQL_FETCH_NEXT)

wxDb::wxDb

wxDb()

Default constructor.

wxDb(HENV& aHenv, bool FwdOnlyCursors=(bool)wxODBC_FWD_ONLY_CURSORS)

Constructor, used to create an ODBC connection to a data source.

Parameters

aHenv

Environment handle used for this connection.

FwdOnlyCursors

Will cursors created for use with this datasource connection only allow forward scrolling cursors.

Remarks

This is the constructor for the `wxDdb` class. The `wxDdb` object must be created and opened before any database activity can occur.

Example

```
wxDdbConnectInf ConnectInf;
....Set values for member variables of ConnectInf here

wxDdb sampleDB(ConnectInf.Henv);
if (!sampleDB.Open(ConnectInf.Dsn, ConnectInf.Uid,
ConnectInf.AuthStr))
{
    // Error opening data source
}
```

wxDdb::Catalog

bool Catalog(char * *userID*, char **fileName* = SQL_CATALOG_FILENAME)

Allows a data "dictionary" of the data source 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 argument. Name of the text file to create and write the DB catalog to.

Return value

Returns TRUE if the catalog request was successful, of FALSE if there was some reason 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
```

wxDdb::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 data source. 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 data source
sampleDB.CommitTrans();

// Delete any remaining wxDbTable objects allocated with new
delete parts;

// Close the wxDb connection when finished with it
sampleDB.Close();

// Free Environment Handle that ODBC uses
if (SQLFreeEnv(Db.Henv) != SQL_SUCCESS)
{
    // Error freeing environment handle
}
```

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. At any time thereafter, you can save your work to the database ("Commit") or roll back all of your changes ("Rollback"). Calling this member function commits all open transactions on this ODBC connection.

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 means you must requery the data source before you can perform any additional work against the wxDbTable object. This is only necessary however if the data source closes the cursor after a commit or rollback. Use the wxDbTable::IsCursorClosedOnCommit() member function to determine the data source's transaction behavior. Note, it would be very inefficient to just assume the data source closes the cursor and always requery. This could put a significant, unnecessary load on data sources that leave the cursors open after a transaction.

wxDb::CreateView

bool CreateView(char * viewName, char * colList, char *pSqlStmt)

Creates a SQL VIEW.

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.

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_NO, 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
```

wxDdb::DispAllErrors

bool DispAllErrors(HENV aHenv, HDBC aHdbc = SQL_NULL_HDBC, HSTMT aHstmt = SQL_NULL_HSTMT)

Logs all database errors that occurred as a result of the last executed database command. This logging also includes debug logging when compiled in debug mode via *wxLogDebug* (p. 1299). If logging is turned on via *wxDdb::SetSqlLogging* (p. 262), then an entry is also logged to the defined log file.

Parameters

aHenv

A handle to the ODBC environment.

aHdbc

A handle to the ODBC connection. Pass this in if the ODBC function call that erred out required a hdbc or hstmt argument.

AHstmt

A handle to the ODBC statement being executed against. Pass this in if the ODBC function call that erred out required a hstmt argument.

Remarks

This member function will display all of the ODBC error messages for the last ODBC function call that was made. Normally used internally within the ODBC class library. Would be used externally if calling ODBC functions directly (i.e. *SQLFreeEnv()*).

See also

wxDdb::SetSqlLogging (p. 262), *wxDdbSqlLog*

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));
```

wxDdb::DispNextError

void DispNextError()

Remarks

This function is normally used internally within the ODBC class library. It could be used externally if calling ODBC functions directly. This function works in conjunction with *wxDdb::GetNextError* (p. 258) 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 (strcmp(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 char *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 data sources.

wxDb::ExecSql

bool ExecSql(char *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 `wxDdb` class and allows you to build and execute ANY VALID SQL statement against the data source. This allows you to extend the class library by being able to issue any SQL statement that the data source is capable of processing.

See also

`wxDdb::GetData` (p. 255), `wxDdb::GetNext` (p. 258)

wxDdb::FwdOnlyCursors

bool FwdOnlyCursors()

Indicates whether this connection to the datasource only allows forward scrolling cursors or not. This state is set at connection creation time.

Remarks

In `wxWindows` v2.4 release, this function will be deprecated to use a renamed version of the function called `wxDdb::IsFwdOnlyCursors()` to match the normal `wxWindows` naming conventions for class member functions.

See also

`wxDdb::IsFwdOnlyCursors` (p. 260), `wxDdb::wxDdb` (p. 247), `wxDdbGetConnection` (p. 247)

wxDdb::GetCatalog

wxDdbInf * GetCatalog(char *userID)

Returns a `wxDdbInf` pointer that points to the catalog(data source) name, schema, number of tables accessible to the current user, and a `wxDdbTableInf` pointer to all data pertaining to all tables in the users catalog.

Parameters

userID

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. `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 via this connection will be returned.

wxDdb::GetColumnCount

```
int GetColumnCount(char *tableName, const char *userID)
```

Parameters

tableName

A table name you wish to obtain column information about.

userID

Name of the user that owns the table(s). 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 the function will return a -1.

wxDdb::GetColumns

```
wxDdbColInf * GetColumns(char *tableName, int *numCols, const char *userID=NULL)
```

```
wxDdbColInf * GetColumns(char *tableName[], const char *userID)
```

Parameters

tableName

A table name you wish to obtain column information about.

numCols

A pointer to an integer 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). 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 an array of *wxDdbColInf* structures. This allows you to obtain information regarding the columns of your table(s). If no columns were found, or an error occurred, this pointer will be zero (null).

THE CALLING FUNCTION IS RESPONSIBLE FOR DELETING THE *wxDdbColInf* MEMORY WHEN IT IS FINISHED WITH IT.

ALL column bindings associated with this *wxDdb* instance are unbound by this function. This function should use its own *wxDdb* instance to avoid undesired unbinding of columns.

See also

wxDdbColInf (p. 265)

Example

```
char *tableList[] = {"PARTS", 0};
wxDdbColInf *colInf = pDb->GetColumns(tableList);
if (colInf)
{
    // Use the column inf
    .....
    // Destroy the memory
    delete [] colInf;
}
```

wxDdb::GetData

bool GetData(**UWORD** *colNo*, **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 *wxDdbTable* instance to access table data).

Parameters

colNo

Ordinal number of column in the result set to be returned.

cType

The C data type that is to be returned.

pData

Memory buffer which will hold the data returned by the call to this function.

maxLen

Maximum size of the buffer that will hold the returned 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 GetData() call has failed.

See also

wxDb::GetNext (p. 258), *wxDb::ExecSql* (p. 252)

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 this row of the result set and store 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 GetData must request the columns in ordinal sequence (1,2,3 or 1,3 or 2,3).

wxDdb::GetDatabaseName

char * GetDatabaseName()

Returns the name of the database engine.

wxDdb::GetDataSource

char * GetDataSource()

Returns the ODBC datasource name.

wxDdb::GetHDBC

HDBC GetHDBC()

Returns the ODBC handle to the database connection.

wxDdb::GetHENV

HENV GetHENV()

Returns the ODBC environment handle.

wxDdb::GetHSTMT

HSTMT GetHSTMT()

Returns the ODBC statement handle associated with this database connection.

wxDdb::GetKeyFields

int GetKeyFields(char *tableName, wxDbCollnf *collnf, intnocols)

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 *wxDdb::GetColumns* (p. 254) 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.

collnf

Data structure containing the column definitions (obtained with *wxDdb::GetColumns* (p. 254)). This function populates the *PkCol*, *PkTableName*, and *FkTableName* members of the *collnf* structure.

nocols

Number of columns defined in the instance of *collnf*.

Return value

Currently always returns TRUE.

See also

wxDdbCollnf (p. 265), *wxDdb::GetColumns* (p. 254)

wxDdb::GetNext

bool GetNext()

Requests the next row in the result set obtained by issuing a query through a direct request using *wxDdb::ExecSql()*.

See also

wxDdb::ExecSql (p. 252), *wxDdb::GetData* (p. 255)

wxDdb::GetNextError

bool GetNextError(HENV *aHenv*, HDBC *aHdbc* = SQL_NULL_HDBC, HSTMT *aHstmt* = SQL_NULL_HSTMT)

Parameters*aHenv*

A handle to the ODBC environment.

aHdbc

A handle to the ODBC connection. Pass this in if the ODBC function call that erred out required a *hdbc* or *hstmt* argument.

AHstmt

A handle to the ODBC statement being executed against. Pass this in if the ODBC function call that erred out requires a *hstmt* argument.

See also

wxDdb::DispNextError (p. 251), *wxDdb::DispAllErrors* (p. 251)

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));
    }

```

wxDdb::GetPassword

char * GetPassword()

Returns the password used to connect to the datasource.

wxDdb::GetTableCount

int GetTableCount()

Returns the number of wxDbTable() instances currently using this data source connection.

wxDdb::GetUsername

char * GetUsername()

Returns the user name used to access the datasource.

wxDdb::Grant

bool Grant(int privileges, char *tableName, char *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

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.

Example

```
db.Grant(DB_GRANT_SELECT | DB_GRANT_INSERT, "PARTS", "mary, sue");
```

wxDdb::IsFwdOnlyCursors

bool IsFwdOnlyCursors()

Indicates whether this connection to the datasource only allows forward scrolling cursors or not. This state is set at connection creation time.

Remarks

Added as of wxWindows v2.4 release, this function is a renamed version of wxDdb::FwdOnlyCursors() to match the normal wxWindows naming conventions for class member functions.

This function is not available in versions prior to v2.4. You should use *wxDdb::FwdOnlyCursors* (p. 253) for wxWindows versions prior to 2.4.

See also

wxDdb::wxDdb (p. 247), *wxDdbGetConnection* (p. 247)

wxDdb::IsOpen

bool IsOpen()

Indicates whether the database connection to the datasource is currently opened.

wxDdb::Open

bool Open(char *Dsn, char *Uid, char *AuthStr)

Parameters

Dsn

Data source name. The name of the ODBC data source as assigned when the data source 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 data source. 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.

Remarks

After a `wxDb` instance is created, it must then be opened. When opening a data source, 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 data source can be performed until it is opened. This would normally be done at program startup and the data source would remain open for the duration of the program run. Note: It is possible to have multiple data sources open at the same time to support distributed database connections.

Example

```
wxDb sampleDB(Db.Henv);
if (!sampleDB.Open("Oracle 7.1 HP/UX", "gtasker", "myPassword"))
{
    // Error opening data source
}
```

wxDb::RollbackTrans

bool RollbackTrans()

Function to "rollback" 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. 249) is called on the database connection.

Remarks

Transactions begin implicitly as soon as you make a change to the database. At any time thereafter, you can save your work to the database (using `wxDb::CommitTrans` (p. 249)) or undo all of your changes using this function.

Calling this member function rolls back ALL open (uncommitted) transactions on this ODBC connection.

See also

wxDdb::CommitTrans (p. 249) for a special note on cursors

wxDdb::SetSqlLogging

bool SetSqlLogging(wxDdbSqlLogState state, const wxChar *filename = SQL_LOG_FILENAME, bool append = FALSE)

Parameters

state

Either *sqlLogOFF* or *sqlLogON* (see *enum wxDdbSqlLogState* (p. 265)). Turns logging of SQL commands sent to the data source OFF or ON.

filename

OPTIONAL. Name of the file to which the log text is to be written.

append

OPTIONAL. Whether the file is appended to or overwritten.

Remarks

When called with *sqlLogON*, all commands sent to the data source engine are logged to the file specified by *filename*. Logging is done by embedded *WriteSqlLog()* calls in the database member functions, or may be manually logged by adding calls to *WriteSqlLog()* in your own source code.

When called with *sqlLogOFF*, the logging file is closed, and any calls to *WriteSqlLog()* are ignored.

wxDdb::TableExists

bool TableExists(const char *tableName, const char *userID=NULL, const char *path=NULL)

Checks the ODBC data source 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. 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.

wxDb::TranslateSqlState

int TranslateSqlState(const wxChar *SQLState)

Parameters

SQLState

Converts an ODBC sqlstate to an internal error code.

Return value

Returns the internal class DB_ERR code. See *wxDb::DB_STATUS* (p. 242) definition.

wxDb::WriteSqlLog

bool WriteSqlLog(const wxChar *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. The passed in string *logMsg* will be written to a log file if SQL logging is turned on (see *wxDb::SetSqlLogging* (p. 262) for details on turning logging on/off).

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. 262)

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 be work with these classes, but these databases returned by this function have been tested and confirmed to work with these ODBC classes.

enum wxDBMS includes:

```
dbmsUNIDENTIFIED
dbmsORACLE
dbmsSYBASE_ASA
dbmsSYBASE_ASE
dbmsMY_SQL_SERVER
dbmsMY_SQL
dbmsPOSTGRES
dbmsACCESS
dbmsDBASE
dbmsINFORMIX
```

There are known issues with conformance to the ODBC standards with several datasources listed above. 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::SetDebugErrorMessages

void SetDebugErrorMessages(bool state)

state

Either TRUE (debug messages are displayed) or FALSE (debug messages are not displayed).

Remarks

Turns on/off debug error messages from the ODBC class library. When this function is passed TRUE, errors are reported to the user 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. 247)

wxDb::LogError

void LogError(const char *errMsg const char *SQLState=0)

errMsg

Free-form text to display describing the error to be logged.

SQLState

Native SQL state error

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. 263)

wxDbColInf

Used with the *wxDb::GetColumns* (p. 254) functions for obtaining all retrievable information about a columns definition.

wxDbColFor

Beginning support for handling international formatting specifically on dates and floats.

Only one function is provided with this class currently:

wxDbColFor::Format

int Format(int Nation, int dbDataType, SWORD sqlDataType, short columnSize, short decimalDigits)

Work in progress, and should be inter-related with wxLocale

wxDblnf

Contains information regarding the database connection (data source 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.

wxDbTable

A wxDbTable instance provides re-usable access to rows of data in a table contained within the associated ODBC data source

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.

```
class wxDbColDef      : Bound column definitions for use by a
wxDbTable            instance

class wxDbColDataPtr : Pointer to dynamic column definitions for use
with                a wxDbTable instance

class wxDbIdxDef     : Used in creation of non-primary indexes
```

Constants

```
wxDB_DEFAULT_CURSOR   Index number of the cursor that each table
will use              by default.

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 specific] - Used when
CanUpdateByRowID() is true. Optimizes updates so they are faster
by
```

rather updating on the Oracle-specific ROWID column than some other index.

`wxDB_DISABLE_VIEW`
not be Use to indicate when a database view should if a table is normally set up to use a view. [Currently unsupported.]

Remarks

See also

`wxDBTable` (p. 266)

`wxDBTable::wxDBTable`

`wxDBTable(wxDB *pwxDb, const char *tblName, const int nCols, const char *qryTblName = 0, bool qryOnly = !wxDB_QUERY_ONLY, const char *tblPath=NULL)`

Default constructor.

Parameters

pSqlStmt

typeOfDel

pWhereClause
Default is 0.

`wxDBTable::wxDBTable`

`virtual ~wxDBTable()`

Virtual default destructor.

`wxDBTable::BuildDeleteStmt`

`void BuildSelectStmt(char *pSqlStmt, int typeOfDel, const char *pWhereClause=0)`

Parameters

pSqlStmt

typeOfDel

pWhereClause
Default is 0.

wxDbTable::BuildSelectStmt

void BuildSelectStmt(char *pSqlStmt, int typeOfSelect, bool distinct)

Parameters

pSqlStmt

typeOfSelect

distinct

wxDbTable::BuildUpdateStmt

void BuildSelectStmt(char *pSqlStmt, int typeOfUpd, const char *pWhereClause=0)

Parameters

pSqlStmt

typeOfUpd

pWhereClause
Default is 0.

wxDbTable::BuildWhereStmt

void BuildSelectStmt(char *pWhereClause, int typeOfWhere, const char *qualTableName=0, const char *useLikeComparison=FALSE)

Parameters

pWhereClause

typeOfWhere

qualTableName
Default is 0.
useLikeComparison
Default is FALSE.

wxDbTable::CanSelectForUpdate

bool CanSelectForUpdate()**Remarks****wxDbTable::CanUpdateByROWID**

bool CanUpdateByROWID()**Remarks****wxDbTable::ClearMemberVars**

void ClearMemberVars()**Remarks****wxDbTable::CloseCursor**

bool CloseCursor(*HSTMT cursor*)**Parameters***cursor***Remarks****wxDbTable::Count**

ULONG Count(const char **args*="*")**Parameters***args*

Default is "*".

Remarks**wxDbTable::CreateIndex**

bool CreateIndex(const char **idxName*, **bool** *unique*, **int** *noldxCols*, **wxDblIdxDef**

pIdxDefs*, **bool *attemptDrop=TRUE*)

Parameters

idxName

unique

nIdxCols

pIdxDefs

attemptDrop

Default is TRUE.

Remarks

wxDbTable::CreateTable

bool CreateTable(**bool** *attemptDrop=TRUE*)

Parameters

attemptDrop

Default is TRUE.

Remarks

wxDbTable::DB_STATUS

bool DB_STATUS()

Accessor function for the private member variable DB_STATUS.

wxDbTable::IsColNull

bool IsColNull(**int** *colNo*)

Parameters

colNo

Remarks

wxDbTable::Delete

bool Delete()

Remarks

wxDbTable::DeleteCursor

bool DeleteCursor(HSTMT *hstmtDel*)

Parameters

hstmtDel

Remarks

wxDbTable::DeleteWhere

bool DeleteWhere(const char **pWhereClause*)

Parameters

pWhereClause

Remarks

wxDbTable::DeleteMatching

bool DeleteMatching()

Remarks

wxDbTable::DropIndex

bool DropIndex(const char **idxName*)

Parameters

idxName

Remarks

wxDbTable::DropTable

bool DropTable()

Remarks

wxDbTable::GetColDefs

wxDbColDef * GetColDefs()

Remarks

wxDbTable::GetCursor

HSTMT GetCursor()

Remarks

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.

Remarks

wxDbTable::GetFirst

bool GetFirst()

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::FwdOnlyCursors (p. 253)

wxDbTable::GetFromClause

const char * GetFromClause()

Accessor function that returns the current FROM setting assigned with the *wxDbTable::SetFromClause* (p. 278).

wxDbTable::GetLast

bool GetLast()

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::FwdOnlyCursors (p. 253)

wxDbTable::GetNewCursor

HSTMT * GetNewCursor(bool setCursor=FALSE, bool bindColumns=TRUE)

Parameters

setCursor

Default is *FALSE*.

bindColumns

Default is *TRUE*.

Remarks

wxDbTable::GetNext

bool GetNext()

Remarks

wxDbTable::GetNumberOfColumns

bool GetNumberOfColumns()

Accessor function that returns the number of columns that are statically bound for access by the *wxDbTable* instance.

wxDbTable::GetOrderByClause

const char * GetOrderByClause()

Accessor function that returns the current ORDER BY setting assigned with the *wxDbTable::SetOrderByClause* (p. 278).

wxDbTable::GetPrev

bool GetPrev()

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::FwdOnlyCursors (p. 253)

wxDbTable::GetQueryTableName

const char * GetQueryTableName()

Remarks

wxDbTable::GetRowNum

UWORD GetRowNum()

Remarks

wxDbTable::GetTableName

const char * GetTableName()

Remarks

wxDbTable::GetTablePath

const char * GetTablePath()

Remarks

wxDbTable::GetWhereClause

const char * GetWhereClause()

Accessor function that returns the current WHERE setting assigned with the *wxDbTable::SetWhereClause* (p. 279)

wxDbTable::Insert

int Insert()**Remarks****wxDbTable::IsCursorClosedOnCommit**

bool IsCursorClosedOnCommit()**Remarks****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()**Remarks****wxDbTable::Query**

virtual bool Query(*bool*forUpdate=FALSE, *bool*distinct=FALSE)**Parameters***forUpdate*

Default is FALSE.

distinct

Default is FALSE.

Remarks

wxDbTable::QueryBySqlStmt

bool QueryBySqlStmt(const char *pSqlStmt)

Parameters

pSqlStmt

Remarks

wxDbTable::QueryMatching

virtual bool QueryMatching(**bool**forUpdate=FALSE, **bool**distinct=FALSE)

Parameters

forUpdate

Default is FALSE.

distinct

Default is FALSE.

Remarks

wxDbTable::QueryOnKeyFields

bool QueryOnKeyFields(**bool**forUpdate=FALSE, **bool**distinct=FALSE)

Parameters

forUpdate

Default is FALSE.

distinct

Default is FALSE.

Remarks

wxDbTable::Refresh

bool Refresh()

Remarks

wxDbTable::SetColDefs

```
void SetColDefs(int index, const char *fieldName, int dataType, void *pData, int cType, int size, bool keyField = FALSE, bool upd = TRUE, bool insAllow = TRUE, bool derivedCol = FALSE)
```

```
wxDbColDataPtr * SetColDefs(wxDbCollnf *collnfs, ULONG numCols)
```

Parameters*index**fieldName**dataType**pData**cType**size**keyField*

Default is FALSE.

upd

Default is TRUE.

insAllow

Default is TRUE.

derivedCol

Default is FALSE.

*collnfs**numCols***Remarks**

wxDbTable::SetCursor

```
bool Open(HSTMT *hstmtActivate = (void **) wxDB_DEFAULT_CURSOR)
```

Parameters*hstmtActivate*

Default is wxDB_DEFAULT_CURSOR.

Remarks**wxDbTable::SetFromClause**

```
void SetFromClause(const wxString&From)
```

Parameters

From

wxDbTable::SetNull

```
bool SetNull(int colNo)
```

```
bool SetNull(const char *colName)
```

Parameters

colNo

colName

Remarks**wxDbTable::SetOrderByClause**

```
void SetOrderByClause(const wxString&OrderBy)
```

Parameters

OrderBy

wxDbTable::SetQueryTimeout

```
bool SetQueryTimeout(UDWORD nSeconds)
```

Parameters

nSeconds

Remarks

wxDbTable::SetWhereClause

void SetWhereClause(const wxString&Where)

Parameters

Where

wxDbTable::Update

bool Update()

bool Update(const char *pSqlStmt)

Parameters

pSqlStmt

Remarks**wxDbTable::UpdateWhere**

bool UpdateWhere(const char *pWhereClause)

Parameters

pWhereClause

Remarks**wxDbTable::operator ++**

bool operator ++()

Same as *wxDbTable::GetNext* (p. 273)

See also

wxDbTable::GetNext (p. 273)

wxDbTable::operator --

bool operator --()

Same as `wxDbTable::GetPrev` (p. 274)

See also

`wxDbTable::GetPrev` (p. 274)

wxDbTableInf

Currently only used by `wxDb::GetCatalog()` internally and `wxDbInf` class, but may be used in future releases for user functions. Contains information describing the table (Name, type, etc). A pointer to a `wxDbCollnf` array instance is included so a program can create a `wxDbCollnf` array instance (using `wxDb::GetColumns` (p. 254)) to maintain all information about the columns of a table in one memory structure.

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.

Derived from

`wxObject` (p. 746)

Include files

<wx/dc.h>

See also

Overview (p. 1391)

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)

Copy from a source DC to this DC, specifying the destination coordinates, size of area to copy, source DC, source coordinates, and logical function.

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. 295).

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:

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.

Remarks

There is partial support for Blit in *wxPostScriptDC*, under X.

See *wxMemoryDC* (p. 678) for typical usage.

See also

wxMemoryDC (p. 678), *wxBitmap* (p. 54), *wxMask* (p. 659)

wxDC::CalcBoundingBox

void CalcBoundingBox(wxCoord x, wxCoord y)

Adds the specified point to the bounding box which can be retrieved with *MinX* (p. 293), *MaxX* (p. 293) and *MinY* (p. 293), *MaxY* (p. 293) functions.

See also

ResetBoundingBox (p. 293)

wxDC::Clear

void Clear()

Clears the device context using the current background brush.

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. 294).

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. 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. Use this function for converting a height, for example.

wxDC::DrawArc

void DrawArc(wxCoord x1, wxCoord y1, wxCoord x2, wxCoord y2, double xc, double 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. 297), *SetTextBackground* (p. 297) and *wxMemoryDC* (p. 678).

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::DrawEllipse

void DrawEllipse(wxCoord x, wxCoord y, wxCoord width, wxCoord height)

Draws an ellipse contained in the 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::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::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.

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.

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 wxWindows automatically closes the first and last points.

wxPython note: The wxPython version of this method accepts a Python list of wxPoint objects.

wxDC::DrawPoint

```
void DrawPoint(wxCoord x, wxCoord y)
```

Draws a point using the current pen.

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.

See also

DrawText (p. 287)

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(wxList **points*)

Draws a spline between all given control points, using the current pen. Doesn't delete the wxList and contents. The spline is drawn using a series of lines, using an algorithm taken from the X drawing program 'XFIG'.

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.

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. 291) 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. 290) 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

void 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.

Note: this function is available in MS Windows only.

wxDC::GetBackground

wxBrush& GetBackground()

const wxBrush& GetBackground() const

Gets the brush used for painting the background (see *wxDC::SetBackground* (p. 294)).

wxDC::GetBackgroundMode

int GetBackgroundMode() const

Returns the current background mode: `wxSOLID` or `wxTRANSPARENT`.

See also

SetBackgroundMode (p. 294)

wxDC::GetBrush

wxBrush& GetBrush()**const wxBrush& GetBrush() const**

Gets the current brush (see *wxDC::SetBrush* (p. 295)).

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.

wxDC::GetFont

wxFont& GetFont()**const wxFont& GetFont() const**

Gets the current font (see `wxDC::SetFont` (p. 295)).

wxDC::GetLogicalFunction

int GetLogicalFunction()

Gets the current logical function (see `wxDC::SetLogicalFunction` (p. 295)).

wxDC::GetMapMode

int GetMapMode()

Gets the *mapping mode* for the device context (see `wxDC::SetMapMode` (p. 296)).

wxDC::GetOptimization

bool GetOptimization()

Returns TRUE if device context optimization is on. See `wxDC::SetOptimization` (p. 296) for details.

wxDC::GetPen

wxPen& GetPen()

const wxPen& GetPen() const

Gets the current pen (see `wxDC::SetPen` (p. 297)).

wxDC::GetPixel

bool GetPixel(wxCoord x, wxCoord y, wxColour *colour)

Sets *colour* to the colour at the specified location. Windows only; an X implementation is being worked on. Not available for `wxPostScriptDC` or `wxMetafileDC`.

wxPython note: For wxPython the `wxColour` value is returned and is not required as a parameter.

wxDC::GetSize

void GetSize(wxCoord *width, wxCoord *height)

For a PostScript device context, this gets the maximum size of graphics drawn so far on the device context.

For a Windows printer device context, this gets the horizontal and vertical resolution. It can be used to scale graphics to fit the page when using a Windows printer device context. 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)

wxDC::GetTextBackground

wxColour& GetTextBackground()

const wxColour& GetTextBackground() const

Gets the current text background colour (see *wxDC::SetTextBackground* (p. 297)).

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. 434), *wxDC::SetFont* (p. 295).

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)

wxDC::GetTextForeground

wxColour& GetTextForeground()

const wxColour& GetTextForeground() const

Gets the current text foreground colour (see *wxDC::SetTextForeground* (p. 297)).

wxDC::GetUserScale

void GetUserScale(double *x, double *y)

Gets the current user scale factor (set by *SetUserScale* (p. 297)).

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. 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. 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. 282)

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::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::SetClippingRegion

void SetClippingRegion(wxCoord x, wxCoord y, wxCoord width, wxCoord height)

void SetClippingRegion(const wxRegion& region)

Sets the clipping region for the DC. 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. 283), *wxRegion* (p. 885)

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. 761) for further details.

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. 80).

See also *wxMemoryDC* (p. 678) for the interpretation of colours when drawing into a monochrome bitmap.

wxDC::SetFont

void SetFont(const wxFont& font)

Sets the current font for the DC.

If the argument is `wxNullFont`, the current font is selected out of the device context, and the original font restored, allowing the current font to be destroyed safely.

See also *wxFont* (p. 434).

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. 281)) 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:

<code>wxAND</code>	<code>src AND dst</code>
<code>wxAND_INVERT</code>	<code>(NOT src) AND dst</code>
<code>wxAND_REVERSE</code>	<code>src AND (NOT dst)</code>
<code>wxCLEAR</code>	<code>0</code>
<code>wxCOPY</code>	<code>src</code>
<code>wxEQUIV</code>	<code>(NOT src) XOR dst</code>
<code>wxINVERT</code>	<code>NOT dst</code>
<code>wxNAND</code>	<code>(NOT src) OR (NOT dst)</code>

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. 297)) scales the text appropriately. In Windows, scaleable TrueType fonts are always used; in X, results depend on availability of fonts, but usually a reasonable match is found.

Note that the coordinate origin should ideally be selectable, but for now is always at the top left of the screen/printer.

Drawing to a Windows printer device context under UNIX 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::SetOptimization

void SetOptimization(bool *optimize*)

If *optimize* is TRUE (the default), this function sets optimization mode on. This currently means that under X, the device context will not try to set a pen or brush property if it is known to be set already. This approach can fall down if non-wxWindows code is using the same device context or window, for example when the window is a panel on which the windowing system draws panel items. The wxWindows device context 'memory' will now be out of step with reality.

Setting optimization off, drawing, then setting it back on again, is a trick that must occasionally be employed.

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. 678) 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. 678) 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 whilst printing.

wxDC::StartPage

bool StartPage()

Starts a document page (only relevant when outputting to a printer).

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. 299) 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. 1061).

Derived from

wxClientBase
wxObject (p. 746)

Include files

<wx/dde.h>

See also

wxDDEServer (p. 303), *wxDDEConnection* (p. 299), *Interprocess communications overview* (p. 1428)

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. 299) member to return your own derived connection object.

wxDDEClient::OnMakeConnection

wxConnectionBase * OnMakeConnection()

The type of *wxDDEConnection* (p. 299) returned from a *wxDDEClient::MakeConnection* (p. 298) 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. 301). 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. 298) object, or by the acceptance of a connection by a *wxDDEServer* (p. 303) 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. 1063).

Derived from

wxConnectionBase
wxObject (p. 746)

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. 298), wxDDEServer (p. 303), *Interprocess communications overview* (p. 1428)

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. 304) and/or *wxDDEClient::OnMakeConnection* (p. 299) 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. 301) 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. 302) in that respect). Causes the server connection's *wxDDEConnection::OnExecute* (p. 302) 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. 301) 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. 303). 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. 302) 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. 302) 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. 302) 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. 302) 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. 1067).

Derived from

wxServerBase

Include files

<wx/dde.h>

See also

wxDDEClient (p. 298), *wxDDEConnection* (p. 299), *IPC overview* (p. 1428)

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 *wxWindows*, 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 in `setup.h`

Derived from

No parent class.

Include files

<wx/memory.h>

See also

Overview (p. 1357)

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::SetCheckPrevious (p. 308)

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. 308)

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. 651) functionality.

See also

wxDebugContext::SetLevel (p. 309)

wxDebugContext::GetStream

ostream& GetStream()

Returns the output stream associated with the debug context.

This is obsolete, replaced by *wxLog* (p. 651) functionality.

See also

wxDebugContext::SetStream (p. 309)

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. 651) 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. 651) functionality.

See also

wxDebugContext::SetStream (p. 309), *wxDebugContext::GetStream* (p. 306)

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. 307)

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. 307)

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. 305)

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. 306)

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. 651) functionality.

See also

`wxDebugContext::GetLevel` (p. 306)

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. 651) 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. 651) 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. 306), `wxDebugContext::HasStream` (p. 307)

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. 651) functionality.

Derived from

streambuf

Include files

<wx/memory.h>

Example

```
wxDebugStreamBuf streamBuf;  
ostream stream(&streamBuf);  
  
stream << "Hello world!" << endl;
```

See also

Overview (p. 1357)

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.

Derived from

wxPanel (p. 764)
wxWindow (p. 1184)
wxEvtHandler (p. 378)
wxObject (p. 746)

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 on other windows is still possible. You specify the type of dialog with the **wxDIALOG_MODAL** and **wxDIALOG_MODELESS** window styles.

A dialog may be loaded from a wxWindows resource file (extension `wxr`), which may itself be created by Dialog Editor. For details, see *The wxWindows resource system* (p. 1379), *wxWindows resource functions* (p. 1293) and the resource sample.

An application can define an *OnCloseWindow* (p. 1208) handler for the dialog to respond to system close events.

Window styles

wxDIALOG_MODAL	Specifies that the dialog box will be modal.
wxCAPTION	Puts a caption on the dialog box.
wxDEFAULT_DIALOG_STYLE	Equivalent to a combination of wxCAPTION , wxSYSTEM_MENU and wxTHICK_FRAME
wxRESIZE_BORDER	Display a resizable frame around the window.
wxSYSTEM_MENU	Display a system menu.
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.

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. 1371).

See also

wxDialog overview (p. 1373), *wxFrame* (p. 452), *Resources* (p. 9), *Validator overview* (p. 1374)

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 wxWindows, depending on platform.

size

The dialog size. A value of (-1, -1) indicates a default size, chosen by either the windowing system or wxWindows, depending on platform.

style

The window style. See *wxDialog* (p. 310).

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. 313)

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. 311) for details.

wxDialog::EndModal

void EndModal(*int* retCode)

Ends a modal dialog, passing a value to be returned from the *wxDialog::ShowModal* (p. 317) invocation.

Parameters

retCode

The value that should be returned by **ShowModal**.

See also

wxDialog::ShowModal (p. 317), *wxDialog::GetReturnCode* (p. 313),
wxDialog::SetReturnCode (p. 316)

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. 317) returns a code to the application.

See also

wxDialog::SetReturnCode (p. 316), *wxDialog::ShowModal* (p. 317), *wxDialog::EndModal* (p. 313)

wxDialog::GetTitle

wxString GetTitle() **const**

Returns the title of the dialog box.

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::OnCharHook

void OnCharHook(wxKeyEvent& *event*)

This member is called to allow the window to intercept keyboard events before they are processed by child windows.

For more information, see `wxWindow::OnCharHook` (p. 1206)

Remarks

`wxDialog` implements this handler to fake a cancel command if the escape key has been pressed. This will dismiss the dialog.

wxDialog::OnApply

void OnApply(wxCommandEvent& event)

The default handler for the `wxID_APPLY` identifier.

Remarks

This function calls `wxWindow::Validate` (p. 1233) and `wxWindow::TransferDataToWindow` (p. 1233).

See also

`wxDialog::OnOK` (p. 315), `wxDialog::OnCancel` (p. 315)

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. 315), `wxDialog::OnApply` (p. 315)

wxDialog::OnOK

void OnOK(wxCommandEvent& event)

The default handler for the `wxID_OK` identifier.

Remarks

The function calls `wxWindow::Validate` (p. 1233), then `wxWindow::TransferDataFromWindow` (p. 1232). 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. 315), *wxDialog::OnApply* (p. 315)

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 *wxWindow::OnSysColourChanged* (p. 1217) to propagate the notification to child windows and controls.

See also

wxSysColourChangedEvent (p. 1034)

wxDialog::SetModal

void SetModal(const bool flag)

NB: This function is deprecated and doesn't work for all ports, just use *ShowModal* (p. 317) 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. 317) returns a code to the application. The function *wxDialog::EndModal* (p. 313) calls **SetReturnCode**.

See also

wxDialog::GetReturnCode (p. 313), *wxDialog::ShowModal* (p. 317), *wxDialog::EndModal* (p. 313)

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. 313).

wxDialog::ShowModal

int ShowModal()

Shows a modal dialog. Program flow does not return until the dialog has been dismissed with *wxDialog::EndModal* (p. 313).

Return value

The return value is the value set with *wxDialog::SetReturnCode* (p. 316).

See also

wxDialog::EndModal (p. 313), *wxDialog::GetReturnCode* (p. 313),
wxDialog::SetReturnCode (p. 316)

wxDialUpEvent

This is the event class for the dialup events sent by *wxDialUpManager* (p. 319).

Derived from

wxEvent (p. 375)
wxObject (p. 746)

Include files

<wx/dialup.h>

wxDialUpEvent::wxDialUpEvent

wxDialUpEvent(bool *isConnected*, bool *isOwnEvent*)

Constructor is only used by *wxDialUpManager* (p. 319).

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 external 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 `wxWindows` classes in that there is at most one instance of this class in the program accessed via `wxDialUpManager::Create()` (p. 320) 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. 318) 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. 1322)
wxDialUpEvent (p. 318)

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. 320) 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. 323) 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. 320)

wxDialUpManager::CancelDialing

bool CancelDialing()

Cancel dialing the number initiated with *Dial* (p. 320) with *async* parameter equal to `TRUE`.

Note that this won't result in `DISCONNECTED` event being sent.

See also

IsDialing (p. 321)

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. 322)) 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. 322)

wxDialUpManager::EnableAutoCheckOnlineStatus

bool EnableAutoCheckOnlineStatus(size_t nSeconds = 60)

Enable automatical 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

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. 320)

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.

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 filenames 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::Exists

static bool Exists(const wxString& dir)

Test for existence of a directory with the given name

wxDir::wxDir

wxDir()

Default constructor, use *Open()* (p. 325) afterwards.

wxDir(const wxString& dir)

Opens the directory for enumeration, use *IsOpened()* (p. 325) 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::Open

bool Open(const wxString& dir)

Open the directory for enumerating, returns TRUE on success or FALSE if an error occurred.

wxDir::IsOpened

bool IsOpened() const

Returns TRUE if the directory was successfully opened by a previous call to *Open* (p. 325).

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::GetNext

bool GetNext(wxString* filename) const

Continue enumerating files satisfying the criteria specified by the last call to *GetFirst* (p. 325).

wxDirDialog

This class represents the directory chooser dialog.

Derived from

wxDialog (p. 310)

wxWindow (p. 1184)

wxEvtHandler (p. 378)

wxObject (p. 746)

Include files

<wx/dirdlg.h>

See also

wxDirDialog overview (p. 1401), *wxFileDialog* (p. 407)

wxDirDialog::wxDirDialog

wxDirDialog(*wxWindow** parent, **const wxString&** message = "Choose a directory", **const wxString&** defaultPath = "", **long** style = 0, **const wxPoint&** pos = *wxDefaultPosition*)

Constructor. Use *wxDirDialog::ShowModal* (p. 327) 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. Not implemented.

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 `wxOK_CANCEL` otherwise.

wxDllLoader

`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.

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 `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. 329)

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. 330).

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. 329) 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 *wxWindows*, and cooperates with the *wxView* (p. 1179), *wxDocument* (p. 351), *wxDocManager* (p. 332) and *wxDocTemplate* (p. 345) classes.

See the example application in `samples/docview`.

Derived from

wxFrame (p. 452)

wxWindow (p. 1184)

wxEvtHandler (p. 378)

wxObject (p. 746)

Include files

<wx/docview.h>

See also

Document/view overview (p. 1402), *wxFrame* (p. 452)

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 `wxWindows`, and cooperates with the `wxView` (p. 1179), `wxDocument` (p. 351) and `wxDocTemplate` (p. 345) classes.

Derived from

`wxEvtHandler` (p. 378)

`wxObject` (p. 746)

Include files

<wx/docview.h>

See also

`wxDocManager` overview (p. 1405), `wxDocument` (p. 351), `wxView` (p. 1179), `wxDocTemplate` (p. 345), `wxFileHistory` (p. 413)

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. 413), 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 mnTemplates

A list of all document templates.

wxDocManager::wxDocManager

void 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. 338) 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, bool deleting)

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::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. 162)

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. 162)

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::GetNoHistoryFiles

int GetNoHistoryFiles()

Returns the number of files currently stored in the file history.

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.

wxDocManager::OnCreateFileHistory

wxFileHistory * OnCreateFileHistory()

A hook to allow a derived class to create a different type of file history. Called from *Initialize* (p. 338).

wxDocManager::OnFileClose

void OnFileClose()

Closes and deletes the currently active document.

wxDocManager::OnFileNew

void OnFileNew()

Creates a document from a list of templates (if more than one template).

wxDocManager::OnFileOpen

void OnFileOpen()

Creates a new document and reads in the selected file.

wxDocManager::OnFileSave

void OnFileSave()

Saves the current document by calling `wxDocument::Save` for the current document.

wxDocManager::OnFileSaveAs

void OnFileSaveAs()

Calls `wxDocument::SaveAs` for the current document.

wxDocManager::OnMenuCommand

void OnMenuCommand(int *cmd*)

Processes menu commands routed from child or parent frames. This deals with the following predefined menu item identifiers:

- `wxID_OPEN` Creates a new document and opens a file into it.
- `wxID_CLOSE` Closes the current document.
- `wxID_NEW` Creates a new document.
- `wxID_SAVE` Saves the document.
- `wxID_SAVE_AS` Saves the document into a specified filename.

Unrecognized commands are routed to the currently active `wxView`'s `OnMenuCommand`.

wxDocManager::RemoveDocument

void RemoveDocument(wxDocument **doc*)

Removes the document from the list of documents.

wxDocManager::SelectDocumentPath

wxDocTemplate * SelectDocumentPath(wxDocTemplate **templates, int noTemplates, const wxString& path, const wxString& bufSize, 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.

wxDocManager::SelectDocumentType

wxDocTemplate * SelectDocumentType(wxDocTemplate **templates, int noTemplates)

Returns a document template by asking the user (if there is more than one template). This function is used in wxDocManager::CreateDocument.

wxDocManager::SelectViewType

wxDocTemplate * SelectViewType(wxDocTemplate **templates, int noTemplates)

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 won't appear because the array of templates only contains those relevant to the document in question, and often there will only be one such.

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 `wxWindows`, and cooperates with the `wxView` (p. 1179), `wxDocument` (p. 351), `wxDocManager` (p. 332) and `wxDocTemplate` (p. 345) classes.

See the example application in `samples/docview`.

Derived from

`wxMDIChildFrame` (p. 666)
`wxFrame` (p. 452)
`wxWindow` (p. 1184)
`wxEvtHandler` (p. 378)
`wxObject` (p. 746)

Include files

<wx/docmdi.h>

See also

Document/view overview (p. 1402), `wxMDIChildFrame` (p. 666)

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. 1179), *wxDocument* (p. 351), *wxDocManager* (p. 332) and *wxDocTemplates* (p. 345) classes.

See the example application in `samples/docview`.

Derived from

wxMDIParentFrame (p. 671)

wxFrame (p. 452)

wxWindow (p. 1184)

wxEvtHandler (p. 378)

wxObject (p. 746)

Include files

<wx/docmdi.h>

See also

Document/view overview (p. 1402), *wxMDIParentFrame* (p. 671)

wxDocMDIParentFrame::wxDocMDIParentFrame

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.

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. 1179), `wxDocument` (p. 351), `wxDocManager` (p. 332) and `wxDocTemplates` (p. 345) classes.

See the example application in `samples/docview`.

Derived from

`wxFrame` (p. 452)
`wxWindow` (p. 1184)
`wxEvtHandler` (p. 378)
`wxObject` (p. 746)

Include files

<wx/docview.h>

See also

Document/view overview (p. 1402), *wxFrame* (p. 452)

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. 746)

Include files

<wx/docview.h>

See also

wxDocTemplate overview (p. 1405), *wxDocument* (p. 351), *wxView* (p. 1179)

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`.

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.

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::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 `wxWindows`, and cooperates with the `wxView` (p. 1179), `wxDocTemplate` (p. 345) and `wxDocManager` (p. 332) classes.

Derived from

`wxEvtHandler` (p. 378)

`wxObject` (p. 746)

Include files

<wx/docview.h>

See also

`wxDocument overview` (p. 1403), `wxView` (p. 1179), `wxDocTemplate` (p. 345), `wxDocManager` (p. 332)

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 implementation 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. 158).

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. 352).

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. 355)

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**.

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. 354)

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. 355).

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 *wxWindows* 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. 355).

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. 158).

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 `wxWindows` 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. 158).

wxDocument::SetDocumentName

void SetDocumentName(const wxString& name)

Sets the document type name for this document. See the comment for *documentTypeName* (p. 352).

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)

Updates all views. If *sender* is non-NULL, does not update this view.

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. 362) and `wxDragImage::GetImageRect` (p. 362).

Please see `samples/dragimag` for an example.

Derived from

wxObject (p. 746)

Include files

<wx/dragimag.h>
<wx/generic/dragimgg.h>

wxDragImage::wxDragImage

wxDragImage()

Default constructor.

wxDragImage(const wxBitmap& image, const wxCursor& cursor = wxNullCursor, const wxPoint& hotspot = wxPoint(0, 0))

Constructs a drag image from a bitmap and optional cursor.

wxDragImage(const wxIcon& image, const wxCursor& cursor = wxNullCursor, const wxPoint& hotspot = 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& hotspot = 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& hotspot = 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. 362) and `wxDragImage::GetImageRect` (p. 362).

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*
Position of the hotspot within the new image.
- 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. 363) and `wxDragImage::Move` (p. 363) to show the image on the screen.

Call `wxDragImage::EndDrag` (p. 362) 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. 362).

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. 362)) 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. 363) and `wxDragImage::Show` (p. 363).

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. 363) has been called previously (for example at the start of the drag).

`pt` is the position in window coordinates (or screen coordinates if no window was specified to `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.

Important note: this is a separate implementation to the more general drag and drop implementation documented *here* (p. 1420). It uses the older, Windows message-based approach of dropping files.

Derived from

`wxEvent` (p. 375)

`wxObject` (p. 746)

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

`wxWindow::OnDropFiles` (p. 1208), *Event handling overview* (p. 1364)

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. 1420) and *wxDataObject overview* (p. 1422) 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
    wxDragCancel     // the operation was cancelled by user (not an
error)
};
```

See also

`wxDropTarget` (p. 368), `wxTextDropTarget` (p. 1090), `wxFileDropTarget` (p. 412)

wxDropSource::wxDropSource

wxDropSource(`wxWindow* win = NULL`, **const wxIconOrCursor& iconCopy = wxNullIconOrCursor**, **const wxIconOrCursor& iconCopy = wxNullIconOrCursor**, **const wxIconOrCursor& iconNone = wxNullIconOrCursor**)

wxDropSource(`wxDataObject& data`, `wxWindow* win = NULL`, **const wxIconOrCursor& iconCopy = wxNullIconOrCursor**, **const wxIconOrCursor& iconCopy = wxNullIconOrCursor**, **const wxIconOrCursor& iconNone = wxNullIconOrCursor**)

The constructors for `wxDataObject`.

If you use the constructor without `data` parameter you must call `SetData` (p. 367) 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. 1268) 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. 196) associated with the drop source. This will not delete any previously associated data.

wxDropSource::DoDragDrop

virtual wxDragResult DoDragDrop(bool allowMove = FALSE)

Do it (call this in response to a mouse button press, for example).

If **allowMove** is FALSE, data can only be copied.

wxDropSource::GiveFeedback

virtual bool GiveFeedback(wxDragResult effect, bool scrolling)

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* 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.

wxDropTarget

This class represents a target for a drag and drop operation. A *wxDataObject* (p. 196) 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. 370), possibly many times *OnDragOver* (p. 370), *OnDrop* (p. 369) and finally *OnData* (p. 369).

See *Drag and drop overview* (p. 1420) and *wxDataObject overview* (p. 1422) 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
    wxDragCancel       // the operation was cancelled by user (not an
error)
};
```

See also

wxDropSource (p. 365), *wxTextDropTarget* (p. 1090), *wxFileDropTarget* (p. 412), *wxDataFormat* (p. 194), *wxDataObject* (p. 196)

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. 369). By default, this method copies the data from the drop source to the *wxDataObject* (p. 196) associated with this drop target, calling its *wxDataObject::SetData* (p. 199) method.

wxDropTarget::OnData

virtual wxDragResult OnData(wxCoord x, wxCoord y, wxDragResult def)

Called after *OnDrop* (p. 369) returns TRUE. By default this will usually *GetData* (p. 369) 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. 370).

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. 196) associated with the drop target and deletes any previously associated data object.

wxEncodingConverter

This class is capable of converting strings between any two 8-bit encodings/charsets. It can also convert from/to Unicode (but only if you compiled wxWindows with wxUSE_WCHAR_T set to 1).

Derived from

wxObject (p. 746)

Include files

<wx/encconv.h>

See also

wxFontMapper (p. 448), *wxMBConv* (p. 661), *Writing non-English applications* (p. 1347)

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. 372) 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 wxWindows or if input or output encoding is not supported.)

wxEncodingConverter::Convert

```
wxString Convert(const wxString& input)
void Convert(const wxChar* input, wxChar* output)
void Convert(wxChar* str)
void Convert(const char* input, wxChar* output)
```

Convert input string according to settings passed to *Init* (p. 371). Note that you must call *Init* before using *Convert*!

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 them 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. 372) 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. 372), 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.

Derived from

wxEvt (p. 375)

wxObject (p. 746)

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

If the **m_DC** member is non-NULL, draw into this device context.

See also

wxWindow::OnEraseBackground (p. 1209), *Event handling overview* (p. 1364)

wxEraseEvent::wxEraseEvent

wxEraseEvent(int id = 0, wxDC* dc = NULL)

Constructor.

wxEraseEvent::m_dc

wxDC* m_dc

The device context associated with the erase event (may be NULL).

wxEraseEvent::GetDC

wxDC* GetDC() const

Returns the device context to draw into. If this is non-NULL, you should draw into it to perform the erase operation.

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).

Derived from

wxObject (p. 746)

Include files

<wx/event.h>

See also

wxCommandEvent (p. 152), *wxMouseEvent* (p. 721)

wxEvent::wxEvent

wxEvent(int *id* = 0)

Constructor. Should not need to be used directly by an application.

wxEvent::m_eventHandle

char* m_eventHandle

Handle of an underlying windowing system event handle, such as XEvent. Not guaranteed to be instantiated.

wxEvent::m_eventObject

wxObject* m_eventObject

The object (usually a window) that the event was generated from, or should be sent to.

wxEvent::m_eventType

WXTYPE m_eventType

The type of the event, such as wxEVENT_TYPE_BUTTON_COMMAND.

wxEvent::m_id

int m_id

Identifier for the window.

wxEvent::m_skipped

bool m_skipped

Set to TRUE by **Skip** if this event should be skipped.

wxEvent::m_timeStamp

long m_timeStamp

Timestamp for this event.

wxEvent::GetEventObject

wxObject* GetEventObject()

Returns the object 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()

Returns the identifier associated with this event, such as a button command id.

wxEvt::GetObjectType

WXTYPE GetObjectType()

Returns the type of the object associated with the event, such as wxTYPE_BUTTON.

wxEvt::GetSkipped

bool GetSkipped()

Returns TRUE if the event handler should be skipped, FALSE otherwise.

wxEvt::GetTimestamp

long GetTimestamp()

Gets the timestamp for the event.

wxEvt::SetEventObject

void SetEventObject(wxObject* object)

Sets the originating object.

wxEvt::SetEventType

void SetEventType(WXTYPE typ)

Sets the event type.

wxEvt::SetId

void SetId(int id)

Sets the identifier associated with this event, such as a button command id.

wxEvt::SetTimestamp

void SetTimestamp(long timeStamp)

Sets the timestamp for the event.

Sets the originating object.

wxEvtHandler::Skip

void Skip(bool *skip* = *TRUE*)

Called by an event handler to tell the event system that the event handler should be skipped, and the next valid handler used instead.

wxEvtHandler

A class that can handle events from the windowing system. `wxWindow` (and therefore all window classes) are derived from this class.

Derived from

`wxObject` (p. 746)

Include files

<wx/event.h>

See also

Event handling overview (p. 1364)

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

virtual void AddPendingEvent(wxEvent& event)

Adds an event to be processed later. The function will return immediately and the event will get processed in idle time using the *wxEvtHandler::ProcessEvent* (p. 382) method.

Parameters

event

Event to add to process queue.

Remarks

Note that this requires that the event has a fully implemented Clone() method so that the event can be duplicated and stored until it gets processed later. Not all events in wxWindows currently have a fully implemented Clone() method, so you may have to look at the source to verify this.

This methods automatically wakes up idle handling even if the underlying window system is currently idle anyway and thus would not send any idle events. (Waking up the idle handling is done calling *::wxWakeUpIdle* (p. 1285).)

This is also the method to call for inter-thread communication. In a multi-threaded program, you will often have to inform the main GUI thread about the status of other working threads and this has to be done using this method - which also means that this method is thread safe by means of using critical sections where needed.

Furthermore, it may be noted that some ports of wxWindows will probably move to using this method more and more in preference over calling ProcessEvent() directly so as to avoid problems with reentrant code.

wxEvtHandler::Connect

void Connect(int id, wxEventType eventType, wxObjectEventFunction function, wxObject* userData = NULL)

void Connect(int id, int lastId, wxEventType eventType, wxObjectEventFunction function, wxObject* userData = 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 'dynamic' sample for usage.

Parameters

id

The identifier (or first of the identifier range) to be associated with the event handler function.

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.

userData

Data to be associated with the event table entry.

Example

```
frame->Connect( wxID_EXIT,
               wxEVT_COMMAND_MENU_SELECTED,
               (wxObjectEventFunction) (wxCommandEventFunction)
               MyFrame::OnQuit );
```

wxEvtHandler::Disconnect

```
bool Disconnect(int id, wxEvtType eventType = wxEVT_NULL,
wxObjectEventFunction function = NULL, wxObject* userData = NULL)
```

```
bool Disconnect(int id, int lastId = -1, wxEvtType eventType = wxEVT_NULL,
wxObjectEventFunction function = NULL, wxObject* userData = 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. 379) 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.

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. 384)

wxEvtHandler::GetEvtHandlerEnabled

bool GetEvtHandlerEnabled()

Returns TRUE if the event handler is enabled, FALSE otherwise.

See also

wxEvtHandler::SetEvtHandlerEnabled (p. 384)

wxEvtHandler::GetNextHandler

wxEvtHandler* GetNextHandler()

Gets the pointer to the next handler in the chain.

See also

wxEvtHandler::SetNextHandler (p. 384), *wxEvtHandler::GetPreviousHandler* (p. 381), *wxEvtHandler::SetPreviousHandler* (p. 385), *wxWindow::PushEventHandler* (p. 1218), *wxWindow::PopEventHandler* (p. 1217)

wxEvtHandler::GetPreviousHandler

wxEvtHandler* GetPreviousHandler()

Gets the pointer to the previous handler in the chain.

See also

wxEvtHandler::SetPreviousHandler (p. 385), *wxEvtHandler::GetNextHandler* (p. 381), *wxEvtHandler::SetNextHandler* (p. 384), *wxWindow::PushEventHandler* (p. 1218), *wxWindow::PopEventHandler* (p. 1217)

wxEvtHandler::ProcessEvent

virtual bool ProcessEvent(wxEvt& 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 *wxEvt::Skip* (p. 378).

Remarks

Normally, your application would not call this function: it is called in the wxWindows 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 wxWindows. 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. 384)) the function skips to step (6).
2. If the object is a *wxWindow*, **ProcessEvent** is recursively called on the window's *wxValidator* (p. 1166). 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. 383)

wxEvtHandler::SearchEventTable

bool SearchEventTable(wxEvtTable& 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 *wxEvt::Skip* (p. 378).

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 *wxEvt::Skip* (p. 378), this function will fail, and searching will continue.

See also

wxEvtHandler::ProcessEvent (p. 382)

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.

See also

wxEvtHandler::GetClientData (p. 381)

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. 381)

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. 381), *wxEvtHandler::SetPreviousHandler* (p. 385), *wxEvtHandler::GetPreviousHandler* (p. 381), *wxWindow::PushEventHandler* (p. 1218), *wxWindow::PopEventHandler* (p. 1217)

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. 381), *wxEvtHandler::SetNextHandler* (p. 384), *wxEvtHandler::GetNextHandler* (p. 381), *wxWindow::PushEventHandler* (p. 1218), *wxWindow::PopEventHandler* (p. 1217)

wxExpr

The **wxExpr** class is the building brick of expressions similar to Prolog clauses, or objects. It can represent an expression of type long integer, float, string, word, or list, and lists can be nested.

Derived from

None

Include files

<wx/wxexpr.h>

See also

wxExpr overview (p. 1359), *wxExprDatabase* (p. 392)

wxExpr::wxExpr

wxExpr(const wxString& functor)

Construct a new clause with this form, supplying the functor name. A clause is an object that will appear in the data file, with a list of attribute/value pairs.

wxExpr(wxExprType type, const wxString& wordOrString = "")

Construct a new empty list, or a word (will be output with no quotes), or a string, depending on the value of *type*.

type can be **wxExprList**, **wxExprWord**, or **wxExprString**. If *type* is **wxExprList**, the value of *wordOrString* will be ignored.

wxExpr(long value)

Construct an integer expression.

wxExpr(float value)

Construct a floating point expression.

wxExpr(wxList* value)

Construct a list expression. The list's nodes' data should themselves be **wxExprs**.

wxExpr no longer uses the **wxList** internally, so this constructor turns the list into its internal format (assuming a non-nested list) and then deletes the supplied list.

wxExpr::~~wxExpr

~wxExpr()

Destructor.

wxExpr::AddAttributeValue

Use these on clauses ONLY. Note that the functions for adding strings and words must be differentiated by function name which is why they are missing from this group (see *wxExpr::AddAttributeValueString* (p. 387) and *wxExpr::AddAttributeValueWord* (p. 387)).

void AddAttributeValue(const wxString& attribute, float value)

Adds an attribute and floating point value pair to the clause.

void AddAttributeValue(const wxString& attribute, long value)

Adds an attribute and long integer value pair to the clause.

void AddAttributeValue(const wxString& attribute, wxList* value)

Adds an attribute and list value pair to the clause, converting the list into internal form and then deleting **value**. Note that the list should not contain nested lists (except if in internal **wxExpr** form.)

void AddAttributeValue(const wxString& attribute, wxExpr* value)

Adds an attribute and wxExpr value pair to the clause. Do not delete *value* once this function has been called.

wxExpr::AddAttributeValueString

void AddAttributeValueString(const wxString& attribute, const wxString& value)

Adds an attribute and string value pair to the clause.

wxExpr::AddAttributeValueStringList

void AddAttributeValueStringList(const wxString& attribute, wxList* value)

Adds an attribute and string list value pair to the clause.

Note that the list passed to this function is a list of strings, NOT a list of **wxExprs**; it gets turned into a list of **wxExprs** automatically. This is a convenience function, since lists of strings are often manipulated in C++.

wxExpr::AddAttributeValueWord

void AddAttributeValueWord(const wxString& attribute, const wxString& value)

Adds an attribute and word value pair to the clause.

wxExpr::Append

void Append(wxExpr* value)

Append the **value** to the end of the list. 'this' must be a list.

wxExpr::Arg

wxExpr* Arg(wxExprType type, int n) const

Get *n*th arg of the given clause (starting from 1). NULL is returned if the expression is not a clause, or *n* is invalid, or the given type does not match the actual type. See also *wxExpr::Nth* (p. 390).

wxExpr::Insert

void Insert(wxExpr* value)

Insert the **value** at the start of the list. 'this' must be a list.

wxExpr::GetAttributeValue

These functions are the easiest way to retrieve attribute values, by passing a pointer to variable. If the attribute is present, the variable will be filled with the appropriate value. If not, the existing value is left alone. This style of retrieving attributes makes it easy to set variables to default values before calling these functions; no code is necessary to check whether the attribute is present or not.

bool GetAttributeValue(const wxString& attribute, wxString& value) const

Retrieve a string (or word) value.

bool GetAttributeValue(const wxString& attribute, float& value) const

Retrieve a floating point value.

bool GetAttributeValue(const wxString& attribute, int& value) const

Retrieve an integer value.

bool GetAttributeValue(const wxString& attribute, long& value) const

Retrieve a long integer value.

bool GetAttributeValue(const wxString& attribute, wxExpr value) const**

Retrieve a wxExpr pointer.

wxExpr::GetAttributeValueStringList

void GetAttributeValueStringList(const wxString& attribute, wxList* value) const

Use this on clauses ONLY. See above for comments on this style of attribute value retrieval. This function expects to receive a pointer to a new list (created by the calling application); it will append strings to the list if the attribute is present in the clause.

wxExpr::AttributeValue

wxExpr* AttributeValue(const wxString& word) const

Use this on clauses ONLY. Searches the clause for an attribute matching *word*, and returns the value associated with it.

wxExpr::Copy

wxExpr* Copy() const

Recursively copies the expression, allocating new storage space.

wxExpr::DeleteAttributeValue

void DeleteAttributeValue(const wxString& attribute)

Use this on clauses only. Deletes the attribute and its value (if any) from the clause.

wxExpr::Functor

wxString Functor() const

Use this on clauses only. Returns the clause's functor (object name).

wxExpr::GetClientData

wxObject* GetClientData() const

Retrieve arbitrary data stored with this clause. This can be useful when reading in data for storing a pointer to the C++ object, so when another clause makes a reference to this clause, its C++ object can be retrieved. See *wxExpr::SetClientData* (p. 390).

wxExpr::GetFirst

wxExpr* GetFirst() const

If this is a list expression (or clause), gets the first element in the list.

See also *wxExpr::GetLast* (p. 389), *wxExpr::GetNext* (p. 390), *wxExpr::Nth* (p. 390).

wxExpr::GetLast

wxExpr* GetLast() const

If this is a list expression (or clause), gets the last element in the list.

See also *wxExpr::GetFirst* (p. 389), *wxExpr::GetNext* (p. 390), *wxExpr::Nth* (p. 390).

wxExpr::GetNext

wxExpr* GetNext() const

If this is a node in a list (any *wxExpr* may be a node in a list), gets the next element in the list.

See also *wxExpr::GetFirst* (p. 389), *wxExpr::GetLast* (p. 389), *wxExpr::Nth* (p. 390).

wxExpr::IntegerValue

long IntegerValue() const

Returns the integer value of the expression.

wxExpr::Nth

wxExpr* Nth(int *n*) const

Get *n*th arg of the given list expression (starting from 0). NULL is returned if the expression is not a list expression, or *n* is invalid. See also *wxExpr::Arg* (p. 387).

Normally, you would use attribute-value pairs to add and retrieve data from objects (clauses) in a data file. However, if the data gets complex, you may need to store attribute values as lists, and pick them apart yourself.

wxExpr::RealValue

float RealValue() const

Returns the floating point value of the expression.

wxExpr::SetClientData

void SetClientData(wxObject **data*)

Associate arbitrary data with this clause. This can be useful when reading in data for storing a pointer to the C++ object, so when another clause makes a reference to this

clause, its C++ object can be retrieved. See *wxExpr::GetClientData* (p. 389).

wxExpr::StringValue

wxString StringValue() const

Returns the string value of the expression.

wxExpr::Type

wxExprType Type() const

Returns the type of the expression. **wxExprType** is defined as follows:

```
typedef enum {
    wxExprNull,
    wxExprInteger,
    wxExprReal,
    wxExprWord,
    wxExprString,
    wxExprList
} wxExprType;
```

wxExpr::WordValue

wxString WordValue() const

Returns the word value of the expression.

wxExpr::WriteClause

void WriteClause(FILE * stream)

Writes the clause to the given stream in Prolog format. Not normally needed, since the whole **wxExprDatabase** will usually be written at once. The format is: functor, open parenthesis, list of comma-separated expressions, close parenthesis, full stop.

wxExpr::WriteExpr

void WriteExpr(FILE * stream)

Writes the expression (not clause) to the given stream in Prolog format. Not normally needed, since the whole **wxExprDatabase** will usually be written at once. Lists are written in square bracketed, comma-delimited format.

Functions and macros

Below are miscellaneous functions and macros associated with `wxExpr` objects.

bool wxExprIsFunctor(wxExpr *expr, const wxString& functor)

Checks that the functor of *expr* is *functor*.

void wxExprCleanUp()

Cleans up the `wxExpr` system (YACC/LEX buffers) to avoid memory-checking warnings as the program exits.

```
#define wxMakeInteger(x) (new wxExpr((long)x))
#define wxMakeReal(x)   (new wxExpr((float)x))
#define wxMakeString(x) (new wxExpr(PrologString, x))
#define wxMakeWord(x)   (new wxExpr(PrologWord, x))
#define wxMake(x)       (new wxExpr(x))
```

Macros to help make `wxExpr` objects.

wxExprDatabase

The **wxExprDatabase** class represents a database, or list, of Prolog-like expressions. Instances of this class are used for reading, writing and creating data files.

Derived from

wxList (p. 615)

wxObject (p. 746)

See also

wxExpr overview (p. 1359), *wxExpr* (p. 385)

wxExprDatabase::wxExprDatabase

void wxExprDatabase(proioErrorHandler handler = 0)

Construct a new, unhashed database, with an optional error handler. The error handler must be a function returning a bool and taking an integer and a string argument. When an error occurs when reading or writing a database, this function is called. The error is given as the first argument (currently one of `WXEXPR_ERROR_GENERAL`, `WXEXPR_ERROR_SYNTAX`) and an error message is given as the second argument. If `FALSE` is returned by the error handler, processing of the `wxExpr` operation stops.

Another way of handling errors is simply to call `wxExprDatabase::GetErrorCount` (p. 394) after the operation, to check whether errors have occurred, instead of installing an error handler. If the error count is more than zero, `wxExprDatabase::Write` (p. 395) and `wxExprDatabase::Read` (p. 395) will return `FALSE` to the application.

For example:

```
bool myErrorHandler(int err, chat *msg)
{
    if (err == WXEXPR_ERROR_SYNTAX)
    {
        wxMessageBox(msg, "Syntax error");
    }
    return FALSE;
}

wxExprDatabase database(myErrorHandler);
```

wxExprDatabase(wxExprType type, const wxString&attribute, int size = 500, proioErrorHandler handler = 0)

Construct a new database hashed on a combination of the clause functor and a named attribute (often an integer identification).

See above for an explanation of the error handler.

wxExprDatabase::~wxExprDatabase

~wxExprDatabase()

Delete the database and contents.

wxExprDatabase::Append

void Append(wxExpr* clause)

Append a clause to the end of the database. If the database is hashing, the functor and a user-specified attribute will be hashed upon, giving the option of random access in addition to linear traversal of the database.

wxExprDatabase::BeginFind

void BeginFind()

Reset the current position to the start of the database. Subsequent `wxExprDatabase::FindClause` (p. 394) calls will move the pointer.

wxExprDatabase::ClearDatabase

void ClearDatabase()

Clears the contents of the database.

wxExprDatabase::FindClause

Various ways of retrieving clauses from the database. A return value of NULL indicates no (more) clauses matching the given criteria. Calling the functions repeatedly retrieves more matching clauses, if any.

wxExpr* FindClause(long *id*)

Find a clause based on the special "id" attribute.

wxExpr* FindClause(const wxString& *attribute*, const wxString& *value*)

Find a clause which has the given attribute set to the given string or word value.

wxExpr* FindClause(const wxString& *attribute*, long *value*)

Find a clause which has the given attribute set to the given integer value.

wxExpr* FindClause(const wxString& *attribute*, float *value*)

Find a clause which has the given attribute set to the given floating point value.

wxExprDatabase::FindClauseByFunctor

wxExpr* FindClauseByFunctor(const wxString& *functor*)

Find the next clause with the specified functor.

wxExprDatabase::GetErrorCount

int GetErrorCount() const

Returns the number of errors encountered during the last read or write operation.

wxExprDatabase::HashFind

wxExpr* HashFind(const wxString& *functor*, long *value*) const

Finds the clause with the given functor and with the attribute specified in the database constructor having the given integer value.

For example,

```
// Hash on a combination of functor and integer "id" attribute when
reading in
wxExprDatabase db(wxExprInteger, "id");

// Read it in
db.ReadProlog("data");

// Retrieve a clause with specified functor and id
wxExpr *clause = db.HashFind("node", 24);
```

This would retrieve a clause which is written: `node(id = 24, ...,)`.

wxExpr* HashFind(const wxString& functor, const wxString& value)

Finds the clause with the given functor and with the attribute specified in the database constructor having the given string value.

wxExprDatabase::Read

bool Read(const wxString& filename)

Reads in the given file, returning TRUE if successful.

wxExprDatabase::ReadFromString

bool ReadFromString(const wxString& buffer)

Reads a Prolog database from the given string buffer, returning TRUE if successful.

wxExprDatabase::Write

bool Write(FILE *stream)

bool Write(const wxString& filename)

Writes the database as a Prolog-format file.

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. 402) for a wrapper around `FILE` structure.

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. 398) and `wxFile::Open` (p. 400).

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. 398) function.

wxFile::read	Open file for reading or test if it can be opened for reading with <code>Access()</code>
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 <code>Access()</code>
wxFile::read_write	Open file for reading and writing; can not be used with <code>Access()</code>
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 <code>Access()</code> . This is the same as wxFile::write if the file doesn't exist.

Other constants defined elsewhere but used by `wxFile` functions are `wxInvalidOffset`

which represents an invalid value of type *off_t* and is returned by functions returning *off_t* on error and the seek mode constants used with *Seek()* (p. 400):

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. 399) 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. 398) for the list of predefined descriptors)

wxFile::~~wxFile

~wxFile()

Destructor will close the file.

NB: it is not virtual so you should 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. 399) 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. 402) is different as *wxFFile::Eof* (p. 404) 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. 400) 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::IsOpened

bool IsOpened() const

Returns TRUE if the file has been opened.

wxFile::Length

off_t 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

off_t **Read**(void* *buffer*, **off_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

off_t **Seek**(**off_t** *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

off_t SeekEnd(*off_t 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

off_t 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*, **off_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

the number of bytes actually written

wxFile::Write

bool Write(const wxString& s)

Writes the contents of the string to the file, returns TRUE on success.

wxFile

wxFile 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)

wxFile::wxFile

wxFile()

Default constructor.

wxFile(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. 404) to check that it didn't fail.

wxFile(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.

fp

An existing file descriptor, such as `stderr`.

wxFile::~wxFile

~wxFile()

Destructor will close the file.

NB: it is not virtual so you should *not* derive from `wxFile`!

wxFile::Attach

void Attach(FILE* fp)

Attaches an existing file pointer to the `wxFile` object.

The descriptor should be already opened and it will be closed by `wxFile` object.

wxFile::Close

bool Close()

Closes the file and returns `TRUE` on success.

wxFile::Detach

void Detach()

Get back a file pointer from `wxFile` object - the caller is responsible for closing the file if this descriptor is opened. `IsOpened()` (p. 404) will return `FALSE` after call to `Detach()`.

wxFile::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. 395) is different as *wxFile::Eof* (p. 399) will return TRUE here as soon as the last byte of the file has been read.

wxFFile::Flush

bool Flush()

Flushes the file and returns TRUE on success.

wxFFile::IsOpened

bool IsOpened() const

Returns TRUE if the file has been opened.

wxFFile::Length

size_t 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.

wxFile::Read

size_t Read(void* *buffer*, off_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.

wxFile::Seek

bool Seek(long *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**.

wxFile::SeekEnd

bool SeekEnd(long *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.

wxFile::Tell

size_t Tell() const

Returns the current position.

wxFile::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.

wxFile::Write

bool Write(const wxString& *s*)

Writes the contents of the string to the file, returns TRUE on success.

wxFileDataObject

wxFileDataObject is a specialization of *wxDataObject* (p. 196) 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 wxWindows application is not currently supported.

Virtual functions to override

None.

Derived from

wxDataObjectSimple (p. 201)
wxDataObject (p. 196)

Include files

<wx/dataobj.h>

See also

wxDataObject (p. 196), *wxDataObjectSimple* (p. 201), *wxTextDataObject* (p. 1083),
wxBitmapDataObject (p. 75), *wxDataObject* (p. 196)

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. 44) of file names.

wxFileDialog

This class represents the file chooser dialog.

Derived from

wxDialog (p. 310)
wxWindow (p. 1184)
wxEvtHandler (p. 378)
wxObject (p. 746)

Include files

<wx/filedlg.h>

See also

wxFileDialog overview (p. 1400), *wxFileSelector* (p. 1257)

Remarks

Pops up a file selector box. In Windows, 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 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 files (*.bmp)|*.bmp|GIF files (*.gif)|*.gif"
```

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. 411) 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 `"*. *"`.

style

A dialog style. A bitlist of:

wxOPEN	This is an open dialog.
wxSAVE	This is a save dialog.
wxHIDE_READONLY	Hide read-only files.
wxOVERWRITE_PROMPT	Prompt for a conformation if a file will be overridden.
wxMULTIPLE	For open dialog only: allows selecting multiple files

pos

Dialog position. Not implemented.

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. 409) for the others.

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. 410) 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. Windows only.

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. 408) for details.

wxFileDialog::SetWildcard

void SetWildcard(const wxString& *wildCard*)

Sets the wildcard, which in Windows can contain multiple file types.

wxFileDialog::ShowModal

int ShowModal()

Shows the dialog, returning `wxID_OK` if the user pressed OK, and `wxID_CANCEL` otherwise.

wxFileDropTarget

A drop target which accepts files (dragged from File Manager or Explorer).

Derived from

wxDropTarget (p. 368)

Include files

<wx/dnd.h>

See also

Drag and drop overview (p. 1420), *wxDropSource* (p. 365), *wxDropTarget* (p. 368), *wxTextDropTarget* (p. 1090)

wxFileDropTarget::wxFileDropTarget

wxFileDropTarget()

Constructor.

wxFileDropTarget::OnDrop

virtual bool OnDrop(long x, long y, const void *data, size_t size)

See *wxDropTarget::OnDrop* (p. 369). This function is implemented appropriately for files, and calls *wxFileDropTarget::OnDropFiles* (p. 412).

wxFileDropTarget::OnDropFiles

virtual bool OnDropFiles(long x, long y, size_t nFiles, const char * constfiles[])

Override this function to receive dropped files.

Parameters

x
The x coordinate of the mouse.

y
The y coordinate of the mouse.

nFiles
The number of files being dropped.

files
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. 746)

Include files

<wx/docview.h>

See also

wxFileHistory overview (p. 1407), *wxDocManager* (p. 332)

wxFileHistory::m_fileHistory

char m_fileHistory**

A character array of strings corresponding to the most recently opened files.

wxFileHistory::m_fileHistoryN

int m_fileHistoryN

The number of files stored in the history array.

wxFileHistory::m_fileMaxFiles

int 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(int maxFiles = 9)

Constructor. Pass the maximum number of files that should be stored and displayed.

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(int *index*) const

Returns the file at this index (zero-based).

wxFileHistory::GetMaxFiles

int GetMaxFiles() const

Returns the maximum number of files that can be stored.

wxFileHistory::GetNoHistoryFiles

int GetNoHistoryFiles() 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. 162)

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. 162)

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. 395) whereas *wxFFileInputStream* (p. 419) is based in the *wxFFile* (p. 402) class.

Note that *wxFile* (p. 395) and *wxFFile* (p. 402) differ in one aspect, namely when to report that the end of the file has been reached. This is documented in *wxFile::Eof* (p. 399) and *wxFFile::Eof* (p. 404) 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.

Derived from

wxInputStream (p. 592)

Include files

<wx/wfstream.h>

See also

wxBufferedInputStream (p. 92), *wxFileOutputStream* (p. 417), *wxFFileOutputStream* (p. 420)

wxFileInputStream::wxFileInputStream

wxFileInputStream(const wxString& filename)

Opens the specified file using its *filename* 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.

wxFileOutputStream

This class represents data written to a file. There are actually two such groups of classes: this one is based on *wxFile* (p. 395) whereas *wxFFileInputStream* (p. 419) is based in the *wxFFile* (p. 402) class.

Note that *wxFile* (p. 395) and *wxFFile* (p. 402) differ in one aspect, namely when to report that the end of the file has been reached. This is documented in *wxFile::Eof* (p. 399) and *wxFFile::Eof* (p. 404) 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.

Derived from

wxOutputStream (p. 751)

Include files

<wx/wfstream.h>

See also

wxBufferedOutputStream (p. 93), *wxFileInputStream* (p. 416), *wxFFileInputStream* (p. 419)

wxFileOutputStream::wxFileOutputStream

wxFileOutputStream(const wxString& *fileName*)

Creates a new file with *filename* 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. 417), *wxFileInputStream* (p. 416)

Include files

<wx/wfstream.h>

See also

wxStreamBuffer (p. 1000)

wxFileStream::wxFileStream

wxFileStream(const wxString& *iofileName*)

Initializes a new file stream in read-write mode using the specified *iofilename* name.

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. 402) whereas *wxFileInputStream* (p. 416) is based in the *wxFile* (p. 395) class.

Note that *wxFile* (p. 395) and *wxFFile* (p. 402) differ in one aspect, namely when to report that the end of the file has been reached. This is documented in *wxFile::Eof* (p. 399) and *wxFFile::Eof* (p. 404) 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.

Derived from

wxInputStream (p. 592)

Include files

<wx/wfstream.h>

See also

wxBufferedInputStream (p. 92), *wxFFileOutputStream* (p. 420), *wxFileOutputStream* (p. 417)

wxFFileInputStream::wxFFileInputStream

wxFFileInputStream(const wxString& *fileName*)

Opens the specified file using its *fileName* name in read-only 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*.

wxFileInputStream::~wxFileInputStream

~wxFileInputStream()

Destructor.

wxFileInputStream::Ok

bool Ok() const

Returns TRUE if the stream is initialized and ready.

wxFileOutputStream

This class represents data written to a file. There are actually two such groups of classes: this one is based on *wxFile* (p. 402) whereas *wxFileInputStream* (p. 419) is based in the *wxFile* (p. 395) class.

Note that *wxFile* (p. 395) and *wxFile* (p. 402) differ in one aspect, namely when to report that the end of the file has been reached. This is documented in *wxFile::Eof* (p. 399) and *wxFile::Eof* (p. 404) and the behaviour of the stream classes reflects this difference, i.e. *wxFileInputStream* will report `wxSTREAM_EOF` after having read the last byte whereas *wxFileInputStream* will report `wxSTREAM_EOF` after trying to read *past* the last byte.

Derived from

wxOutputStream (p. 751)

Include files

<wx/wfstream.h>

See also

wxBufferedOutputStream (p. 93), *wxFileInputStream* (p. 419), *wxFileInputStream* (p. 416)

wxFileOutputStream::~wxFileOutputStream

wxFFileOutputStream(const wxString& ofileName)

Creates a new file with *ofilename* name and initializes the stream in write-only 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. 420), *wxFFileInputStream* (p. 419)

Include files

<wx/wfstream.h>

See also

wxStreamBuffer (p. 1000)

wxFFileStream::wxFFileStream

wxFFileStream(const wxString& iofilename)

Initializes a new file stream in read-write mode using the specified *iofilename* name.

wxFilenameListValidator

This class validates a filename for a *property list view* (p. 834), allowing the user to edit it textually and also popping up a file selector in "detailed editing" mode.

See also

Validator classes (p. 1453)

wxFilenameListValidator::wxFilenameListValidator

```
void wxFilenameListValidator(wxString message = "Select a file", wxString wildcard = "*. *", long flags=0)
```

Constructor. Supply an optional message and wildcard.

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. 424) to provide access to user-defined virtual file systems.

Derived from

wxObject (p. 746)

Include files

<wx/filesys.h>

See Also

wxFileSystemHandler (p. 424), *wxFSFile* (p. 464), *Overview* (p. 1362)

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. 424) 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. 424) is relative to this path.

Caution! Unless *is_dir* is TRUE the *location* parameter is not directory name but the name of the file in this directory!! All these commands change 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 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. 423)).

wxFileSystem::FindFirst

wxString FindFirst(const wxString& wildcard, int flags = 0)

Works like *wxFindFirstFile* (p. 1248). Returns name of the first filename (withing 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 next filename that matches parameters passed to *FindFirst* (p. 424).

wxFileSystem::OpenFile

wxFSFile* OpenFile(const wxString& location)

Opens file and returns pointer to *wxFSFile* (p. 464) 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.

wxFileSystemHandler

Classes derived from *wxFileSystemHandler* are used to access virtual file systems. Its public interface consists of two methods: *CanOpen* (p. 425) and *OpenFile* (p. 427). 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. 1362) if you don't know how locations are constructed.

Also consult *list of available handlers* (p. 1362).

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. 746)

Include files

`<wx/filesys.h>`

See also

`wxFileSystem` (p. 422), `wxFSFile` (p. 464), `Overview` (p. 1362)

`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. 465) 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. 1248). Returns name of the first filename (withing 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. 425) returns TRUE.

wxFileSystemHandler::FindNext

virtual wxString FindNext()

Returns next filename that matches parameters passed to *FindFirst* (p. 424).

This method is only called if *CanOpen* (p. 425) 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. 716) and *wxMimeTypesManager::GetFileTypeFromExtension* (p. 715) 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. 713)

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. 430) like this:

```

wxString command;
if ( filetype->GetOpenCommand(&command,
                             MailMessageParameters("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. 713) 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. 429) 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. 715) function in the first place.

wxFileType::GetIcon

bool GetIcon(wxIcon* icon)

If the function returns TRUE, the icon associated with this file type will be created and assigned to the *icon* parameter.

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)

If the function returns TRUE, the string pointed to by *command* is filled with the command which must be executed (see *wxExecute* (p. 1273)) in order to open the file of the given type. The name of the file is retrieved from *MessageParameters* (p. 428) class.

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. 1273)) in order to print the file of the given type. The name of the file is retrieved from *MessageParameters* (p. 428) 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 specifier	with
%s	the full file name
%t	the MIME type
%{param}	the value of the parameter <i>param</i>

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.

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.

Derived from

wxGridSizer (p. 924)

wxSizer (p. 924)

wxObject (p. 746)

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.

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 unencrypt the data which are read from another stream and pass it to the requester.

Derived from

wxInputStream (p. 592)

wxStreamBase (p. 998)

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. 751)

wxStreamBase (p. 998)

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.

wxFocusEvent

A focus event is sent when a window's focus changes.

Derived from

wxEvent (p. 375)

wxObject (p. 746)

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

wxWindow::OnSetFocus (p. 1216), *wxWindow::OnKillFocus* (p. 1211), *Event handling overview* (p. 1364)

wxFocusEvent::wxFocusEvent

wxFocusEvent(WXTYPE *eventType* = 0, int *id* = 0)

Constructor.

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.

Derived from

wxGDIObject (p. 474)
wxObject (p. 746)

Include files

<wx/font.h>

Predefined objects

Objects:

wxNullFont

Pointers:

wxNORMAL_FONT
wxSMALL_FONT
wxITALIC_FONT
wxSWISS_FONT

See also

wxFont overview (p. 1392), *wxDC::SetFont* (p. 295), *wxDC::DrawText* (p. 287), *wxDC::GetTextExtent* (p. 291), *wxFontDialog* (p. 444)

wxFont::wxFont

wxFont()

Default constructor.

wxFont(int *pointSize*, int *family*, int *style*, int *weight*, const bool *underline* = FALSE, const wxString& *faceName* = "", wxFontEncoding *encoding* = wxFONTENCODING_DEFAULT)

Creates a font object (see *font encoding overview* (p. 1393) for the meaning of the last parameter).

Parameters*pointSize*

Size in points.

family

Font family, a generic way of referring to fonts without specifying actual facename. One of:

wxDEFAULT	Chooses a default font.
wxDECORATIVE	A decorative font.
wxROMAN	A formal, serif font.
wxSCRIPT	A handwriting font.
wxSWISS	A sans-serif font.
wxMODERN	A fixed pitch font.

style

One of **wxNORMAL**, **wxSLANT** and **wxITALIC**.

weight

One of **wxNORMAL**, **wxLIGHT** and **wxBOLD**.

underline

The value can be TRUE or FALSE. At present this has an effect on Windows only.

faceName

An optional string specifying the actual typeface to be used. If the empty string, a default typeface will 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. 438)

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.

Remarks

If the desired font does not exist, the closest match will be chosen. Under Windows, only scaleable TrueType fonts are used.

Underlining only works under Windows at present.

See also *wxDC::SetFont* (p. 295), *wxDC::DrawText* (p. 287) and *wxDC::GetTextExtent* (p. 291).

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 wxWindows 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::GetDefaultEncoding

static wxFontEncoding GetDefaultEncoding()

Returns the current applications default encoding.

See also

Font encoding overview (p. 1393), *SetDefaultEncoding* (p. 438)

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. 438)

wxFont::GetFamily

int GetFamily() const

Gets the font family. See *wxFont::wxFont* (p. 435) for a list of valid family identifiers.

See also

wxFont::SetFamily (p. 439)

wxFont::GetFontId

int GetFontId() const

Returns the font id, if the portable font system is in operation. See *Font overview* (p. 1392) for further details.

wxFont::GetPointSize

int GetPointSize() const

Gets the point size.

See also

wxFont::SetPointSize (p. 439)

wxFont::GetStyle

int GetStyle() const

Gets the font style. See *wxFont::wxFont* (p. 435) for a list of valid styles.

See also

wxFont::SetStyle (p. 439)

wxFont::GetUnderlined

bool GetUnderlined() const

Returns TRUE if the font is underlined, FALSE otherwise.

See also

wxFont::SetUnderlined (p. 440)

wxFont::GetWeight

int GetWeight() const

Gets the font weight. See *wxFont::wxFont* (p. 435) for a list of valid weight identifiers.

See also

wxFont::SetWeight (p. 440)

wxFont::SetDefaultEncoding

static void SetDefaultEncoding(wxFontEncoding *encoding*)

Sets the default font encoding.

See also

Font encoding overview (p. 1393), *GetDefaultEncoding* (p. 436)

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, `wxWindows` will first search for the specific face, and then for a font belonging to the same family.

See also

`wxFont::GetFaceName` (p. 437), `wxFont::SetFamily` (p. 439)

wxFont::SetFamily

void SetFamily(int *family*)

Sets the font family.

Parameters

family

One of:

wxDEFAULT	Chooses a default font.
wxDECORATIVE	A decorative font.
wxROMAN	A formal, serif font.
wxSCRIPT	A handwriting font.
wxSWISS	A sans-serif font.
wxMODERN	A fixed pitch font.

See also

`wxFont::GetFamily` (p. 437), `wxFont::SetFaceName` (p. 438)

wxFont::SetPointSize

void SetPointSize(int *pointSize*)

Sets the point size.

Parameters

pointSize

Size in points.

See also

`wxFont::GetPointSize` (p. 437)

wxFont::SetStyle

void SetStyle(int style)

Sets the font style.

Parameters

style

One of **wxNORMAL**, **wxSLANT** and **wxITALIC**.

See also

wxFont::GetStyle (p. 437)

wxFont::SetUnderlined

void SetUnderlined(const bool underlined)

Sets underlining.

Parameters

underlining

TRUE to underline, FALSE otherwise.

See also

wxFont::GetUnderlined (p. 438)

wxFont::SetWeight

void SetWeight(int weight)

Sets the font weight.

Parameters

weight

One of **wxNORMAL**, **wxLIGHT** and **wxBOLD**.

See also

wxFont::GetWeight (p. 438)

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. 1399)

This class holds a variety of information related to font dialogs.

Derived from

wxObject (p. 746)

Include files

<wx/cmndata.h>

See also

Overview (p. 1399), *wxFontDialog* (p. 444)

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::~~wxFontData

~wxFontData()

Destructor.

wxFontData::EnableEffects

void EnableEffects(bool *enable*)

Enables or disables 'effects' under MS Windows only. This refers to the controls for manipulating colour, strikethrough 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::Show` returned TRUE), this returns a new font which is now 'owned' by the application, and should be deleted if not required. If the user pressed Cancel (`wxFontDialog::Show` returned FALSE) or the colour dialog has not been invoked yet, this will return NULL.

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. 310)

wxWindow (p. 1184)

wxEvtHandler (p. 378)

wxObject (p. 746)

Include files

<wx/fontdlg.h>

See also

Overview (p. 1399), *wxFontData* (p. 441)

wxFontDialog::wxFontDialog

wxFontDialog(wxWindow* parent, wxFontData* data = NULL)

Constructor. Pass a parent window, and optionally a pointer to a block of font data, which will be copied to the font dialog's font data.

wxFontDialog::~~wxFontDialog

~wxFontDialog()

Destructor.

wxFontDialog::GetFontData

wxFontData& GetFontData()

Returns the *font data* (p. 441) 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 (ShowModal returns wxID_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. 1393).

To do this, you just have to call one of `EnumerateXXX()` functions - either `EnumerateFacenames` (p. 446) or `EnumerateEncodings` (p. 446) and the corresponding callback (`OnFacename` (p. 447) or `OnFontEncoding` (p. 447)) 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. 447) or `OnFontEncoding` (p. 447) should be overridden depending on whether you plan to call `EnumerateFacenames` (p. 446) or `EnumerateEncodings` (p. 446). 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. 1393), *Font sample* (p. 1323), `wxFont` (p. 434), `wxFontMapper` (p. 448)

wxFontEnumerator::EnumerateFacenames

```
virtual bool EnumerateFacenames( wxFontEncoding encoding =  
wxFONTENCODING_SYSTEM, bool fixedWidthOnly = FALSE)
```

Call `OnFacename` (p. 447) 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. 447) 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. 446). This is convenience function. It is based on default implementation of *OnFontEncoding* (p. 447) 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. 446). This is convenience function. It is based on default implementation of *OnFacename* (p. 447) so don't expect it to work if you overwrite that method.

wxFontEnumerator::OnFacename

virtual bool OnFacename(const wxString& font)

Called by *EnumerateFacenames* (p. 446) 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. 446) 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. 615)

wxObject (p. 746)

Include files

<wx/gdicmn.h>

See also

wxFont (p. 434)

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 *wxWindows* 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. 435) for details of the arguments.

wxFontList::RemoveFont

void RemoveFont(*wxFont *font*)

Used by *wxWindows* 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. 371) for "equivalent" encodings (e.g. iso8859-2 and cp1250) and tries them.

Global variables

`wxFontMapper` *`wxTheFontMapper` is defined.

Using `wxFontMapper` in conjunction with `wxEncodingConverter`

If you need to display text in encoding which is not available at host system (see *IsEncodingAvailable* (p. 450)), you may use these two classes to a) find font in some similar encoding (see *GetAltForEncoding* (p. 450)) and b) convert the text to this encoding (`wxEncodingConverter::Convert` (p. 372)).

Following code snippet demonstrates it:

```
if (!wxTheFontMapper->IsEncodingAvailable(enc, facename))
{
    wxFontEncoding alternative;
    if (wxTheFontMapper->GetAltForEncoding(enc, &alternative,
                                          facename, FALSE))
    {
        wxEncodingConverter encconv;
        if (!encconv.Init(enc, alternative))
            ...failure...
        else
            text = encconv.Convert(text);
    }
    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. 371), *Writing non-English applications* (p. 1347)

wxFontMapper::wxFontMapper

wxFontMapper()

Default ctor.

wxFontMapper::~~wxFontMapper

~wxFontMapper()

Virtual dtor for a base class.

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 wxWindows' internal use while the second one is better suitable for general use -- it returns wxFontEncoding which can consequently be passed to wxFont constructor.

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::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.

wxFontMapper::GetEncodingName

static wxString GetEncodingName(wxFontEncoding encoding)

Return internal string identifier for the encoding (see also *GetEncodingDescription()* (p. 451))

wxFontMapper::GetEncodingDescription

static wxString GetEncodingDescription(wxFontEncoding encoding)

Return user-readable string describing the given 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::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

wxWindow (p. 1184)
wxEvtHandler (p. 378)
wxObject (p. 746)

Include files

<wx/frame.h>

Window styles

wxDEFAULT_FRAME_STYLE Defined as **wxMINIMIZE_BOX | wxMAXIMIZE_BOX | wxRESIZE_BOX | wxSYSTEM_MENU | wxCAPTION.**

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.

wxSTAY_ON_TOP Stay on top of other windows. Windows only.

wxSYSTEM_MENU Displays a system menu.

wxSIMPLE_BORDER Displays no border or decorations. GTK and Windows only.

wxRESIZE_BORDER Displays a resizable border around the window (Unix only).

wxFRAME_FLOAT_ON_PARENT Causes the frame to be above the parent window in the z-order and not shown in the taskbar. Without this style, frames are created as top-level windows that may be obscured by the parent window, and frame titles are shown in the taskbar. Windows and GTK.

wxFRAME_TOOL_WINDOW Causes a frame with a small titlebar to be created;

the frame title does not appear in the taskbar. Windows only.

The default frame style is for normal, resizable frames. To create a frame which can not be resized by user, you may use the following combination of styles:

`wxDEFAULT_FRAME_STYLE & | wxRESIZE_BOX | wxMAXIMIZE_BOX`). See also *window styles overview* (p. 1371).

Remarks

An application should normally define an *OnCloseWindow* (p. 1208) 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. 671), *wxMDIChildFrame* (p. 666), *wxMiniFrame* (p. 716), *wxDialog* (p. 310)

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 wxWindows, depending on platform.

size

The window size. A value of (-1, -1) indicates a default size, chosen by either the windowing system or `wxWindows`, depending on platform.

style

The window style. See `wxFrame` (p. 452).

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. 455)

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::Command

void Command(int id)

Simulate a menu command.

Parameters*id*

The identifier for a menu item.

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. 453) 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. 991) for a list of valid styles.

id

The status bar window identifier. If -1, an identifier will be chosen by wxWindows.

name

The status bar window name.

Return value

A pointer to the 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. 458).

Note that you can put controls and other windows on the status bar if you wish.

See also

wxFrame::SetStatusText (p. 462), *wxFrame::OnCreateStatusBar* (p. 458),
wxFrame::GetStatusBar (p. 457)

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. 1117) for a list of valid styles.

id

The toolbar window identifier. If -1, an identifier will be chosen by wxWindows.

name

The toolbar window name.

Return value

A pointer to the 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. 459).

When a toolbar has been created with this function, or made known to the frame with *wxFrame::SetToolBar* (p. 463), the frame will manage the toolbar position and adjust the return value from *wxWindow::GetClientSize* (p. 1195) to reflect the available space for application windows.

See also

wxFrame::CreateStatusBar (p. 455), *wxFrame::OnCreateToolBar* (p. 459),
wxFrame::SetToolBar (p. 463), *wxFrame::GetToolBar* (p. 457)

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. 461), *wxMenuBar* (p. 693), *wxMenu* (p. 683)

wxFrame::GetStatusBar

wxStatusBar* GetStatusBar() const

Returns a pointer to the status bar currently associated with the frame (if any).

See also

wxFrame::CreateStatusBar (p. 455), *wxStatusBar* (p. 991)

wxFrame::GetTitle

wxString GetTitle() const

Gets a string containing the frame title. See *wxFrame::SetTitle* (p. 463).

wxFrame::GetToolBar

wxToolBar* GetToolBar() const

Returns a pointer to the toolbar currently associated with the frame (if any).

See also

wxFrame::CreateToolBar (p. 456), *wxToolBar* (p. 1117), *wxFrame::SetToolBar* (p. 463)

wxFrame::Iconize

void Iconize(bool iconize)

Iconizes or restores the frame. Windows only.

Parameters

iconize

If TRUE, iconizes the frame; if FALSE, shows and restores it.

See also

wxFrame::IsIconized (p. 458), *wxFrame::Maximize* (p. 458).

wxFrame::IsIconized

bool IsIconized() const

Returns TRUE if the frame is iconized. Windows only.

wxFrame::IsMaximized

bool IsMaximized() const

Returns TRUE if the frame is maximized.

wxFrame::Maximize

void Maximize(bool maximize)

Maximizes or restores the frame.

Parameters

maximize

If TRUE, maximizes the frame, otherwise it restores it.

Remarks

This function only works under Windows.

See also

wxFrame::Iconize (p. 457)

wxFrame::OnActivate

void OnActivate(wxActivateEvent& event)

Called when a window is activated or deactivated (MS Windows only). See also *wxActivateEvent* (p. 20).

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. 455).

Parameters

number

The number of fields to create.

style

The window style. See *wxStatusBar* (p. 991) for a list of valid styles.

id

The window identifier. If -1, an identifier will be chosen by wxWindows.

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. 991).

See also

wxFrame::CreateStatusBar (p. 455), *wxStatusBar* (p. 991).

wxFrame::OnCreateToolBar

virtual wxToolBar* OnCreateToolBar(long style, wxWindowID id, const wxString& name)

Virtual function called when a toolbar is requested by *wxFrame::CreateToolBar* (p. 456).

Parameters

style

The toolbar style. See *wxToolBar* (p. 1117) for a list of valid styles.

id

The toolbar window identifier. If -1, an identifier will be chosen by wxWindows.

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. 1117).

See also

wxFrame::CreateToolBar (p. 456), *wxToolBar* (p. 1117).

wxFrame::OnMenuCommand

void OnMenuCommand(wxCommandEvent& event)

See *wxWindow::OnMenuCommand* (p. 1212).

wxFrame::OnMenuHighlight

void OnMenuHighlight(wxMenuEvent& event)

See *wxWindow::OnMenuHighlight* (p. 1212).

wxFrame::OnSize

void OnSize(wxSizeEvent& event)

See *wxWindow::OnSize* (p. 1216).

The default **wxFrame::OnSize** implementation looks for a single subwindow, and if one is found, resizes it to fit inside the frame. Override this member if more complex behaviour is required (for example, if there are several subwindows).

wxFrame::SetIcon

void SetIcon(const wxIcon& icon)

Sets the icon for this frame.

Parameters

icon

The icon to associate with this frame.

Remarks

The frame 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.

Under Windows, instead of using **SetIcon**, you can add the following lines to your MS Windows resource file:

```
wxSTD_MDIPARENTFRAME ICON icon1.ico
wxSTD_MDICHILDFRAME  ICON icon2.ico
wxSTD_FRAME           ICON icon3.ico
```

where icon1.ico will be used for the MDI parent frame, icon2.ico will be used for MDI child frames, and icon3.ico will be used for non-MDI frames.

If these icons are not supplied, and **SetIcon** is not called either, then the following defaults apply if you have included wx.rc.

```
wxDEFAULT_FRAME           ICON std.ico
wxDEFAULT_MDIPARENTFRAME  ICON mdi.ico
wxDEFAULT_MDICHILDFRAME  ICON child.ico
```

You can replace std.ico, mdi.ico and child.ico with your own defaults for all your wxWindows application. Currently they show the same icon.

Note: a wxWindows application linked with subsystem equal to 4.0 (i.e. marked as a Windows 95 application) doesn't respond properly to wxFrame::SetIcon. To work around this until a solution is found, mark your program as a 3.5 application. This will also ensure that Windows provides small icons for the application automatically.

See also *wxIcon* (p. 558).

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 call to `wxFrame::OnSize` (p. 460) is generated, so be sure to initialize data members properly before calling **SetMenuBar**.

Note that it is not possible to call this function twice for the same frame object.

See also

`wxFrame::GetMenuBar` (p. 457), `wxMenuBar` (p. 693), `wxMenu` (p. 683).

wxFrame::SetStatusBar

void SetStatusBar(`wxStatusBar* statusBar`)

Associates a status bar with the frame.

See also

`wxFrame::CreateStatusBar` (p. 455), `wxStatusBar` (p. 991), `wxFrame::GetStatusBar` (p. 457)

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. 455), `wxStatusBar` (p. 991)

wxFrame::SetStatusWidths

virtual void SetStatusWidths(`int n`, `int *widths`)

Sets the widths of the fields in the status bar.

Parameters

n The number of fields in the status bar. It must be the same used in *CreateStatusBar* (p. 455).

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.

wxFrame::SetToolBar

void SetToolBar(wxToolBar* *toolBar*)

Associates a toolbar with the frame.

See also

wxFrame::CreateToolBar (p. 456), *wxToolBar* (p. 1117), *wxFrame::GetToolBar* (p. 457)

wxFrame::SetTitle

virtual void SetTitle(const wxString& *title*)

Sets the frame title.

Parameters

title

The frame title.

See also

wxFrame::GetTitle (p. 457)

wxFrame::ShowFullScreen

bool ShowFullScreen(bool *show*, long *style* = wxFULLSCREEN_ALL)

Passing TRUE to *shows* shows the frame full-screen, and passing FALSE restores the frame again. *style* is a bit list containing some or all of the following values, which indicate what elements of the frame to hide in full-screen mode:

- wxFULLSCREEN_NOMENUBAR
- wxFULLSCREEN_NOTOOLBAR
- wxFULLSCREEN_NOSTATUSBAR
- wxFULLSCREEN_NOBORDER
- wxFULLSCREEN_NOCAPTION
- wxFULLSCREEN_ALL (all of the above)

This function only works on Windows and has not been tested with MDI frames.

wxFSFile

This class represents a single file opened by *wxFileSystem* (p. 422). It provides more information than *wxWindow*'s input stream (stream, filename, mime type, anchor).

Note: Any pointer returned by *wxFSFile*'s member is valid only as long as *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. 746)

Include files

<wx/filesys.h>

See Also

wxFileSystemHandler (p. 424), *wxFileSystem* (p. 422), *Overview* (p. 1362)

wxFSFile::wxFSFile

wxFSFile(*wxInputStream* **stream*, **const wxString&** *loc*, **const wxString&** *mimetype*, **const wxString&** *anchor*)

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. 465) 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.wxwindows.org
http://www.ms.mff.cuni.cz/~vs1a8348/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**Derived from**

wxProtocol (p. 849)

Include files

<wx/protocol/ftp.h>

See also

wxSocketBase (p. 938)

wxFTP::SendCommand

bool SendCommand(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::GetLastResult

const wxString& GetLastResult()

Returns the last command result.

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::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::GetList

bool GetList(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.

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. 751)

wxFTP::GetInputStream

wxInputStream * GetInputStream(const wxString& path)

Creates a new input stream on the 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->StreamSize()];

in_stream->Read(data, in_stream->StreamSize());
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. 592)

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. 176)
wxWindow (p. 1184)
wxEvtHandler (p. 378)
wxObject (p. 746)

Include files

<wx/gauge.h>

Window styles

wxGA_HORIZONTAL	Creates a horizontal gauge.
wxGA_VERTICAL	Creates a vertical gauge.
wxGA_PROGRESSBAR	Under Windows 95, creates a horizontal progress bar.
wxGA_SMOOTH	Under Windows 95, creates smooth progress bar with one pixel wide update step.

See also *window styles overview* (p. 1371).

Event handling

wxGauge is read-only so generates no events.

See also

wxSlider (p. 929), *wxScrollBar* (p. 903)

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. 470).*name*

Window name.

Remarks

Under Windows 95, there are two different styles of gauge: normal gauge, and progress bar (when the **wxGA_PROGRESSBAR** style is used). A progress bar is always horizontal.

See also*wxGauge::Create* (p. 472)**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. 471) for further details.

wxGauge::GetBezelFace

int GetBezelFace() const

Returns the width of the 3D bezel face.

RemarksWindows only, not for **wxGA_PROGRESSBAR**.**See also**

wxGauge::SetBezelFace (p. 473)

wxGauge::GetRange

int GetRange() const

Returns the maximum position of the gauge.

See also

wxGauge::SetRange (p. 474)

wxGauge::GetShadowWidth

int GetShadowWidth() const

Returns the 3D shadow margin width.

Remarks

Windows only, not for **wxGA_PROGRESSBAR**.

See also

wxGauge::SetShadowWidth (p. 474)

wxGauge::GetValue

int GetValue() const

Returns the current position of the gauge.

See also

wxGauge::SetValue (p. 474)

wxGauge::SetBezelFace

void SetBezelFace(int width)

Sets the 3D bezel face width.

Remarks

Windows only, not for **wxGA_PROGRESSBAR**.

See also

wxGauge::GetBezelFace (p. 472)

wxGauge::SetRange

void SetRange(int *range*)

Sets the range (maximum value) of the gauge.

See also

wxGauge::GetRange (p. 473)

wxGauge::SetShadowWidth

void SetShadowWidth(int *width*)

Sets the 3D shadow width.

Remarks

Windows only, not for **wxGA_PROGRESSBAR**.

wxGauge::SetValue

void SetValue(int *pos*)

Sets the position of the gauge.

Parameters

pos
Position for the gauge level.

See also

wxGauge::GetValue (p. 473)

wxGDIObject

This class allows platforms to implement functionality to optimise GDI objects, such as `wxPen`, `wxBrush` and `wxFont`. On Windows, the underlying 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. 746)

Include files

<wx/gdiobj.h>

See also

wxPen (p. 771), *wxBrush* (p. 80), *wxFont* (p. 434)

wxGDIObject::wxGDIObject

wxGDIObject()

Default constructor.

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. 476) to direct normal OpenGL commands to the window, and then call *wxGLCanvas::SwapBuffers* (p. 476) to show the OpenGL buffer on the window.

Please note that despite deriving from *wxScrolledWindow*, scrolling is not enabled for this class under Windows.

To switch *wxGLCanvas* support on under Windows, edit *setup.h* and set *wxUSE_GLCANVAS* to 1. On Unix, pass *--with-opengl* to *configure* to compile using OpenGL or Mesa.

Derived from

wxScrolledWindow (p. 911)

wxWindow (p. 1184)

wxEvtHandler (p. 378)

wxObject (p. 746)

Include files

<wx/glcanvas.h>

Window styles

There are no specific window styles for this class.

See also *window styles overview* (p. 1371).

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.

wxGLCanvas::SetCurrent

```
void SetCurrent()
```

Sets this canvas as the current recipient of OpenGL calls.

wxGLCanvas::SetColour

```
void SetColour(const char* colour)
```

Sets the current colour for this window, using the wxWindows colour database to find a named colour.

wxGLCanvas::SwapBuffers

void SwapBuffers()

Displays the previous OpenGL commands on the window.

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. 1374).

Derived from

wxValidator (p. 1166)
wxEvtHandler (p. 378)
wxObject (p. 746)

Include files

<wx/valgen.h>

See also

Validator overview (p. 1374), *wxValidator* (p. 1166), *wxTextValidator* (p. 1092)

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 TransferToWindow()`

Transfers the value to the window.

`wxGenericValidator::TransferToWindow`

`virtual bool TransferToWindow()`

Transfers the window value to the appropriate data type.

wxGrid

wxGrid is a class for displaying and editing tabular information.

Note: there is a new grid implementation from wxWindows 2.1.14, with an API that is backwardly compatible with the one documented here. This documentation is awaiting updates for the new and improved API.

Derived from

wxPanel (p. 764)

wxWindow (p. 1184)

wxEvtHandler (p. 378)

wxObject (p. 746)

Include files

<wx/grid.h>

Window styles

There are no specific window styles for this class, but you may use different SetXXX() functions to change the controls behaviour (for example, to enable in-place editing).

See also *window styles overview* (p. 1371).

See also

wxGrid classes overview (p. 1417)

wxGrid::wxGrid

void wxGrid(wxWindow* parent, wxWindowID id, const wxPoint& pos, const wxSize& size, long style=0, const wxString& name="grid")

Constructor. Before using a wxGrid object, you must call CreateGrid to set up the required rows and columns.

wxGrid::AdjustScrollbars

void AdjustScrollbars()

Call this function whenever a change has been made via the API that might alter the scrollbar characteristics: particularly when adding or deleting rows, or changing row or column dimensions. For example, removing rows might make it unnecessary to show the vertical scrollbar.

wxGrid::AppendCols

bool AppendCols(int *n*=1, bool *updateLabels*=TRUE)

Appends *n* columns to the grid. If *updateLabels* is TRUE, the function `OnChangeLabels` is called to give the application the opportunity to relabel.

wxGrid::AppendRows

bool AppendRows(int *n*=1, bool *updateLabels*=TRUE)

Appends *n* rows to the grid. If *updateLabels* is TRUE, the function `OnChangeLabels` is called to give the application the opportunity to relabel.

wxGrid::BeginBatch

void BeginBatch()

Start a `BeginBatch/EndBatch` pair between which, calls to `SetCellValue` or `SetCellBitmap` will not cause a refresh. This allows you to speed up some operations (for example, setting several hundred cell values). You can nest, but not overlap, these two functions.

See also `wxGrid::EndBatch` (p. 481), `wxGrid::GetBatchCount` (p. 481).

wxGrid::CellHitTest

bool CellHitTest(int *x*, int *y*, int **row*, int **col*)

Returns TRUE if the *x*, *y* panel position coincides with a cell. If so, *row* and *col* are returned.

wxGrid::CreateGrid

bool CreateGrid(int *rows*, int *cols*, wxString *cellValues*=NULL, short **widths*=NULL, short *defaultWidth*=wxGRID_DEFAULT_CELL_WIDTH, short *defaultHeight*=wxGRID_DEFAULT_CELL_HEIGHT)**

Creates a grid *rows* high and *cols* wide. You can optionally specify an array of initial values and widths, and/or default cell width and height.

Call this function after creating the `wxGrid` object.

wxPython note: Currently the `cellValues` and `widths` parameters don't exist in the wxPython version of this method. So in other words, the definition of the wxPython version of this method looks like this:

```
CreateGrid(rows, cols,
           defaultWidth = wxGRID_DEFAULT_CELL_WIDTH,
           defaultHeight = wxGRID_DEFAULT_CELL_HEIGHT)
```

wxGrid::CurrentCellVisible

bool CurrentCellVisible()

Returns TRUE if the currently selected cell is visible, FALSE otherwise.

wxGrid::DeleteCols

bool DeleteCols(int pos=0, int n=1, bool updateLabels=TRUE)

Deletes *n* columns from the grid at position *pos*. If *updateLabels* is TRUE, the function `OnChangeLabels` is called to give the application the opportunity to relabel.

wxGrid::DeleteRows

bool DeleteRows(int pos=0, int n=1, bool updateLabels=TRUE)

Deletes *n* rows from the grid at position *pos*. If *updateLabels* is TRUE, the function `OnChangeLabels` is called to give the application the opportunity to relabel.

wxGrid::EndBatch

void EndBatch()

End a `BeginBatch/EndBatch` pair between which, calls to `SetCellValue` or `SetCellBitmap` will not cause a refresh. This allows you to speed up some operations (for example, setting several hundred cell values). You can nest, but not overlap, these two functions.

See also `wxGrid::BeginBatch` (p. 480), `wxGrid::GetBatchCount` (p. 481).

wxGrid::GetBatchCount

int GetBatchCount() const

Return the level of batch nesting. This is initially zero, and will be incremented every time `BeginBatch` is called, and decremented when `EndBatch` is called. When the batch count is more zero, some functions (such as `SetCellValue` and `SetCellBitmap`) will not refresh the cell.

See also `wxGrid::BeginBatch` (p. 480), `wxGrid::EndBatch` (p. 481).

wxGrid::GetCell

wxGridCell * GetCell(int row, int col) const

Returns the grid cell object associated with this position.

wxGenericGrid implementation only.

wxGrid::GetCellAlignment

int GetCellAlignment(int row, int col) const

int GetCellAlignment() const

Sets the text alignment for the cell at the given position, or the global alignment value. The return value is `wxLEFT`, `wxRIGHT` or `wxCENTRE`.

wxPython note: In place of a single overloaded method name, wxPython implements the following methods:

GetCellAlignment(row, col)
GetDefCellAlignment()

wxGrid::GetCellBackgroundColour

wxColour& GetCellBackgroundColour(int row, int col) const

wxColour& GetCellBackgroundColour() const

Gets the background colour for the cell at the given position, or the global background colour.

wxPython note: In place of a single overloaded method name, wxPython implements the following methods:

GetCellBackgroundColour(row, col)
GetDefCellBackgroundColour()

wxGrid::GetCells

wxGridCell * GetCells() const**

Returns the array of grid cell object associated with this wxGrid.

wxGrid::GetCellTextColour

wxColour& GetCellTextColour(int row, int col) const**wxColour& GetCellTextColour() const**

Gets the text colour for the cell at the given position, or the global text colour.

wxPython note: In place of a single overloaded method name, wxPython implements the following methods:

GetCellTextColour(row, col)
GetDefCellTextColour()**wxGrid::GetCellTextFont**

const wxFont& GetCellTextFont(int row, int col) const**wxFont& GetCellTextFont() const**

Gets the text font for the cell at the given position, or the global text font.

wxPython note: In place of a single overloaded method name, wxPython implements the following methods:

GetCellTextFont(row, col)
GetDefCellTextFont()**wxGrid::GetCellValue**

wxString& GetCellValue(int row, int col) const

Returns the cell value at the given position.

wxGrid::GetCols

int GetCols() const

Returns the number of columns in the grid.

wxGrid::GetColumnWidth

int GetColumnWidth(int col) const

Gets the width in pixels for column *col*.

wxGrid::GetCurrentRect

wxRectangle * GetCurrentRect() const

Returns a pointer to the rectangle enclosing the currently selected cell. Do not delete this pointer.

wxGrid::GetCursorColumn

int GetCursorColumn() const

Returns the column position of the currently selected cell.

wxGrid::GetCursorRow

int GetCursorRow() const

Returns the row position of the currently selected cell.

wxGrid::GetEditable

bool GetEditable() const

Returns TRUE if the grid cells can be edited.

wxGrid::GetEditInPlace

bool GetEditInPlace() const

Returns TRUE if editing in-place is enabled.

wxGrid::GetHorizScrollBar

wxScrollBar * GetHorizScrollBar() const

Returns a pointer to the horizontal scrollbar.

wxGrid::GetLabelAlignment

int GetLabelAlignment(int *orientation*) const

Gets the row or column label alignment. *orientation* should be wxHORIZONTAL to specify column label, wxVERTICAL to specify row label. *alignment* should be wxCENTRE, wxLEFT or wxRIGHT.

wxGrid::GetLabelBackgroundColour

wxColour& GetLabelBackgroundColour() const

Gets a row and column label text colour.

wxGrid::GetLabelSize

int GetLabelSize(int *orientation*) const

Gets the row label height, or column label width, in pixels. *orientation* should be wxHORIZONTAL to specify column label, wxVERTICAL to specify row label.

wxGrid::GetLabelTextColour

wxColour& GetLabelTextColour() const

Gets a row and column label text colour.

wxGrid::GetLabelTextFont

wxFont& GetLabelTextFont() const

Gets the font to be used for the row and column labels.

wxGrid::GetLabelValue

wxString& GetLabelValue(int *orientation*, int *pos*) const

Gets a row or column label value. *orientation* should be wxHORIZONTAL to specify

column label, wxVERTICAL to specify row label. *pos* is the label position.

wxGrid::GetRowHeight

int GetRowHeight(int row) const

Gets the height in pixels for row *row*.

wxGrid::GetRows

int GetRows() const

Returns the number of rows in the grid.

wxGrid::GetScrollPosX

int GetScrollPosX() const

Returns the column scroll position.

wxGrid::GetScrollPosY

int GetScrollPosY() const

Returns the row scroll position.

wxGrid::GetTextItem

wxTextCtrl * GetTextItem() const

Returns a pointer to the text item used for entering text into a cell.

wxGrid::GetVertScrollBar

wxScrollBar * GetVertScrollBar() const

Returns a pointer to the vertical scrollbar.

wxGrid::InsertCols

bool InsertCols(int pos=0, int n=1, bool updateLabels=TRUE)

Inserts *n* number of columns before position *pos*. If *updateLabels* is TRUE, the function `OnChangeLabels` is called to give the application the opportunity to relabel.

wxGrid::InsertRows

bool InsertRows(int pos=0, int n=1, bool updateLabels=TRUE)

Inserts *n* number of rows before position *pos*. If *updateLabels* is TRUE, the function `OnChangeLabels` is called to give the application the opportunity to relabel.

wxGrid::OnActivate

void OnActivate(bool active)

Sets the text item to have the focus. Call this function when the `wxGrid` window should have the focus, for example from `wxFrame::OnActivate`.

wxGrid::OnChangeLabels

void OnChangeLabels()

Called when rows and columns are created or deleted, to allow the application an opportunity to update the labels. By default, columns are labelled alphabetically, and rows numerically.

wxGrid::OnChangeSelectionLabel

void OnChangeSelectionLabel()

Called when a cell is selected, to allow the application an opportunity to update the selection label (the label of the `wxTextCtrl` used for entering cell text). By default, the cell column letter and row number are concatenated to form the selection label.

wxGrid::OnCreateCell

wxGridCell * OnCreateCell()

Override this virtual function if you want to replace the normal `wxGridCell` with a derived class.

wxGrid::OnCellLeftClick

void OnLeftClick(int row, int col, int x, int y, bool control, bool shift)

Virtual function called when the left button is depressed within a cell, just after `OnSelectCell` is called.

wxGrid::OnCellRightClick

void OnRightClick(int row, int col, int x, int y, bool control, bool shift)

Virtual function called when the right button is depressed within a cell, just after `OnSelectCell` is called.

wxGrid::OnLabelLeftClick

void OnLeftClick(int row, int col, int x, int y, bool control, bool shift)

Virtual function called when the left button is depressed within a label.

row will be *-1* if the click is in the top labels.

col will be *-1* if the click is in the left labels.

row and *col* will be *-1* if the click is in the upper left corner.

wxGrid::OnLabelRightClick

void OnRightClick(int row, int col, int x, int y, bool control, bool shift)

Virtual function called when the right button is depressed within a label.

row will be *-1* if the click is in the top labels.

col will be *-1* if the click is in the left labels.

row and *col* will be *-1* if the click is in the upper left corner.

wxGrid::OnSelectCell

void OnSelectCell(int row, int col)

Virtual function called when the user left-clicks on a cell.

wxGrid::OnSelectCellImplementation

void OnSelectCellImplementation(wxDC *dc, int row, int col)

Virtual function called when the user left-clicks on a cell. If you override this function, call `wxGrid::OnSelectCell` to apply the default behaviour.

wxGrid::SetCellAlignment

void SetCellAlignment(int *alignment*, int *row*, int *col*)

void SetCellAlignment(int *alignment*)

Sets the text alignment for the cell at the given position, or for the whole grid. *alignment* may be `wxLEFT`, `wxRIGHT` or `wxCENTRE`.

wxPython note: In place of a single overloaded method name, wxPython implements the following methods:

SetCellAlignment(alignment, row, col)
SetDefCellAlignment(alignment)

wxGrid::SetCellBackgroundColour

void SetCellBackgroundColour(const wxColour& *colour*, int *row*, int *col*)

void SetCellBackgroundColour(const wxColour& *colour*)

Sets the background colour for the cell at the given position, or for the whole grid.

wxPython note: In place of a single overloaded method name, wxPython implements the following methods:

SetCellBackgroundColour(colour, row, col)
SetDefCellBackgroundColour(colour)

wxGrid::SetCellTextColour

void SetCellTextColour(const wxColour& *colour*, int *row*, int *col*)

void SetCellTextColour(const wxColour& *colour*)

Sets the text colour for the cell at the given position, or for the whole grid.

wxPython note: In place of a single overloaded method name, wxPython implements the following methods:

SetCellTextColour(colour, row, col)
SetDefCellTextColour(colour)

wxGrid::SetCellTextFont

void SetCellTextFont(const wxFont& font, int row, int col)

void SetCellTextFont(const wxFont& font)

Sets the text font for the cell at the given position, or for the whole grid.

wxPython note: In place of a single overloaded method name, wxPython implements the following methods:

SetCellTextFont(font, row, col)

SetDefCellTextFont(font)

wxGrid::SetCellValue

void SetCellValue(const wxString& val, int row, int col)

Sets the cell value at the given position.

wxGrid::SetColumnWidth

void SetColumnWidth(int col, int width)

Sets the width in pixels for column *col*.

wxGrid::SetDividerPen

void SetDividerPen(const wxPen& pen)

Specifies the pen to be used for drawing the divisions between cells. The default is a light grey. If NULL is specified, the divisions will not be drawn.

wxGrid::SetEditable

void SetEditable(bool editable)

If *editable* is TRUE (the default), the grid cells will be editable by means of the text edit control. If FALSE, the text edit control will be hidden and the user will not be able to edit the cell contents.

wxGrid::SetEditInPlace

void SetEditInPlace(bool *edit* = *TRUE*)

Enables (if *edit* is *TRUE*, default value) or disables in-place editing. When it is enabled, the cells contents can be changed by typing text directly in the cell.

wxGrid::SetGridCursor

void SetGridCursor(int *row*, int *col*)

Sets the position of the selected cell.

wxGrid::SetLabelAlignment

void SetLabelAlignment(int *orientation*, int *alignment*)

Sets the row or column label alignment. *orientation* should be *wxHORIZONTAL* to specify column label, *wxVERTICAL* to specify row label. *alignment* should be *wxCENTRE*, *wxLEFT* or *wxRIGHT*.

wxGrid::SetLabelBackgroundColour

void SetLabelBackgroundColour(const wxColour& *value*)

Sets a row or column label background colour.

wxGrid::SetLabelSize

void SetLabelSize(int *orientation*, int *size*)

Sets the row label height, or column label width, in pixels. *orientation* should be *wxHORIZONTAL* to specify column label, *wxVERTICAL* to specify row label.

If a dimension of zero is specified, the row or column labels will not be shown.

wxGrid::SetLabelTextColour

void SetLabelTextColour(const wxColour& *value*)

Sets a row and column label text colour.

wxGrid::SetLabelTextFont

void SetLabelTextFont(const wxFont& font)

Sets the font to be used for the row and column labels.

wxGrid::SetLabelValue

void SetLabelValue(int orientation, const wxString& value, int pos)

Sets a row or column label value. *orientation* should be wxHORIZONTAL to specify column label, wxVERTICAL to specify row label. *pos* is the label position.

wxGrid::SetRowHeight

void SetRowHeight(int row, int height)

Sets the height in pixels for row *row*.

wxGrid::UpdateDimensions

void UpdateDimensions()

Call this function whenever a change has been made via the API that might alter size characteristics. You may also need to follow it with a call to AdjustScrollbars.

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. 924)
wxObject (p. 746)

wxGridSizer::wxGridSizer

wxGridSizer(int cols, int rows, 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.

wxHashTable

This class provides hash table functionality for `wxWindows`, and for an application if it wishes. Data can be hashed on an integer or string key.

Derived from

`wxObject` (p. 746)

Include files

<wx/hash.h>

Example

Below is an example of using a hash table.

```
wxHashTable table(KEY_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. 615)

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

wxNode * 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 **wxNode** pointer (or NULL if there are no more nodes). See the description for *wxNode* (p. 735). The user will probably only wish to use the **wxNode::Data** function 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.

wxList::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 Windows, `wxWinHelpController` is used.
- On all other platforms, `wxHelpControllerHtml` is used if `wxHTML` is compiled into `wxWindows`; 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 0 in `setup.h`, and link your application with Microsoft's `htmlhelp.lib`. Currently VC++ only.
- `wxExtHelpController`, for controlling external browsers under Unix. The default browser is Netscape Navigator. The 'help' sample shows its use.
- `wxHelpControllerHtml`, using `wxHTML` (p. 1432) to display help. See `wx/helpwxht.h` for details of use.
- `wxHtmlHelpController` (p. 519), a more sophisticated help controller using `wxHTML` (p. 1432), 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.

Derived from

`wxHelpControllerBase`
`wxObject` (p. 746)

Include files

`<wx/help.h>` (`wxWindows` 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/generic/helpwxht.h>` (simple wxHTML-based help controller)
`<wx/html/helpctrl.h>` (advanced wxHTML based help controller: `wxHtmlHelpController`)

See also

`wxHtmlHelpController` (p. 519), `wxHTML` (p. 1432)

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. For `wxHelpControllerHtml` (the standard HTML help controller), the filename is assumed to be a directory containing the HTML files.

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. 498).

wxHtmlHelpController: *sectionNo* is an identifier as specified in the .hhc file. See *Help files format* (p. 1433).

This function is for backward compatibility only, and applications should use *wxHelpController* (p. 498) instead.

wxHelpController::DisplayContents

virtual bool DisplayContents()

If the help viewer is not running, runs it and displays the contents.

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, external HTML help and simple wxHTML 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/simple wxHTML help: *wxExtHelpController* and *wxHelpControllerHtml* implement *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. 1433).

See also the help sample for notes on how to specify section numbers for various help file formats.

wxHelpController::GetFrameParameters

virtual wxFrame * GetFrameParameters(const wxSize * size = NULL, const wxPoint * pos = NULL, bool *newFrameEachTime = NULL)

This reads the current settings for the help frame in the case of the wxHelpControllerHtml, setting the frame size, position and the newFrameEachTime parameters to the last values used. It also returns the pointer to the last opened help frame. This can be used for example, to automatically close the help frame on program shutdown.

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)

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.

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. 522).

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 wxHelpControllerHtml, this allows the application to set the default frame title, size and position for the frame. If the title contains %s, this will be replaced with the page title. If the parameter newFrameEachTime is set, the controller will open a new help frame each time it is called.

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.

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. 542) and *wxHtmlWinParser* (p. 548) 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. 746)

Include files

<wx/html/htmlcell.h>

See Also

Cells Overview (p. 1435), *wxHtmlContainerCell* (p. 507)

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. 501) when the cell is certainly out of the screen (and thus invisible). This is not nonsense - some tags (like *wxHtmlColourCell* (p. 506) 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

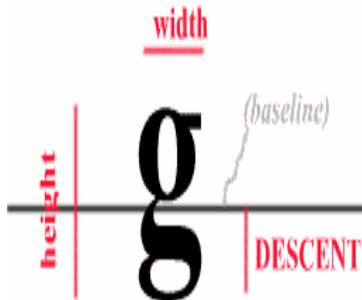
wxHTML_COND_ISANCHOR Finds particular anchor. *param* is pointer to `wxString` with name of the anchor.

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:



wxHtmlCell::GetHeight

int GetHeight() const

Returns height of the cell (`m_Height` member).

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. 529). (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. 504) 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. 504) 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::OnClick

virtual void OnClick(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. 542) this method of that cell is called. Default behavior is that it calls *wxHtmlWindow::LoadPage* (p. 545).

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::SetLink

void SetLink(const wxHtmlLinkInfo& *link*)

Sets the hypertext link associated with this cell. (Default value is *wxHtmlLinkInfo* (p. 529)("", "")) (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. 508).

wxHtmlCell::SetParent

void SetParent(*wxHtmlContainerCell *p*)

Sets parent container of this cell. This is called from *wxHtmlContainerCell::InsertCell* (p. 508).

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. 501)

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. 501)

Include files

<wx/html/htmlcell.h>

See Also

Cells Overview (p. 1435)

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::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. 502) 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. 508) 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. 508) and *SetAlignVer* (p. 509).

wxHtmlContainerCell::SetAlignHor

void SetAlignHor(int al)

Sets the container's *horizontal alignment*. During *Layout* (p. 504) each line is aligned according to *al* value.

Parameters

a

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 *a*)

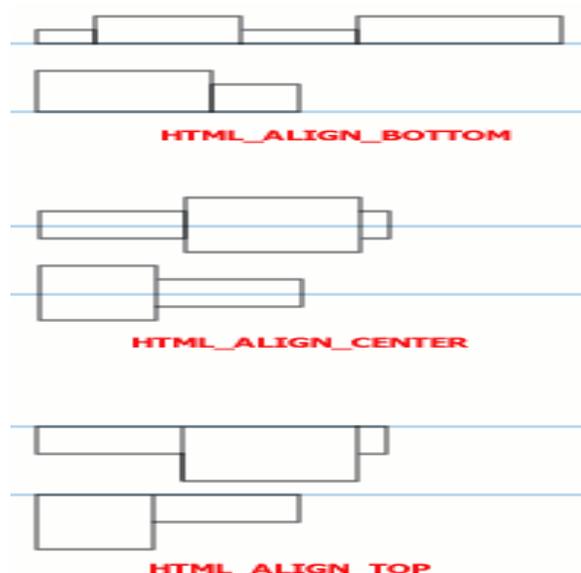
Sets the container's *vertical alignment*. This is per-line alignment!

Parameters

a

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



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

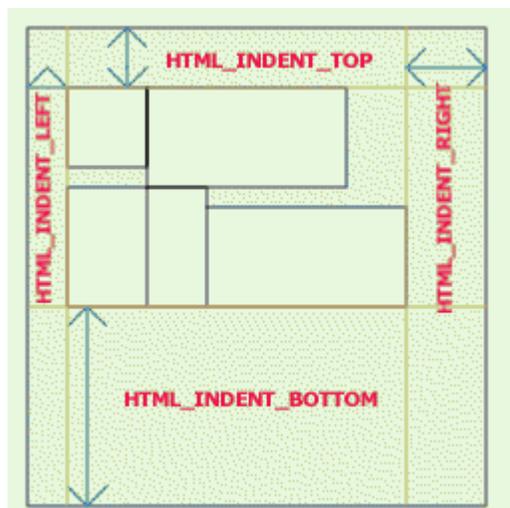
i

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



units

Units of *i*. This parameter affects interpretation of *i* 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. 504) 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 *w* 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. 515) or *wxHtmlPrintout* (p. 534) is strongly recommended.

Derived from

wxObject (p. 746)

Include files

<wx/html/htmpriint.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.)

Parameters

maxwidth

width of the area (on this DC) that is equivalent to screen's width, in pixels (you should set it to page width minus margins).

Note: In the current implementation the screen width is always 800 pixels: it gives best results and ensures (almost) same printed outputs across platforms and differently configured desktops.

See also *SetSize* (p. 513).

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. 514)! (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. 514) 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. 422) 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. 513))

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. 513)
- *SetSize* (p. 513)
- *SetHtmlText* (p. 513)

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. 513)) is smaller than total height and thus the page cannot fit into it. In that case you're supposed to call *Render* (p. 514) 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. 746)

Include files

<wx/html/htmprint.h>

wxHtmlEasyPrinting::wxHtmlEasyPrinting

```
wxHtmlEasyPrinting(const wxString& name = "Printing", wxFrame* parent_frame = NULL)
```

Constructor.

Parameters

name

Name of the printing. Used by preview frames and setup dialogs.

parent_frame

pointer to the frame that will own 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. 804) 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. 804) 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. 804) 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. 804) 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::PrinterSetup

void PrinterSetup()

Display printer setup dialog and allows the user to modify settings.

wxHtmlEasyPrinting::PageSetup

void PageSetup()

Display page setup dialog and allows the user to modify settings.

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. 792) instance used by this class. You can set its parameters (via SetXXXX methods).

wxHtmlEasyPrinting::GetPageSetupData

wxPageSetupDialogData* GetPageSetupData()

Returns a pointer to *wxPageSetupDialogData* (p. 752) instance used by this class. You can set its parameters (via SetXXXX methods).

wxHtmlFilter

This class is an input filter for *wxHtmlWindow* (p. 542). It allows you to read and display files of different file formats.

Derived from

wxObject (p. 746)

Include files

<wx/html/htmlfilt.h>

See Also

Overview (p. 1435)

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 wxWindows help controllers as documented by *wxHelpController* (p. 495), 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. 520)). A book is a logical section of documentation (for example "User's Guide" or "Programmer's Guide" or "C++ Reference" or "wxWindows Reference"). The help controller can handle as many books as you want.

wxHTML uses Microsoft's HTML Help Workshop project files (.hlp, .hhk, .hhc) as its native format. The file format is described *here* (p. 1433). 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.

In order to use the controller in your application under Windows you must have the following line in your `.rc` file:

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

Note

It is strongly recommended to use preprocessed **.hhp.cached** version of projects. It can be either created on-the-fly (see *SetTempDir* (p. 522)) 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.

Derived from

wxHelpControllerBase

Include files

<wx/html/helpctrl.h>

wxHtmlHelpController::wxHtmlHelpController

wxHtmlHelpController(int *style* = `wxHF_DEFAULTSTYLE`)

Constructor.

Parameters

style is combination of these flags:

wxHF_TOOLBAR	Help frame has toolbar.
wxHF_FLATTOOLBAR	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_OPENFILES	Allow user to open arbitrary HTML document.
wxHF_PRINT	Toolbar contains "print" button.

Default value: everything but `wxHF_OPENFILES` enabled.

wxHtmlHelpController::AddBook

bool AddBook(const wxString& *book*, **bool** *show_wait_msg*)

Adds book (*.hhp file* (p. 1433) - HTML Help Workshop project file) into the list of loaded books. This must be called at least once before displaying any help.

book 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("help.zip")` is possible and, in fact, recommended way.

If *show_wait_msg* is TRUE then a decorationless window with progress message is displayed.

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)

Displays help window, focuses search panel and starts searching. Returns TRUE if the keyword was found.

Important: KeywordSearch searches only pages listed in .hmc 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 .hmc 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. 170) and *wxConfigBase::Set* (p. 174)).

wxHtmlHelpController::WriteCustomization

void WriteCustomization(wxConfigBase* *cfg*, wxString *path* = wxEmptyString)

Stores controllers setting (position of window etc.)

wxHtmlHelpData

This class is used by *wxHtmlHelpController* (p. 519) and *wxHtmlHelpFrame* (p. 525) 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. 746)

Include files

<wx/html/helpdata.h>

wxHtmlHelpData::wxHtmlHelpData

wxHtmlHelpData()

Constructor.

wxHtmlHelpData::AddBook

bool AddBook(const wxString& book)

Adds new book. 'book' is location 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::GetContents

wxHtmlContentsItem* GetContents()

Returns contents lists pointer.

wxHtmlHelpData::GetContentsCnt

int GetContentsCnt()

Returns size of contents list.

wxHtmlHelpData::GetIndex

wxHtmlContentsItem* GetIndex()

Returns pointer to index items list.

wxHtmlHelpData::GetIndexCnt

int GetIndexCnt()

Returns size of index list.

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. 519) 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. 452)**Include files**

<wx/html/helpfrm.h>

wxHtmlHelpFrame::wxHtmlHelpFrame

wxHtmlHelpFrame(wxHtmlHelpData* data = NULL)**wxHtmlHelpFrame(wxWindow* parent, int wxWindowID, const wxString& title = wxEmptyString, int style = wxHF_DEFAULTSTYLE, wxHtmlHelpData* data = NULL)**

Constructor.

style is combination of these flags:

wxHF_TOOLBAR	Help frame has toolbar.
wxHF_FLATTOOLBAR	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_OPENFILES	Allow user to open arbitrary HTML document.
wxHF_PRINT	Toolbar contains "print" button.

wxHtmlHelpFrame::Create

bool Create(wxWindow* *parent*, wxWindowID *id*, const wxString& *title* = wxEmptyString, int *style* = wxHF_DEFAULTSTYLE)

Creates the frame.

style is combination of these flags:

wxHF_TOOLBAR	Help frame has toolbar.
wxHF_FLATTOOLBAR	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_OPENFILES	Allow user to open arbitrary HTML document.
wxHF_PRINT	Toolbar contains "print" button.

wxHtmlHelpFrame::CreateContents

void CreateContents(bool *show_progress* = FALSE)

Creates contents panel. (May take some time.)

wxHtmlHelpFrame::CreateIndex

void CreateIndex(bool *show_progress* = FALSE)

Creates index panel. (May take some time.)

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)

Search for given keyword.

wxHtmlHelpFrame::ReadCustomization

void ReadCustomization(wxConfigBase* *cfg*, const wxString& *path* = wxEmptyString)

Reads user's settings for this frame (see *wxHtmlHelpController::ReadCustomization* (p. 522))

wxHtmlHelpFrame::RefreshLists

void RefreshLists(bool *show_progress* = FALSE)

Refresh all panels. This is necessary if a new book was added.

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. 523)).

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. 746)

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. 545), NULL otherwise.

wxHtmlLinkInfo::GetHtmlCell

const wxHtmlCell * GetHtmlCell()

Return pointer to the cell that was clicked. Valid only within `wxHtmlWindow::OnLinkClicked` (p. 545), 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. 529) opened).

wxHtmlParser

This class handles the **generic** parsing of HTML document: 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. 532) method.

Derived from

`wxObject`

Include files

`<wx/html/htmlpars.h>`

See also

Cells Overview (p. 1435), *Tag Handlers Overview* (p. 1436), *wxHtmlTag* (p. 536)

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. 536) 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. 531) 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 its 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::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. 532)
2. calls *DoParsing* (p. 531)
3. calls *GetProduct* (p. 532)
4. calls *DoneParser* (p. 531)
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. 540)). 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. 534)).

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. 533).

wxHtmlParser::SetFS

void SetFS(wxFileSystem *fs)

Sets the virtual file system that will be used to request additional files. (For example tag handler requests wxFSFile with the image data.)

wxHtmlPrintout

This class serves as printout class for HTML documents.

Derived from

wxPrintout (p. 807)

Include files

<wx/html/htmprint.h>

wxHtmlPrintout::wxHtmlPrintout

wxHtmlPrintout(const wxString& title = "Printout")

Constructor.

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. 422) 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. 1436).

Derived from

wxObject

Include files

<wx/html/htmltag.h>

wxHtmlTag::wxHtmlTag

wxHtmlTag(const wxString& source, int pos, int end_pos, wxHtmlTagsCache* cache)

Constructor. You will probably never have to construct a wxHtmlTag object yourself. Feel free to ignore the constructor parameters. Have a look at lib/htmlparser.cpp if you're interested in creating it.

wxHtmlTag::GetAllParams

const wxString& GetAllParams() const

Returns string with all params.

Example : tag contains . 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 intenal 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 intenal 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 intenal 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 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 param exists or not (use *HasParam* (p. 538)) first.

Parameters

par

The parameter's name in uppercase

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::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 parameter of the given name. Example : has two parameters named "SIZE" and "COLOR".

Parameters

par

the parameter you're looking for. It must *always* be in uppercase!

wxHtmlTag::IsEnding

bool IsEnding() const

Returns TRUE if this tag is ending one. (is ending tag, is not)

wxHtmlTag::ScanParam

wxString ScanParam(const wxString& par, const char *format, fuck) const

This method scans given parameter. Usage is exactly the same as sscanf's usage except that you don't pass string but param name as the first parameter.

Parameters

par

The name of tag you want to query (in uppercase)

format

scanf()-like format string.

Cygwin and Mingw32

If you're using Cygwin beta 20 or Mingw32 compiler please remember that ScanParam() is only partially implemented! The problem is that under Cygnus' GCC vsscanf() function is not implemented. I worked around this in a way which causes that you can use only one parameter in ... (and only one % in *format*).

wxHtmlTagHandler

Derived from

wxObject (p. 746)

Include files

<wx/html/htmlpars.h>

See Also

Overview (p. 1436), *wxHtmlTag* (p. 536)

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. 536) for details).

Return value

TRUE if *ParseInner* (p. 540) 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 ending one.

wxHtmlTagHandler::ParseInner

void ParseInner(const wxHtmlTag& tag)

This method calls parser's *DoParsing* (p. 531) method for the string between this tag and 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 set of *TAGS_MODULE_** macros (p. 1436)

Derived from

wxModule (p. 719)

Include files

<wx/html/winpars.h>

See Also

Tag Handlers (p. 1436), *wxHtmlTagHandler* (p. 539), *wxHtmlWinTagHandler* (p. 555),

wxHtmlTagsModule::FillHandlersTable**virtual void FillHandlersTable(wxHtmlWinParser *parser)**

You must override this method. In most common case it is body consists only of lines of following type:

```
parser -> AddTagHandler(new MyHandler);
```

I recommend using **TAGS_MODULE_*** macros.

Parameters

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. 501)

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. 547) or `LoadPage(filename)` (p. 545).

Derived from

`wxScrolledWindow` (p. 911)

Include files

<wx/html/htmlwin.h>

wxHtmlWindow::wxHtmlWindow

wxHtmlWindow()

Default constructor.

`wxHtmlWindow(wxWindow *parent, wxWindowID id = -1, const wxPoint& pos = wxDefaultPosition, const wxSize& size = wxDefaultSize, long style = wxHW_SCROLLBAR_AUTO, const wxString& name = "htmlWindow")`

Constructor. The parameters are the same as for the `wxScrolledWindow` (p. 911) constructor.

Parameters

style

`wxHW_SCROLLBAR_NEVER`, or `wxHW_SCROLLBAR_AUTO`. Affects the appearance of vertical scrollbar in the window.

wxHtmlWindow::AddFilter

static void AddFilter(wxHtmlFilter *filter)

Adds *input filter* (p. 1435) 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::GetInternalRepresentation

wxHtmlContainerCell* GetInternalRepresentation() const

Returns pointer to the top-level container.

See also: *Cells Overview* (p. 1435), *Printing Overview* (p. 1433)

wxHtmlWindow::GetOpenedAnchor

wxString GetOpenedAnchor()

Returns anchor within currently opened page (see *GetOpenedPage* (p. 544)). 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. 545) is stored in history list.)

wxHtmlWindow::HistoryClear

void HistoryClear()

Clears history.

wxHtmlWindow::HistoryForward

bool HistoryForward()

Moves to next page in history.

wxHtmlWindow::LoadPage

bool LoadPage(const wxString& location)

Unlike `SetPage` this function first loads HTML page from *location* and then displays it. See example:

```
htmlwin -> SetPage("help/myproject/index.htm");
```

Parameters

location

The address of document. See *wxFileSystem* (p. 422) for details on address format and behaviour of "opener".

Return value

FALSE if an error occurred, TRUE otherwise

wxHtmlWindow::OnLinkClicked

virtual void OnLinkClicked(const wxHtmlLinkInfo& link)

Called when user clicks on hypertext link. Default behaviour is to call *LoadPage* (p. 545) and do nothing else.

Also see *wxHtmlLinkInfo* (p. 529).

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.

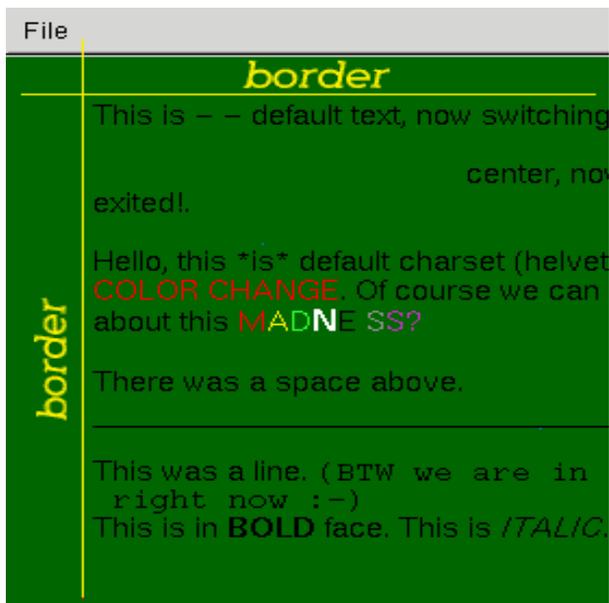
path

Optional path in config tree. If not given current path is used.

wxHtmlWindow::SetBorders

void SetBorders(int b)

This function sets the space between border of window and HTML contents. See image:



Parameters

b

indentation from borders in pixels

wxHtmlWindow::SetFont

void SetFont(wxString normal_face, wxString fixed_face, const int *sizes)

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 (to)

Defaults

Under wxGTK:

```
SetFont("", "", {10, 12, 14, 16, 19, 24, 32});
```

Under Windows:

```
SetFont("", "", {7, 8, 10, 12, 16, 22, 30});
```

Although it seems different the fact is that the fonts are of approximately same size under both platforms (due to wxMSW / wxGTK inconsistency)

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. 545) 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. 548), this sets statusbar slot where messages will be displayed. (Default is -1 = no messages.)

Parameters

bar
statusbar slot number (0..n)

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 *SetFont*s, *SetBorder*s.

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. 530) and its main goal is to parse HTML input so that it can be displayed in *wxHtmlWindow* (p. 542). It uses a special *wxHtmlWinTagHandler* (p. 555).

Notes

The product of parsing is a *wxHtmlCell* (resp. *wxHtmlContainer*) object.

Derived from

wxHtmlParser (p. 530)

Include files

<wx/html/winpars.h>

See Also

Handlers overview (p. 1436)

wxHtmlWinParser::wxHtmlWinParser

wxHtmlWinParser()

wxHtmlWinParser(wxWindow *wnd)

Constructor. Don't use the default one, use constructor with *wnd* parameter (*wnd* is pointer to associated *wxHtmlWindow* (p. 542))

wxHtmlWinParser::AddModule

static void AddModule(wxHtmlTagsModule *module)

Adds *module* (p. 1436) 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. 1435)).

wxHtmlWinParser::CreateCurrentFont

virtual wxFont* CreateCurrentFont()

Creates font based on current setting (see *SetFontSize* (p. 554), *SetFontBold* (p. 553), *SetFontItalic* (p. 554), *SetFontFixed* (p. 554), *SetFontUnderlined* (p. 554)) 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 correctly opened container (see *Overview* (p. 1435)). 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. 371) class used to do conversion between *input encoding* (p. 552) and *output encoding* (p. 552).

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. 529) string if the parser is between `<A>` and `` 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

wxWindow* 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. 1435)).

wxHtmlWinParser::SetActualColor

void SetActualColor(const wxColour& clr)

Sets actual text colour. Note: this DOESN'T change the colour! You must create *wxHtmlColourCell* (p. 506) yourself.

wxHtmlWinParser::SetAlign

void SetAlign(int a)

Sets default horizontal alignment (see *wxHtmlContainerCell::SetAlignHor* (p. 508).) 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. 532)! *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 or FALSE.

wxHtmlWinParser::SetFontItalic

void SetFontItalic(int x)

Sets italic flag of actualfont. *x* is either TRUE or 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 or FALSE.

wxHtmlWinParser::SetFont

void SetFont(wxString normal_face, wxString fixed_face, const int *sizes)

Sets fonts. This method is identical to *wxHtmlWindow::SetFont* (p. 547)

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. 529) 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. 539)

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. 849)

Include files

<wx/protocol/http.h>

See also

wxSocketBase (p. 938), *wxURL* (p. 1164)

wxHTTP::GetInputStream

wxInputStream * GetInputStream(const wxString& path)

Creates a new input stream on the 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 Seekl 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. 1000). 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. 593): 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. 592)

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.

Constructor.

wxIdleEvent::RequestMore

void RequestMore(bool *needMore* = *TRUE*)

Tells wxWindows 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. 558), *wxApp::OnIdle* (p. 27)

wxIdleEvent::MoreRequested

bool MoreRequested() const

Returns TRUE if the OnIdle function processing this event requested more processing time.

See also

wxIdleEvent::RequestMore (p. 558), *wxApp::OnIdle* (p. 27)

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. 54)

wxGDIObject (p. 474)

wxObject (p. 746)

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. 1134) and *wxListCtrl* (p. 630).

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. 1388).

See also

Bitmap and icon overview (p. 1388), *supported bitmap file formats* (p. 1389), *wxDC::DrawIcon* (p. 285), *wxCursor* (p. 184)

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.

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

Creates a new icon.

wxlcon(char** *bits*)

wxlcon(const char** *bits*)

Creates an icon from XPM data.

wxlcon(const wxString& *name*, long *type*, int *desiredWidth* = -1, int *desiredHeight* = -1)

Loads an icon from a file or resource.

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.

desiredHeight

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.

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 wxWindows configuration. If all possible wxWindows 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 wxWindows 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::LoadFile (p. 562)

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 `wxWindows` 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. 562)

wxIcon::LoadFile

bool LoadFile(const wxString& name, long 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 wxWindows configuration.

Return value

TRUE if the operation succeeded, FALSE otherwise.

See also

wxIcon::wxIcon (p. 559)

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.

wxlcon::SetHeight

void SetHeight(int *height*)

Sets the height member (does not affect the icon data).

Parameters

height
Icon height in pixels.

wxlcon::SetOk

void SetOk(int *isOk*)

Sets the validity member (does not affect the icon data).

Parameters

isOk
Validity flag.

wxlcon::SetWidth

void SetWidth(int *width*)

Sets the width member (does not affect the icon data).

Parameters

width
Icon width in pixels.

wxlcon::operator =

wxlcon& operator =(const wxlcon& *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.

wxImage

This class encapsulates a platform-independent image. An image can be created from data, or using the constructor taking a *wxBitmap* object. 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. 280). Instead, a platform-specific *wxBitmap* (p. 54) object must be created from it using the *ConvertToBitmap* (p.

569) function. This bitmap can then be drawn in a device context, using `wxDC::DrawBitmap` (p. 284).

One colour value of the image may be used as a mask colour which will lead to the automatic creation of a `wxMask` (p. 659) object associated to the bitmap object.

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. 568) or `wxInitAllImageHandlers` (p. 1279).

wxBMPHandler	Only for loading, always installed.
wxPNGHandler	For loading 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.

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 PNM only works for ASCII or raw RGB images. When saving in PNM format, **wxPNMHandler** will always save as raw RGB.

Derived from

`wxObject` (p. 746)

Include files

<wx/image.h>

See also

`wxBitmap` (p. 54), `wxInitAllImageHandlers` (p. 1279)

wxImage::wxImage

wxImage()

Default constructor.

wxImage(const wxImage& image)

Copy constructor.

wxImage(const wxBitmap& *bitmap*)

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*)

Creates an image with the given width and height.

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)

wxImage(const wxString& *name*, const wxString& *mimetype*)

Loads an image from a file.

wxImage(wxInputStream& *stream*, long *type* = wxBITMAP_TYPE_ANY)

wxImage(wxInputStream& *stream*, const wxString& *mimetype*)

Loads an image from an input stream.

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_ANY	Will try to autodetect the format.

mimetype

MIME type string (for example 'image/jpeg')

Remarks

Depending on how wxWindows has been configured, not all formats may be available.

Note: any handler other than BMP must be previously initialized with *wxImage::AddHandler* (p. 568) or *wxInitAllImageHandlers* (p. 1279).

See also

wxImage::LoadFile (p. 573)

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

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. 580)

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 `wxWindows` on exit.

wxImage::ConvertToBitmap

wxBitmap ConvertToBitmap() const

Converts the image to a platform-specific bitmap object. 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 colour depth of the current system 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.

wxImage::Copy

wxImage Copy() const

Returns an identical copy of the image.

wxImage::Create

bool Create(int width, int height)

Creates a fresh image.

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

bool Destroy()

Destroys the image data.

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. 580)

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 RGBGBRGB... format.

wxImage::GetGreen

unsigned char GetGreen(int x, int y) const

Returns the green intensity at the given coordinate.

wxImage::GetRed

unsigned char GetRed(int x, int y) const

Returns the red intensity at the given coordinate.

wxImage::GetHandlers

static wxList& GetHandlers()

Returns the static list of image format handlers.

See also

wxImageHandler (p. 580)

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::GetPalette

const wxPalette& GetPalette() const

Returns the palette associated with the image. Currently the palette is only used in `ConvertToBitmap` under Windows.

Eventually `wxImage` handlers will set the palette if one exists in the image file.

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. 571)

wxImage::HasMask

bool HasMask() const

Returns TRUE if there is a mask active, FALSE otherwise.

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 *wxWindows* on startup, and shouldn't be called by the user.

See also

wxImageHandler (p. 580), *wxInitAllImageHandlers* (p. 1279)

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.

See also

wxImageHandler (p. 580)

wxImage::LoadFile

bool LoadFile(const *wxString&* name, long type = *wxBITMAP_TYPE_ANY*)

bool LoadFile(const *wxString&* name, const *wxString&* mimetype)

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)

bool LoadFile(wxInputStream& stream, const wxString& mimetype)

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_ANY	Will try to autodetect the format.

mimetype

MIME type string (for example 'image/jpeg')

Remarks

Depending on how wxWindows has been configured, not all formats may be available.

Return value

TRUE if the operation succeeded, FALSE otherwise.

See also

wxImage::SaveFile (p. 575)

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

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. 580)

wxImage::SaveFile

bool SaveFile(const wxString& name, int type)**bool SaveFile(const wxString& name, const wxString& mimetype)**

Saves a image in the named file.

bool SaveFile(wxOutputStream& stream, int type)**bool SaveFile(wxOutputStream& stream, const wxString& mimetype)**

Saves a 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 three types can be used:

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).

mimetype
MIME type.

Return value

TRUE if the operation succeeded, FALSE otherwise.

Remarks

Depending on how wxWindows has been configured, not all formats may be available.

See also

wxImage::LoadFile (p. 573)

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

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: after a call to this function, the image will have the given width and height.

Returns the (modified) image itself.

See also

Scale (p. 577)

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::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);
    bmp = image.Scale(32, 32).ConvertToBitmap();
}
```

```
    // another possibility:  
    image.Rescale(32, 32);  
    bmp = image;  
}
```

See also

Rescale (p. 576)

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.

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 blue, unsigned char green)

Sets the mask colour for this image (and tells the image to use the mask).

wxImage::SetPalette

void SetPalette(const wxPalette& palette)

Associates a palette with the image. Currently, the palette is not used.

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. 571) method.

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)

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)

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. 568) in your application initialisation.

Note (Legal Issue)

This software is based in part on the work of the Independent JPEG Group.

(Applies when wxWindows is linked with JPEG support. wxJPEGHandler uses libjpeg created by IJG.)

Derived from

wxObject (p. 746)

Include files

<wx/image.h>

See also

wxImage (p. 565), *wxInitAllImageHandlers* (p. 1279)

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 handles, this defaults to 1.

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. 573), *wxImage::SaveFile* (p. 575), *wxImageHandler::SaveFile* (p. 582)

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. 573), *wxImage::SaveFile* (p. 575), *wxImageHandler::LoadFile* (p.

581)

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. 1134) and `wxListCtrl` (p. 630) classes.

Derived from

`wxObject` (p. 746)

Include files

<wx/imaglist.h>

See also

`wxTreeCtrl` (p. 1134), `wxListCtrl` (p. 630)

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. 586)

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. 584) 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 focussed state.

solidBackground

For optimisation - drawing can be faster if the function is told that the background is solid.

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.

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. 746)

Include files

<wx/layout.h>

See also

Overview and examples (p. 1376), *wxLayoutConstraints* (p. 613), *wxWindow::SetConstraints* (p. 1223).

Edges and relationships

The `wxEdge` enumerated type specifies the type of edge or dimension of a window.

<code>wxLeft</code>	The left edge.
<code>wxTop</code>	The top edge.
<code>wxRight</code>	The right edge.
<code>wxBottom</code>	The bottom edge.
<code>wxCentreX</code>	The x-coordinate of the centre of the window.
<code>wxCentreY</code>	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.

<code>wxUnconstrained</code>	The edge or dimension is unconstrained (the default for edges).
<code>wxAsIs</code>	The edge or dimension is to be taken from the current window position or size (the default for dimensions).
<code>wxAbove</code>	The edge should be above another edge.
<code>wxBelow</code>	The edge should be below another edge.
<code>wxLeftOf</code>	The edge should be to the left of another edge.
<code>wxRightOf</code>	The edge should be to the right of another edge.
<code>wxSameAs</code>	The edge or dimension should be the same as another edge or dimension.
<code>wxPercentOf</code>	The edge or dimension should be a percentage of another edge or dimension.
<code>wxAbsolute</code>	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.

Derived from

wxEvent (p. 375)

wxObject (p. 746)

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

wxWindow::OnInitDialog (p. 1211), *Event handling overview* (p. 1364)

wxInitDialogEvent::wxInitDialogEvent

wxInitDialogEvent(int *id* = 0)

Constructor.

wxInputStream

wxInputStream is an abstract base class which may not be used directly.

Derived from

wxStreamBase (p. 998)

Include files

<wx/stream.h>

wxInputStream::wxInputStream

wxInputStream()

Creates a dummy input stream.

wxInputStream::~~wxInputStream

~wxInputStream()

Destructor.

wxInputStream::GetC

char GetC()

Returns the first character in the input queue and removes it.

wxInputStream::Eof

wxInputStream Eof() const

Returns TRUE if the end of stream has been reached.

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.

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::Seekl

off_t Seekl(off_t pos, wxSeekMode mode = wxFromStart)

Changes the stream current position.

wxInputStream::Tell

off_t Tell() 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 `Seekl()` on the same stream. Any call to `Seekl()` will invalidate any previous call to this method (otherwise you could `Seekl()` to one position, "unread" a few bytes there, `Seekl()` 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.

wxIntegerFormValidator

This class validates a range of integer values for a form view. The associated control must be a `wxTextCtrl` or `wxSlider`.

See also

Validator classes (p. 1453)

wxIntegerFormValidator::wxIntegerFormValidator

void wxIntegerFormValidator(long min=0, long max=0, long flags=0)

Constructor. Assigning zero to minimum and maximum values indicates that there is no range to check.

wxIntegerListValidator

This class validates a range of integer values for a list view.

See also

Validator classes (p. 1453)

wxIntegerListValidator::wxIntegerListValidator

void wxIntegerListValidator(long min=0, long max=0, long flags=wxPROP_ALLOW_TEXT_EDITING)

Constructor. Assigning zero to minimum and maximum values indicates that there is no range to check.

wxIPv4address

Derived from

wxSockAddress (p. 937)

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).

wxIPV4address::Hostname

wxString Hostname()

Returns the hostname which matches the IP address.

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).

wxIPV4address::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).

wxIPV4address::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. 597), 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. 596), 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. 746)

Include files

<wx/joystick.h>

See also

wxJoystickEvent (p. 604)

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. A bitlist of wxJOY_BUTTONn identifiers, where n is 1, 2, 3 or 4.

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 discrete units.

wxJoystick::GetPOVCTSPosition

int GetPOVCTSPosition() const

Returns the point-of-view position, expressed in continuous, one-hundredth of a degree units.

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. 603), *wxJoystickEvent* (p. 604)

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. 603), *wxJoystickEvent* (p. 604)

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. 375)

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.

See also

wxJoystick (p. 597)

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::ButtonsDown

bool ButtonsDown(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.

Derived from

wxEvent (p. 375)

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_CHAR(func)	Process a <i>wxEVT_CHAR</i> event (a non-modifier key has been pressed).
EVT_KEY_DOWN(func)	Process a <i>wxEVT_KEY_DOWN</i> event (any key has been pressed).
EVT_KEY_UP(func)	Process a <i>wxEVT_KEY_UP</i> event (any key has been released).
EVT_CHAR(func)	Process a <i>wxEVT_CHAR</i> event.
EVT_CHAR_HOOK(func)	Process a <i>wxEVT_CHAR_HOOK</i> event.

See also

wxWindow::OnChar (p. 1205), *wxWindow::OnCharHook* (p. 1206),
wxWindow::OnKeyDown (p. 1209), *wxWindow::OnKeyUp* (p. 1210)

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. 1304) 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::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. 1304) for a full list of the virtual key codes.

wxKeyEvent::GetX

long GetX() const

Returns the X position of the event.

wxKeyEvent::GetY

long GetY() const

Returns the Y position of the event.

wxKeyEvent::MetaDown

bool MetaDown() const

Returns TRUE if the Meta key was down at the time of the key event.

wxKeyEvent::GetPosition

wxPoint GetPosition() const**void GetPosition(long *x, long *y) const**

Obtains the position at which the key was pressed.

wxKeyEvent::HasModifiers

bool HasModifiers() const

Returns TRUE if either of CTRL, ALT or META keys was down at the time of the key event. Note that this function does not take into account the SHIFT key state.

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. 894) 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 inbetween 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. 746)

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 <code>wxEVT_QUERY_LAYOUT_INFO</code> event, to get size, orientation and alignment from a window. See <i>wxQueryLayoutInfoEvent</i> (p. 856).
EVT_CALCULATE_LAYOUT(func)	Process a <code>wxEVT_CALCULATE_LAYOUT</code> event, which asks the window to take a 'bite' out of a rectangle provided by the algorithm. See <i>wxCalculateLayoutEvent</i> (p. 94).

Data types

```
enum wxLayoutOrientation {
    wxLAYOUT_HORIZONTAL,
    wxLAYOUT_VERTICAL
};

enum wxLayoutAlignment {
    wxLAYOUT_NONE,
    wxLAYOUT_TOP,
    wxLAYOUT_LEFT,
    wxLAYOUT_RIGHT,
    wxLAYOUT_BOTTOM,
};
```

See also

wxSashEvent (p. 892), *wxSashLayoutWindow* (p. 894), *Event handling overview* (p. 1364)

wxCalculateLayoutEvent (p. 94), *wxQueryLayoutInfoEvent* (p. 856), *wxSashLayoutWindow* (p. 894), *wxSashWindow* (p. 897)

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. 613).

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.

wxLayoutConstraints

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.

Derived from

wxObject (p. 746)

Include files

<wx/layout.h>

See also

Overview and examples (p. 1376), *wxIndividualLayoutConstraint* (p. 588),
wxWindow::SetConstraints (p. 1223)

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 wxWindows, 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. See *wxHashTable* (p. 493) for a faster method of storage when random access is required.

While wxList class in the previous versions of wxWindows 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 stricter 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 wxWindows 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.

In the documentation of the list classes below, you should replace `wxNode` with `wxListName::Node` and `wxObject` with the list element type (i.e. the first parameter of `WX_DECLARE_LIST`) for the template lists.

Derived from

wxObject (p. 746)

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. 735) for members that retrieve the data associated with a node, and members for getting to the next or previous node.

See also

wxNode (p. 735), *wxStringList* (p. 1029), *wxArray* (p. 32)

wxList::wxList

wxList()

wxList(unsigned int *key_type*)

wxList(int *n*, wxObject **objects*[])

wxList(wxObject **object*, ...)

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 * Append(wxObject **object*)

wxNode * Append(long *key*, wxObject **object*)

wxNode * Append(const wxString& *key*, wxObject **object*)

Appends a new **wxNode** 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 *node)

Deletes the given node from the list, returning TRUE if successful.

wxList::DeleteObject

bool DeleteObject(wxObject *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::Find

wxNode * Find(long key)

wxNode * 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 * GetFirst()

Returns the first node in the list (NULL if the list is empty).

wxList::GetLast

wxNode * GetLast()

Returns the last node in the list (NULL if the list is empty).

wxList::IndexOf

int IndexOf(wxObject* obj)

Returns the index of *obj* within the list or NOT_FOUND if *obj* is not found in the list.

wxList::Insert

wxNode * Insert(wxObject *object)

Insert object at front of list.

wxNode * Insert(size_t position, wxObject *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. 619); if it is equal to it, this is the same as calling *Append* (p. 618).

wxNode * Insert(wxNode *node, wxObject *object)

Inserts the object before the given *node*.

wxList::Item

wxNode * Item(size_t index) const

Returns the node at given position in the list.

wxList::Member

wxNode * Member(wxObject *object)

NB: This function is deprecated, use *Find* (p. 619) instead.

Returns the node associated with *object* if it is in the list, NULL otherwise.

wxList::Nth

wxNode * Nth(int n)

NB: This function is deprecated, use *Item* (p. 620) 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. 619) 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. The sort function receives pointers to wxObject pointers (wxObject**), so be careful to dereference appropriately.

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);
}
```

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

wxControl (p. 176)
wxWindow (p. 1184)
wxEvtHandler (p. 378)
wxObject (p. 746)

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. No effect for GTK.

See also *window styles overview* (p. 1371).

Event handling

EVT_LISTBOX(id, func)	Process a <code>wxEVT_COMMAND_LISTBOX_SELECTED</code> event, when an item on the list is selected.
EVT_LISTBOX_DCLICK(id, func)	Process a <code>wxEVT_COMMAND_LISTBOX_DOUBLECLICKED</code> event, when the listbox is doubleclicked.

See also

wxChoice (p. 113), *wxComboBox* (p. 143), *wxListCtrl* (p. 630), *wxCommandEvent* (p. 152)

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")
```

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. 621).

validator

Window validator.

name

Window name.

See also

wxListBox::Create (p. 624), *wxValidator* (p. 1166)

wxPython note: The *wxListBox* constructor in wxPython reduces the `n` and `choices` arguments to a single argument, which is a list of strings.

wxListBox::~wxListBox

```
void ~wxListBox()
```

Destructor, destroying the list box.

wxListBox::Append

void Append(const wxString& item)

Adds the item to the end of the list box.

void Append(const wxString& item, void* clientData)

Adds the item to the end of the list box, associating the given data with the item.

Parameters

item

String to add.

clientData

Client data to associate with the item.

wxListBox::Clear

void Clear()

Clears all strings from 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")

Creates the listbox for two-step construction. See *wxListBox::wxListBox* (p. 622) for further details.

wxListBox::Delete

void Delete(int n)

Deletes an item from the listbox.

Parameters

n

The zero-based item index.

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::FindString

int FindString(const wxString& *string*)

Finds an item matching the given string.

Parameters

string
String to find.

Return value

The zero-based position of the item, or -1 if the string was not found.

wxListBox::GetClientData

void* GetClientData(int *n*) const

Returns a pointer to the client data associated with the given item (if any).

Parameters

n
The zero-based position of the item.

Return value

A pointer to the client data, or NULL if not present.

wxListBox::GetSelection

int GetSelection() const

Gets the position of the selected item.

Return value

The position of the current selection.

Remarks

Applicable to single selection list boxes only.

See also

wxListBox::SetSelection (p. 629), *wxListBox::GetStringSelection* (p. 627),
wxListBox::GetSelections (p. 626)

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

wxListBox::GetSelection (p. 626), *wxListBox::GetStringSelection* (p. 627),
wxListBox::SetSelection (p. 629)

wxPython note: The wxPython version of this method takes no parameters and returns a tuple of the selected items.

wxListBox::GetString

wxString GetString(int *n*) const

Returns the string at the given position.

Parameters

n
The zero-based position.

Return value

The string, or an empty string if the position was invalid.

wxListBox::GetStringSelection

wxString GetStringSelection() const

Gets the selected string - for single selection list boxes only. This must be copied by the calling program if long term use is to be made of it.

See also

wxListBox::GetSelection (p. 626), *wxListBox::GetSelections* (p. 626),
wxListBox::SetSelection (p. 629)

wxListBox::InsertItems

void InsertItems(int *nItems*, const wxString *items*, 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.

wxListBox::Number

int Number() const

Returns the number of items in the listbox.

wxListBox::Selected

bool Selected(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*)

Clears the list box and adds the given strings. Not implemented for GTK.

Parameters

n
The number of strings to set.

choices
An array of strings to set.

Remarks

Deallocate the array from the calling program after this function has been called.

wxListBox::SetClientData

void SetClientData(int *n*, void* *data*)

Associates the given client data pointer with the given item.

Parameters

n

The zero-based item index.

data

The client data to associate with the item.

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.

wxListBox::SetSelection

void SetSelection(int *n*, const bool *select* = TRUE)

Selects or deselects the given item. This does not cause a `wxEVT_COMMAND_LISTBOX_SELECT` event to get emitted.

Parameters

n

The zero-based item index.

select

If TRUE, will select the item. If FALSE, will deselect it.

wxListBox::SetString

void SetString(int *n*, const wxString& *string*)

Sets the string value of an item.

Parameters

n

The zero-based item index.

string

The string to set.

wxListBox::SetStringSelection**void SetStringSelection(const wxString& *string*, const bool *select* = TRUE)**

Sets the current selection. This does not cause a wxEVT_COMMAND_LISTBOX_SELECT event to get emitted.

Parameters*string*

The item to select.

select

If TRUE, will select the item. If FALSE, will deselect it.

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.

Using many of wxListCtrl is shown in the *corresponding sample* (p. 1325).To intercept events from a list control, use the event table macros described in *wxListEvent* (p. 644).**Derived from***wxControl* (p. 176)*wxWindow* (p. 1184)*wxEvtHandler* (p. 378)*wxObject* (p. 746)**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_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_USER_TEXT	The application provides label text on demand, except for column headers. Win32 only.
wxLC_EDIT_LABELS	Labels are editable: the application will be notified when editing starts.
wxLC_NO_HEADER	No header in report mode. Win32 only.
wxLC_SINGLE_SEL	Single selection.
wxLC_SORT_ASCENDING	Sort in ascending order (must still supply a comparison callback in <code>SortItems</code>).
wxLC_SORT_DESCENDING	Sort in descending order (must still supply a comparison callback in <code>SortItems</code>).

See also *window styles overview* (p. 1371).

Event handling

To process input from a list control, use these event handler macros to direct input to member functions that take a *wxCommandEvent* (p. 644) 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 <i>Veto()</i> (p. 746).
EVT_LIST_END_LABEL_EDIT(id, func)	Finish editing a label. This can be prevented by calling <i>Veto()</i> (p. 746).
EVT_LIST_DELETE_ITEM(id, func)	Delete an item.
EVT_LIST_DELETE_ALL_ITEMS(id, func)	Delete all items.
EVT_LIST_GET_INFO(id, func)	Request information from the application, usually the item text.
EVT_LIST_SET_INFO(id, func)	Information is being supplied (not implemented).
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_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.

See also

wxListCtrl overview (p. 1397), *wxListBox* (p. 621), *wxTreeCtrl* (p. 1134), *wxImageList* (p. 584), *wxCommandEvent* (p. 644)

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 = "listCtrl")

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. 630).

validator

Window validator.

name

Window name.

See also

wxListCtrl::Create (p. 633), *wxValidator* (p. 1166)

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::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* = "listCtrl")

Creates the list control. See *wxListCtrl::wxListCtrl* (p. 632) for further details.

wxListCtrl::ClearAll

void ClearAll()

Deletes all items and all columns.

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. 633)

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. 633)).

wxListCtrl::DeleteColumn

bool DeleteColumn(int col)

Deletes a column.

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 the item *start*.**long FindItem(long start, long data)**Find an item whose data matches this data, starting from the item *start*.**long FindItem(long start, const wxPoint& pt, int direction)**Find an item nearest this position in the specified direction, starting from the item *start*.

NB: The meaning of parameters *start* has changed in wxMSW version 2.2.0, previously the item with this index was excluded from the search but it is included now.

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)

wxListCtrl::GetColumn

bool GetColumn(int col, wxListItem& item) const

Gets information about this column. See *wxListCtrl::SetItem* (p. 641) for more information.

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

Gets the edit control for editing labels.

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. 641) for more information.

You must call `info.SetId()` to set 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.

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`.

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`.

wxListCtrl::GetItemState

int GetItemState(long item, long stateMask) const

Gets the item state. For a list of state flags, see `wxListCtrl::SetItem` (p. 641).

The **stateMask** indicates which state flags are of interest.

wxListCtrl::GetItemCount

int GetItemCount() const

Returns the number of items in the list control.

wxListCtrl::GetItemSpacing

int GetItemSpacing(bool *isSmall*) constRetrieves the spacing between icons in pixels. If *small* is TRUE, gets the spacing for the small icon view, otherwise the large icon view.

wxListCtrl::GetItemText

wxString GetItemText(long *item*) const

Gets the item text for this item.

wxListCtrl::GetNextItem

long GetNextItem(long *item*, int *geometry* = wxLIST_NEXT_ALL, int *state* = wxLIST_STATE_DONTCARE) constSearches 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.

<code>wxLIST_NEXT_LEFT</code>	Searches for an item to the left of the specified item.
<code>wxLIST_NEXT_RIGHT</code>	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:

<code>wxLIST_STATE_DONTCARE</code>	Don't care what the state is.
<code>wxLIST_STATE_DROPHILITED</code>	The item indicates it is a drop target.
<code>wxLIST_STATE_FOCUSED</code>	The item has the focus.
<code>wxLIST_STATE_SELECTED</code>	The item is selected.
<code>wxLIST_STATE_CUT</code>	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::HitTest

long HitTest(const wxPoint& *point*, int& *flags*)

Determines which item (if any) is at the specified point, giving details in *flags*. *flags* will be a combination of the following flags:

<code>wxLIST_HITTEST_ABOVE</code>	Above the client area.
<code>wxLIST_HITTEST_BELOW</code>	Below the client area.
<code>wxLIST_HITTEST_NOWHERE</code>	In the client area but below the last item.
<code>wxLIST_HITTEST_ONITEMICON</code>	On the bitmap associated with an item.
<code>wxLIST_HITTEST_ONITEMLABEL</code>	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**, and **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.

wxListCtrl::InsertColumn

long InsertColumn(long col, wxListItem& info)

For list view mode (only), inserts a column. For more details, see *wxListCtrl::SetItem* (p. 641).

long InsertColumn(long col, const wxString& heading, int format = wxLIST_FORMAT_LEFT, int width = -1)

For list view mode (only), inserts a column. For more details, see *wxListCtrl::SetItem* (p. 641).

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.
InsertColumnInfo(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.

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.

If in icon, small icon or list view mode, *dy* specifies the number of pixels to scroll. If in report view mode, *dy* specifies the number of lines to scroll.

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. 641) 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).

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:

<code>long m_mask</code>	Indicates which fields are valid. See the list of valid mask flags below.
<code>long m_itemId</code>	The zero-based item position.
<code>int m_col</code>	Zero-based column, if in report mode.
<code>long m_state</code>	The state of the item. See the list of valid state flags below.
<code>long m_stateMask</code>	A mask indicating which state flags are valid. See the list of valid state flags below.
<code>wxString m_text</code>	The label/header text.
<code>int m_image</code>	The zero-based index into an image list.
<code>long m_data</code>	Application-defined data.
<code>int m_format</code>	For columns only: the format. Can be

`int m_width` `wxLIST_FORMAT_LEFT`, `wxLIST_FORMAT_RIGHT` or
 `wxLIST_FORMAT_CENTRE`.
 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:

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.

`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.

long SetItem(long index, int col, const wxString& label, int imageld = -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, imageld) Sets a string or image at a given location.

wxListCtrl::SetItemData

bool SetItemData(long item, long data)

Associates application-defined data with this item.

wxListCtrl::SetItemImage

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.

wxListCtrl::SetItemPosition

bool SetItemPosition(long item, const wxPoint& pos)

Sets the position of the item, in icon or small icon view.

wxListCtrl::SetItemState

bool SetItemState(long item, long state, long stateMask)

Sets the item state. For a list of state flags, see *wxListCtrl::SetItem* (p. 641).

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::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.

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. 642) if you want to be able to sort the items in the control.

Please see the *listctrl sample* (p. 1325) 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.

wxListEvent

A list event holds information about events associated with `wxListCtrl` objects.

Derived from

wxNotifyEvent (p. 745)

wxCommandEvent (p. 152)

wxEvent (p. 375)

wxObject (p. 746)

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 <code>Veto()</code> (p. 746).
EVT_LIST_END_LABEL_EDIT(id, func)	Finish editing a label. This can be prevented by calling <code>Veto()</code> (p. 746).
EVT_LIST_DELETE_ITEM(id, func)	Delete an item.
EVT_LIST_DELETE_ALL_ITEMS(id, func)	Delete all items.
EVT_LIST_GET_INFO(id, func)	Request information from the application, usually the item text.
EVT_LIST_SET_INFO(id, func)	Information is being supplied (not implemented).
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_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.

See also

`wxListCtrl` (p. 630)

`wxListEvent::wxListEvent`

`wxListEvent(WXTYPE commandType = 0, int id = 0)`

Constructor.

`wxListEvent::GetCode`

`int GetCode() const`

Key code if the event is a keypress event.

wxCommandEvent::GetIndex

long GetIndex() const

The item index.

wxCommandEvent::GetOldIndex

long GetOldIndex() const

The old item index.

wxCommandEvent::GetColumn

int GetColumn() const

The column position.

wxCommandEvent::Cancelled

bool Cancelled() const

TRUE if this event is an end edit event and the user cancelled the edit.

wxCommandEvent::GetPoint

wxPoint GetPoint() const

The position of the mouse pointer if the event is a drag event.

wxCommandEvent::GetLabel

const wxString& GetLabel() const

The label.

wxCommandEvent::GetText

const wxString& GetText() const

The text.

wxCommandEvent::GetImage

int GetImage() const

The image.

wxCommandEvent::GetData

long GetData() const

The data.

wxCommandEvent::GetMask

long GetMask() const

The mask.

wxCommandEvent::GetItem

const wxListItem& GetItem() constAn item object, used by some events. See also *wxListCtrl::SetItem* (p. 641).**wxStringListValidator**

This class validates a list of strings for a list view. When editing the property, a dialog box is presented for adding, deleting or editing entries in the list. At present no constraints may be supplied.

You can construct a string list property value by constructing a `wxStringList` object.

For example:

```
myListValidatorRegistry.RegisterValidator((wxString)"stringlist",
    new wxStringListValidator);

wxStringList *strings = new wxStringList("earth", "fire", "wind",
"water", NULL);

sheet->AddProperty(new wxProperty("fred", strings, "stringlist"));
```

See also

Validator classes (p. 1453)

wxListOfStringsListValidator::wxListOfStringsListValidator

void wxListOfStringsListValidator(long flags=0)

Constructor.

wxLocale

wxLocale class encapsulates all language-dependent settings and is a generalization of the C locale concept.

In wxWindows this class manages message catalogs which contain the translations of the strings used to the current language.

Derived from

No base class

See also

18n overview (p. 1346)

Include files

<wx/intl.h>

wxLocale::wxLocale

wxLocale()

This is the default constructor and it does nothing to initialize the object: *Init()* (p. 650) must be used to do that.

wxLocale(const char *szName, const char *szShort = NULL, const char *szLocale = NULL, bool bLoadDefault = TRUE, bool bConvertEncoding = FALSE)

The parameters have the following meaning:

- szName is the name of the locale and is only used in diagnostic messages

- `szShort` is the standard 2 letter locale abbreviation and is used as the directory prefix when looking for the message catalog files
- `szLocale` is the parameter for the call to `setlocale()`
- `bLoadDefault` may be set to `FALSE` to prevent loading of the message catalog for the given locale containing the translations of standard `wxWindows` 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. 1347) 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()` will try to translate the messages using the message catalogs for this locale. Finally, unless `bLoadDefault` parameter is `FALSE`, the method also loads the `wxstd.mo` catalog (which is looked for in all the usual places and, additionally, under the location specified by the environment variable `WXDIR` if it is set) which will allow to translate all the messages generated by the library itself.

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. 650) documentation are rolled back.

wxLocale::AddCatalog

bool AddCatalog(const char *szDomain)

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. 649).

All loaded catalogs will be used for message lookup by `GetString()` 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).

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/LC_MESSAGES and prefix (in this order).

This only applies to subsequent invocations of `AddCatalog()`!

wxLocale::GetLocale

const char* GetLocale() const

Returns the locale name as passed to the constructor or *Init()* (p. 650).

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

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. 1353) to suppress it).

Remarks

Domains are searched in the last to first order, i.e. catalogs added later override those added before.

wxLocale::Init

bool Init(const char *szName, const char *szShort = NULL, const char *szLocale = NULL, bool bLoadDefault = TRUE)

The parameters have the following meaning:

- *szName* is the name of the locale and is only used in diagnostic messages
- *szShort* is the standard 2 letter locale abbreviation and is used as the directory prefix when looking for the message catalog files
- *szLocale* is the parameter for the call to *setlocale()*
- *bLoadDefault* may be set to `FALSE` to prevent loading of the message catalog for the given locale containing the translations of standard `wxWindows`

messages. This parameter would be rarely used in normal circumstances.

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.

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. 649)

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 `wxWindows` logging functions as explained in the *wxLog overview* (p. 1353). 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. 1353) for the descriptions of `wxWindows` 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()* is called by the *wxLogXXX()* functions and invokes the *DoLog()* of the active log target if any. Get/Set methods are used to install/query the current active target and, finally, *DontCreateOnDemand()* 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. 654)

GetActiveTarget (p. 654)

SetActiveTarget (p. 654)

DontCreateOnDemand (p. 654)

Message buffering

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()* shows them all and clears the buffer contents. Although this function doesn't do anything if the buffer is already empty, *HasPendingMessages()* is also provided which allows to explicitly verify it.

Flush (p. 654)

FlushActive (p. 655)

HasPendingMessages (p. 655)

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. 1298). 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. 656) and *GetTraceMask* (p. 656) and using *wxLogTrace* (p. 1299) which takes an integer mask or by using *AddTraceMask* (p. 654) 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. 1299) 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 % specifiers 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. 655) explicitly.

AddTraceMask (p. 654)
RemoveTraceMask (p. 656)
IsAllowedTraceMask (p. 656)
SetVerbose (p. 655)
GetVerbose (p. 655)
SetTimestamp (p. 655)
GetTimestamp (p. 655)
SetTraceMask (p. 656)
GetTraceMask (p. 656)

wxLog::AddTraceMask

static void AddTraceMask(const wxString& mask)

Add the *mask* to the list of allowed masks for *wxLogTrace* (p. 1299).

See also: *RemoveTraceMask* (p. 656)

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).

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.

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. 654)

wxLog::HasPendingMessages

bool HasPendingMessages() const

Returns true if there are any messages in the buffer (not yet shown to the user). (Almost) for internal use only.

wxLog::SetVerbose

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

bool GetVerbose() const

Returns whether the verbose mode is currently active.

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 specifiers, 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. 652) section for details.

wxLog::GetTraceMask

Returns the current trace mask, see *Customization* (p. 652) section for details.

wxLog::IsAllowedTraceMask

static bool IsAllowedTraceMask(const wxChar *mask)

Returns TRUE if the *mask* is one of allowed masks for *wxLogTrace* (p. 1299).

See also: *AddTraceMask* (p. 654), *RemoveTraceMask* (p. 656)

wxLog::RemoveTraceMask

static void RemoveTraceMask(const wxString& mask)

Remove the *mask* from the list of allowed masks for *wxLogTrace* (p. 1299).

See also: *AddTraceMask* (p. 654)

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 a native (i.e. supported directly by the compiler) 64 bit integer type was found a typedef `wxLongLong_t` will be defined to correspond it.

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 l)

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 l)

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).

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.

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::operator+

wxLongLong operator+(const wxLongLong& //) const

Adds 2 wxLongLongs together and returns the result.

wxLongLong::operator+=

wxLongLong& operator+(const wxLongLong& //)

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.

wxLongLong::operator-

wxLongLong operator-(const wxLongLong& //) const

Subtracts 2 wxLongLongs and returns the result.

wxLongLong::operator-=

wxLongLong& operator-(const wxLongLong& //)

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. 746)

Include files

<wx/bitmap.h>

Remarks

A mask may be associated with a *wxBitmap* (p. 54). It is used in *wxDC::Blit* (p. 281) when the source device context is a *wxMemoryDC* (p. 678) with *wxBitmap* selected into it that contains a mask.

See also

wxBitmap (p. 54), *wxDC::Blit* (p. 281), *wxMemoryDC* (p. 678)

wxMask::wxMask

wxMask()

Default constructor.

wxMask(const wxBitmap& *bitmap*)

Constructs a mask from a monochrome bitmap.

wxPython note: This is the default constructor for *wxMask* in *wxPython*.

wxMask(const wxBitmap& *bitmap*, const wxColour& *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.

wxMBCConv

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

wxCSCConv (p. 180), *wxEncodingConverter* (p. 371), *wxMBConv classes overview* (p. 1343)

wxMBConv::wxMBConv

wxMBConv()

Constructor.

wxMBConv::MB2WC

virtual size_t MB2WC(wchar_t* buf, const char* psz, size_t n) const

Converts from multibyte encoding to Unicode, using the libc routine `mbstowcs()` (this is overridden by derived classes). Returns the size of the destination buffer.

wxMBConv::WC2MB

virtual size_t WC2MB(char* buf, const wchar_t* psz, size_t n) const

Converts from Unicode to multibyte encoding, using the libc routine `wcstombs()` (this is overridden by derived classes). Returns the size of the destination buffer.

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.

wxMBCConv::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.

wxMBCConv::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).

wxMBCConv::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).

wxMBCConv::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).

wxMBCConv::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 converts file names between filesystem multibyte encoding and Unicode. It has one predefined instance, **wxConvFile**. Since some platforms (e.g. Win32) use Unicode in the filenames, and others (e.g. Unix) use multibyte encodings, this class should only be used directly if `wxMBFILES` is defined to 1. A convenience macro, `wxFNCONV`, is defined to `wxConvFile.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. 661)

Include files

<wx/strconv.h>

See also

wxMBConv classes overview (p. 1343)

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**. Unfortunately, this class is not quite implemented yet.

Derived from

wxMBConv (p. 661)

Include files

<wx/strconv.h>

See also

wxMBConvUTF8 (p. 665), *wxMBConv classes overview* (p. 1343)

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. 661)

Include files

<wx/strconv.h>

See also

wxMBConvUTF7 (p. 665), *wxMBConv classes overview* (p. 1343)

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 unambiguates 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.

wxMDIChildFrame

An MDI child frame is a frame that can only exist on a *wxMDIClientWindow* (p. 670), which is itself a child of *wxMDIParentFrame* (p. 671).

Derived from

wxFrame (p. 452)
wxWindow (p. 1184)
wxEvtHandler (p. 378)
wxObject (p. 746)

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 resizable 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. 1371).

Remarks

Although internally an MDI child frame is a child of the MDI client window, in *wxWindows* you create it as a child of *wxMDIParentFrame* (p. 671). 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. 670), *wxMDIParentFrame* (p. 671), *wxFrame* (p. 452)

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 *wxWindows*, depending on platform.

size

The window size. A value of (-1, -1) indicates a default size, chosen by either the windowing system or *wxWindows*, depending on platform.

style

The window style. See *wxMDIChildFrame* (p. 666).

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. 669)

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. 669), *wxMDIChildFrame::Restore* (p. 669)

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. 668) for further details.

wxMDIChildFrame::Maximize

void Maximize()

Maximizes this MDI child frame.

See also

wxMDIChildFrame::Activate (p. 669), *wxMDIChildFrame::Restore* (p. 669)

wxMDIChildFrame::Restore

void Restore()

Restores this MDI child frame (unmaximizes).

See also

wxMDIChildFrame::Activate (p. 669), *wxMDIChildFrame::Maximize* (p. 669)

wxMDIClientWindow

An MDI client window is a child of *wxMDIParentFrame* (p. 671), and manages zero or more *wxMDIChildFrame* (p. 666) objects.

Derived from

wxWindow (p. 1184)
wxEvtHandler (p. 378)
wxObject (p. 746)

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. 677) 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. 666), *wxMDIParentFrame* (p. 671), *wxFrame* (p. 452)

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. 677).

See also

wxMDIParentFrame::wxMDIParentFrame (p. 673), *wxMDIParentFrame::OnCreateClient* (p. 677)

wxMDIClientWindow::~~wxMDIClientWindow

~wxMDIClientWindow()

Destructor.

wxMDIClientWindow::CreateClient

bool CreateClient(*wxMDIParentFrame** parent, **long** style = 0)

Used in two-step frame construction. See *wxMDIClientWindow::wxMDIClientWindow* (p. 670) 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. 452)

wxWindow (p. 1184)

wxEvtHandler (p. 378)
wXObject (p. 746)

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 either *wxMDIChildFrame* (p. 666), or *wxFrame* (p. 452).

An MDI parent frame always has a *wxMDIClientWindow* (p. 670) 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 <i>client window</i> , 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 resizable 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 <i>client window</i> , 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. 1371).

See also

wxMDIChildFrame (p. 666), *wxMDIClientWindow* (p. 670), *wxFrame* (p. 452), *wxDialog* (p. 310)

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 *wxWindows*, depending on platform.

size

The window size. A value of (-1, -1) indicates a default size, chosen by either the windowing system or *wxWindows*, depending on platform.

style

The window style. See *wxMDIParentFrame* (p. 671).

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. 670), override *wxMDIParentFrame::OnCreateClient* (p. 677).

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. 675), *wxMDIParentFrame::OnCreateClient* (p. 677)

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. 674)

wxMDIParentFrame::ActivatePrevious

void ActivatePrevious()

Activates the MDI child preceding the currently active one.

See also

wxMDIParentFrame::ActivateNext (p. 674)

wxMDIParentFrame::ArrangeIcons

void ArrangeIcons()

Arranges any iconized (minimized) MDI child windows.

See also

wxMDIParentFrame::Cascade (p. 675), *wxMDIParentFrame::Tile* (p. 678)

wxMDIParentFrame::Cascade

void Cascade()

Arranges the MDI child windows in a cascade.

See also

wxMDIParentFrame::Tile (p. 678), *wxMDIParentFrame::ArrangeIcons* (p. 674)

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. 673) 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. 677), 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. 676), `wxMDIParentFrame::SetToolBar` (p. 677), `wxWindow` (p. 1216), `wxMDIClientWindow` (p. 670)

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. 677)

wxMDIParentFrame::GetToolBar

virtual wxWindow* GetToolBar() const

Returns the window being used as the toolbar for this frame.

See also

wxMDIParentFrame::SetToolBar (p. 677)

wxMDIParentFrame::GetWindowMenu

wxMenu* GetWindowMenu() const

Returns the current Window menu (added by wxWindows 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. 670) 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@ukonline.co.uk).

See also

wxMDIParentFrame::GetClientWindow (p. 676), *wxMDIClientWindow* (p. 670)

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. 675) 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. 676), `wxMDIParentFrame::GetClientSize` (p. 675)

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()

Tiles the MDI child windows.

See also

`wxMDIParentFrame::Cascade` (p. 675), `wxMDIParentFrame::ArrangeIcons` (p. 674)

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. 280)
wXObject (p. 746)

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. 54), *wxDC* (p. 280)

wxMemoryDC::wxMemoryDC

wxMemoryDC()

Constructs a new memory device context.

Use the *Ok* member to test whether the constructor was successful in creating a useable device context. Don't forget to select a bitmap into the DC before drawing on it.

wxMemoryDC::SelectObject

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. 285) 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. 422) 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;

    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. 424)

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.

Note that when storing image/bitmap, you must use image format that wxWindows can write (e.g. JPG, PNG, see *wxImage documentation* (p. 565))!

wxMemoryFSHandler::RemoveFile

static void RemoveFile(const wxString& filename)

Remove file from memory FS and free occupied memory.

wxMemoryInputStream

Derived from

wxInputStream (p. 592)

Include files

<wx/mstream.h>

See also

wxStreamBuffer (p. 1000), *wxMemoryOutputStream* (p. 682)

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::~~wxMemoryInputStream

~wxMemoryInputStream()

Destructor.

wxMemoryOutputStream

Derived from

wxOutputStream (p. 751)

Include files

<wx/mstream.h>

See also

wxStreamBuffer (p. 1000)

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.

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.

Derived from

wxEvtHandler (p. 378)

wxObject (p. 746)

Include files

```
<wx/menu.h>
```

Event handling

If the menu is part of a menubar, then *wxMenuBar* (p. 693) 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. 152). This method is deprecated and should not be used in the new code, it is provided for backwards compatibility only.

See also

wxMenuBar (p. 693), *wxWindow::PopupMenu* (p. 1217), *Event handling overview* (p. 1364)

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.

wxMenu(long style)

Constructs a wxMenu object.

Parameters

style

If set to wxMENU_TEAROFF, the menu will be detachable.

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

void Append(int *id*, const wxString& *item*, const wxString& *helpString* = "", const bool *checkable* = FALSE)

Adds a string item to the end of the menu.

void Append(int *id*, const wxString& *item*, wxMenu* *subMenu*, const wxString& *helpString* = "")

Adds a pull-right submenu to the end of the menu.

void 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.

checkable

If TRUE, this item is checkable.

helpString

An optional help string associated with the item. By default, `wxFrame::OnMenuHighlight` (p. 460) 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.

See also

wxMenu::AppendSeparator (p. 686), *wxMenu::Insert* (p. 690), *wxMenu::SetLabel* (p. 692), *wxMenu::GetHelpString* (p. 689), *wxMenu::SetHelpString* (p. 692), *wxMenuItem* (p. 702)

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::AppendSeparator

void AppendSeparator()

Adds a separator to the end of the menu.

See also

wxMenu::Append (p. 685)

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. 690)

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. 687) 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. 688), *wxMenu::Destroy* (p. 687), *wxMenu::Remove* (p. 691)

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. 691) 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. 688), *wxMenu::Deletes* (p. 687), *wxMenu::Remove* (p. 691)

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. 691)

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.

Parameters

itemString

Menu item string to find.

id

Menu item identifier.

menu

If the pointer is not NULL, it will be filled with the items 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::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. 692), *wxMenu::Append* (p. 685)

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. 692)

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.

See also

wxMenu::SetTitle (p. 692)

wxMenu::Insert

bool Insert(size_t pos, wxMenuItem *item)

Inserts the given *item* before the position *pos*. Inserting the item at the position *GetMenuItemCount* (p. 689) is the same as appending it.

See also

wxMenu::Append (p. 685)

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. 686)

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. 688)

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. 689)

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. 685), *wxMenu::GetLabel* (p. 689)

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.

See also

wxMenu::SetTitle (p. 692)

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. 1217), but the application may call it at other times if required.

See also

wxUpdateUIEvent (p. 1160)

wxMenuBar

A menu bar is a series of menus accessible from the top of a frame.

Derived from

wxEvtHandler (p. 378)

wxObject (p. 746)

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.

Note that menu commands (and UI update events for menus) are first sent to the focus window within the frame. If no window within the frame has the focus, then the events are sent directly to the frame. This allows command and UI update handling to be processed by specific windows and controls, and not necessarily by the application frame.

See also

wxMenu (p. 683), *Event handling overview* (p. 1364)

wxMenuBar::wxMenuBar

void wxMenuBar(long style = 0)

Default constructor.

void wxMenuBar(int n, wxMenu* menus[], const wxString titles[])

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` 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. 699)

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. 701)

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.

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. 702)

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. 698) is the same as calling *Append()* (p. 694).

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. 694)

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. 699) to change the menubar dynamically.

See also

`wxMenuBar::Replace` (p. 700)

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 the position *pos*. The caller is responsible for deleting it.

See also

wxMenuBar::Insert (p. 699), *wxMenuBar::Remove* (p. 700)

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. 697)

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. 697)

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. 698)

wxMenuItem

A menu item represents an item in a popup menu. Note that the majority of this class is only implemented under Windows so far, but everything except fonts, colours and bitmaps can be achieved via *wxMenu* on all platforms.

Derived from

wxOwnerDrawn (Windows only)

wxObject (p. 746)

Include files

<wx/menuitem.h>

See also

wxMenuBar (p. 693), *wxMenu* (p. 683)

wxMenuItem::wxMenuItem

wxMenuItem(**wxMenu*** *parentMenu* = *NULL*, **int** *id* = *ID_SEPARATOR*, **const wxString&** *text* = "", **const wxString&** *helpString* = "", **bool** *checkable* = *FALSE*, **wxMenu*** *subMenu* = *NULL*,)

Constructs a wxMenuItem object.

Parameters

parentMenu

Menu that the menu item belongs to.

id

Identifier for this menu item, or ID_SEPARATOR to indicate a separator.

text

Text for the menu item, as shown on the menu.

helpString

Optional help string that will be shown on the status bar.

checkable

TRUE if this menu item is checkable.

subMenu

If non-NULL, indicates that the menu item is a submenu.

wxMenuItem::~~wxMenuItem

~wxMenuItem()

Destructor.

wxMenuItem::Check

void Check(**bool** *check*)

Checks or unchecks the menu item.

wxMenuItem::DeleteSubMenu

void DeleteSubMenu()

Deletes the submenu, if any.

wxMenuItem::Enable

void Enable(bool *enable*)

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::GetLabel

wxString GetLabel() const

Returns the text associated with the menu item without any accelerator characters it might contain.

See also

GetText (p. 705), *GetLabelFromText* (p. 705)

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. 705), *GetLabel* (p. 704)

wxMenuItem::GetMarginWidth

int GetMarginWidth() const

Gets the width of the menu item checkmark bitmap (Windows only).

wxMenuItem::GetName

wxString GetName() const

Returns the text associated with the menu item.

NB: this function is deprecated, please use *GetText* (p. 705) or *GetLabel* (p. 704) 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. 704), *GetLabelFromText* (p. 705)

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::SetBackgroundColour

void SetBackgroundColour(const wxColour& colour) const

Sets the background colour associated with the menu item (Windows only).

wxMenuItem::SetBitmaps

void SetBitmaps(const wxBitmap& *checked*, const wxBitmap& *unchecked* = wxNullBitmap) const

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*) const

Sets the font associated with the menu item (Windows only).

wxMenuItem::SetHelp

void SetHelp(const wxString& *helpString*) const

Sets the help string.

wxMenuItem::SetMarginWidth

void SetMarginWidth(int *width*) const

Sets the width of the menu item checkmark bitmap (Windows only).

wxMenuItem::SetName

void SetName(const wxString& *text*) const

Sets the text associated with the menu item.

wxMenuItem::SetTextColour

void SetTextColour(const wxColour& *colour*) const

Sets the text colour associated with the menu item (Windows only).

wxMenuEvent

This class is used for a variety of menu-related events. Note that these do not include

menu command events.

Derived from

wxEvt (p. 375)

wxObject (p. 746)

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.

EVT_MENU_CHAR(func)	Process a <i>wxEVT_MENU_CHAR</i> event (a keypress when a menu is showing). Windows only; not yet implemented.
EVT_MENU_INIT(func)	Process a <i>wxEVT_MENU_INIT</i> event (the menu is about to pop up). Windows only; not yet implemented.
EVT_MENU_HIGHLIGHT(func)	Process a <i>wxEVT_MENU_HIGHLIGHT</i> event (a menu item is being highlighted). Windows only; not yet implemented.
EVT_POPUP_MENU(func)	Process a <i>wxEVT_POPUP_MENU</i> event (a menu item is being highlighted). Windows only; not yet implemented.
EVT_CONTEXT_MENU(func)	Process a <i>wxEVT_CONTEXT_MENU</i> event (F1 has been pressed with a particular menu item highlighted). Windows only; not yet implemented.

See also

wxWindow::OnMenuHighlight (p. 1212), *Event handling overview* (p. 1364)

wxMenuEvent::wxMenuEvent

wxMenuEvent(WXTYPE id = 0, int id = 0, wxDC* dc = NULL)

Constructor.

wxMenuEvent::m_menuId

int m_menuId

The relevant menu identifier.

wxMenuEvent::GetMenuId

int GetMenuId() const

Returns the menu identifier associated with the event.

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. 310)
wxWindow (p. 1184)
wxEvtHandler (p. 378)
wxObject (p. 746)

Include files

<wx/msgdlg.h>

See also

wxMessageDialog overview (p. 1401)

wxMessageDialog::wxMessageDialog

wxMessageDialog(*wxWindow** parent, **const wxString&** message, **const wxString&** caption = "Message box", **long** style = wxOK | wxCANCEL | wxCENTRE, **const wxPoint&** pos = wxDefaultPosition)

Constructor. Use *wxMessageDialog::ShowModal* (p. 710) 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.
wxCENTRE	Centre the message. Not Windows.
wxICON_EXCLAMATION	Shows an exclamation mark icon.
wxICON_HAND	Shows a hand icon.
wxICON_QUESTION	Shows a question mark icon.
wxICON_INFORMATION	Shows an information (i) icon.

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 `wxWindows`, 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. 746)

Include files

<wx/metafile.h>

See also

wxMetafileDC (p. 712)

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. 711) 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 an 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. 713) 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. 710)).

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. 1262) after closing your disk-based metafile device context.

Derived from

`wxDC` (p. 280)
`wxObject` (p. 746)

Include files

<wx/metafile.h>

See also

`wxMetafile` (p. 710), `wxDC` (p. 280)

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).

wxMimeTypeManager

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 `wxMimeTypeManager::GetFileTypeFromMimeType` (p. 716) and `wxMimeTypeManager::GetFileTypeFromExtension` (p. 715) may be called: they will return a `wxFileType` (p. 427) 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 `wxMimeTypeManager::ReadMailcap` (p. 716) and `wxMimeTypeManager::ReadMimeType` (p. 716) are provided to load additional files.

If GNOME or KDE desktop environment is installed, then `wxMimeTypeManager` gathers MIME information from respective files (e.g. `.kdeInk` files under KDE).

NB: Currently, `wxMimeTypeManager` is limited to reading MIME type information but it will support modifying it as well in the future versions.

Global objects

Global instance of `wxMimeTypeManager` is always available. It is defined as follows:

```
wxMimeTypeManager *wxTheMimeTypeManager;
```

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. 427)

Helper functions

All of these functions are static (i.e. don't need a `wxMimeTypeManager` 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. 716)

Constructor and destructor

NB: You won't normally need to use more than one `wxMimeTypeManager` object in a program.

wxMimeTypeManager (p. 715)

~wxMimeTypeManager (p. 715)

Query database

These functions are the heart of this class: they allow to find a *file type* (p. 427) 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. 716)

GetFileTypeFromExtension (p. 715)

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. 716)

ReadMimeType (p. 716)

AddFallbacks (p. 715)

wxMimeTypeManager::wxMimeTypeManager

wxMimeTypeManager()

Constructor puts the object in the "working" state, no additional initialization are needed - but *ReadXXX* (p. 715) may be used to load additional mailcap/mime.types files.

wxMimeTypeManager::~~wxMimeTypeManager

~wxMimeTypeManager()

Destructor is not virtual, so this class should not be derived from.

wxMimeTypeManager::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.

wxMimeTypeManager::GetFileTypeFromExtension

wxFileType* GetFileTypeFromExtension(const wxString& extension)

Gather information about the files with given extension and return the corresponding *wxFileType* (p. 427) object or NULL if the extension is unknown.

wxMimeTypeManager::GetFileTypeFromMimeType

wxFileType* GetFileTypeFromMimeType(const wxString& mimeType)

Gather information about the files with given MIME type and return the corresponding *wxFileType* (p. 427) object or NULL if the MIME type is unknown.

wxMimeTypeManager::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 done by this function is case insensitive so it is not necessary to convert the strings to the same case before calling it.

wxMimeTypeManager::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.

wxMimeTypeManager::ReadMimeType

bool ReadMimeType(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. 452)
wxWindow (p. 1184)
wxEvtHandler (p. 378)
wxObject (p. 746)

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 .
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).
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	Displays a small horizontal caption. Use instead of wxCAPTION .
wxTINY_CAPTION_VERT	Under Windows, displays a small vertical caption. Use instead of wxCAPTION .
wxRESIZE_BORDER	Displays a resizable border around the window (Motif only; for Windows, it is implicit in wxTHICK_FRAME).

See also *window styles overview* (p. 1371). 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. 671), *wxMDIChildFrame* (p. 666), *wxFrame* (p. 452), *wxDialog* (p. 310)

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 wxWindows, depending on platform.

size

The window size. A value of (-1, -1) indicates a default size, chosen by either the windowing system or wxWindows, depending on platform.

style

The window style. See *wxMiniFrame* (p. 716).

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. 719)

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. 718) for further details.

wxModule

The module system is a very simple mechanism to allow applications (and parts of wxWindows itself) to define initialization and cleanup functions that are automatically called on wxWindows 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, wxWindows will find all classes derived from wxModule, create an instance of each, and call each OnInit function. On exit, wxWindows 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. 746)

Include files

<wx/module.h>

wxModule::wxModule

wxModule()

Constructs a wxModule object.

wxModule::~~wxModule

~wxModule()

Destructor.

wxModule::CleanupModules

static void CleanupModules()

Calls Exit for each module instance. Called by wxWindows on exit, so there is no need for an application to call it.

wxModule::Exit

void Exit()

Calls OnExit. This function is called by wxWindows and should not need to be called by an application.

wxModule::Init

bool Init()

Calls OnInit. This function is called by wxWindows and should not need to be called by an application.

wxModule::InitializeModules

static bool InitializeModules()

Calls Init for each module instance. Called by wxWindows on startup, so there is no need for an application to call it.

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, wxWindows will exit immediately.

wxModule::RegisterModule

static void RegisterModule(wxModule* module)

Registers this module with wxWindows. Called by wxWindows on startup, so there is no need for an application to call it.

wxModule::RegisterModules

static bool RegisterModules()

Creates instances of and registers all modules. Called by wxWindows on startup, so there is no need for an application to call it.

wxMouseEvent

This event class contains information about mouse events. See *wxWindow::OnMouseEvent* (p. 1213).

NB: Note that under Windows mouse enter and leave events are not natively supported by the system but are generated by wxWindows 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. 727) and *LeftIsDown* (p. 727): 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. 727) to test whether the left mouse button is (still) depressed. Also, by convention, if *LeftDown* (p. 727) returns `TRUE`, *LeftIsDown* (p. 727) will also return `TRUE` in `wxWindows` 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. 375)

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 <code>wxEVT_LEFT_DOWN</code> event.
EVT_LEFT_UP(func)	Process a <code>wxEVT_LEFT_UP</code> event.
EVT_LEFT_DCLICK(func)	Process a <code>wxEVT_LEFT_DCLICK</code> event.
EVT_MIDDLE_DOWN(func)	Process a <code>wxEVT_MIDDLE_DOWN</code> event.
EVT_MIDDLE_UP(func)	Process a <code>wxEVT_MIDDLE_UP</code> event.
EVT_MIDDLE_DCLICK(func)	Process a <code>wxEVT_MIDDLE_DCLICK</code> event.
EVT_RIGHT_DOWN(func)	Process a <code>wxEVT_RIGHT_DOWN</code> event.
EVT_RIGHT_UP(func)	Process a <code>wxEVT_RIGHT_UP</code> event.
EVT_RIGHT_DCLICK(func)	Process a <code>wxEVT_RIGHT_DCLICK</code> event.
EVT_MOTION(func)	Process a <code>wxEVT_MOTION</code> event.
EVT_ENTER_WINDOW(func)	Process a <code>wxEVT_ENTER_WINDOW</code> event.
EVT_LEAVE_WINDOW(func)	Process a <code>wxEVT_LEAVE_WINDOW</code> 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_leftDown

bool m_leftDown

TRUE if the left 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::wxMouseEvent

wxMouseEvent(WXTYPE *mouseEventType* = 0, int *id* = 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**

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 1, 2 or 3 for left, middle and right buttons respectively.

Not all mice have middle buttons so a portable application should avoid this one.

wxMouseEvent::ButtonDClick

bool ButtonDClick(int but = -1)

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 (1, 2 or 3 for left, middle and right buttons respectively).

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 (1, 2 or 3 for left, middle and right buttons respectively).

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 (1, 2 or 3 for left, middle and right buttons respectively).

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).

wxMouseEvent::Entering

bool Entering()

Returns TRUE if the mouse was entering the window.

See also *wxMouseEvent::Leaving* (p. 726).

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.

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::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::Leaving

bool Leaving() const

Returns TRUE if the mouse was leaving the window.

See also *wxMouseEvent::Entering* (p. 725).

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. 727) 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 (no buttons depressed).

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. 375)

wxObject (p. 746)

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 <code>wxEVT_MOVE</code> event, which is generated when a window is moved.
-----------------------	---

See also

wxWindow::OnMove (p. 1213), *wxPoint* (p. 785), *Event handling overview* (p. 1364)

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.

wxMultipleChoiceDialog

This class represents a dialog that shows a list of strings, and allows the user to select one or more.

NOTE: this class is not yet implemented.

Derived from

wxDialog (p. 310)

wxWindow (p. 1184)

wxEvtHandler (p. 378)

wxObject (p. 746)

Include files

<wx/choicdlg.h>

See also

wxMultipleChoiceDialog overview (p. 1402)

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. Only one thread at a time can own a mutex object.

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.

Derived from

None.

Include files

<wx/thread.h>

See also

wxThread (p. 1101), *wxCondition* (p. 160), *wxMutexLocker* (p. 733), *wxCriticalSection* (p. 178)

wxMutex::wxMutex

wxMutex()

Default constructor.

wxMutex::~~wxMutex

~wxMutex()

Destroys the wxMutex object.

wxMutex::IsLocked

bool IsLocked() const

Returns TRUE if the mutex is locked, FALSE otherwise.

wxMutex::Lock

wxMutexError Lock()

Locks the mutex object.

Return value

One of:

wxMUTEX_NO_ERROR
wxMUTEX_DEAD_LOCK
wxMUTEX_BUSY

There was no error.
A deadlock situation was detected.
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
wxMUTEX_DEAD_LOCK
wxMUTEX_BUSY

There was no error.
A deadlock situation was detected.
The mutex is already locked by another thread.

wxMutex::Unlock

wxMutexError Unlock()

Unlocks the mutex object.

Return value

One of:

wxMUTEX_NO_ERROR
wxMUTEX_DEAD_LOCK
wxMUTEX_BUSY
wxMUTEX_UNLOCKED

There was no error.
A deadlock situation was detected.
The mutex is already locked by another thread.
The calling thread tries to unlock an unlocked mutex.

wxMutexLocker

This is a small helper class to be used with *wxMutex* (p. 730) 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. 730) for an example of *wxMutexLocker* usage.

Derived from

None.

Include files

<wx/thread.h>

See also

wxMutex (p. 730), *wxCriticalSectionLocker* (p. 179)

wxMutexLocker::wxMutexLocker

wxMutexLocker(**wxMutex** *mutex)

Constructs a *wxMutexLocker* object associated with *mutex* which must be non-NULL and locks it. Call *IsLocked* (p. 734) to check if the mutex was successfully locked.

wxMutexLocker::~~wxMutexLocker

~wxMutexLocker()

Destuctor 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.

wxNotebookSizer

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.

In order to query the size of notebook page, this page needs to have its own sizer, otherwise the *wxNotebookSizer* will ignore it. Notebook pages get there sizer by assigning one to them using *wxWindow::SetSizer* (p. 1230) and setting the auto-layout option to TRUE using *wxWindow::SetAutoLayout* (p. 1221). Here is one example showing how to add a notebook page that the notebook sizer is aware of:

```
wxNotebook *notebook = new wxNotebook( &dialog, -1 );
wxNotebookSizer *nbs = new wxNotebookSizer( notebook );
```

```

// Add panel as notebook page
wxPanel *panel = new wxPanel( notebook, -1 );
notebook->AddPage( panel, "My Notebook Page" );

wxBoxSizer *panelsizer = new wxBoxSizer( wxVERTICAL );

// Add controls to panel and panelsizer here...

panel->SetAutoLayout( TRUE );
panel->SetSizer( panelsizer );

```

See also *wxSizer* (p. 924), *wxNotebook* (p. 736).

Derived from

wxSizer (p. 924)
wxObject (p. 746)

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.

wxNodeBase

A node structure used in linked lists (see *wxList* (p. 615)) and derived classes. You should never use *wxNodeBase* class directly because it works with untyped (void *) data and this is unsafe. Use *wxNode*-derived classes which are defined by `WX_DECLARE_LIST` and `WX_DEFINE_LIST` macros instead as described in *wxList* (p. 615) documentation (see example there). *wxNode* is defined for compatibility as *wxNodeBase* containing "wxObject *" pointer, but usage of this class is deprecated.

Derived from

None.

Include files

<wx/list.h>

See also

wxList (p. 615), *wxHashTable* (p. 493)

wxNodeBase::GetData

void * Data()

Retrieves the client data pointer associated with the node.

wxNodeBase::GetNext

wxNodeBase * Next()

Retrieves the next node (NULL if at end of list).

wxNodeBase::GetPrevious

wxNodeBase * GetPrevious()

Retrieves the previous node (NULL if at start of list).

wxNodeBase::SetData

void SetData(void *data)

Sets the data associated with the node (usually the pointer will have been set when the node was created).

wxNodeBase::IndexOf

int IndexOf()

Returns the zero-based index of this node within the list. The return value will be `NOT_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. 738) or *InsertPage* (p. 741), 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. 176)
wxWindow (p. 1184)
wxEvtHandler (p. 378)
wxObject (p. 746)

Include files

<wx/notebook.h>

Window styles

<code>wxNB_FIXEDWIDTH</code>	(Windows only) All tabs will have same width.
<code>wxNB_LEFT</code>	Place tabs on the left side.
<code>wxNB_RIGHT</code>	Place tabs on the right side.
<code>wxNB_BOTTOM</code>	Place tabs under instead of above the notebook pages.

See also *window styles overview* (p. 1371).

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. 743) argument.

<code>EVT_NOTEBOOK_PAGE_CHANGED(id, func)</code>	The page selection was changed.
<code>EVT_NOTEBOOK_PAGE_CHANGING(id, func)</code>	The page selection is about to be changed. This can be prevented by calling <i>Veto()</i> (p. 746).

See also

wxNotebookEvent (p. 743), *wxImageList* (p. 584), *wxTabCtrl* (p. 1052)

wxNotebook::wxNotebook

wxNotebook()

Default constructor.

wxNotebook(*wxWindow** parent, **wxWindowID** id, **const wxPoint&** pos = *wxDefaultPosition*, **const wxSize&** size, **long** style = 0, **const wxString&** name = "notebook")

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. 736).

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.

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. 741)

wxNotebook::AdvanceSelection

void AdvanceSelection(bool *forward* = TRUE)

Cycles through the tabs.

wxNotebook::Create

bool Create(wxWindow* *parent*, wxWindowID *id*, const wxPoint& *pos* = wxDefaultPosition, const wxSize& *size*, long *style* = 0, const wxString& *name* = "notebook")

Creates a notebook control. See *wxNotebook::wxNotebook* (p. 737) for a description of the parameters.

wxNotebook::DeleteAllPages

bool DeleteAllPages()

Deletes all pages.

wxNotebook::DeletePage

bool DeletePage(int page)

Deletes the specified page, and the associated window.

wxNotebook::GetImageList

wxImageList* GetImageList() const

Returns the associated image list.

See also

wxImageList (p. 584), *wxNotebook::SetImageList* (p. 742)

wxNotebook::GetPage

wxNotebookPage* GetPage(int page)

Returns the window at the given page position.

wxNotebook::GetPageCount

int GetPageCount() const

Returns the number of pages in the notebook control.

wxNotebook::GetPageImage

int GetPageImage(int nPage) const

Returns the image index for the given page.

wxNotebook::GetPageText

wxString GetPageText(int 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.

wxNotebook::InsertPage

bool InsertPage(int *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. 738)

wxNotebook::OnSelChange

void OnSelChange(wxNotebookEvent& event)

An event handler function, called when the page selection is changed.

See also

wxNotebookEvent (p. 743)

wxNotebook::RemovePage

bool RemovePage(int page)

Deletes the specified page, without deleting the associated window.

wxNotebook::SetImageList

void SetImageList(wxImageList* imageList)

Sets the image list for the page control.

See also

wxImageList (p. 584)

wxNotebook::SetPadding

void SetPadding(const wxSize& padding)

Sets the amount of space around each page's icon and label, in pixels.

wxNotebook::SetPageSize

void SetPageSize(const wxSize& size)

Sets the width and height of the pages.

wxNotebook::SetPageImage

bool SetPageImage(int 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. 742).

wxNotebook::SetPageText

bool SetPageText(int page, const wxString& text)

Sets the text for the given page.

wxNotebook::SetSelection

int SetSelection(int page)

Sets the selection for the given page, returning the previous selection.

See also

wxNotebook::GetSelection (p. 741)

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 to the program to examine the current page (which can be retrieved with *GetOldSelection()* (p. 744)) and to veto the page change by calling *Veto()* (p. 746) 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.

NB: under Windows, *GetSelection()* will return the same value as *GetOldSelection()* when called from `PAGE_CHANGING` handler and not the page which is going to be selected if the handler doesn't call *Veto()*.

Derived from

wxNotifyEvent (p. 745)
wxCommandEvent (p. 152)
wxEvent (p. 375)
wxEvtHandler (p. 378)
wxObject (p. 746)

Include files

<wx/notebook.h>

Event table macros

To process a notebook event, use these event handler macros to direct input to member functions that take a `wxNotebookEvent` 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. 746).

See also

wxNotebook (p. 736), *wxTabCtrl* (p. 1052), *wxTabEvent* (p. 1058)

`wxNotebookEvent::wxNotebookEvent`

`wxNotebookEvent(wxEventType eventType = wxEVT_NULL, int id = 0, int sel = -1, int oldSel = -1)`

Constructor (used internally by `wxWindows` 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.

`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. 744)

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. 743)).

It (or an object of a derived class) is sent when the controls state is being changed and allows the program to *Veto()* (p. 746) this change if it wants to prevent it from happening.

In some rare cases (so far only dragging the items in the tree control) the event is disabled by default in which case *Allow()* (p. 746) may be called to allow it.

If both *Allow()* (p. 746) and *Veto()* (p. 746) are called, only the last method called counts.

Derived from

wxCommandEvent (p. 152)

wxEvent (p. 375)

wxEvtHandler (p. 378)

wxObject (p. 746)

Include files

<wx/event.h>

Event table macros

None

See also

wxNotebookEvent (p. 743)

wxNotifyEvent::wxNotifyEvent

wxNotifyEvent(**wxEventType** *eventType* = *wxEVT_NULL*, **int** *id* = 0)

Constructor (used internally by wxWindows only).

wxNotifyEvent::Allow

void Allow(**bool** *allow* = *TRUE*)

Allow the action signalled by this event to be carried.

wxNotifyEvent::IsAllowed

bool IsAllowed() **const**

Returns TRUE if the change is allowed (*Veto()* (p. 746) 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 wxWindows 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. 119), *Debugging overview* (p. 1356), *wxObjectRefData* (p. 750)

wxObject::wxObject

wxObject()

Default constructor.

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. 748), *wxObject::UnRef* (p. 749), *wxObject::SetRefData* (p. 749), *wxObject::GetRefData* (p. 748), *wxObjectRefData* (p. 750)

wxObject::Dump

void Dump(ostream& stream)

A virtual function that should be redefined by derived classes to allow dumping of memory states.

Parameters

stream

Stream on which to output dump information.

Remarks

Currently *wxWindows* 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 just writes the class name of the object in debug build (`__WXDEBUG__` defined), otherwise it does nothing.

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.

See also

wxObject::Ref (p. 748), *wxObject::UnRef* (p. 749), *wxObject::m_refData* (p. 747), *wxObject::SetRefData* (p. 749), *wxObjectRefData* (p. 750)

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. 749) 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. 749), *wxObject::m_refData* (p. 747), *wxObject::SetRefData* (p. 749), *wxObject::GetRefData* (p. 748), *wxObjectRefData* (p. 750)

wxObject::SetRefData

void SetRefData(wxObjectRefData* data)

Sets the *m_refData* pointer.

See also

wxObject::Ref (p. 748), *wxObject::UnRef* (p. 749), *wxObject::m_refData* (p. 747), *wxObject::GetRefData* (p. 748), *wxObjectRefData* (p. 750)

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. 748), *wxObject::m_refData* (p. 747), *wxObject::SetRefData* (p. 749), *wxObject::GetRefData* (p. 748), *wxObjectRefData* (p. 750)

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. 746)

See also

wxObject (p. 746)

wxObjectRefData::m_count

int m_count

Reference count. When this goes to zero during a *wxObject::UnRef* (p. 749), 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. 998)

Include files

<wx/stream.h>

wxOutputStream::wxOutputStream**wxOutputStream()**

Creates a dummy wxOutputStream object.

wxOutputStream::~~wxOutputStream**~wxOutputStream()**

Destructor.

wxOutputStream::LastWrite**size_t LastWrite() const**

Returns the number of bytes written during the last Write().

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)

Changes the stream current position.

wxOutputStream::TellO

off_t TellO() const

Returns the current stream position.

wxOutputStream::Write

wxOutputStream& Write(const void *buffer, size_t size)Writes the specified amount of bytes using the data of *buffer*. *WARNING!* The buffer absolutely needs to have at least the specified size.

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.

wxPageSetupDialogDataThis class holds a variety of information related to *wxPageSetupDialog* (p. 758).It contains a *wxPrintData* (p. 792) 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. 746)**Include files**

<wx/cmndata.h>

See also

wxPageSetupDialog (p. 758)

wxPageSetupDialogData::wxPageSetupDialogData

wxPageSetupDialogData()

Default constructor.

wxPageSetupDialogData(wxPageSetupDialogData& data)

Copy constructor.

wxPrintDialogData(wxPrintData& printData)

Construct an object from a print dialog 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. 795).

wxPageSetupDialogData::GetPaperSize

wxSize GetPaperSize() const

Returns the paper size in millimetres.

wxPageSetupDialogData::GetPrintData

wxPrintData& GetPrintData()

Returns a reference to the *print data* (p. 792) associated with this object.

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 GetMarginTopLeft(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. 795).

Calling this function overrides the explicit paper dimensions passed in *wxPageSetupDialogData::SetPaperSize* (p. 757).

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. 792) 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.

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 wxWindows 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. 752) 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. 310)
wxWindow (p. 1184)
wxEvtHandler (p. 378)
wxObject (p. 746)

Include files

<wx/printdlg.h>

See also

wxPrintDialog (p. 797), *wxPageSetupDialogData* (p. 752)

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. 752) associated with the dialog.

wxPageSetupDialog::ShowModal**int ShowModal()**

Shows the dialog, returning `wxID_OK` if the user pressed OK, and `wxID_CANCEL` otherwise.

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. 120) object.

To draw on the whole window including decorations, construct a `wxWindowDC` (p. 1234) object (Windows only).

Derived from

`wxWindowDC` (p. 1234)

`wxDC` (p. 280)

Include files

<wx/dcclient.h>

See also

wxDC (p. 280), *wxMemoryDC* (p. 678), *wxPaintDC* (p. 759), *wxWindowDC* (p. 1234), *wxScreenDC* (p. 901)

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. 375)

wxObject (p. 746)

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

wxWindow::OnPaint (p. 1214), *Event handling overview* (p. 1364)

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. 474)
wxObject (p. 746)

Include files

<wx/palette.h>

Predefined objects

Objects:

wxNullPalette

See also

wxDC::SetPalette (p. 294), *wxBitmap* (p. 54)

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. 762)

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. 761)

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. 763)

wxPalette::GetRGB

bool GetPixel(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. 763)

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. 1184)
wxEvtHandler (p. 378)
wxObject (p. 746)

Include files

<wx/panel.h>

Window styles

There are no specific styles for this window.

See also *window styles overview* (p. 1371).

Remarks

By default, a panel has the same colouring as a dialog.

A panel may be loaded from a `wxWindows` resource file (extension `wxr`).

See also

wxDialog (p. 310)

wxPanel::wxPanel

wxPanel()

Default constructor.

wxPanel(*wxWindow** parent, **wxWindowID** id, **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 wxWindows, depending on platform.

size

The panel size. A value of (-1, -1) indicates a default size, chosen by either the windowing system or wxWindows, depending on platform.

style

The window style. See *wxPanel* (p. 764).

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. 766)

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. 765) for details.

wxPanel::GetDefaultItem

wxButton* 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 an *wxWindow::OnInitDialog* (p. 1211) event, which in turn transfers data to the dialog via validators.

See also

wxWindow::OnInitDialog (p. 1211)

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 *wxWindow::OnSysColourChanged* (p. 1217) to propagate the notification to child windows and controls.

See also

wxSysColourChangedEvent (p. 1034)

wxPanel::SetDefaultItem

void SetDefaultItem(wxButton *btn)

Changes the default button for the panel.

See also

GetDefaultItem (p. 766)

wxPanelTabView

The *wxPanelTabView* is responsible for input and output on a *wxPanel*.

Derived from

wxTabView (p. 1044)

wxObject (p. 746)

Include files

<wx/tab.h>

See also

wxTabView overview (p. 1412), *wxTabView* (p. 1044)

wxPanelTabView::wxPanelTabView

void wxPanelTabView(wxPanel *panel, long style = wxTAB_STYLE_DRAW_BOX | wxTAB_STYLE_COLOUR_INTERIOR)

Constructor. *panel* should be a *wxTabbedPanel* or *wxTabbedDialog*: the type will be checked by the view at run time.

style may be a bit list of the following:

wxTAB_STYLE_DRAW_BOX	Draw a box around the view area. Most commonly used for dialogs.
wxTAB_STYLE_COLOUR_INTERIOR	Draw tab backgrounds in the specified colour. Omitting this style will ensure that the tab background matches the dialog background.

wxPanelTabView::~~wxPanelTabView

void ~wxPanelTabView()

Destructor. This destructor deletes all the panels associated with the view. If you do not wish this to happen, call *ClearWindows* with argument *FALSE* before the view is likely to be destroyed. This will clear the list of windows, without deleting them.

wxPanelTabView::AddTabWindow

void AddTabPanel(int id, wxWindow *window)

Adds a window to the view. The window is associated with the tab identifier, and will be shown or hidden as the tab is selected or deselected.

wxPanelTabView::ClearWindows

void ClearWindows(bool *deleteWindows* = TRUE)

Removes the child windows from the view. If *deleteWindows* is TRUE, the windows will be deleted.

wxPanelTabView::GetCurrentWindow

wxPanel * GetCurrentWindow()

Returns the child window currently being displayed on the tabbed panel or dialog box.

wxPanelTabView::GetTabWindow

wxWindow * GetTabWindow(int *id*)

Returns the window associated with the tab identifier.

wxPanelTabView::ShowWindowForTab

void ShowWindowForTab(int *id*)

Shows the child window corresponding to the tab identifier, and hides the previously shown window.

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. 615)

wxObject (p. 746)

Include files

<wx/filefn.h>

See also

wxList (p. 615)

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` 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::FindAbsolutePath

wxString FindAbsolutePath(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 exists, a temporary pointer to the absolute path 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 exists, a temporary pointer to the full path 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. 474)

wxObject (p. 746)

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, `wxWindows` 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. 778).

`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. 778), *wxDC* (p. 280), *wxDC::SetPen* (p. 297)

wxPen::wxPen

wxPen()

Default constructor. The pen will be uninitialised, and *wxPen::Ok* (p. 775) will return FALSE.

wxPen(const wxColour& colour, int width, int style)

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 <i>wxPen::SetDashes</i> (p. 776).
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. 777), *wxPen::SetColour* (p. 776), *wxPen::SetWidth* (p. 777), *wxPen::SetStipple* (p. 777)

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 wxWindows 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. 776)

wxPen::GetColour

wxColour& GetColour() const

Returns a reference to the pen colour.

See also

wxPen::SetColour (p. 776)

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. 776)

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. 776)

wxPen::GetStipple

wxBitmap* GetStipple() const

Gets a pointer to the stipple bitmap.

[See also](#)

wxPen::SetStipple (p. 777)

wxPen::GetStyle

int GetStyle() const

Returns the pen style.

[See also](#)

wxPen::wxPen (p. 772), *wxPen::SetStyle* (p. 777)

wxPen::GetWidth

int GetWidth() const

Returns the pen width.

[See also](#)

wxPen::SetWidth (p. 777)

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. 774)

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. 774)

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. 774)

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. 775)

wxPen::SetStipple

void SetStipple(`wxBitmap*` *stipple*)

Sets the bitmap for stippling.

See also

`wxPen::GetStipple` (p. 775)

wxPen::SetStyle

void SetStyle(`int` *style*)

Set the pen style.

See also

`wxPen::wxPen` (p. 772)

wxPen::SetWidth

void SetWidth(`int` *width*)

Sets the pen width.

See also

`wxPen::GetWidth` (p. 775)

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 wxWindows 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 wxWindows to keep track of pens in order to clean them up on exit. It is also kept for backward compatibility with earlier versions of wxWindows.

See also

wxPen (p. 771)

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 wxWindows 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. 140).

width
Width of pen.

style
Pen style. See *wxPen::wxPen* (p. 772) for a list of styles.

wxPenList::RemovePen

void RemovePen(wxPen* pen)

Used by wxWindows to remove a pen from the list.

wxPlotCurve

The `wxPlotCurve` class represents a curve displayed in a `wxPlotWindow` (p. 782). It is a virtual curve, i.e. it acts only as an interface, leaving it to the programmer to care for how the values pairs are matched. `wxPlotWindow` and `wxPlotCurve` are designed to display large amounts of data, i.e. most typically data measured by some sort of machine.

This class is abstract, i.e. you have to derive your own class and implement the pure virtual functions (`GetStartX()` (p. 781), `GetEndX()` (p. 780) and `GetY()` (p. 780)).

Derived from

`wxObject` (p. 746)

wxPlotCurve::wxPlotCurve

wxPlotCurve(int offsetY, double startY, double endY)

Constructor assigning start values. See below for interpretation.

wxPlotCurve::GetEndX

wxInt32 GetEndX()

Must be overridden. This function should return the index of the last value of this curve, typically 99 if 100 values have been measured.

wxPlotCurve::GetEndY

double GetEndY()

See `SetStartY` (p. 781).

wxPlotCurve::GetOffsetY

int GetOffsetY()

Returns the vertical offset.

wxPlotCurve::GetY

double GetY(wxInt32 x)

Must be overridden. This function will return the actual Y value corresponding to the given X value. The x value is of an integer type because it is considered to be an index in row of measured values.

wxPlotCurve::GetStartX

wxInt32 GetStartX()

Must be overridden. This function should return the index of the first value of this curve, typically zero.

wxPlotCurve::GetStartY

double GetStartY()

See *SetStartY* (p. 781).

wxPlotCurve::SetEndY

void SetEndY(double endY)

The value returned by this function tells the plot window what the highest values in the curve will be so that a suitable scale can be found for the display. If the Y values in this curve are in the range of -1.5 to 0.5, this function should return 0.5 or maybe 1.0 for nicer aesthetics.

wxPlotCurve::SetOffsetY

void SetOffsetY(int offsetY)

When displaying several curves in one window, it is often useful to assign different offsets to the curves. You should call *wxPlotWindow::Move* (p. 784) to set this value after you have added the curve to the window.

wxPlotCurve::SetStartY

void SetStartY(double startY)

The value returned by this function tells the plot window what the lowest values in the curve will be so that a suitable scale can be found for the display. If the Y values in this curve are in the range of -1.5 to 0.5, this function should return -1.5 or maybe -2.0 for nicer aesthetics.

wxPlotWindow

wxPlotWindow is a specialized window designed to display data that typically has been measured by machines, i.e. that may have thousands of values. One example of such data would be the well known ECG measuring the electrical activity of your heart: the measuring device will produce thousands of values per minute, several measurements are done simultaneously and you might want to have a look at parts of the curves, enlarging them or scrolling from one position to another. Note that this window is not useful for real-time measuring or for displaying charts with error bars etc.

A single curve in the plot window is represented by the *wxPlotCurve* (p. 780) class.

The wxPlotWindow interacts with program using events, for example when clicking or double clicking on a curve or when selecting one by clicking on it (which can be vetoed). Future versions will hopefully feature selecting values or sections of the displayed curves etc.

Derived from

wxScrolledWindow (p. 911)

wxPanel (p. 764)

wxWindow (p. 1184)

wxEvtHandler (p. 378)

wxObject (p. 746)

Window styles

wxPLOT_BUTTON_MOVE	Display buttons to allow moving individual curves up or down.
wxPLOT_BUTTON_ENLARGE	Display buttons to allow enlarging individual curves vertically.
wxPLOT_BUTTON_ZOOM	Display buttons to allow zooming all curves horizontally.
wxPLOT_BUTTON_ALL	Display all buttons.
wxPLOT_Y_AXIS	Display an Y axis to the left of the drawing area.
wxPLOT_X_AXIS	Display a X axis at the bottom of the drawing area.
wxPLOT_DEFAULT	All of the above options.

wxPlotWindow::wxPlotWindow

wxPlotWindow()

wxPlotWindow(wxWindow* parent, wxWindowID id, const wxPoint& pos, const

wxSize& *size*, **int** *flags* = *wxPLOT_DEFAULT*)

Constructor.

wxPlotWindow::~~wxPlotWindow

~wxPlotWindow()

The destructor will not delete the curves associated to the window.

wxPlotWindow::Add

void Add(wxPlotCurve* curve)

Add a curve to the window.

wxPlotWindow::GetCount

size_t GetCount()

Returns number of curves.

wxPlotWindow::GetAt

wxPlotCurve* GetAt(size_t n)

Get the nth curve.

wxPlotWindow::SetCurrent

void SetCurrent(wxPlotCurve* current)

Make one curve the current curve. This will emit a wxPlotEvent.

wxPlotWindow::GetCurrent

wxPlotCurve* GetCurrent()

Returns a pointer to the current curve, or NULL.

wxPlotWindow::Delete

void Delete(wxPlotCurve* curve)

Removes a curve from the window and delete is on screen. This does not delete the actual curve. If the curve removed was the current curve, the current curve will be set to NULL.

wxPlotWindow::Move

void Move(wxPlotCurve* curve, int pixels_up)

Move the curve *curve* up by *pixels_up* pixels. Down if the value is negative.

wxPlotWindow::Enlarge

void Enlarge(wxPlotCurve* curve, double factor)

Changes the representation of the given curve. A *factor* of more than one will stretch the curve vertically. The Y axis will change accordingly.

wxPlotWindow::SetUnitsPerValue

void SetUnitsPerValue(double upv)

This sets the virtual units per value. Normally, you will not be interested in what measured value you see, but what it stands for. If you want to display seconds on the X axis and the measuring device produced 50 values per second, set this value to 50. This will affect all curves being displayed.

wxPlotWindow::GetUnitsPerValue

double GetUnitsPerValue()

See *SetUnitsPerValue* (p. 784).

wxPlotWindow::SetZoom

void SetZoom(double zoom)

This functions zooms all curves in their horizontal dimension. The X axis will be changed accordingly.

wxPlotWindow::GetZoom

double GetZoom()

See *SetZoom* (p. 784).

wxPlotWindow::RedrawEverything

void RedrawEverything()

Helper function which redraws both axes and the central area.

wxPlotWindow::RedrawXAxis

void RedrawXAxis()

Helper function which redraws the X axis.

wxPlotWindow::RedrawYAxis

void RedrawYAxis()

Helper function which redraws the Y axis.

wxPlotWindow::SetScrollOnThumbRelease

void SetScrollOnThumbRelease(bool *onrelease* = TRUE)

This function controls if the plot area will get scrolled only if the scrollbar thumb has been release or also if the thumb is being dragged. When displaying large amounts of data, it might become impossible to display the data fast enough to produce smooth scrolling and then this function should be called.

wxPlotWindow::SetEnlargeAroundWindowCentre

void SetEnlargeAroundWindowCentre(bool *aroundwindow* = TRUE)

Depending on the kind of data you display, enlarging the individual curves might have different desired effects. Sometimes, the data will be supposed to get enlarged with the fixed point being the origin, sometimes the fixed point should be the centre of the current drawing area. This function controls this behaviour.

wxPoint

A **wxPoint** is a useful data structure for graphics operations. It simply contains integer *x* and *y* members.

See also *wxRealPoint* (p. 868) for a floating point version.

Derived from

None

Include files

<wx/gdicmn.h>

See also

wxRealPoint (p. 868)

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 wxWindows Encapsulated PostScript device context, which can write PostScript files on any platform. See *wxDC* (p. 280) for descriptions of the member

functions.

Derived from

wxDC (p. 280)

wXObject (p. 746)

Include files

<wx/dcps.h>

wxPostScriptDC::wxPostScriptDC

wxPostScriptDC(const wxPrintData& *printData*)

Constructs a PostScript printer device context from a *wxPrintData* (p. 792) 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 useable device context.

See *Printer settings* (p. 1263) 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::GetResolution

static int GetResolution()

Return resolution used in PostScript output. See *SetResolution* (p. 787).

wxPostScriptDC::SetResolution

static void SetResolution(int *ppi*)

Set resolution (in pixels per inch) that will be used in PostScript output. Default is 720ppi.

wxPreviewCanvas

A preview canvas is the default canvas used by the print preview system to display the preview.

Derived from

wxScrolledWindow (p. 911)
wxWindow (p. 1184)
wxEventHandler (p. 378)
wxObject (p. 746)

Include files

<wx/print.h>

See also

wxPreviewFrame (p. 790), *wxPreviewControlBar* (p. 788), *wxPrintPreview* (p. 810)

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. 813) 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. 764)
wxWindow (p. 1184)
wxEvtHandler (p. 378)
wxObject (p. 746)

Include files

<wx/print.h>

See also

wxPreviewFrame (p. 790), *wxPreviewCanvas* (p. 787), *wxPrintPreview* (p. 810)

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. 452)
wxWindow (p. 1184)
wxEvtHandler (p. 378)
wxObject (p. 746)

Include files

<wx/print.h>

See also

wxPreviewCanvas (p. 787), *wxPreviewControlBar* (p. 788), *wxPrintPreview* (p. 810)

wxPreviewFrame::wxPreviewFrame

wxPreviewFrame(*wxPrintPreview** preview, *wxFrame** parent, **const wxString&** title, **const wxPoint&** pos = *wxDefaultPosition*, **const wxSize&** size = *wxDefaultSize*, **long** style = *wxDEFAULT_FRAME_STYLE*, **const wxString&** name = "frame")

Constructor. Pass a print preview object plus other normal frame arguments.

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. 746)

Include files

<wx/cmndata.h>

See also

`wxPrintDialog` (p. 797), `wxPageSetupDialog` (p. 758), `wxPrintDialogData` (p. 799), `wxPageSetupDialogData` (p. 752), `wxPrintDialog Overview` (p. 1400), `wxPrinterDC` (p. 806), `wxPostScriptDC` (p. 786)

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::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. 795).

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

wxPaperQuality 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::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_LETTERSMAIL,  // 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(wxPaperQuality 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. 806) device context from a successfully dismissed print dialog.

Derived from

`wxDialog` (p. 310)
`wxWindow` (p. 1184)
`wxEvtHandler` (p. 378)
`wxObject` (p. 746)

Include files

<wx/printdlg.h>

See also

wxPrintDialog Overview (p. 1400)

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. 799)

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. 799) 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. 798).

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. 746)

Include files

<wx/cmndata.h>

See also

`wxPrintDialog` (p. 797), `wxPrintDialog Overview` (p. 1400)

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::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).

Note that the setup dialog is (according to Microsoft) obsolete from Windows 95, though retained for backward compatibility.

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. 746)

Include files

<wx/print.h>

See also

Printing framework overview (p. 1419), *wxPrinterDC* (p. 806), *wxPrintDialog* (p. 797), *wxPrintout* (p. 807), *wxPrintPreview* (p. 810).

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. 799), *wxPrintData* (p. 792)

wxPrinter::~~wxPrinter

~wxPrinter()

Destructor.

wxPrinter::Abort

bool Abort()

Returns TRUE if the user has aborted the print job.

wxPrinter::CreateAbortWindow

void CreateAbortWindow(wxWindow* parent, wxPrintout* printout)

Creates the default printing abort window, with a cancel button.

wxPrinter::GetLastError

static wxPrinterError GetLastError()

Return last error. Valid after calling *Print* (p. 805), *PrintDialog* (p. 805) or *wxPrintPreview::Print* (p. 813). 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. 792) 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. 804) 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. 804) 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. 280) for further information on device contexts, and *wxDC::GetSize* (p. 290) for advice on achieving the correct scaling for the page.

Derived from

wxDC (p. 280)

wXObject (p. 280)

Include files

<wx/dcprint.h>

See also

wxDC (p. 280), *Printing framework overview* (p. 1419)

wxPrinterDC::wxPrinterDC

wxPrinterDC(const wxPrintData& printData)

Pass a *wxPrintData* (p. 792) object with information necessary for setting up a suitable printer device context. This is the recommended way to construct a *wxPrinterDC*.

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 useable 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. 746)

Include files

<wx/print.h>

See also

Printing framework overview (p. 1419), *wxPrinterDC* (p. 806), *wxPrintDialog* (p. 797), *wxPrinter* (p. 803), *wxPrintPreview* (p. 810)

wxPrintout::wxPrintout

`wxPrintout(const wxString& title = "Printout")`

Constructor. Pass an optional title argument (currently unused).

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.

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.

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. 290) 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.

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.

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.

wxPython note: This method returns the output-only parameters as a tuple.

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. 746)

Include files

<wx/print.h>

See also

Printing framework overview (p. 1419), *wxPrinterDC* (p. 806), *wxPrintDialog* (p. 797), *wxPrintout* (p. 807), *wxPrinter* (p. 803), *wxPreviewCanvas* (p. 787), *wxPreviewControlBar* (p. 788), *wxPreviewFrame* (p. 790).

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::DrawBlankPage

bool DrawBlankPage(**wxWindow*** *window*)

Draws a representation of the blank page into the preview window. Used internally.

wxPrintPreview::GetCanvas

wxWindow* 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::GetPrintData

wxPrintData& GetPrintData()

Returns a reference to the internal print data.

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(wxWindow* window)

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. 804) 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(wxWindow* 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.

wxPrivateDropTarget

wxPrivateDropTarget is for...

Derived from

wxDropTarget (p. 368)

Include files

<wx/dnd.h>

See also

wxDropTarget (p. 368)

wxPrivateDropTarget::wxPrivateDropTarget

wxPrivateDropTarget()

wxPrivateDropTarget::SetId

void SetId(const wxString& id)

You have to override `OnDrop` to get at the data. The string ID identifies the format of clipboard or DnD data. A word processor would e.g. add a `wxTextDataObject` and a `wxPrivateDataObject` to the clipboard - the latter with the Id "WXWORD_FORMAT".

wxPrivateDropTarget::GetId

virtual wxString GetId() const

wxProcess

The objects of this class are used in conjunction with `wxExecute` (p. 1273) function. When a `wxProcess` object is passed to `wxExecute()`, its `OnTerminate()` (p. 817) 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. 816) for others).

`wxProcess` also supports IO redirection of the child process. For this, you have to call its `Redirect` (p. 817) method before passing it to `wxExecute` (p. 1273). If the child process was launched successfully, `GetInputStream` (p. 817), `GetOutputStream` (p. 817) and `GetErrorStream` (p. 817) can then be used to retrieve the streams corresponding to the child process standard output, input and error output respectively.

Derived from

wxEvtHandler (p. 378)

Include files

<wx/process.h>

See also

wxExecute (p. 1273)
exec sample (p. 1323)

wxProcess::wxProcess

wxProcess(wxEvtHandler * parent = NULL, int id = -1)

Constructs a process object. *id* is only used in the case you want to use wxWindows 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*.

Parameters

parent
The event handler parent.

id
id of an event.

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 a output 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. 817).

wxProcess::GetOutputStream

wxOutputStream* GetOutputStream() const

It returns an output stream corresponding to the input stream of the subprocess. If it is `NULL`, you have not turned on the redirection. See *wxProcess::Redirect* (p. 817).

wxProcess::OnTerminate

void OnTerminate(int *pid*, int *status*) const

It is called when the process with the `pid` *pid* finishes. It raises a `wxWindows` event when it isn't overridden.

pid

The `pid` of the process which has just terminated.

status

The exit code of the process.

wxProcess::Redirect

void Redirect()

It turns on the 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.

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. 452)
`wxWindow` (p. 1184)
`wxEvtHandler` (p. 378)
`wxObject` (p. 746)

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.

message

Message to show on the dialog.

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	By default, the progress dialog will disappear from screen as soon as the maximum value of the progress meter has been reached. This flag prevents it from doing it - instead the dialog will wait until the user closes it.
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 <i>Update()</i> (p. 819) will return FALSE.
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.
wxPD_SMOOTH	This flag tells the dialog that it should use smooth gauge (has effect only under 32bit Windows).

wxProgressDialog::~wxProgressDialog

~wxMessageDialog()

Destructor. Deletes the dialog and enables all top level windows.

wxProgressDialog::Update

bool Update(int value = -1, const char * newmsg = NULL,)

Updates the dialog, setting the progress bar to the new value and, if given changes the message above it. Returns TRUE if the ABORT button has *not* 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. 820) function.

value

The new value of the progress meter. It must be strictly less than the maximum value given to the constructor (i.e., as usual in C, the index runs from 0 to maximum-1).

newmsg

The new messages for the progress dialog text, if none is given the message is not changed.

wxProgressDialog::Resume

void Resume()

Can be used to continue with the dialog, after the user had chosen ABORT.

wxProcessEvent

A process event is sent when a process is terminated.

Derived from

wxEvent (p. 375)

wxObject (p. 746)

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. 815), *Event handling overview* (p. 1364)

wxProcessEvent::wxProcessEvent

wxProcessEvent(int *id* = 0, int *pid* = 0)

Constructor. Takes a wxProcessObject or window id, and a process id.

wxProcessEvent::m_pid

int m_pid

Contains the process id.

wxProcessEvent::GetPid

int GetPid() const

Returns the process id.

wxProcessEvent::SetPid

void SetPid(int *pid*)

Sets the process id.

wxProperty

The **wxProperty** class represents a property, with a *wxPropertyValue* (p. 841) containing the actual value, a name a role, an optional validator, and an optional associated window.

A property might correspond to an actual C++ data member, or it might correspond to a conceptual property, such as the width of a window. There is no explicit data member *wxWindow::width*, but it may be convenient to invent such a property for the purposes of editing attributes of the window. The properties in the property sheet can be mapped to "reality" by whatever means (in this case by calling *wxWindow::SetSize* when the user has finished editing the property sheet).

A validator may be associated with the property in order to ensure that this and only this validator will be used for editing and validating the property. An alternative method is to

use the *role* parameter to specify what kind of validator would be appropriate; for example, specifying "filename" for the role would allow the property view to find an appropriate validator at edit time.

wxProperty::wxProperty

void wxProperty()

void wxProperty(wxProperty& prop)

**void wxProperty(wxString name, wxString role, wxPropertyValidator
*validator=NULL)**

**void wxProperty(wxString name, const wxPropertyValue& val, wxString role,
wxPropertyValidator *validator=NULL)**

Constructors.

wxProperty::~~wxProperty

void ~wxProperty()

Destructor. Destroys the wxPropertyValue, and the property validator if there is one. However, if the actual C++ value in the wxPropertyValue is a pointer, the data in that variable is not destroyed.

wxProperty::GetValue

wxPropertyValue& GetValue()

Returns a reference to the property value.

wxProperty::GetValidator

wxPropertyValidator * GetValidator()

Returns a pointer to the associated property validator (if any).

wxProperty::GetName

wxString& GetName()

Returns the name of the property.

wxProperty::GetRole

wxRole& GetRole()

Returns the role of the property, to be used when choosing an appropriate validator.

wxProperty::GetWindow

wxWindow * GetWindow()

Returns the window associated with the property (if any).

wxProperty::SetValue

void SetValue(wxPropertyValue& *val*)

Sets the value of the property.

wxProperty::SetName

void SetName(wxString& *name*)

Sets the name of the property.

wxProperty::SetRole

void SetRole(wxString& *role*)

Sets the role of the property.

wxProperty::SetValidator

void SetValidator(wxPropertyValidator **validator*)

Sets the validator: this will be deleted when the property is deleted.

wxProperty::SetWindow

void SetWindow(wxWindow **win*)

Sets the window associated with the property.

wxProperty::operator =

```
void operator =(const wxPropertyValue& val)
```

Assignment operator.

wxPropertyFormDialog

The **wxPropertyFormDialog** class is a prepackaged dialog which can be used for viewing a form property sheet. Pass a property form view object, and the dialog will pass OnClose and OnDefaultAction listbox messages to the view class for processing.

wxPropertyFormDialog::wxPropertyFormDialog

```
void wxPropertyFormDialog(wxPropertyFormView *view, wxWindow *parent, char
 *title, bool modal=FALSE, int x=-1, int y=-1, int width=-1, int height=-1, long
 style=wxDEFAULT_DIALOG_STYLE, char *name="dialogBox")
```

Constructor.

wxPropertyFormDialog::~wxPropertyFormDialog

```
void ~wxPropertyFormDialog()
```

Destructor.

wxPropertyFormFrame

The **wxPropertyFormFrame** class is a prepackaged frame which can be used for viewing a property form. Pass a property form view object, and the frame will pass OnClose messages to the view class for processing.

Call Initialize to create the panel and associate the view; override OnCreatePanel if you wish to use a panel class other than the default wxPropertyFormPanel.

wxPropertyFormFrame::wxPropertyFormFrame

```
void wxPropertyFormFrame(wxPropertyFormView *view, wxFrame *parent, char
*title, int x=-1, int y=-1, int width=-1, int height=-1, long style=wxSDI |
wxDEFAULT_FRAME, char *name="frame")
```

Constructor.

wxPropertyFormFrame::~~wxPropertyFormFrame

```
void ~wxPropertyFormFrame()
```

Destructor.

wxPropertyFormFrame::GetPropertyPanel

```
wxPanel * GetPropertyPanel()
```

Returns the panel associated with the frame.

wxPropertyFormFrame::Initialize

```
bool Initialize()
```

Must be called to create the panel and associate the view with the panel and frame.

wxPropertyFormFrame::OnCreatePanel

```
wxPanel * OnCreatePanel(wxFrame *parent, wxPropertyFormView *view)
```

Creates a panel. Override this to create a panel type other than wxPropertyFormPanel.

wxPropertyFormPanel

The **wxPropertyFormPanel** class is a prepackaged panel which can be used for viewing a property form. Pass a property form view object, and the panel will pass OnDefaultAction listbox messages to the view class for processing.

wxPropertyFormPanel::wxPropertyFormPanel

```
void wxPropertyFormPanel(wxPropertyFormView *view, wxWindow *parent, int x=-1, int y=-1, int width=-1, int height=-1, long style=0, char *name="panel")
```

Constructor.

wxPropertyFormPanel::~~wxPropertyFormPanel

```
void ~wxPropertyFormPanel()
```

Destructor.

wxPropertyFormValidator

The **wxPropertyFormValidator** class defines a base class for form validators. By overriding virtual functions, the programmer can create custom behaviour for kinds of property.

See also

wxPropertyFormValidator overview (p. 1450)

wxPropertyFormValidator::wxPropertyFormValidator

```
void wxPropertyFormValidator(long flags = 0)
```

Constructor.

wxPropertyFormValidator::~~wxPropertyFormValidator

```
void ~wxPropertyFormValidator()
```

Destructor.

wxPropertyFormValidator::OnCommand

```
bool OnCommand(wxProperty *property, wxPropertyFormView *view, wxWindow *parentWindow, wxCommandEvent& event)
```

Called when the control corresponding to the property receives a command (if not intercepted by a callback associated with the actual control).

wxPropertyFormValidator::OnCheckValue

bool OnCheckValue(wxProperty *property, wxPropertyFormView *view, wxWindow *parentWindow) Called when the view checks the property value. The value checked by this validator should be taken from the panel item corresponding to the property.

wxPropertyFormValidator::OnDisplayValue

bool OnDisplayValue(wxProperty *property, wxPropertyFormView *view, wxWindow *parentWindow)

Should display the property value in the appropriate control.

wxPropertyFormValidator::OnDoubleClick

bool OnDoubleClick(wxProperty *property, wxPropertyFormView *view, wxWindow *parentWindow)

Called when the control corresponding to the property is double clicked (listboxes only).

wxPropertyFormValidator::OnRetrieveValue

bool OnRetrieveValue(wxProperty *property, wxPropertyFormView *view, wxWindow *parentWindow)

Should do the transfer from the property editing area to the property itself.

wxPropertyFormView

The **wxPropertyFormView** class shows a **wxPropertySheet** as a view onto a panel or dialog box which has already been created.

See also

wxPropertyFormView overview (p. 1451)

wxPropertyFormView::wxPropertyFormView

void wxPropertyFormView(long flags = 0)

Constructor.

wxPropertyFormView::~wxPropertyFormView

void ~wxPropertyFormView()

Destructor.

wxPropertyFormView::AssociateNames

void AssociateNames()

Associates the properties with the controls on the panel. For each panel item, if the panel item name is the same as a property name, the two objects will be associated. This function should be called manually since the programmer may wish to do the association manually.

wxPropertyFormView::Check

bool Check()

Checks all properties by calling the appropriate validators; returns FALSE if a validation failed.

wxPropertyFormView::GetPanel

wxPanel * GetPanel()

Returns the panel associated with the view.

wxPropertyFormView::GetManagedWindow

wxWindow * GetManagedWindow()

Returns the managed window (a frame or dialog) associated with the view.

wxPropertyFormView::OnOk

void OnOk()

Virtual function that will be called when the OK button on the physical window is

pressed. By default, checks and updates the form values, closes and deletes the frame or dialog, then deletes the view.

wxPropertyFormView::OnCancel

void OnCancel()

Virtual function that will be called when the Cancel button on the physical window is pressed. By default, closes and deletes the frame or dialog, then deletes the view.

wxPropertyFormView::OnHelp

void OnHelp()

Virtual function that will be called when the Help button on the physical window is pressed. This needs to be overridden by the application for anything interesting to happen.

wxPropertyFormView::OnRevert

void OnRevert()

Virtual function that will be called when the Revert button on the physical window is pressed. By default transfers the wxProperty values to the panel items (in effect undoing any unsaved changes in the items).

wxPropertyFormView::OnUpdate

void OnUpdate()

Virtual function that will be called when the Update button on the physical window is pressed. By default transfers the displayed values to the wxProperty objects.

wxPropertyFormView::SetManagedWindow

void SetManagedWindow(wxWindow *win)

Sets the managed window (a frame or dialog) associated with the view.

wxPropertyFormView::TransferToDialog

bool TransferToDialog()

Transfers property values to the controls in the dialog.

wxPropertyFormView::TransferToPropertySheet

bool TransferToPropertySheet()

Transfers property values from the controls in the dialog to the property sheet.

wxPropertyListDialog

The **wxPropertyListDialog** class is a prepackaged dialog which can be used for viewing a property list. Pass a property list view object, and the dialog will pass `OnClose` and `OnDefaultAction` listbox messages to the view class for processing.

wxPropertyListDialog::wxPropertyListDialog

void wxPropertyListDialog(wxPropertyListView *view, wxWindow *parent, char *title, bool modal=FALSE, int x=-1, int y=-1, int width=-1, int height=-1, long style=wxDEFAULT_DIALOG_STYLE, char *name="dialogBox")

Constructor.

wxPropertyListDialog::~~wxPropertyListDialog

void ~wxPropertyListDialog()

Destructor.

wxPropertyListFrame

The **wxPropertyListFrame** class is a prepackaged frame which can be used for viewing a property list. Pass a property list view object, and the frame will pass `OnClose` messages to the view class for processing.

Call `Initialize` to create the panel and associate the view; override `OnCreatePanel` if you wish to use a panel class other than the default `wxPropertyListPanel`.

wxPropertyListFrame::wxPropertyListFrame

```
void wxPropertyListFrame(wxPropertyListView *view, wxFrame *parent, char *title,
int x=-1, int y=-1, int width=-1, int height=-1, long style=wxSDI | wxDEFAULT_FRAME,
char *name="frame")
```

Constructor.

wxPropertyListFrame::~~wxPropertyListFrame

```
void ~wxPropertyListFrame()
```

Destructor.

wxPropertyListFrame::GetPropertyPanel

```
wxPanel * GetPropertyPanel()
```

Returns the panel associated with the frame.

wxPropertyListFrame::Initialize

```
bool Initialize()
```

Must be called to create the panel and associate the view with the panel and frame.

wxPropertyListFrame::OnCreatePanel

```
wxPanel * OnCreatePanel(wxFrame *parent, wxPropertyListView *view)
```

Creates a panel. Override this to create a panel type other than wxPropertyListPanel.

wxPropertyListPanel

The **wxPropertyListPanel** class is a prepackaged panel which can be used for viewing a property list. Pass a property list view object, and the panel will pass OnDefaultAction listbox messages to the view class for processing.

wxPropertyListPanel::wxPropertyListPanel

void wxPropertyListPanel(wxPropertyListView *view, wxWindow *parent, int x=-1, int y=-1, int width=-1, int height=-1, long style=0, char *name="panel")

Constructor.

wxPropertyListPanel::~~wxPropertyListPanel

void ~wxPropertyListPanel()

Destructor.

wxPropertyListValidator

The **wxPropertyListValidator** abstract class is the base class for deriving validators for property lists.

See also

wxPropertyListValidator overview (p. 1450)

wxPropertyListValidator::wxPropertyListValidator

void wxPropertyListValidator(long flags = wxPROP_ALLOW_TEXT_EDITING)

Constructor.

wxPropertyListValidator::~~wxPropertyListValidator

void ~wxPropertyListValidator()

Destructor.

wxPropertyListValidator::OnCheckValue

bool OnCheckValue(wxProperty *property, wxPropertyListView *view, wxWindow *parentWindow) Called when the Tick (Confirm) button is pressed or focus is list. Return FALSE if the value was invalid, which is a signal restores the old value. Return TRUE if

the value was valid.

wxPropertyListValidator::OnClearControls

bool OnClearControls(wxProperty *property, wxPropertyListView *view, wxWindow *parentWindow) Allows the clearing (enabling, disabling) of property list controls, when the focus leaves the current property.

wxPropertyListValidator::OnClearDetailControls

bool OnClearDetailControls(wxProperty *property, wxPropertyListView *view, wxWindow *parentWindow) Called when the focus is lost, if the validator is in detailed editing mode.

wxPropertyListValidator::OnDisplayValue

bool OnDisplayValue(wxProperty *property, wxPropertyListView *view, wxWindow *parentWindow)

Should display the value in the appropriate controls. The default implementation gets the textual value from the property and inserts it into the text edit control.

wxPropertyListValidator::OnDoubleClick

bool OnDoubleClick(wxProperty *property, wxPropertyListView *view, wxWindow *parentWindow)

Called when the property is double clicked. Extra functionality can be provided, such as cycling through possible values.

wxPropertyListValidator::OnEdit

bool OnEdit(wxProperty *property, wxPropertyListView *view, wxWindow *parentWindow) Called when the Edit (detailed editing) button is pressed. The default implementation calls `wxPropertyListView::BeginDetailedEditing`; a filename validator (for example) overrides this function to show the file selector.

wxPropertyListValidator::OnPrepareControls

bool OnPrepareControls(wxProperty *property, wxPropertyListView *view, wxWindow *parentWindow)

Called to allow the validator to setup the display, such enabling or disabling buttons, and setting the values and selection in the standard listbox control (the one optionally used for displaying value options).

wxPropertyListValidator::OnPrepareDetailControls

bool OnPrepareDetailControls(wxProperty *property, wxPropertyListView *view, wxWindow *parentWindow) Called when the property is edited 'in detail', i.e. when the Edit button is pressed.

wxPropertyListValidator::OnRetrieveValue

bool OnRetrieveValue(wxProperty *property, wxPropertyListView *view, wxWindow *parentWindow)

Called when Tick (Confirm) is pressed or focus is lost or view wants to update the property list. Should do the transfer from the property editing area to the property itself

wxPropertyListValidator::OnSelect

bool OnSelect(bool select, wxProperty *property, wxPropertyListView *view, wxWindow *parentWindow)

Called when the property is selected or deselected: typically displays the value in the edit control (having chosen a suitable control to display: (non)editable text or listbox).

wxPropertyListValidator::OnValueListSelect

bool OnValueListSelect(wxProperty *property, wxPropertyListView *view, wxWindow *parentWindow)

Called when the value listbox is selected. The default behaviour is to copy string to text control, and retrieve the value into the property.

wxPropertyListView

The **wxPropertyListView** class shows a wxPropertySheet as a Visual Basic-style property list.

See also

wxPropertyListView overview (p. 1451)

wxPropertyListView::wxPropertyListView

void wxPropertyListView(long flags = wxPROP_BUTTON_DEFAULT)

Constructor.

The *flags* argument can be a bit list of the following:

- wxPROP_BUTTON_CLOSE
- wxPROP_BUTTON_OK
- wxPROP_BUTTON_CANCEL
- wxPROP_BUTTON_CHECK_CROSS
- wxPROP_BUTTON_HELP
- wxPROP_DYNAMIC_VALUE_FIELD
- wxPROP_PULLDOWN

wxPropertyListView::~~wxPropertyListView

void ~wxPropertyListView()

Destructor.

wxPropertyListView::AssociatePanel

void AssociatePanel(wxPanel *panel)

Associates the window on which the controls will be displayed, with the view (sets an internal pointer to the window).

wxPropertyListView::BeginShowingProperty

bool BeginShowingProperty(wxProperty *property)

Finds the appropriate validator and loads the property into the controls, by calling `wxPropertyValidator::OnPrepareControls` and then `wxPropertyListView::DisplayProperty`.

wxPropertyListView::DisplayProperty

bool DisplayProperty(wxProperty *property)

Calls `wxPropertyValidator::OnDisplayValue` for the current property's validator. This function gets called by `wxPropertyListView::BeginShowingProperty`, which is in turn called from `ShowProperty`, called by `OnPropertySelect`, called by the listbox callback when selected.

wxPropertyListView::EndShowingProperty

bool EndShowingProperty(wxProperty *property)

Finds the appropriate validator and unloads the property from the controls, by calling `wxPropertyListView::RetrieveProperty`, `wxPropertyValidator::OnClearControls` and (if we're in detailed editing mode) `wxPropertyValidator::OnClearDetailControls`.

wxPropertyListView::GetPanel

wxPanel * GetPanel()

Returns the panel associated with the view.

wxPropertyListView::GetManagedWindow

wxWindow * GetManagedWindow()

Returns the managed window (a frame or dialog) associated with the view.

wxPropertyListView::GetWindowCancelButton

wxButton * GetWindowCancelButton()

Returns the window cancel button, if any.

wxPropertyListView::GetWindowCloseButton

wxButton * GetWindowCloseButton()

Returns the window close or OK button, if any.

wxPropertyListView::GetWindowHelpButton

wxButton * GetWindowHelpButton()

Returns the window help button, if any.

wxPropertyListView::SetManagedWindow

void SetManagedWindow(wxWindow *win)

Sets the managed window (a frame or dialog) associated with the view.

wxPropertyListView::UpdatePropertyDisplayInList

bool UpdatePropertyDisplayInList(wxProperty *property)

Updates the display for the given changed property.

wxPropertyListView::UpdatePropertyList

bool UpdatePropertyList(bool clearEditArea = TRUE)

Updates the whole property list display.

wxPropertySheet

The **wxPropertySheet** class is used for storing a number of wxProperty objects (essentially names and values).

See also

wxPropertySheet overview (p. 1452)

wxPropertySheet::wxPropertySheet

void wxPropertySheet(const wxString name = "")

Constructor. Sets property sheet's name to name if present.

wxPropertySheet::~~wxPropertySheet

void ~wxPropertySheet()

Destructor. Destroys all contained properties.

wxPropertySheet::AddProperty

void AddProperty(wxProperty *property)

Adds a property to the sheet.

wxPropertySheet::Clear

void Clear()

Clears all the properties from the sheet (deleting them).

wxPropertySheet::GetName

wxString GetName()

Gets the sheet's name.

wxPropertySheet::GetProperty

wxProperty * GetProperty(wxString name)

Gets a property by name.

wxPropertySheet::GetProperties

wxList& GetProperties()

Returns a reference to the internal list of properties.

wxPropertySheet::HasProperty

bool HasProperty(wxString propname)

Returns true if sheet contains property propname.

wxPropertySheet::RemoveProperty

void RemoveProperty(wxString propname)

Removes property propname from sheet, deleting it.

wxPropertySheet::SetName

void SetName(wxString sheetname)

Set the sheet's name to sheetname

wxPropertySheet::SetProperty

bool SetProperty(wxString propname, wxPropertyValue value)

Sets property propname to value. Returns false if property is not a member of sheet.

wxPropertySheet::SetAllModified

void SetAllModified(bool flag)

Sets the 'modified' flag of each property value.

wxPropertyValidator

The **wxPropertyValidator** abstract class is the base class for deriving validators for properties.

See also

wxPropertyValidator overview (p. 1449)

wxPropertyValidator::wxPropertyValidator

void wxPropertyValidator(long flags = 0)

Constructor.

wxPropertyValidator::~~wxPropertyValidator

void ~wxPropertyValidator()

Destructor.

wxPropertyValidator::GetFlags

long GetFlags()

Returns the flags for the validator.

wxPropertyValidator::GetValidatorProperty

wxProperty * GetValidatorProperty()

Gets the property for the validator.

wxPropertyValidator::SetValidatorProperty

void SetValidatorProperty(wxProperty *property)

Sets the property for the validator.

wxPropertyValidatorRegistry

The **wxPropertyValidatorRegistry** class is used for storing validators, indexed by the 'role name' of the property, by which groups of property can be identified for the purpose of validation and editing.

wxPropertyValidatorRegistry::wxPropertyValidatorRegistry

void wxPropertyValidatorRegistry()

Constructor.

wxPropertyValidatorRegistry::~~wxPropertyValidatorRegistry

void ~wxPropertyValidatorRegistry()

Destructor.

wxPropertyValidatorRegistry::Clear

void ClearRegistry()

Clears the registry, deleting the validators.

wxPropertyValidatorRegistry::GetValidator

wxPropertyValidator * GetValidator(wxString& roleName)

Retrieve a validator by the property role name.

wxPropertyValidatorRegistry::RegisterValidator

void RegisterValidator(wxString& roleName, wxPropertyValidator *validator)

Register a validator with the registry. *roleName* is a name indicating the role of the property, such as "filename". Later, when a validator is chosen for editing a property, this role name is matched against the class names of the property, if the property does not already have a validator explicitly associated with it.

wxPropertyValue

The **wxPropertyValue** class represents the value of a property, and is normally associated with a wxProperty object.

A wxPropertyValue has one of the following types:

- wxPropertyValueNull
- wxPropertyValueInteger
- wxPropertyValueReal
- wxPropertyValueBool
- wxPropertyValueString
- wxPropertyValueList
- wxPropertyValueIntegerPtr
- wxPropertyValueRealPtr
- wxPropertyValueBoolPtr
- wxPropertyValueStringPtr

wxPropertyValue::wxPropertyValue

void wxPropertyValue()

Default constructor.

void wxPropertyValue(const wxPropertyValue& copyFrom)

Copy constructor.

void wxPropertyValue(char *val)

Construction from a string value.

void wxPropertyValue(long val)

Construction from an integer value. You may need to cast to (long) to avoid confusion with other constructors (such as the bool constructor).

void wxPropertyValue(bool val)

Construction from a boolean value.

void wxPropertyValue(float val)

Construction from a floating point value.

void wxPropertyValue(double val)

Construction from a floating point value.

void wxPropertyValue(wxList * val)

Construction from a list of wxPropertyValue objects. The list, but not each contained wxPropertyValue, will be deleted by the constructor. The wxPropertyValues will be assigned to this wxPropertyValue list. In other words, so do not delete wxList or its data after calling this constructor.

void wxPropertyValue(wxStringList * val)

Construction from a list of strings. The list (including the strings contained in it) will be deleted by the constructor, so do not destroy val explicitly.

void wxPropertyValue(char **val)

Construction from a string pointer.

void wxPropertyValue(long *val)

Construction from an integer pointer.

void wxPropertyValue(bool *val)

Construction from an boolean pointer.

void wxPropertyValue(float *val)

Construction from a floating point pointer.

The last four constructors use pointers to various C++ types, and do not store the types themselves; this allows the values to stand in for actual data values defined elsewhere.

wxPropertyValue::~~wxPropertyValue

void ~wxPropertyValue()

Destructor.

wxPropertyValue::Append

void Append(wxPropertyValue *expr)

Appends a property value to the list.

wxPropertyValue::BoolValue

bool BoolValue()

Returns the boolean value.

wxPropertyValue::BoolValuePtr

bool * BoolValuePtr()

Returns the pointer to the boolean value.

wxPropertyValue::ClearList

void ClearList()

Deletes the contents of the list.

wxPropertyValue::Delete

void Delete(wxPropertyValue *expr)

Deletes *expr* from this list.

wxPropertyValue::GetFirst

wxPropertyValue * GetFirst()

Gets the first value in the list.

wxPropertyValue::GetLast

wxPropertyValue * GetFirst()

Gets the last value in the list.

wxPropertyValue::GetModified

bool GetModified()

Returns TRUE if the value was modified since being created (or since SetModified was called).

wxPropertyValue::GetNext

wxPropertyValue * GetNext()

Gets the next value in the list (the one after 'this').

wxPropertyValue::GetStringRepresentation

wxString GetStringRepresentation()

Gets a string representation of the value.

wxPropertyValue::IntegerValue

long IntegerValue()

Returns the integer value.

wxPropertyValue::Insert

void Insert(wxPropertyValue **expr*)

Inserts a property value at the front of a list.

wxPropertyValue::IntegerValuePtr

long * IntegerValuePtr()

Returns the pointer to the integer value.

wxPropertyValue::Nth

wxPropertyValue * Nth(int *n*)

Returns the *n*th value of a list expression (starting from zero).

wxPropertyValue::Number

int Number()

Returns the number of elements in a list expression.

wxPropertyValue::RealValue

float RealValue()

Returns the floating point value.

wxPropertyValue::RealValuePtr

float * RealValuePtr()

Returns the pointer to the floating point value.

wxPropertyValue::SetModified

void SetModified(bool *flag*)

Sets the 'modified' flag.

wxPropertyValue::StringValue

char * StringValue()

Returns the string value.

wxPropertyValue::StringValuePtr

```
char ** StringValuePtr()
```

Returns the pointer to the string value.

wxPropertyValue::Type

```
wxPropertyValueType Type()
```

Returns the value type.

wxPropertyValue::operator =

```
void operator =(const wxPropertyValue& val)
```

```
void operator =(const char *val)
```

```
void operator =(const long val)
```

```
void operator =(const bool val)
```

```
void operator =(const float val)
```

```
void operator =(const char **val)
```

```
void operator =(const long *val)
```

```
void operator =(const bool *val)
```

```
void operator =(const float *val)
```

Assignment operators.

wxPropertyView

The **wxPropertyView** abstract class is the base class for views of property sheets, acting as intermediaries between properties and actual windows.

See also

wxPropertyView overview (p. 1451)

wxPropertyView::wxPropertyView

void wxPropertyView(long flags = wxPROP_BUTTON_DEFAULT)

Constructor.

The *flags* argument can be a bit list of the following:

- wxPROP_BUTTON_CLOSE
- wxPROP_BUTTON_OK
- wxPROP_BUTTON_CANCEL
- wxPROP_BUTTON_CHECK_CROSS
- wxPROP_BUTTON_HELP
- wxPROP_DYNAMIC_VALUE_FIELD
- wxPROP_PULLDOWN

wxPropertyView::~~wxPropertyView

void ~wxPropertyView()

Destructor.

wxPropertyView::AddRegistry

void AddRegistry(wxPropertyValidatorRegistry *registry)

Adds a registry (list of property validators) the view's list of registries, which is initially empty.

wxPropertyView::FindPropertyValidator

wxPropertyValidator * FindPropertyValidator(wxProperty *property)

Finds the property validator that is most appropriate to this property.

wxPropertyView::GetPropertySheet

wxPropertySheet * GetPropertySheet()

Gets the property sheet for this view.

wxPropertyView::GetRegistryList

wxList& GetRegistryList()

Returns a reference to the list of property validator registries.

wxPropertyView::OnOk

void OnOk()

Virtual function that will be called when the OK button on the physical window is pressed (if it exists).

wxPropertyView::OnCancel

void OnCancel()

Virtual function that will be called when the Cancel button on the physical window is pressed (if it exists).

wxPropertyView::OnClose

bool OnClose()

Virtual function that will be called when the physical window is closed. The default implementation returns FALSE.

wxPropertyView::OnHelp

void OnHelp()

Virtual function that will be called when the Help button on the physical window is pressed (if it exists).

wxPropertyView::OnPropertyChanged

void OnPropertyChanged(wxProperty *property)

Virtual function called by a view or validator when a property's value changed. Validators must be written correctly for this to be called. You can override this function to respond immediately to property value changes.

wxPropertyView::OnUpdateView

bool OnUpdateView()

Called by the viewed object to update the view. The default implementation just returns

FALSE.

wxPropertyView::SetPropertySheet

void SetPropertySheet(wxPropertySheet *sheet)

Sets the property sheet for this view.

wxPropertyView::ShowView

void ShowView(wxPropertySheet *sheet, wxPanel *panel)

Associates this view with the given panel, and shows the view.

wxProtocol

Derived from

wxSocketClient (p. 956)

Include files

<wx/protocol/protocol.h>

See also

wxSocketBase (p. 938), *wxURL* (p. 1164)

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 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. 592)

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_NOHANDLER	.
wxPROTO_NOFILE	The remote file doesn't exist.
wxPROTO_ABORT	Last action aborted.
wxPROTO_RCNECT	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.

wxQueryCol

Every ODBC data column is represented by an instance of this class.

Derived from

wxObject (p. 746)

Include files

<wx/odbc.h>

See also

wxQueryCol overview (p. 1425), *wxDatabase overview* (p. 1425)

wxQueryCol::wxQueryCol

void wxQueryCol()

Constructor. Sets the attributes of the column to default values.

wxQueryCol::~~wxQueryCol

void ~wxQueryCol()

Destructor. Deletes the wxQueryField list.

wxQueryCol::BindVar

void * BindVar(void *v, long sz)

Binds a user-defined variable to a column. Whenever a column is bound to a variable, it will automatically copy the data of the current field into this buffer (to a maximum of sz bytes).

wxQueryCol::FillVar

void FillVar(int recnum)

Fills the bound variable with the data of the field recnum. When no variable is bound to the column nothing will happen.

wxQueryCol::GetData

void * GetData(int field)

Returns a pointer to the data of the field.

wxQueryCol::GetName

wxString GetName()

Returns the name of a column.

wxQueryCol::GetType

short GetType()

Returns the data type of a column.

wxQueryCol::GetSize

long GetSize(int field)

Return the size of the data of the field *field*.

wxQueryCol::IsRowDirty

bool IsRowDirty(int *field*)

Returns TRUE if the given field has been changed, but not saved.

wxQueryCol::IsNullable

bool IsNullable()

Returns TRUE if a column may contain no data.

wxQueryCol::AppendField

void AppendField(void **buf*, long *len*)

Appends a `wxQueryField` instance to the field list of the column. *len* bytes from *buf* will be copied into the field's buffer.

wxQueryCol::SetData

bool SetData(int *field*, void **buf*, long *len*)

Sets the data of a field. This function finds the `wxQueryField` corresponding to *field* and calls `wxQueryField::SetData` with *buf* and *len* arguments.

wxQueryCol::SetName

void SetName(const wxString& *name*)

Sets the name of a column. Only useful when creating new tables or appending columns.

wxQueryCol::SetNullable

void SetNullable(bool *nullable*)

Determines whether a column may contain no data. Only useful when creating new tables or appending columns.

wxQueryCol::SetFieldDirty

void SetFieldDirty(int field, bool dirty = TRUE)

Sets the dirty tag of a given field.

wxQueryCol::SetType

void SetType(short type) Sets the data type of a column. Only useful when creating new tables or appending columns.

wxQueryField

Represents the data item for one or several columns.

Derivation

wxObject (p. 746)

See also

wxQueryField overview (p. 1425), *wxDatabase overview* (p. 1425)

wxQueryField::wxQueryField

wxQueryField()

Constructor. Sets type and size of the field to default values.

wxQueryField::~~wxQueryField

~wxQueryField()

Destructor. Frees the associated memory depending on the field type.

wxQueryField::AllocData

bool AllocData()

Allocates memory depending on the size and type of the field.

wxQueryField::ClearData

void ClearData()

Deletes the contents of the field buffer without deallocating the memory.

wxQueryField::GetData

void * GetData()

Returns a pointer to the field buffer.

wxQueryField::GetSize

long GetSize()

Returns the size of the field buffer.

wxQueryField::GetType

short GetType()

Returns the type of the field (currently SQL_CHAR, SQL_VARCHAR or SQL_INTEGER).

wxQueryField::IsDirty

bool IsDirty()

Returns TRUE if the data of a field has been changed, but not saved.

wxQueryField::SetData

bool SetData(void *data, long sz)

Allocates memory of the size *sz* and copies the contents of *d* into the field buffer.

wxQueryField::SetDirty

void SetDirty(bool dirty = TRUE)

Sets the dirty tag of a field.

wxQueryField::SetSize

void SetSize(long size)

Resizes the field buffer. Stored data will be lost.

wxQueryField::SetType

void SetType(short *type*)

Sets the type of the field. Currently the types SQL_CHAR, SQL_VARCHAR and SQL_INTEGER are supported.

wxQueryLayoutInfoEvent

This event is sent when *wxLayoutAlgorithm* (p. 610) 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. 375)

wxObject (p. 746)

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. 94), *wxSashLayoutWindow* (p. 894), *wxLayoutAlgorithm* (p. 610).

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. 176)
wxWindow (p. 1184)
wxEvtHandler (p. 378)
wxObject (p. 746)

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.

See also *window styles overview* (p. 1371).

Event handling

EVT_RADIOBOX(id, func) Process a `wxEVT_COMMAND_RADIOBOX_SELECTED` event, when a radiobutton is clicked.

See also

Event handling overview (p. 1364), *wxRadioButton* (p. 864), *wxCheckBox* (p. 108)

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")

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. 858).

validator

Window validator.

name

Window name.

See also

`wxRadioBox::Create` (p. 861), `wxValidator` (p. 1166)

wxPython note: The `wxRadioBox` constructor in `wxPython` reduces the `n` and `choices` arguments to a single argument, which is a list of strings.

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")

Creates the radiobox for two-step construction. See `wxRadioBox::wxRadioBox` (p. 859) for further details.

wxRadioBox::Enable

void Enable(const bool enable)

Enables or disables the entire radiobox.

void Enable(int n, const bool enable)

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.

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::GetLabel

wxString GetLabel() const

Returns the radiobox label.

wxString GetLabel(int *n*) const

Returns the label for the given button.

Parameters

n

The zero-based button index.

See also

wxRadioBox::SetLabel (p. 863)

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::Number

int Number() const

Returns the number of buttons in the radiobox.

wxRadioBox::SetLabel

void SetLabel(const wxString& *label*)

Sets the radiobox label.

void SetLabel(int *n*, const wxString& *label*)

Sets a label for a radio button.

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

void Show(const bool show)

Shows or hides the entire radiobox.

void Show(int item, const bool show)

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.

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.

wxRadioBox::GetString

wxString GetString(int n) const

Returns the label for the button at the given position.

Parameters

n

The zero-based button position.

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.

Derived from

wxControl (p. 176)
wxWindow (p. 1184)
wxEvtHandler (p. 378)
wxObject (p. 746)

Include files

<wx/radiobut.h>

Window styles

wxRB_GROUP Marks the beginning of a new group of radio buttons.

See also *window styles overview* (p. 1371).

Event handling

EVT_RADIOBUTTON(id, func) Process a `wxEVT_COMMAND_RADIOBUTTON_SELECTED` event, when the radiobutton is clicked.

See also

Event handling overview (p. 1364), *wxRadioBox* (p. 858), *wxCheckBox* (p. 108)

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. 864).

validator

Window validator.

name

Window name.

See also

wxRadioButton::Create (p. 866), *wxValidator* (p. 1166)

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. 865) 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.

wxRealFormValidator

This class validates a range of real values for form views. The associated panel item must be a `wxText`.

See also

Validator classes (p. 1453)

wxRealFormValidator::wxRealFormValidator

void wxRealFormValidator(float min=0.0, float max=0.0, long flags=0)

Constructor. Assigning zero to minimum and maximum values indicates that there is no range to check.

wxRealListValidator

This class validates a range of real values for property lists.

See also

Validator classes (p. 1453)

wxPropertySheet overview (p. 1452)

wxRealListValidator::wxRealListValidator

void wxRealListValidator(float min=0.0, float max=0.0, long flags=wxPROP_ALLOW_TEXT_EDITING)

Constructor. Assigning zero to minimum and maximum values indicates that there is no range to check.

wxRealPoint

A **wxRealPoint** is a useful data structure for graphics operations. It contains floating point *x* and *y* members. See also *wxPoint* (p. 785) for an integer version.

Derived from

None

Include files

<wx/gdicmn.h>

See also

wxPoint (p. 785)

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. 785), *wxSize* (p. 922)

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::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::GetBottom

int GetBottom() const

Gets the bottom point of the rectangle.

wxRect::GetHeight

int GetHeight() const

Gets the height member.

wxRect::GetLeft

int GetLeft() constGets the left point of the rectangle (the same as *wxRect::GetX* (p. 871)).**wxRect::GetPosition**

wxPoint GetPosition() const

Gets the position.

wxRect::GetRight

int GetRight() const

Gets the right point of the rectangle.

wxRect::GetSize

wxSize GetSize() const

Gets the size.

wxRect::GetTop

int GetTop() constGets the top point of the rectangle (the same as *wxRect::GetY* (p. 871)).**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)**Increase the rectangle size by *dx* in x direction and *dy* in y direction. Both (or one of) parameters may be negative to decrease the rectangle size.The second form uses the same *diff* for both *dx* and *dy*.**wxRect::SetHeight**

void SetHeight(int *height*)

Sets the height.

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::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.

wxRecordSet

Each `wxRecordSet` represents an ODBC database query. You can make multiple queries at a time by using multiple `wxRecordSets` with a `wxDatabase` or you can make your queries in sequential order using the same `wxRecordSet`.

Note: this class is considered obsolete, replaced by the Remstar `wxDB/wxTable` classes (documented separately in Word and PDF format, as `odbc.doc` and `odbc.pdf`).

Derived from

`wxObject` (p. 746)

Include files

<wx/odbc.h>

See also

`wxRecordSet` overview (p. 1425), `wxDatabase` overview (p. 1425)

`wxRecordSet::wxRecordSet`

`wxRecordSet(wxDatabase *db, int type = wxOPEN_TYPE_DYNASET, int opt = wxOPTION_DEFAULT)`

Constructor. `db` is a pointer to the `wxDatabase` instance you wish to use the `wxRecordSet` with. Currently there are two possible values of `type`:

- `wxOPEN_TYPE_DYNASET`: Loads only one record at a time into memory. The other data of the result set will be loaded dynamically when moving the cursor. This is the default type.
- `wxOPEN_TYPE_SNAPSHOT`: Loads all records of a result set at once. This will need much more memory, but will result in faster access to the ODBC data.

The `option` parameter is not used yet.

The constructor appends the `wxRecordSet` object to the parent database's list of `wxRecordSet` objects, for later destruction when the `wxDatabase` is destroyed.

`wxRecordSet::~wxRecordSet`

`~wxRecordSet()`

Destructor. All data except that stored in user-defined variables will be lost. It also unlinks the `wxRecordSet` object from the parent database's list of `wxRecordSet` objects.

`wxRecordSet::AddNew`

void AddNew()

Not implemented.

wxRecordSet::BeginQuery

bool BeginQuery(int *openType*, const wxString& *sql* = NULL, int *options* = wxOPTION_DEFAULT)

Not implemented.

wxRecordSet::BindVar

void * BindVar(int *col*, void **buf*, long *size*)

Binds a user-defined variable to the column *col*. Whenever the current field's data changes, it will be copied into *buf* (maximum *size* bytes).

void * BindVar(const wxString& *col*, void **buf*, long *size*)

The same as above, but uses the column name as the identifier.

wxRecordSet::CanAppend

bool CanAppend()

Not implemented.

wxRecordSet::Cancel

void Cancel()

Not implemented.

wxRecordSet::CanRestart

bool CanRestart()

Not implemented.

wxRecordSet::CanScroll

bool CanScroll()

Not implemented.

wxRecordSet::CanTransact

bool CanTransact()

Not implemented.

wxRecordSet::CanUpdate

bool CanUpdate()

Not implemented.

wxRecordSet::ConstructDefaultSQL

bool ConstructDefaultSQL()

Not implemented.

wxRecordSet::Delete

bool Delete()

Deletes the current record. Not implemented.

wxRecordSet::Edit

void Edit()

Not implemented.

wxRecordSet::EndQuery

bool EndQuery()

Not implemented.

wxRecordSet::ExecuteSQL

bool ExecuteSQL(const wxString& sql)

Directly executes a SQL statement. The data will be presented as a normal result set. Note that the recordset must have been created as a snapshot, not dynaset. Dynasets will be implemented in the near future.

Examples of common SQL statements are given in *A selection of SQL commands* (p. 1426).

wxRecordSet::FillVars

void FillVars(int recnum)

Fills in the user-defined variables of the columns. You can set these variables with `wxQueryCol::BindVar`. This function will be automatically called after every successful database operation.

wxRecordSet::GetColName

wxString GetColName(int col)

Returns the name of the column at position *col*. Returns NULL if *col* does not exist.

wxRecordSet::GetColType

short GetColType(int col)

Returns the data type of the column at position *col*. Returns `SQL_TYPE_NULL` if *col* does not exist.

short GetColType(const wxString& name)

The same as above, but uses the column name as the identifier.

See *ODBC SQL data types* (p. 1426) for a list of possible data types.

wxRecordSet::GetColumns

bool GetColumns(const wxString& table = NULL)

Returns the columns of the table with the specified name. If no name is given the class member *tablename* will be used. If both names are NULL nothing will happen. The data will be presented as a normal result set, organized as follows:

0 (VARCHAR)	TABLE_QUALIFIER
1 (VARCHAR)	TABLE_OWNER

2 (VARCHAR)	TABLE_NAME
3 (VARCHAR)	COLUMN_NAME
4 (SMALLINT)	DATA_TYPE
5 (VARCHAR)	TYPE_NAME
6 (INTEGER)	PRECISION
7 (INTEGER)	LENGTH
8 (SMALLINT)	SCALE
9 (SMALLINT)	RADIX
10 (SMALLINT)	NULLABLE
11 (VARCHAR)	REMARKS

wxRecordSet::GetCurrentRecord

long GetCurrentRecord()

Not implemented.

wxRecordSet::GetDatabase

wxDatabase * GetDatabase()

Returns the wxDatabase object bound to a wxRecordSet.

wxRecordSet::GetDataSources

bool GetDataSources()

Gets the currently-defined data sources via the ODBC manager. The data will be presented as a normal result set. See the documentation for the ODBC function SQLDataSources for how the data is organized.

Example:

```
wxDatabase Database;

wxRecordSet *Record = new wxRecordSet(&Database);

if (!Record->GetDataSources()) {
    char buf[300];
    sprintf(buf, "%s %s\n", Database.GetErrorClass(),
Database.GetErrorMessage());
    frame->output->SetValue(buf);
}
else {
    do {
        frame->DataSource->Append((char*)Record->GetFieldDataPtr(0,
SQL_CHAR));
    } while (Record->MoveNext());
}
```

wxRecordSet::GetDefaultConnect

wxString GetDefaultConnect()

Not implemented.

wxRecordSet::GetDefaultSQL

wxString GetDefaultSQL()

Not implemented.

wxRecordSet::GetErrorCode

wxRETCODE GetErrorCode()

Returns the error code of the last ODBC action. This will be one of:

SQL_ERROR	General error.
SQL_INVALID_HANDLE	An invalid handle was passed to an ODBC function.
SQL_NEED_DATA	ODBC expected some data.
SQL_NO_DATA_FOUND	No data was found by this ODBC call.
SQL_SUCCESS	The call was successful.
SQL_SUCCESS_WITH_INFO	The call was successful, but further information can be obtained from the ODBC manager.

wxRecordSet::GetFieldData

bool GetFieldData(int col, int dataType, void *dataPtr)Copies the current data of the column at position *col* into the buffer *dataPtr*. To be sure to get the right type of data, the user has to pass the correct data type. The function returns FALSE if *col* does not exist or the wrong data type was given.**bool GetFieldData(const wxString& name, int dataType, void *dataPtr)**

The same as above, but uses the column name as the identifier.

See *ODBC SQL data types* (p. 1426) for a list of possible data types.

wxRecordSet::GetFieldDataPtr

void * GetFieldDataPtr(int col, int dataType)

Returns the current data pointer of the column at position *col*. To be sure to get the right type of data, the user has to pass the data type. Returns NULL if *col* does not exist or if *dataType* is incorrect.

void * GetFieldDataPtr(const wxString& name, int dataType)

The same as above, but uses the column name as the identifier.

See *ODBC SQL data types* (p. 1426) for a list of possible data types.

wxRecordSet::GetFilter

wxString GetFilter()

Returns the current filter.

wxRecordSet::GetForeignKeys

bool GetPrimaryKeys(const wxString& ptable = NULL, const wxString& ftable = NULL)

Returns a list of foreign keys in the specified table (columns in the specified table that refer to primary keys in other tables), or a list of foreign keys in other tables that refer to the primary key in the specified table.

If *ptable* contains a table name, this function returns a result set containing the primary key of the specified table.

If *ftable* contains a table name, this functions returns a result set of containing all of the foreign keys in the specified table and the primary keys (in other tables) to which they refer.

If both *ptable* and *ftable* contain table names, this function returns the foreign keys in the table specified in *ftable* that refer to the primary key of the table specified in *ptable*. This should be one key at most.

GetForeignKeys returns results as a standard result set. If the foreign keys associated with a primary key are requested, the result set is ordered by FKTABLE_QUALIFIER, FKTABLE_OWNER, FKTABLE_NAME, and KEY_SEQ. If the primary keys associated with a foreign key are requested, the result set is ordered by PKTABLE_QUALIFIER, PKTABLE_OWNER, PKTABLE_NAME, and KEY_SEQ. The following table lists the columns in the result set.

0 (VARCHAR)	PKTABLE_QUALIFIER
1 (VARCHAR)	PKTABLE_OWNER
2 (VARCHAR)	PKTABLE_NAME
3 (VARCHAR)	PKCOLUMN_NAME

4 (VARCHAR)	FKTABLE_QUALIFIER
5 (VARCHAR)	FKTABLE_OWNER
6 (VARCHAR)	FKTABLE_NAME
7 (VARCHAR)	FKCOLUMN_NAME
8 (SMALLINT)	KEY_SEQ
9 (SMALLINT)	UPDATE_RULE
10 (SMALLINT)	DELETE_RULE
11 (VARCHAR)	FK_NAME
12 (VARCHAR)	PK_NAME

wxRecordSet::GetNumberCols

long GetNumberCols()

Returns the number of columns in the result set.

wxRecordSet::GetNumberFields

int GetNumberFields()

Not implemented.

wxRecordSet::GetNumberParams

int GetNumberParams()

Not implemented.

wxRecordSet::GetNumberRecords

long GetNumberRecords()

Returns the number of records in the result set.

wxRecordSet::GetPrimaryKeys

bool GetPrimaryKeys(const wxString& table = NULL)

Returns the column names that comprise the primary key of the table with the specified name. If no name is given the class member *tablename* will be used. If both names are NULL nothing will happen. The data will be presented as a normal result set, organized as follows:

0 (VARCHAR)	TABLE_QUALIFIER
1 (VARCHAR)	TABLE_OWNER
2 (VARCHAR)	TABLE_NAME
3 (VARCHAR)	COLUMN_NAME
4 (SMALLINT)	KEY_SEQ

5 (VARCHAR)

PK_NAME

wxRecordSet::GetOptions

int GetOptions()

Returns the options of the wxRecordSet. Options are not supported yet.

wxRecordSet::GetResultSet

bool GetResultSet()

Copies the data presented by ODBC into wxRecordSet. Depending on the wxRecordSet type all or only one record(s) will be copied. Usually this function will be called automatically after each successful database operation.

wxRecordSet::GetSortString

wxString GetSortString()

Not implemented.

wxRecordSet::GetSQL

wxString GetSQL()

Not implemented.

wxRecordSet::GetTableName

wxString GetTableName()

Returns the name of the current table.

wxRecordSet::GetTables

bool GetTables()

Gets the tables of a database. The data will be presented as a normal result set, organized as follows:

0 (VARCHAR)	TABLE_QUALIFIER
1 (VARCHAR)	TABLE_OWNER
2 (VARCHAR)	TABLE_NAME
3 (VARCHAR)	TABLE_TYPE (TABLE, VIEW, SYSTEM TABLE, GLOBAL TEMPORARY, LOCAL TEMPORARY, ALIAS, SYNONYM, or database-specific type)

4 (VARCHAR)

REMARKS

wxRecordSet::GetType

int GetType()

Returns the type of the wxRecordSet: wxOPEN_TYPE_DYNASET or wxOPEN_TYPE_SNAPSHOT. See the wxRecordSet description for details.

wxRecordSet::GoTo

bool GoTo(long *n*)

Moves the cursor to the record with the number *n*, where the first record has the number 0.

wxRecordSet::IsBOF

bool IsBOF()

Returns TRUE if the user tried to move the cursor before the first record in the set.

wxRecordSet::IsFieldDirty

bool IsFieldDirty(int *field*)

Returns TRUE if the given field has been changed but not saved yet.

bool IsFieldDirty(const wxString& *name*)

Same as above, but uses the column name as the identifier.

wxRecordSet::IsFieldNull

bool IsFieldNull(int *field*)

Returns TRUE if the given field has no data.

bool IsFieldNull(const wxString& *name*)

Same as above, but uses the column name as the identifier.

wxRecordSet::IsColNullable

bool IsColNullable(int col)

Returns TRUE if the given column may contain no data.

bool IsColNullable(const wxString& name)

Same as above, but uses the column name as the identifier.

wxRecordSet::IsEOF

bool IsEOF()

Returns TRUE if the user tried to move the cursor behind the last record in the set.

wxRecordSet::IsDeleted

bool IsDeleted()

Not implemented.

wxRecordSet::IsOpen

bool IsOpen()

Returns TRUE if the parent database is open.

wxRecordSet::Move

bool Move(long rows)

Moves the cursor a given number of rows. Negative values are allowed.

wxRecordSet::MoveFirst

bool MoveFirst()

Moves the cursor to the first record.

wxRecordSet::MoveLast

bool MoveLast()

Moves the cursor to the last record.

wxRecordSet::MoveNext

bool MoveNext()

Moves the cursor to the next record.

wxRecordSet::MovePrev

bool MovePrev()

Moves the cursor to the previous record.

wxRecordSet::Query

bool Query(const wxString& columns, const wxString& table, const wxString& filter = NULL)

Start a query. An SQL string of the following type will automatically be generated and executed: "SELECT columns FROM table WHERE filter".

wxRecordSet::RecordCountFinal

bool RecordCountFinal()

Not implemented.

wxRecordSet::Requery

bool Requery()

Re-executes the last query. Not implemented.

wxRecordSet::SetFieldDirty

void SetFieldDirty(int field, bool dirty = TRUE)

Sets the dirty tag of the field field. Not implemented.

void SetFieldDirty(const wxString& name, bool dirty = TRUE)

Same as above, but uses the column name as the identifier.

wxRecordSet::SetDefaultSQL

void SetDefaultSQL(const wxString& s)

Not implemented.

wxRecordSet::SetFieldNull

void SetFieldNull(void *p, bool isNull = TRUE)

Not implemented.

wxRecordSet::SetOptions

void SetOptions(int *opt*)

Sets the options of the wxRecordSet. Not implemented.

wxRecordSet::SetTableName

void SetTableName(const wxString& *tablename*)

Specify the name of the table you want to use.

wxRecordSet::SetType

void SetType(int *type*)

Sets the type of the wxRecordSet. See the wxRecordSet class description for details.

wxRecordSet::Update

bool Update()

Writes back the current record. Not implemented.

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. 474)

wxObject (p. 746)

Include files

<wx/region.h>

See also

wxRegionIterator (p. 889)

wxRegion::wxRegion

wxRegion()

Default constructor.

wxRegion(long x, long y, long width, long 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::~~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.

wxRegion::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::GetBox

void GetBox(long& *x*, long& *y*, long& *width*, long& *height*) const

Returns the outer bounds of the region.

wxRect GetBox() const

Returns the outer bounds of the region.

wxRegion::Intersect

bool Intersect(long *x*, long *y*, long *width*, long *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::Union

bool Union(long *x*, long *y*, long *width*, long *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.

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(long *x*, long *y*, long *width*, long *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 *wxWindow::OnPaint* (p. 1214) for an example of use.

Derived from

wxObject (p. 746)

Include files

<wx/region.h>

See also

wxWindow::OnPaint (p. 1214)

wxRegionIterator::wxRegionIterator

wxRegionIterator()

Default constructor.

wxRegionIterator(const wxRegion& region)

Creates an iterator object given a region.

wxRegionIterator::GetX

long GetX() const

Returns the x value for the current region.

wxRegionIterator::GetY

long GetY() const

Returns the y value for the current region.

wxRegionIterator::GetW

long GetW() const

An alias for GetWidth.

wxRegionIterator::GetWidth

long GetWidth() const

Returns the width value for the current region.

wxRegionIterator::GetH

long GetH() const

An alias for GetHeight.

wxRegionIterator::GetHeight

long GetWidth() const

Returns the width value for the current region.

wxRegionIterator::GetRect

wxRect GetRect() const

Returns the current rectangle.

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.

wxSashEvent

A sash event is sent when the sash of a *wxSashWindow* (p. 897) has been dragged by the user.

Derived from

wxCommandEvent (p. 152)

wxEvent (p. 375)

wxObject (p. 746)

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. 893) 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. 893) to see whether the drag was OK or out of the current allowed range.

See also

`wxSashWindow` (p. 897), *Event handling overview* (p. 1364)

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 range.

wxSashLayoutWindow

`wxSashLayoutWindow` responds to `OnCalculateLayout` events generated by `wxLayoutAlgorithm` (p. 610). 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. 610) explains the purpose of this class in more detail.

Derived from

`wxSashWindow` (p. 897)
`wxWindow` (p. 1184)
`wxEvtHandler` (p. 378)
`wxObject` (p. 746)

Include files

<wx/laywin.h>

Window styles

See `wxSashWindow` (p. 897).

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. 897) for relevant event information.

See also `wxLayoutAlgorithm` (p. 610) for information about the layout events.

See also

`wxLayoutAlgorithm` (p. 610), `wxSashWindow` (p. 897), *Event handling overview* (p. 1364)

wxSashLayoutWindow::wxSashLayoutWindow

wxSashLayoutWindow()

Default constructor.

wxSashLayoutWindow(wxSashLayoutWindow* parent, wxSashLayoutWindowID id, const wxPoint& pos = wxDefaultPosition, const wxSize& size = wxDefaultSize,

long *style* = *wxCAPTION* | *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. 894).

name

Window name.

wxSashLayoutWindow::~~wxSashLayoutWindow

~wxSashLayoutWindow()

Destructor.

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. 610) and *wxCalculateLayoutEvent* (p. 94).

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. 610) and *wxQueryLayoutInfoEvent* (p. 856).

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. 892) notification.

Derived from

wxWindow (p. 1184)
wxEvtHandler (p. 378)
wxObject (p. 746)

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. 1371).

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. 892), *wxSashLayoutWindow* (p. 894), *Event handling overview* (p. 1364)

wxSashWindow::wxSashWindow

wxSashWindow()

Default constructor.

wxSashWindow(**wxSashWindow*** *parent*, **wxSashWindowID** *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. 897).

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. 900)

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(int *min*) 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::SetSashBorder (p. 901)

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::GetSashVisible (p. 899)

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::HashBorder (p. 900)

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. 280)

Include files

<wx/dcscreen.h>

See also

wxDC (p. 280), *wxMemoryDC* (p. 678), *wxPaintDC* (p. 759), *wxClientDC* (p. 120), *wxWindowDC* (p. 1234)

wxScreenDC::wxScreenDC

wxScreenDC()

Constructor.

wxScreenDC::StartDrawingOnTop

bool StartDrawingOnTop(wxWindow* window)**bool StartDrawingOnTop(wxRect* rect = NULL)**

Use this in conjunction with *EndDrawingOnTop* (p. 902) 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. 973) 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. 902).

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. 176)

`wxWindow` (p. 1184)

`wxEvtHandler` (p. 378)

`wxObject` (p. 746)

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

wxB_HORIZONTAL Specifies a horizontal scrollbar.
wxB_VERTICAL Specifies a vertical scrollbar.

See also *window styles overview* (p. 1371).

Event handling

To process input from a scrollbar, use one of these event handler macros to direct input to member functions that take a *wxB_ScrollEvent* (p. 909) argument:

EVT_COMMAND_SCROLL(id, func) Catch all scroll commands.
EVT_COMMAND_SCROLL_TOP(id, func) Catch a command to put the scroll thumb at the maximum position.
EVT_COMMAND_SCROLL_BOTTOM(id, func) Catch a command to put the scroll thumb at the maximum position.
EVT_COMMAND_SCROLL_LINEUP(id, func) Catch a line up command.
EVT_COMMAND_SCROLL_LINEDOWN(id, func) Catch a line down command.
EVT_COMMAND_SCROLL_PAGEUP(id, func) Catch a page up command.
EVT_COMMAND_SCROLL_PAGEDOWN(id, func) Catch a page down command.
EVT_COMMAND_SCROLL_THUMBTRACK(id, func) Catch a thumbtrack command (continuous movement of the scroll thumb).

See also

Scrolling overview (p. 1386), *Event handling overview* (p. 1364), *wxB_ScrolledWindow* (p. 911)

wxB_ScrollBar::wxB_ScrollBar

wxB_ScrollBar()

Default constructor.

wxB_ScrollBar(wxB_Window* parent, wxWindowID id, const wxPoint& pos = wxDefaultPosition, const wxSize& size = wxDefaultSize, long style = wxB_HORIZONTAL, const wxValidator& validator = wxDefaultValidator, const wxString& name = "scrollBar")

Constructor, creating and showing a scrollbar.

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 *wxScrollBar* (p. 903).

validator

Window validator.

name

Window name.

See also

wxScrollBar::Create (p. 905), *wxValidator* (p. 1166)

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. 904) for details.

wxScrollBar::GetRange

int GetRange() const

Returns the length of the scrollbar.

See also

wxScrollBar::SetScrollbar (p. 907)

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. 907)

wxScrollBar::GetThumbPosition

int GetThumbPosition() const

Returns the current position of the scrollbar thumb.

See also

wxScrollBar::SetThumbPosition (p. 906)

wxScrollBar::GetThumbLength

int GetThumbLength() const

Returns the thumb or 'view' size.

See also

wxScrollBar::SetScrollbar (p. 907)

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. 906)

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 `wxWindow::OnSize` (p. 1216) event handler function.

See also

Scrolling overview (p. 1386), `wxWindow::SetScrollbar` (p. 1227), `wxScrolledWindow` (p. 911)

wxScrollWinEvent

A scroll event holds information about events sent from scrolling windows.

Derived from

`wxEvent` (p. 375)
`wxObject` (p. 746)

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 <code>wxEVT_SCROLLWIN_TOP</code> scroll-to-top events.
EVT_SCROLLWIN_BOTTOM(func)	Process <code>wxEVT_SCROLLWIN_TOP</code> scroll-to-bottom events.
EVT_SCROLLWIN_LINEUP(func)	Process <code>wxEVT_SCROLLWIN_LINEUP</code> line up events.
EVT_SCROLLWIN_LINEDOWN(func)	Process <code>wxEVT_SCROLLWIN_LINEDOWN</code> line down events.
EVT_SCROLLWIN_PAGEUP(func)	Process <code>wxEVT_SCROLLWIN_PAGEUP</code> page up events.
EVT_SCROLLWIN_PAGEDOWN(func)	Process <code>wxEVT_SCROLLWIN_PAGEDOWN</code> page down events.
EVT_SCROLLWIN_THUMBTRACK(func)	Process <code>wxEVT_SCROLLWIN_THUMBTRACK</code> thumbtrack events (frequent events sent as the user drags the thumbtrack).
EVT_SCROLLWIN_THUMBRELEASE(func)	Process

`wxEVT_SCROLLWIN_THUMBRELEASE`
thumb release events.

See also

wxWindow::OnScroll (p. 1215), *wxScrollEvent* (p. 909), *Event handling overview* (p. 1364)

wxScrollWinEvent::wxScrollWinEvent

`wxScrollWinEvent(WXTYPE commandType = 0, int id = 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.

wxScrollEvent

A scroll event holds information about events sent from stand-alone scrollbars, spin-buttons and sliders. Note that starting from `wxWindows 2.1`, scrolled windows send the *wxScrollWinEvent* (p. 908) 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. 152)
wxEvent (p. 375)
wxObject (p. 746)

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 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 <code>wxEVT_SCROLL_TOP</code> scroll-to-top events.
EVT_SCROLL_BOTTOM(func)	Process <code>wxEVT_SCROLL_BOTTOM</code> scroll-to-bottom events.
EVT_SCROLL_LINEUP(func)	Process <code>wxEVT_SCROLL_LINEUP</code> line up events.
EVT_SCROLL_LINEDOWN(func)	Process <code>wxEVT_SCROLL_LINEDOWN</code> line down events.
EVT_SCROLL_PAGEUP(func)	Process <code>wxEVT_SCROLL_PAGEUP</code> page up events.
EVT_SCROLL_PAGEDOWN(func)	Process <code>wxEVT_SCROLL_PAGEDOWN</code> page down events.
EVT_SCROLL_THUMBTRACK(func)	Process <code>wxEVT_SCROLL_THUMBTRACK</code> thumbtrack events (frequent events sent as the user drags the thumbtrack).
EVT_SCROLL_THUMBRELEASE(func)	Process <code>wxEVT_SCROLL_THUMBRELEASE</code> thumb release events.
EVT_COMMAND_SCROLL(id, func)	Process all scroll events.
EVT_COMMAND_SCROLL_TOP(id, func)	Process <code>wxEVT_SCROLL_TOP</code> scroll-to-top events.
EVT_COMMAND_SCROLL_BOTTOM(id, func)	Process <code>wxEVT_SCROLL_BOTTOM</code> scroll-to-bottom events.
EVT_COMMAND_SCROLL_LINEUP(id, func)	Process <code>wxEVT_SCROLL_LINEUP</code> line up events.
EVT_COMMAND_SCROLL_LINEDOWN(id, func)	Process <code>wxEVT_SCROLL_LINEDOWN</code> line down events.
EVT_COMMAND_SCROLL_PAGEUP(id, func)	Process <code>wxEVT_SCROLL_PAGEUP</code> page up events.
EVT_COMMAND_SCROLL_PAGEDOWN(id, func)	Process <code>wxEVT_SCROLL_PAGEDOWN</code> page down events.
EVT_COMMAND_SCROLL_THUMBTRACK(id, func)	Process <code>wxEVT_SCROLL_THUMBTRACK</code> thumbtrack events (frequent events sent as the user drags the thumbtrack).

EVT_COMMAND_SCROLL_THUMBRELEASE(func) Process
 wxEVT_SCROLL_THUMBRELEASE thumb
 release events.

Remarks

Note that unless specifying a scroll control identifier, you will need to test for scrollbar orientation with `wxScrollEvent::GetOrientation` (p. 911), since horizontal and vertical scroll events are processed using the same event handler.

See also

`wxScrollBar` (p. 903), `wxSlider` (p. 929), `wxSpinButton` (p. 963),
`wxScrollWinEvent` (p. 908), *Event handling overview* (p. 1364)

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.

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.

As with all windows, an application can draw onto a `wxScrolledWindow` using a *device*

context (p. 1391).

You have the option of handling the `OnPaint` handler or overriding the `OnDraw` (p. 917) function, which is passed a pre-scrolled device context (prepared by `PrepareDC` (p. 916)).

If you don't wish to calculate your own scrolling, you must call `PrepareDC` 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. 918) 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. 764)
`wxWindow` (p. 1184)
`wxEvtHandler` (p. 378)
`wxObject` (p. 746)

Include files

<wx/scrolwin.h>

Window styles

wxRETAINED Uses a backing pixmap to speed refreshes. Motif only.

See also *window styles overview* (p. 1371).

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. 903), *wxClientDC* (p. 120), *wxPaintDC* (p. 759)

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. 911).

name

Window name.

Remarks

The window is initially created without visible scrollbars. Call *wxScrolledWindow::SetScrollbars* (p. 917) 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, 0, &xx, &yy)` will return 10 in `yy`.

See also

CalcUnscrolledPosition (p. 914)

wxPython note: The wxPython version of this methods accepts only two parameters and returns `xx` and `yy` as a tuple of values.

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, 10, &xx, &yy)` will return 0 in `yy`.

See also

CalcScrolledPosition (p. 914)

wxPython note: The wxPython version of this methods accepts only two parameters and returns `xx` and `yy` as a tuple of values.

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. 913) 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. 917). 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. 917), *wxScrolledWindow::GetVirtualSize* (p. 915)

wxPython note: The wxPython version of this methods accepts no parameters and returns a tuple of values for xUnit and yUnit.

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. 283) and *wxDC::DeviceToLogicalY* (p. 283) to translate these units to logical units.

See also

wxScrolledWindow::SetScrollbars (p. 917), *wxScrolledWindow::GetScrollPixelsPerUnit* (p. 915)

wxPython note: The wxPython version of this methods accepts no parameters and returns a tuple of values for *x* and *y*.

wxScrolledWindow::IsRetained

bool IsRetained() const

Motif only: TRUE if the window has a backing bitmap.

wxScrolledWindow::PrepareDC

void PrepareDC(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.

PrepareDC is called automatically within the default *wxScrolledWindow::OnPaint* event handler, so your *wxScrolledWindow::OnDraw* (p. 917) 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);
    PrepareDC(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::PrepareDC` (p. 916).

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 `PrepareDC()` yourself.

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. 917), `wxScrolledWindow::GetScrollPixelsPerUnit` (p. 915)

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.

wxScrolledWindow::SetTargetWindow

void SetTargetWindow(wxWindow* window)

Call this function to tell `wxScrolledWindow` to perform the actual scrolling on a different

window (not on itself).

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. 1195), 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. 917)

wxPython note: The wxPython version of this methods accepts no parameters and returns a tuple of values for *x* and *y*.

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. 310)
wxWindow (p. 1184)
wxEvtHandler (p. 378)
wxObject (p. 746)

Include files

<wx/choicdlg.h>

See also

wxSingleChoiceDialog overview (p. 1401)

wxSingleChoiceDialog::wxSingleChoiceDialog

wxSingleChoiceDialog(*wxWindow** parent, **const wxString&** message, **const wxString&** caption, **int** n, **const wxString*** choices, **void**** clientData = NULL, **long** style = wxOK | wxCANCEL | wxCENTRE, **const wxPoint&** pos = wxDefaultPosition)

Constructor, taking an array of wxString choices and optional client data.

wxSingleChoiceDialog(*wxWindow** parent, **const wxString&** message, **const wxString&** caption, **const wxStringList&** choices, **void**** clientData = NULL, **long** style = wxOK | wxCANCEL | wxCENTRE, **const wxPoint&** pos = wxDefaultPosition)

Constructor, taking a string list 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.

style

A dialog style (bitlist) containing flags chosen from the following:

wxOK	Show an OK button.
wxCANCEL	Show a Cancel button.
wxCENTRE	Centre the message. Not Windows.

pos

Dialog position. Not Windows.

Remarks

Use `wxSingleChoiceDialog::ShowModal` (p. 921) 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.

wxSingleChoiceDialog::~wxSingleChoiceDialog

~wxSingleChoiceDialog()

Destructor.

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`.

wxSize

A **wxSize** is a useful data structure for graphics operations. It simply contains integer *width* and *height* members.

wxSize is used throughout wxWindows 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. 785), *wxRealPoint* (p. 868)

wxSize::wxSize

wxSize()

wxSize(int width, int height)

Creates a size object.

wxSize::GetWidth

int GetWidth() const

Gets the width member.

wxSize::GetHeight

int GetHeight() const

Gets the height member.

wxSize::Set

void Set(int *width*, int *height*)

Sets the width and height members.

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.

Derived from*wxEvent* (p. 375)*wxObject* (p. 746)**Include files**

<wx/event.h>

Event table macrosTo 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

wxWindow::OnSize (p. 1216), *wxSize* (p. 922), *Event handling overview* (p. 1364)

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 *wxBoxSizer* (p. 77), *wxStaticBoxSizer* (p. 986) or *wxNotebookSizer* (p. 734).

The layout algorithm used by sizers in wxWindows 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 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 do 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 wxWindows 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.

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. 746)

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

void Add(wxWindow* window, int option = 0, int flag = 0, int border = 0, wxObject* userData = NULL)

void Add(wxSizer* sizer, int option = 0, int flag = 0, int border = 0, wxObject* userData = NULL)

void Add(int width, int height, int option = 0, int flag = 0, int border = 0, wxObject* userData = NULL)

Adds the *window* to the sizer. As `wxSizer` itself is an abstract class, the parameters have no meaning in the `wxSizer` class itself, but as there currently is only one class deriving directly from `wxSizer` and this class does not override these methods, the meaning of the parameters is 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. 929).

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 *option* 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.

option

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 changable 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 where the border may be (`wxTOP`, `wxBOTTOM`, `wxLEFT`, `wxRIGHT` or `wxALL`). The other flags determine the child window's behaviour if the size of the sizer changes. However this is not - in contrast to the *option* flag - in the main orientation, but in the respectively other orientation. So if you created a `wxBoxSizer` with the `wxVERTICAL` option, these flags will be relevant if the sizer changes its horizontal size. A child may get resized to completely fill out the new size (using either `wxGROW` or `wxEXPAND`), it may get proportionally resized (`wxSHAPED`), it may get centered (`wxALIGN_CENTER` or `wxALIGN_CENTRE`) or it may get aligned to either side (`wxALIGN_LEFT` and `wxALIGN_TOP` are set to 0 and thus represent the default, `wxALIGN_RIGHT` and `wxALIGN_BOTTOM` have their obvious meaning). With proportional resize, a child may also be centered in the main orientation using `wxALIGN_CENTER_VERTICAL` (same as `wxALIGN_CENTRE_VERTICAL`) and `wxALIGN_CENTER_HORIZONTAL` (same as `wxALIGN_CENTRE_HORIZONTAL`) flags.

border

Determines the border width, if the *flag* parameter is set to any border.

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 *option* and *flag* will allow for.

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::Fit

void 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. 77).

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. 928), depending on which is bigger.

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

```
void Prepend(wxWindow* window, int option = 0, int flag = 0, int border = 0,
wxObject* userData = NULL)
```

```
void Prepend(wxSizer* sizer, int option = 0, int flag = 0, int border = 0, wxObject*
userData = NULL)
```

```
void Prepend(int width, int height, int option = 0, int flag = 0, int border= 0, wxObject*
userData = NULL)
```

Same as `wxSizer::Add` (p. 925), but prepends the items to the beginning of the list of items (windows, subsizers or spaces) owned by this sizer.

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(int nth)
```

Removes a child from the sizer. *window* is the window to be removed, *sizer* is the equivalent sizer and *nth* 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 and does not delete the window itself. Call `wxSizer::Layout` (p. 927) to update the layout "on screen" after removing a child from the sizer.

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. 925) and `Prepend` (p. 927) 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. 927) 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(int pos, 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 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. 77) if the window is resizable (as are many dialogs under Unix and frames on probably all platforms).

wxSlider

A slider is a control with a handle which can be pulled back and forth to change the value.

In Windows versions below Windows 95, a scrollbar is used to simulate the slider. In Windows 95, the track bar control is used.

Slider events are handled in the same way as a scrollbar.

Derived from

wxControl (p. 176)

wxWindow (p. 1184)

wxEvtHandler (p. 378)

wxObject (p. 746)

Include files

<wx/slider.h>

Window styles

wxSL_HORIZONTAL	Displays the slider horizontally.
wxSL_VERTICAL	Displays the slider vertically.
wxSL_AUTOTICKS	Displays tick marks.
wxSL_LABELS	Displays minimum, maximum and value labels. (NB: only displays the current value label under wxGTK)
wxSL_LEFT	Displays ticks on the left, if a vertical slider.
wxSL_RIGHT	Displays ticks on the right, if a vertical slider.
wxSL_TOP	Displays ticks on the top, if a horizontal slider.
wxSL_SELRange	Allows the user to select a range on the slider. Windows 95 only.

See also *window styles overview* (p. 1371).

Event handling

To process input from a slider, use one of these event handler macros to direct input to member functions that take a *wxCommandEvent* (p. 909) argument:

EVT_COMMAND_SCROLL(id, func)	Catch all scroll commands.
EVT_COMMAND_TOP(id, func)	Catch a command to put the scroll thumb at the maximum position.
EVT_COMMAND_BOTTOM(id, func)	Catch a command to put the scroll thumb at the maximum position.
EVT_COMMAND_LINEUP(id, func)	Catch a line up command.
EVT_COMMAND_LINEDOWN(id, func)	Catch a line down command.
EVT_COMMAND_PAGEUP(id, func)	Catch a page up command.
EVT_COMMAND_PAGEDOWN(id, func)	Catch a page down command.
EVT_COMMAND_THUMBTRACK(id, func)	Catch a thumbtrack command (continuous movement of the scroll thumb).
EVT_SLIDER(id, func)	Process a wxEVT_COMMAND_SLIDER_UPDATED event, when the slider is moved. Though provided for backward compatibility, this is obsolete.

See also

Event handling overview (p. 1364), *wxScrollBar* (p. 903)

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. 929).

validator

Window validator.

name

Window name.

See also

wxSlider::Create (p. 932), *wxValidator* (p. 1166)

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. 931) for further details.

wxSlider::GetLineSize

int GetLineSize() const

Returns the line size.

See also

wxSlider::SetLineSize (p. 935)

wxSlider::GetMax

int GetMax() const

Gets the maximum slider value.

See also

wxSlider::GetMin (p. 933), *wxSlider::SetRange* (p. 935)

wxSlider::GetMin

int GetMin() const

Gets the minimum slider value.

See also

wxSlider::GetMin (p. 933), *wxSlider::SetRange* (p. 935)

wxSlider::GetPageSize

int GetPageSize() const

Returns the page size.

See also

wxSlider::SetPageSize (p. 936)

wxSlider::GetSelEnd

int GetSelEnd() const

Returns the selection end point.

Remarks

Windows 95 only.

See also

wxSlider::GetSelStart (p. 933), *wxSlider::SetSelection* (p. 936)

wxSlider::GetSelStart

int GetSelStart() const

Returns the selection start point.

Remarks

Windows 95 only.

See also

wxSlider::GetSelEnd (p. 933), *wxSlider::SetSelection* (p. 936)

wxSlider::GetThumbLength

int GetThumbLength() const

Returns the thumb length.

Remarks

Windows 95 only.

See also

wxSlider::SetThumbLength (p. 936)

wxSlider::GetTickFreq

int GetTickFreq() const

Returns the tick frequency.

Remarks

Windows 95 only.

See also

wxSlider::SetTickFreq (p. 935)

wxSlider::GetValue

int GetValue() const

Gets the current slider value.

See also

wxSlider::GetMin (p. 933), *wxSlider::GetMax* (p. 932), *wxSlider::SetValue* (p. 937)

wxSlider::SetRange

void SetRange(int *minValue*, int *maxValue*)

Sets the minimum and maximum slider values.

See also

wxSlider::GetMin (p. 933), *wxSlider::GetMax* (p. 932)

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. 934)

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. 932)

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. 933)

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. 933), *wxSlider::GetSelEnd* (p. 933)

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. 934)

wxSlider::SetTick

void SetTick(int *tickPos*)

Sets a tick position.

Parameters

tickPos

The tick position.

Remarks

Windows 95 only.

See also

wxSlider::SetTickFreq (p. 935)

wxSlider::SetValue

void SetValue(int *value*)

Sets the slider position.

Parameters

value

The slider position.

See also

wxSlider::GetValue (p. 934)

wxSocketAddress

You are unlikely to need to use this class: only `wxSocketBase` uses it.

Derived from

`wxObject` (p. 746)

Include files

`<wx/socket.h>`

See also

`wxSocketBase` (p. 938) `wxIPv4address` (p. 595)

`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.

Derived from

`wxObject` (p. 746)

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 <code>wxSocket</code> .
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. 956) or accepted with *Accept* (p. 960). 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. 958) argument.

EVT_SOCKET(id, func) Process a wxEVT_SOCKET event.

See also

wxSocketEvent (p. 958), *wxSocketClient* (p. 956), *wxSocketServer* (p. 959), *Sockets sample* (p. 1326)

Construction and destruction

wxSocketBase (p. 942)
~wxSocketBase (p. 942)
Destroy (p. 943)

Socket state

Functions to retrieve current state and miscellaneous info.

Error (p. 944)
GetLocal (p. 944)
GetPeer (p. 945) *IsConnected* (p. 945)
IsData (p. 945)
IsDisconnected (p. 945)
LastCount (p. 946)
LastError (p. 946)
Ok (p. 946)
SaveState (p. 947)
RestoreState (p. 947)

Basic IO

Functions that perform basic IO functionality.

Close (p. 943)
Discard (p. 944)
Peek (p. 949)
Read (p. 950)
ReadMsg (p. 951)
Unread (p. 952)
Write (p. 954)
WriteMsg (p. 955)

Functions that perform a timed wait on a certain IO condition.

InterruptWait (p. 945)
Wait (p. 952)
WaitForLost (p. 953)
WaitForRead (p. 953)
WaitForWrite (p. 954)

and also:

wxSocketServer::WaitForAccept (p. 961)
wxSocketClient::WaitOnConnect (p. 957)

Functions that allow applications to customize socket IO as needed.

GetFlags (p. 944)
SetFlags (p. 948)
SetTimeout (p. 949)

Handling socket events

Functions that allow applications to receive socket events.

Notify (p. 946)
SetNotify (p. 949)
GetClientData (p. 944)
SetClientData (p. 947)
SetEventHandler (p. 947)

Callback functions are also available, but they are provided for backwards compatibility only. Their use is strongly discouraged in favour of events, and should be considered deprecated. Callbacks may be unsupported in future releases of wxWindows.

Callback (p. 942)
CallbackData (p. 942)

wxSocketBase::wxSocketBase

wxSocketBase()

Default constructor. Don't use it directly; instead, use *wxSocketClient* (p. 956) to construct a socket client, or *wxSocketServer* (p. 959) to construct a socket server.

wxSocketBase::~~wxSocketBase

~wxSocketBase()

Destructor. Do not destroy a socket using the delete operator directly; use *Destroy* (p. 943) instead. Also, do not create socket objects in the stack.

wxSocketBase::Callback

wxSocketBase::wxSockCbk Callback(wxSocketBase::wxSockCbk callback)

You can setup a callback function to be called when an event occurs. The function will be called only for those events for which notification has been enabled with *Notify* (p. 946) and *SetNotify* (p. 949). The prototype of the callback must be as follows:

```
void SocketCallback(wxSocketBase& sock, wxSocketNotify evt, char
*cdata);
```

The first parameter is a reference to the socket object in which the event occurred. The second parameter tells you which event occurred. (See *wxSocket events* (p. 938)). The third parameter is the user data you specified using *CallbackData* (p. 942).

Return value

A pointer to the previous callback.

Remark/Warning

Note that callbacks are now deprecated and unsupported, and they remain for backwards compatibility only. Use events instead.

See also

wxSocketBase::CallbackData (p. 942), *wxSocketBase::SetNotify* (p. 949),
wxSocketBase::Notify (p. 946)

wxSocketBase::CallbackData

char * CallbackData(char *cdata)

This function sets the the user data which will be passed to a callback function set via *Callback* (p. 942).

Return value

A pointer to the previous user data.

Remark/Warning

Note that callbacks are now deprecated and unsupported, and they remain for backwards compatibility only. Use events instead.

See also

wxSocketBase::Callback (p. 942), *wxSocketBase::SetNotify* (p. 949),
wxSocketBase::Notify (p. 946)

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. 943) 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. 946) to verify the number of bytes actually discarded.

If you use *Error* (p. 944), 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. 947)

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. 948)

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, ...).

Return value

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. 943) a socket (and thus also upon socket destruction), so you don't need to use it in these cases.

wxSocketBase::Wait (p. 952), *wxSocketServer::WaitForAccept* (p. 961),
wxSocketBase::WaitForLost (p. 953), *wxSocketBase::WaitForRead* (p. 953),
wxSocketBase::WaitForWrite (p. 954), *wxSocketClient::WaitOnConnect* (p. 957)

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. 938).

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. 944) 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. 949) 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. 956), Ok won't return TRUE unless the client is connected to a server.

For *wxSocketServer* (p. 959), 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. 944) instead for that purpose.

wxSocketBase::RestoreState

void RestoreState()

This function restores the previous state of the socket, as saved with *SaveState* (p. 947)

Calls to *SaveState* and *RestoreState* can be nested.

See also

wxSocketBase::SaveState (p. 947)

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. 948), event mask, as set with *SetNotify* (p. 949) and *Notify* (p. 946), user data, as set with *SetClientData* (p. 947), and asynchronous callback settings, as set with *Callback* (p. 942) and *CallbackData* (p. 942).

Calls to *SaveState* and *RestoreState* can be nested.

See also

wxSocketBase::RestoreState (p. 947)

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. 959) 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. 949) and *Notify* (p. 946).

Parameters

handler

Specifies the event handler you want to use.

id

The id of socket event.

See also

`wxSocketBase::SetNotify` (p. 949), `wxSocketBase::Notify` (p. 946), `wxSocketEvent` (p. 958), `wxEvtHandler` (p. 378)

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.

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.

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.

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. 938).

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. 952) 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. 946) to verify the number of bytes actually peeked.

Use *Error* (p. 944) 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. 948)

See also

`wxSocketBase::Error` (p. 944), `wxSocketBase::LastError` (p. 946),
`wxSocketBase::LastCount` (p. 946), `wxSocketBase::SetFlags` (p. 948)

wxSocketBase::Read

wxSocketBase& Read(void * *buffer*, wxUInt32 *nbytes*)

This function reads a buffer of *nbytes* bytes from the socket.

Use *LastCount* (p. 946) to verify the number of bytes actually read.

Use *Error* (p. 944) 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. 948).

See also

`wxSocketBase::Error` (p. 944), `wxSocketBase::LastError` (p. 946),
`wxSocketBase::LastCount` (p. 946), `wxSocketBase::SetFlags` (p. 948)

wxSocketBase::ReadMsg

wxSocketBase& ReadMsg(void * *buffer*, wxUint32 *nbytes*)

This function reads a buffer sent by `WriteMsg` (p. 955) 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. 946) to verify the number of bytes actually read.

Use `Error` (p. 944) 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. 948).

See also

`wxSocketBase::Error` (p. 944), `wxSocketBase::LastError` (p. 946),
`wxSocketBase::LastCount` (p. 946), `wxSocketBase::SetFlags` (p. 948),
`wxSocketBase::WriteMsg` (p. 955)

wxSocketBase::Unread

wxSocketBase& Unread(const void * *buffer*, wxUInt32 *nbytes*)

This function unread 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. 946), it will always return *nbytes*.

If you use *Error* (p. 944), 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. 944), *wxSocketBase::LastCount* (p. 946),
wxSocketBase::LastError (p. 946)

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. 956) only)
- An incoming connection request has arrived (*wxSocketServer* (p. 959) 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. 949).

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. 945), *wxSocketServer::WaitForAccept* (p. 961), *wxSocketBase::WaitForLost* (p. 953), *wxSocketBase::WaitForRead* (p. 953), *wxSocketBase::WaitForWrite* (p. 954), *wxSocketClient::WaitOnConnect* (p. 957)

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. 949).

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. 945), *wxSocketBase::Wait* (p. 952)

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. 949).

millisecond

Number of milliseconds to wait.

Return value

Returns TRUE if the socket becomes readable, FALSE on timeout.

See also

wxSocketBase::InterruptWait (p. 945), *wxSocketBase::Wait* (p. 952)

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. 949).

millisecond

Number of milliseconds to wait.

Return value

Returns TRUE if the socket becomes writable, FALSE on timeout.

See also

wxSocketBase::InterruptWait (p. 945), *wxSocketBase::Wait* (p. 952)

wxSocketBase::Write

wxSocketBase& Write(const void * buffer, wxUint32 nbytes)

This function writes a buffer of *nbytes* bytes to the socket.

Use *LastCount* (p. 946) to verify the number of bytes actually written.

Use *Error* (p. 944) 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. 948).

See also

`wxSocketBase::Error` (p. 944), `wxSocketBase::LastError` (p. 946),
`wxSocketBase::LastCount` (p. 946), `wxSocketBase::SetFlags` (p. 948)

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. 951) 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. 946) to verify the number of bytes actually written.

Use *Error* (p. 944) 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. 948).

See also

`wxSocketBase::Error` (p. 944), `wxSocketBase::LastError` (p. 946),
`wxSocketBase::LastCount` (p. 946), `wxSocketBase::SetFlags` (p. 948),
`wxSocketBase::ReadMsg` (p. 951)

wxSocketClient

Derived from

`wxSocketBase` (p. 938)

Include files

<wx/socket.h>

wxSocketClient::wxSocketClient

`wxSocketClient(wxSocketFlags flags = wxSOCKET_NONE)`

Constructor.

Parameters

flags

Socket flags (See `wxSocketBase::SetFlags` (p. 948))

wxSocketClient::~~wxSocketClient

`~wxSocketClient()`

Destructor. Please see `wxSocketBase::Destroy` (p. 943).

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. 957), 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. 957) or by watching **wxSOCKET_CONNECTION** and **wxSOCKET_LOST** events.

See also

wxSocketClient::WaitOnConnect (p. 957), *wxSocketBase::SetNotify* (p. 949),
wxSocketBase::Notify (p. 946)

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. 956) 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. 949).

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. 945) 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. 956), *wxSocketBase::InterruptWait* (p. 945),
wxSocketBase::IsConnected (p. 945)

wxSocketEvent

This event class contains information about socket events.

Derived from

wxEvent (p. 375)

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. 938), *wxSocketClient* (p. 956), *wxSocketServer* (p. 959)

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. 947).

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.

wxSocketServer

Derived from

wxSocketBase (p. 938)

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. 946).

Parameters

address

Specifies the local address for the server (e.g. port number).

flags

Socket flags (See *wxSocketBase::SetFlags* (p. 948))

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. 938) 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. 961) 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. 961), *wxSocketBase::SetNotify* (p. 949),
wxSocketBase::Notify (p. 946), *wxSocketServer::AcceptWith* (p. 961)

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. 961), *wxSocketBase::SetNotify* (p. 949),
wxSocketBase::Notify (p. 946), *wxSocketServer::Accept* (p. 960)

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. 960) or *AcceptWith* (p. 961) 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. 949).

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. 960), *wxSocketServer::AcceptWith* (p. 961), *wxSocketBase::InterruptWait* (p. 945)

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. 592)

Include files

<wx/sckstrm.h>

See also

wxSocketBase (p. 938)

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. 751)

Include files

<wx/sckstrm.h>

See also

wxSocketBase (p. 938)

wxSocketOutputStream::wxSocketOutputStream

wxSocketInputStream(wxSocketBase& s)

Creates a new write-only socket stream using the specified initialized socket connection.

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. 966) instead as *wxSpinButton* is not implemented for all platforms (Win32 and GTK only currently).

NB: the range supported by this control (and *wxSpinCtrl*) depends on the platform but is at least `SHRT_MIN` to `SHRT_MAX`.

Derived from

wxControl (p. 176)
wxWindow (p. 1184)
wxEvtHandler (p. 378)
wxObject (p. 746)

See also

wxSpinCtrl (p. 966)

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.
wxSP_WRAP	The value wraps at the minimum and maximum.

See also *window styles overview* (p. 1371).

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. 969) 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.

See also

Event handling overview (p. 1364)

wxSpinButton::wxSpinButton

wxSpinButton()

Default constructor.

wxSpinButton(wxWindow* parent, wxWindowID id, const wxPoint& pos = wxDefaultPosition, const wxSize& size = wxDefaultSize, long style = wxSP_HORIZONTAL, const wxValidator& validator = wxDefaultValidator, 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. 963).

validator

Window validator.

name

Window name.

See also

wxSpinButton::Create (p. 965), *wxValidator* (p. 1166)

wxSpinButton::~~wxSpinButton

void ~wxSpinButton()

Destructor, destroying the spin button.

wxSpinButton::Create

bool Create(wxWindow* parent, wxWindowID id, const wxPoint& pos = wxDefaultPosition, const wxSize& size = wxDefaultSize, long style = wxSP_HORIZONTAL, const wxValidator& validator = wxDefaultValidator, const wxString& name = "spinButton")

Scrollbar creation function called by the spin button constructor. See *wxSpinButton::wxSpinButton* (p. 964) for details.

wxSpinButton::GetMax

int GetMax() const

Returns the maximum permissible value.

See also

wxSpinButton::SetRange (p. 966)

wxSpinButton::GetMin

int GetMin() const

Returns the minimum permissible value.

See also

wxSpinButton::SetRange (p. 966)

wxSpinButton::GetValue

int GetValue() const

Returns the current spin button value.

See also

wxSpinButton::SetValue (p. 966)

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. 965), *wxSpinButton::GetMax* (p. 965)

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. 965)

wxSpinCtrl

wxSpinCtrl combines *wxTextCtrl* (p. 1070) and *wxSpinButton* (p. 963) in one control.

Derived from

wxControl (p. 176)
wxWindow (p. 1184)
wxEvtHandler (p. 378)
wxObject (p. 746)

Include files

<wx/spinctrl.h>

Window styles

wxSP_ARROW_KEYS The user can use arrow keys.
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. 969) argument:

EVT_SPINCTRL(id, func) Generated whenever spinctrl is updated

See also

Event handling overview (p. 1364), *wxSpinButton* (p. 963), *wxControl* (p. 176)

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. 963).

min

Minimal value.

max

Maximal value.

initial

Initial value.

name

Window name.

See also

wxSpinCtrl::Create (p. 968)

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. 967) 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::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. 963) and *wxSpinCtrl* (p. 966).

Derived from

wxNotifyEvent (p. 745)
wxCommandEvent (p. 152)
wxEvent (p. 375)
wxObject (p. 746)

Include files

<wx/spinbutt.h> or <wx/spinctrl.h>

See also

wxSpinButton (p. 963) and *wxSpinCtrl* (p. 966)

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.

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

wxCommandEvent (p. 152)

wxEvent (p. 375)

wxObject (p. 746)

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. 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 <code>wxEVT_COMMAND_SPLITTER_SASH_POS_CHANGING</code> 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 <code>wxEVT_COMMAND_SPLITTER_SASH_POS_CHANGED</code> event.
EVT_SPLITTER_UNSPPLIT(id, func)	The splitter has been just unsplit. Processes a <code>wxEVT_COMMAND_SPLITTER_UNSPPLIT</code> event.
EVT_SPLITTER_DOUBLECLICKED(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 <code>wxEVT_COMMAND_SPLITTER_DOUBLECLICKED</code> event.

See also

wxSplitterWindow (p. 973), *Event handling overview* (p. 1364)

wxSplitterEvent::wxSplitterEvent

wxSplitterEvent(`wxEventType` *eventType* = `wxEVT_NULL`,
wxSplitterWindow * *splitter* = `NULL`)

Constructor. Used internally by wxWindows 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_UNSPILT events.

wxSplitterEvent::SetSashPosition

void SetSashPosition(int pos)

In the case of wxEVT_COMMAND_SPLITTER_SASH_POS_CHANGED events, sets the 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. 1394)

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.

Appropriate 3D shading for the Windows 95 user interface is an option. This is also recommended for GTK.

Window styles

wxSP_3D	Draws a 3D effect border and sash.
wxSP_3DSASH	Draws a 3D effect sash.
wxSP_3DBORDER	Draws a 3D effect border.
wxSP_FULLSASH	Draws the ends of the sash (so the window can be used without a border).
wxSP_BORDER	Draws a thin black border around the window.
wxSP_NOBORDER	No border, and a black sash.
wxSP_PERMIT_UNSPPLIT	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. 1371).

Derived from

wxWindow (p. 1184)
wxEvtHandler (p. 378)
wxObject (p. 746)

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. 970) 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 <code>wxEVT_COMMAND_SPLITTER_SASH_POS_CHANGING</code> 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 <code>wxEVT_COMMAND_SPLITTER_SASH_POS_CHANGED</code> event.
EVT_SPLITTER_UNSPPLIT(id, func)	The splitter has been just unsplit. Processes a <code>wxEVT_COMMAND_SPLITTER_UNSPPLIT</code> event.
EVT_SPLITTER_DOUBLECLICKED(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 <code>wxEVT_COMMAND_SPLITTER_DOUBLECLICKED</code> event.

See also

wxSplitterEvent (p. 970)

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. 973).

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. 977), *wxSplitterWindow::SplitVertically* (p. 981) and *wxSplitterWindow::SplitHorizontally* (p. 980) 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. 977), *wxSplitterWindow::SplitVertically* (p. 981), *wxSplitterWindow::SplitHorizontally* (p. 980), *wxSplitterWindow::Create* (p. 976)

wxSplitterWindow::~~wxSplitterWindow

~wxSplitterWindow()

Destroys the wxSplitterWindow and its children.

wxSplitterWindow::Create

bool Create(wxWindow* parent, wxWindowID id, int x, 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. 975) for details.

wxSplitterWindow::GetMinimumPaneSize

int GetMinimumPaneSize() const

Returns the current minimum pane size (defaults to zero).

See also

wxSplitterWindow::SetMinimumPaneSize (p. 980)

wxSplitterWindow::GetSashPosition

int GetSashPosition()

Returns the current sash position.

See also

wxSplitterWindow::SetSashPosition (p. 979)

wxSplitterWindow::GetSplitMode

int GetSplitMode() const

Gets the split mode.

See also

wxSplitterWindow::SetSplitMode (p. 980), *wxSplitterWindow::SplitVertically* (p. 981),

wxSplitterWindow::SplitHorizontally (p. 980).

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.

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. 981), *wxSplitterWindow::SplitHorizontally* (p. 980)

wxSplitterWindow::IsSplit

bool IsSplit() const

Returns TRUE if the window is split, FALSE otherwise.

wxSplitterWindow::OnDoubleClickSash

virtual void OnDoubleClickSash(int x, int y)

Application-overrideable 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. 982) if the minimum pane size is zero.

See also

wxSplitterWindow::Unsplit (p. 982)

wxSplitterWindow::OnUnsplit

virtual void OnUnsplit(wxWindow* *removed*)

Application-overrideable 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-overrideable 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. 976)

See also

`wxSplitterWindow::Unsplit` (p. 982)

`wxSplitterWindow::SplitVertically` (p. 981)

`wxSplitterWindow::SplitHorizontally` (p. 980)

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. 976)

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_UNSPPLIT` 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. 976)

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. 976), *wxSplitterWindow::SplitVertically* (p. 981), *wxSplitterWindow::SplitHorizontally* (p. 980).

wxSplitterWindow::SplitHorizontally

bool SplitHorizontally(wxWindow* window1, wxWindow* window2, int sashPosition = 0)

Initializes the top and bottom panes of the splitter window.

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. 977).

See also

wxSplitterWindow::SplitVertically (p. 981), *wxSplitterWindow::IsSplit* (p. 977), *wxSplitterWindow::Unsplit* (p. 982)

wxSplitterWindow::SplitVertically

bool SplitVertically(wxWindow* window1, wxWindow* window2, int sashPosition = 0)

Initializes the left and right panes of the splitter window.

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, its 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. 977).

See also

wxSplitterWindow::SplitHorizontally (p. 980), *wxSplitterWindow::IsSplit* (p. 977), *wxSplitterWindow::Unsplit* (p. 982).

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. 978) which can be overridden for the desired behaviour. By default, the pane being removed is hidden.

See also

wxSplitterWindow::SplitHorizontally (p. 980), *wxSplitterWindow::SplitVertically* (p. 981), *wxSplitterWindow::IsSplit* (p. 977), *wxSplitterWindow::OnUnsplit* (p. 978)

wxStaticBitmap

A static bitmap control displays a bitmap.

Derived from

wxControl (p. 176)
wxWindow (p. 1184)
wxEvtHandler (p. 378)
wxObject (p. 746)

Include files

<wx/statbmp.h>

Window styles

There are no special styles for this control.

See also *window styles overview* (p. 1371).

See also

wxStaticBitmap (p. 982), *wxStaticBox* (p. 985)

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. 982).

name

Window name.

See also

wxStaticBitmap::Create (p. 984)

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. 983).

wxStaticBitmap::GetBitmap

wxBitmap& GetBitmap() const

Returns a reference to the label bitmap.

See also

wxStaticBitmap::SetBitmap (p. 984)

wxStaticBitmap::SetBitmap

virtual void SetBitmap(const wxBitmap& label)

Sets the bitmap label.

Parameters

label

The new bitmap.

See also

wxStaticBitmap::GetBitmap (p. 984)

wxStaticBox

A static box is a rectangle drawn around other panel items to denote a logical grouping of items.

Derived from

wxControl (p. 176)

wxWindow (p. 1184)

wxEvtHandler (p. 378)

wxObject (p. 746)

Include files

<wx/statbox.h>

Window styles

There are no special styles for this control.

See also *window styles overview* (p. 1371).

See also

wxStaticText (p. 989)

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. 985).

name

Window name.

See also

wxStaticBox::Create (p. 986)

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. 985) for further details.

wxStaticBoxSizer

`wxStaticBoxSizer` is a sizer derived from `wxBoxSizer` but adds a static box around the sizer. Note that this static box has to be created separately.

See also `wxSizer` (p. 924), `wxStaticBox` (p. 985) and `wxBoxSizer` (p. 77).

Derived from

`wxBoxSizer` (p. 77)

`wxSizer` (p. 924)

`wxObject` (p. 746)

`wxStaticBoxSizer::wxStaticBoxSizer`

`wxStaticBoxSizer(wxStaticBox* box, int orient)`

Constructor. It takes an associated static box and the orientation *orient* as parameters - orient can be either of `wxVERTICAL` or `wxHORIZONTAL`.

`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. 176)

`wxWindow` (p. 1184)

`wxEvtHandler` (p. 378)

`wxObject` (p. 746)

Include files

<wx/statline.h>

Window styles

wxLI_HORIZONTAL	Creates a horizontal line.
wxLI_VERTICAL	Creates a vertical line.

See also

wxStaticBox (p. 985)

wxStaticLine::wxStaticLine

wxStaticLine()

Default constructor.

wxStaticLine(*wxWindow** parent, **wxWindowID** id, **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 is horizontal or vertical) is ignored.

style

Window style (either *wxLI_HORIZONTAL* or *wxLI_VERTICAL*).

name

Window name.

See also

wxStaticLine::Create (p. 989)

wxStaticLine::Create

```
bool Create(wxWindow* parent, wxWindowID id, 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. 988) 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. 176)
wxWindow (p. 1184)
wxEvtHandler (p. 378)
wxObject (p. 746)

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 <i>SetLabel</i> (p. 991) 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. 1371).

See also

wxStaticBitmap (p. 982), *wxStaticBox* (p. 985)

wxStaticText::wxStaticText

wxStaticText()

Default constructor.

wxStaticText(wxWindow* parent, wxWindowID id, const wxString& label, const wxPoint& pos, 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. 989).

name

Window name.

See also

wxStaticText::Create (p. 991)

wxStaticText::Create

bool Create(*wxWindow** parent, *wxWindowID* id, **const wxString&** label, **const wxPoint&** pos, **const wxSize&** size = *wxDefaultSize*, **long** style = 0, **const wxString&** name = "staticText")

Creation function, for two-step construction. For details see *wxStaticText::wxStaticText* (p. 990).

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. 1184)

wxEvtHandler (p. 378)

wxObject (p. 746)

Derived from

wxWindow (p. 1184)

wxEvtHandler (p. 378)

wxObject (p. 746)

Include files

<wx/statusbr.h>

Window styles

wxSB_SIZEGRIP On Windows 95, displays a gripper at right-hand side of the status bar.

See also *window styles overview* (p. 1371).

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. 452), *Status bar sample* (p. 1326)

wxStatusBar::wxStatusBar

wxStatusBar()

Default constructor.

wxStatusBar(**wxWindow*** *parent*, **wxWindowID** *id*, **const wxPoint&** *pos* = *wxDefaultPosition*, **const wxSize&** *size* = *wxDefaultSize*, **long** *style* = 0, **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.

pos

The window position. A value of (-1, -1) indicates a default position, chosen by either the windowing system or wxWindows, depending on platform.

size

The window size. A value of (-1, -1) indicates a default size, chosen by either the

windowing system or `wxWindows`, depending on platform.

style

The window style. See `wxStatusBar` (p. 991).

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. 993)

wxStatusBar::~~wxStatusBar

void ~wxStatusBar()

Destructor.

wxStatusBar::Create

bool Create(wxWindow* parent, wxWindowID id, const wxPoint& pos = wxDefaultPosition, const wxSize& size = wxDefaultSize, long style = 0, const wxString& name = "statusBar")

Creates the window, for two-step construction.

See `wxStatusBar::wxStatusBar` (p. 992) for details.

wxStatusBar::GetFieldRect

virtual bool GetFieldRect(int i, wxRect& rect) const

Returns the size and position of a fields 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. 868)

wxStatusBar::GetFieldsCount

int GetFieldsCount() const

Returns the number of fields in the status bar.

wxStatusBar::GetStatusText

virtual wxString GetStatusText(int *ir* = 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. 996)

wxStatusBar::DrawField

virtual void DrawField(wxDC& *dc*, int *i*)

Draws a field, including shaded borders and text.

Parameters

dc

The device context to draw onto.

i

The field to be drawn.

See also

wxStatusBar::DrawFieldText (p. 995)

wxStatusBar::DrawFieldText

virtual void DrawFieldText(wxDC& *dc*, int *i*)

Draws a field's text.

Parameters

dc

The device context to draw onto.

i

The field whose text is to be drawn.

See also

wxStatusBar::DrawField (p. 994)

wxStatusBar::InitColours

virtual void InitColours()

Sets up the background colour and shading pens using suitable system colours (Windows) or tasteful shades of grey (other platforms).

Remarks

This function is called when the window is created, and also from *wxStatusBar::OnSysColourChanged* (p. 995) on Windows.

See also

wxStatusBar::OnSysColourChanged (p. 995)

wxStatusBar::OnSysColourChanged

void OnSysColourChanged(wxSysColourChangedEvent& *event*)

Handles a system colour change by calling *wxStatusBar::InitColours* (p. 995), and refreshes the window.

Parameters

event

The colour change event.

See also

wxStatusBar::InitColours (p. 995)

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.

Parameters

number

The number of fields.

widths

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.

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. 994), *wxFrame::SetStatusText* (p. 462)

wxStatusBar::SetStatusWidths

virtual void SetStatusWidths(int *n*, int **widths*)

Sets the widths of the fields in the status line.

Parameters

n

The number of fields in the status bar.

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.

See also

wxStatusBar::SetFieldsCount (p. 996), *wxFrame::SetStatusWidths* (p. 462)

wxPython note: Only a single parameter is required, a Python list of integers.

wxStopWatch

The `wxStopWatch` class allow you to measure time intervals.

Include files

<wx/timer.h>

See also

::wxStartTimer (p. 1302), *::wxGetElapsedTime* (p. 1300), *wxTimer* (p. 1113)

wxStopWatch::wxStopWatch

wxStopWatch()

Constructor. This starts the stop watch.

wxStopWatch::Pause

void Pause()

Pauses the stop watch. Call *wxStopWatch::Resume* (p. 998) to resume time measuring again.

wxStopWatch::Start

void Start(long milliseconds = 0)

(Re)starts the stop watch with a given initial value.

wxStopWatch::Resume

void Resume()

Resumes the stop watch after having been paused with *wxStopWatch::Pause* (p. 998).

wxStopWatch::Time

long Time()

Returns the time in milliseconds since the start (or restart) or the last call of *wxStopWatch::Pause* (p. 998).

wxStreamBase

This class is the base class of most stream related classes in wxWindows. It must not be used directly.

Derived from

None

Include files

<wx/stream.h>

See also

wxStreamBuffer (p. 1000)

wxStreamBase::wxStreamBase

wxStreamBase()

Creates a dummy stream object. It doesn't do anything.

wxStreamBase::~~wxStreamBase

~wxStreamBase()

Destructor.

wxStreamBase::IsOk

wxStreamError IsOk() const

Returns TRUE if no error occurred on the stream.

See also

LastError (p. 999)

wxStreamBase::LastError

wxStreamError LastError() 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::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 called when the stream needs to know the real position.

wxStreamBase::OnSysWrite

size_t OnSysWrite(void *buffer, size_t bufsize)

See *OnSysRead* (p. 999).

wxStreamBase::GetSize

size_t GetSize() const

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 an invalid size represented by

```
~(size_t)0
```

wxStreamBuffer

Derived from

None

Include files

<wx/stream.h>

See also

wxStreamBase (p. 998)

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 may not work: it isn't completely finished.

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. 1004)

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 *LastError* (p. 999).

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. 1002)

wxStreamBuffer::Write

size_t Write(const void **buffer*, **size_t** *size*)

Writes a block of the specified *size* using datas of *buffer*. The datas are cached in a buffer before being sent in one block to the stream.

size_t Write(wxStreamBuffer **buffer*)

See *Read* (p. 1002).

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. 1002)

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. 1002)

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. 1001)).

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. 1001)

wxStreamBuffer::Fixed (p. 1006)

wxStreamBuffer::Flushable (p. 1006)

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. 1006)

wxStreamBuffer::Flushable (p. 1006)

wxStreamBuffer::GetBufferStart

char * GetBufferStart() const

Returns a pointer on the start of the stream buffer.

wxStreamBuffer::GetBufferEnd

char * GetBufferEnd() const

Returns a pointer on the end of the stream buffer.

wxStreamBuffer::GetBufferPos

char * 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()

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: specify 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. 1004)

wxStreamBuffer::Flushable

void Flushable(bool *flushable*)

Toggles the flushable flag. If *flushable* is disabled, no datas 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 datas in the buffer.

wxStreamBuffer::Stream

wxStreamBase* Stream()

Returns the parent stream of the stream buffer.

wxString

wxString is a class representing a character string. Please see the *wxString overview* (p. 1331) 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.

Derived from

None

Include files

<wx/string.h>

Predefined objects

Objects:

wxEmptyString

See also

Overview (p. 1331)

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. 1014)
operator = (p. 1026)
~wxString (p. 1014)

String length

These functions return the string length and check whether the string is empty or empty it.

Len (p. 1021)
IsEmpty (p. 1019)

operator! (p. 1026)
Empty (p. 1017)
Clear (p. 1016)

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. 1356), 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. 1016) method for the sake of clarity. Also see *overview* (p. 1333) for the cases where it is necessary to use it.

GetChar (p. 1018)
GetWritableChar (p. 1019)
SetChar (p. 1024)
Last (p. 1021)
operator [] (p. 1027)
c_str (p. 1016)
*operator const char** (p. 1028)

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. 1028)
operator += (p. 1027)
operator + (p. 1027)
Append (p. 1015)
Prepend (p. 1022)

Comparison

The default comparison function *Cmp* (p. 1016) is case-sensitive and so is the default version of *IsSameAs* (p. 1020). For case insensitive comparisons you should use *CmpNoCase* (p. 1016) 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. 1022) is a poor man's regular expression matcher: it only understands '*'

and '?' metacharacters in the sense of DOS command line interpreter.

StartsWith (p. 1024) 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. 1016)
CmpNoCase (p. 1016)
IsSameAs (p. 1020)
Matches (p. 1022)
StartsWith (p. 1024)

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. 1022)
operator() (p. 1027)
Left (p. 1021)
Right (p. 1023)
BeforeFirst (p. 1015)
BeforeLast (p. 1016)
AfterFirst (p. 1015)
AfterLast (p. 1015)
StartsWith (p. 1024)

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. 1022)
Upper (p. 1026)
MakeLower (p. 1022)
Lower (p. 1021)

Searching and replacing

These functions replace the standard *strchr()* and *strstr()* functions.

Find (p. 1017)
Replace (p. 1023)

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. 1025)

ToULong (p. 1025)

ToDouble (p. 1025)

Writing values into the string

Both formatted versions (*Printf* (p. 1022)) and stream-like insertion operators exist (for basic types only). Additionally, the *Format* (p. 1018) 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. 1018)

FormatV (p. 1018)

Printf (p. 1022)

PrintfV (p. 1023)

operator << (p. 1028)

Memory management

These are "advanced" functions and they will be needed quite rarely. *Alloc* (p. 1014) and *Shrink* (p. 1024) are only interesting for optimization purposes. *GetWriteBuf* (p. 1019) 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. 1026) **must** be called!

Alloc (p. 1014)

Shrink (p. 1024)

GetWriteBuf (p. 1019)

UngetWriteBuf (p. 1026)

Miscellaneous

Other string functions.

Trim (p. 1025)

Pad (p. 1022)

Truncate (p. 1026)

wxWindows 1.xx compatibility functions

These functions are deprecated, please consider using new wxWindows 2.0 functions instead of them (or, even better, `std::string` compatible variants).

SubString (p. 1024)
sprintf (p. 1024)
CompareTo (p. 1017)
Length (p. 1021)
Freq (p. 1018)
LowerCase (p. 1021)
UpperCase (p. 1026)
Strip (p. 1024)
Index (p. 1019)
Remove (p. 1023)
First (p. 1017)
Last (p. 1021)
Contains (p. 1017)
IsNull (p. 1020)
IsAscii (p. 1019)
IsNumber (p. 1020)
IsWord (p. 1020)

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.

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.

wxString(const wchar_t* psz)

Constructs a string from the wide (UNICODE) string.

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 of reallocating 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. 1026) 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 char * c_str() const

Returns a pointer to the string data.

wxString::Clear

void Clear()

Empties the string and frees memory occupied by it.

See also: *Empty* (p. 1017)

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. 1016), *IsSameAs* (p. 1020).

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. 1016), *IsSameAs* (p. 1020).

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. 1016).

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

size_t First(char c)

size_t First(const char* psz) const

size_t First(const wxString& str) const

size_t First(const char *ch*) const

Returns the first occurrence of the item.

wxString::Format

static wxString Format(const wxChar **format*, ...)

This static function returns the string containing the result of calling *Printf* (p. 1022) with the passed parameters on it.

See also

FormatV (p. 1018), *Printf* (p. 1022)

wxString::FormatV

static wxString Format(const wxChar **format*, va_list *argptr*)

This static function returns the string containing the result of calling *PrintfV* (p. 1023) with the passed parameters on it.

See also

Format (p. 1018), *PrintfV* (p. 1023)

wxString::Freq

int Freq(char *ch*) const

Returns the number of occurrences of *ch* in the string.

wxString::GetChar

char GetChar(size_t *n*) const

Returns the character at position *n* (read-only).

wxString::GetData

const char* GetData() const

wxWindows 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

char* GetWriteBuf(size_t len)Returns a writable buffer of at least *len* bytes.Call *wxString::UngetWriteBuf* (p. 1026) as soon as possible to put the string back into a reasonable state.

wxString::Index

size_t Index(char ch, int startpos = 0) constSame as *wxString::Find* (p. 1017).**size_t Index(const char* sz) const**Same as *wxString::Find* (p. 1017).**size_t Index(const char* sz, bool caseSensitive = TRUE, bool fromEnd = FALSE) const**

Search the element in the array, starting from either side.

If *fromEnd* is TRUE, reverse search direction.If **caseSensitive**, comparison is case sensitive (the default).

Returns the index of the first item matched, or NOT_FOUND.

wxString::IsAscii

bool IsAscii() const

Returns TRUE if the string is ASCII.

wxString::IsEmpty

bool IsEmpty() const

Returns TRUE if the string is NULL.

wxString::IsNull

bool IsNull() const

Returns TRUE if the string is NULL (same as IsEmpty).

wxString::IsNumber

bool IsNumber() const

Returns TRUE if the string is a positive or negative integer. Will return FALSE for decimals.

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. 1016), *CmpNoCase* (p. 1016), *IsSameAs* (p. 1020)

wxString::IsSameAs

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. 1016), *CmpNoCase* (p. 1016), *IsSameAs* (p. 1020)

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) constReturns the first *count* characters.**wxString Left(char ch) const**Returns all characters before the first occurrence of *ch*. Returns the whole string if *ch* is not found.**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

void MakeLower()

Converts all characters to lower case.

wxString::MakeUpper

void MakeUpper()

Converts all characters to upper case.

wxString::Matches

bool Matches(const char* szMask) const

Returns TRUE if the string contents matches a mask containing '*' and '?'.

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. 1014) 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. 1022) instead (but note that parameters have different meaning).

Returns the part of the string between the indices *from* and *to* inclusive.

wxString::ToDouble

bool ToDouble(double *va) 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 *va*) or FALSE if the string does not represent such number.

See also

wxString::ToLong (p. 1025),
wxString::ToULong (p. 1025)

wxString::ToLong

bool ToLong(long *va) const

Attempts to convert the string to a signed integer. Returns TRUE on success (the number is stored in the location pointed to by *va*) or FALSE if the string does not represent such number.

See also

wxString::ToDouble (p. 1025),
wxString::ToULong (p. 1025)

wxString::ToULong

bool ToULong(unsigned long *va) const

Attempts to convert the string to an unsigned integer. Returns TRUE on success (the number is stored in the location pointed to by *va*) or FALSE if the string does not represent such number.

See also

wxString::ToDouble (p. 1025),
wxString::ToLong (p. 1025)

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()

Puts the string back into a reasonable state, after *wxString::GetWriteBuf* (p. 1019) was called.

wxString::Upper

wxString Upper() *const*

Returns this string converted to upper case.

wxString::UpperCase

void UpperCase()

The same as *MakeUpper*.

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. 1019).

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. 1014)).

wxString::operator +

Concatenation: all these operators return a new string equal to the sum 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. 1022) 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.

wxStringFormValidator

This class validates a string value for a form view, with an optional choice of possible values. The associated panel item must be a wxText, wxListBox or wxChoice. For wxListBox and wxChoice items, if the item is empty, the validator attempts to initialize the item from the strings in the validator. Note that this does not happen for XView wxChoice items since XView cannot reinitialize a wxChoice.

See also

Validator classes (p. 1453)

wxStringFormValidator::wxStringFormValidator

void wxStringFormValidator(wxStringList *list=NULL, long flags=0)

Constructor. Supply a list of strings to indicate a choice, or no strings to allow the user to freely edit the string. The string list will be deleted when the validator is deleted.

wxStringList

A string list is a list which is assumed to contain strings. Memory is allocated when strings are added to the list, and deallocated by the destructor or by the **Delete** member.

Derived from

wxList (p. 615)
wxObject (p. 746)

Include files

<wx/list.h>

See also

wxString (p. 1007), *wxList* (p. 615)

wxStringList::wxStringList

wxStringList()

Constructor.

void wxStringList(char* first, ...)

Constructor, taking NULL-terminated string argument list. *wxStringList* allocates memory for the strings.

wxStringList::~~wxStringList

~wxStringList()

Deletes string list, deallocating strings.

wxStringList::Add

wxNode * Add(const wxString& s)

Adds string to list, allocating memory.

wxStringList::Clear

void Clear()

Clears all strings from the list.

wxStringList::Delete

void Delete(const wxString& s)

Searches for string and deletes from list, deallocating memory.

wxStringList::ListToArray

char* ListToArray(bool new_copies = FALSE)

Converts the list to an array of strings, only allocating new memory if **new_copies** is TRUE.

wxStringList::Member

bool Member(const wxString& s)

Returns TRUE if **s** is a member of the list (tested using **strcmp**).

wxStringList::Sort

void Sort()

Sorts the strings in ascending alphabetical order. Note that all nodes (but not strings) get deallocated and new ones allocated.

wxStringListValidator

This class validates a string value, with an optional choice of possible values.

See also

Validator classes (p. 1453)

wxStringListValidator::wxStringListValidator

void wxStringListValidator(wxStringList *list=NULL, long flags=0)

Constructor. Supply a list of strings to indicate a choice, or no strings to allow the user to freely edit the string. The string list will be deleted when the validator is deleted.

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. 1033) may be called repeatedly until it `HasMoreTokens` (p. 1033) returns `FALSE`.

For example:

```
wxStringTokenizer tkz("first:second:third:fourth", ":");
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. 1033)/*SetString* (p. 1034) parameter `mode` which may be one of the following:

<code>wxTOKEN_DEFAULT</code>	Default behaviour (as described above): same as <code>wxTOKEN_STRTOK</code> if the delimiter string contains only whitespaces, same as <code>wxTOKEN_RET_EMPTY</code> otherwise
<code>wxTOKEN_RET_EMPTY</code>	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'.
<code>wxTOKEN_RET_EMPTY_ALL</code>	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.
<code>wxTOKEN_RET_DELIMS</code>	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 <code>wxTOKEN_RET_EMPTY</code> .
<code>wxTOKEN_STRTOK</code>	In this mode the class behaves exactly like the standard

`strtok()` function. The empty tokens are never returned.

Derived from

wxObject (p. 746)

Include files

<wx/tokenzr.h>

wxStringTokenizer::wxStringTokenizer

wxStringTokenizer()

Default constructor. You must call *SetString* (p. 1034) before calling any other methods.

wxStringTokenizer(const wxString& str, const wxString& delims = " \\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 in the input string.

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. 375)

wxObject (p. 746)

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

wxWindow::OnSysColourChanged (p. 1217), *Event handling overview* (p. 1364)

wxSysColourChangedEvent::wxSysColourChanged

wxSysColourChanged()

Constructor.

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

wxObject (p. 746)

Include files

<wx/settings.h>

See also

wxFont (p. 434), *wxColour* (p. 135)

wxSystemSettings::wxSystemSettings

wxSystemSettings()

Default constructor. You don't need to create an instance of wxSystemSettings since all of its functions are static.

wxSystemSettings::GetSystemColour

static wxColour GetSystemColour(int *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_GetSystemColour`

wxSystemSettings::GetSystemFont

static wxFont GetSystemFont(int *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. Not available in versions of Windows earlier than Windows 95 or Windows NT 4.0.

wxPython note: This static method is implemented in Python as a standalone function named `wxSystemSettings_GetSystemFont`

wxSystemSettings::GetSystemMetric

static int GetSystemMetric(int *index*)

Returns a system 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.

wxPython note: This static method is implemented in Python as a standalone function named `wxSystemSettings_GetSystemMetric`

wxTabbedDialog

A dialog suitable for handling tabs.

Please note that the preferred class for programming tabbed windows is *wxNotebook* (p. 736). This class is retained for backward compatibility.

Derived from

wxDialog (p. 310)

wxEvtHandler (p. 378)

wxObject (p. 746)

Include files

<wx/tab.h>

See also

Tab classes overview (p. 1408)

wxTabbedDialog::wxTabbedDialog

wxTabbedDialog(*wxWindow* *parent, *wxWindowID* id, **const wxString&** title, **const wxPoint&** pos, **const wxSize&** size, **long** style=wxDEFAULT_DIALOG_STYLE, **const wxString&** name="dialogBox")

Constructor.

wxTabbedDialog::~~wxTabbedDialog

~wxTabbedDialog()

Destructor. This destructor deletes the tab view associated with the dialog box. If you do not wish this to happen, set the tab view to NULL before destruction (for example, in the OnCloseWindow event handler).

wxTabbedDialog::SetTabView

void SetTabView(*wxTabView* *view)

Sets the tab view associated with the dialog box.

wxTabbedDialog::GetTabView

wxTabView * GetTabView()

Returns the tab view associated with the dialog box.

wxTabbedPanel

A panel suitable for handling tabs.

Please note that the preferred class for programming tabbed windows is *wxNotebook* (p. 736). This class is retained for backward compatibility.

Derived from

wxPanel (p. 764)

wxEvtHandler (p. 378)

wxObject (p. 746)

Include files

<wx/tab.h>

See also

Tab classes overview (p. 1408)

wxTabbedPanel::wxTabbedPanel

wxTabbedPanel(wxWindow *parent, wxWindowID id, const wxPoint& pos, const wxSize& size, long style=0, const wxString& name="panel")

Constructor.

wxTabbedPanel::SetTabView

void SetTabView(wxTabView *view)

Sets the tab view associated with the panel.

wxTabbedPanel::GetTabView

wxTabView * GetTabView()

Returns the tab view associated with the panel.

wxTabControl

You will rarely need to use this class directly.

Please note that the preferred class for programming tabbed windows is *wxNotebook* (p. 736). This class is retained for backward compatibility.

Derived from

wxObject (p. 746)

Include files

<wx/tab.h>

See also

Tab classes overview (p. 1408)

wxTabControl::wxTabControl

```
void wxTabControl(wxTabView *view = NULL)
```

Constructor.

wxTabControl::GetColPosition

```
int GetColPosition()
```

Returns the position of the tab in the tab column.

wxTabControl::GetFont

```
wxFont * GetFont()
```

Returns the font to be used for this tab.

wxTabControl::GetHeight

int GetHeight()

Returns the tab height.

wxTabControl::GetId

int GetId()

Returns the tab identifier.

wxTabControl::GetLabel

wxString GetLabel()

Returns the tab label.

wxTabControl::GetRowPosition

int GetRowPosition()

Returns the position of the tab in the layer or row.

wxTabControl::GetSelected

bool GetSelected()

Returns the selected flag.

wxTabControl::GetWidth

int GetWidth()

Returns the tab width.

wxTabControl::GetX

int GetX()

Returns the x offset from the top-left of the view area.

wxTabControl::GetY

int GetY()

Returns the y offset from the top-left of the view area.

wxTabControl::HitTest

bool HitTest(int x, int y)

Returns TRUE if the point x, y is within the tab area.

wxTabControl::OnDraw

void OnDraw(wxDC& dc, bool lastInRow)

Draws the tab control on the given device context.

wxTabControl::SetColPosition

void SetColPosition(int pos)

Sets the position in the column.

wxTabControl::SetFont

void SetFont(wxFont *font)

Sets the font to be used for this tab.

wxTabControl::SetId

void SetId(int id)

Sets the tab identifier.

wxTabControl::SetLabel

void SetLabel(const wxString& str)

Sets the label for the tab.

wxTabControl::SetPosition

void SetPosition(int x, int y)

Sets the x and y offsets for this tab, measured from the top-left of the view area.

wxTabControl::SetRowPosition

void SetRowPosition(int pos)

Sets the position on the layer (row).

wxTabControl::SetSelected

void SetSelected(bool selected)

Sets the selection flag for this tab (does not set the current tab for the view; use `wxTabView::SetSelectedTab` for that).

wxTabControl::SetSize

void SetSize(int width, int height)

Sets the width and height for this tab.

wxTabView

Responsible for drawing tabs onto a window, and dealing with input.

Please note that the preferred class for programming tabbed windows is `wxNotebook` (p. 736). This class is retained for backward compatibility.

Derived from

`wxObject` (p. 746)

Include files

<wx/tab.h>

See also

`wxTabView` overview (p. 1412), `wxPanelTabView` (p. 767)

wxTabView::wxTabView

wxTabView(long *style* = wxTAB_STYLE_DRAW_BOX | wxTAB_STYLE_COLOUR_INTERIOR)

Constructor.

style may be a bit list of the following:

wxTAB_STYLE_DRAW_BOX	Draw a box around the view area. Most commonly used for dialogs.
wxTAB_STYLE_COLOUR_INTERIOR	Draw tab backgrounds in the specified colour. Omitting this style will ensure that the tab background matches the dialog background.

wxTabView::AddTab

wxTabControl * AddTab(int *id*, const wxString& *label*, wxTabControl **existingTab*=NULL)

Adds a tab to the view.

id is the application-chosen identifier for the tab, which will be used in subsequent tab operations.

label is the label to give the tab.

existingTab maybe NULL to specify a new tab, or non-NULL to indicate that an existing tab should be used.

A new layer (row) is started when the current layer has been filled up with tabs.

wxTabView::CalculateTabWidth

int CalculateTabWidth(int *noTabs*, bool *adjustView* = FALSE)

The application can specify the tab width using this function, in terms of the number of tabs per layer (row) which will fit the view area, which should have been set previously with SetViewRect.

noTabs is the number of tabs which should take up the full width of the view area.

adjustView can be set to TRUE in order to readjust the view width to exactly fit the given number of tabs.

The new tab width is returned.

wxTabView::ClearTabs

void ClearTabs(bool *deleteTabs=TRUE*)

Clears the tabs, deleting them if *deleteTabs* is TRUE.

wxTabView::Draw

void Draw(wxDC& *dc*)

Draws the tabs and (optionally) a box around the view area.

wxTabView::FindTabControlForId

wxTabControl * FindTabControlForId(int *id*)

Finds the wxTabControl corresponding to *id*.

wxTabView::FindTabControlForPosition

wxTabControl * FindTabControlForPosition(int *layer*, int *position*)

Finds the wxTabControl at layer *layer*, position in layer *position*, both starting from zero. Note that tabs change layer as they are selected or deselected.

wxTabView::GetBackgroundBrush

wxBrush * GetBackgroundBrush()

Returns the brush used to draw in the background colour. It is set when SetBackgroundColour is called.

wxTabView::GetBackgroundColour

wxColour GetBackgroundColour()

Returns the colour used for each tab background. By default, this is light grey. To ensure a match with the dialog or panel background, omit the wxTAB_STYLE_COLOUR_INTERIOR flag from the wxTabView constructor.

wxTabView::GetBackgroundPen

wxPen * GetBackgroundPen()

Returns the pen used to draw in the background colour. It is set when `SetBackgroundColour` is called.

wxTabView::GetHighlightColour

wxColour GetHighlightColour()

Returns the colour used for bright highlights on the left side of '3D' surfaces. By default, this is white.

wxTabView::GetHighlightPen

wxPen * GetHighlightPen()

Returns the pen used to draw 3D effect highlights. This is set when `SetHighlightColour` is called.

wxTabView::GetHorizontalTabOffset

int GetHorizontalTabOffset()

Returns the horizontal spacing by which each tab layer is offset from the one below.

wxTabView::GetNumberOfLayers

int GetNumberOfLayers()

Returns the number of layers (rows of tabs).

wxTabView::GetSelectedTabFont

wxFont * GetSelectedTabFont()

Returns the font to be used for the selected tab label.

wxTabView::GetShadowColour

wxColour GetShadowColour()

Returns the colour used for shadows on the right-hand side of '3D' surfaces. By default,

this is dark grey.

wxTabView::GetTabHeight

int GetTabHeight()

Returns the tab default height.

wxTabView::GetTabFont

wxFont * GetTabFont()

Returns the tab label font.

wxTabView::GetTabSelectionHeight

int GetTabSelectionHeight()

Returns the height to be used for the currently selected tab; normally a few pixels higher than the other tabs.

wxTabView::GetTabStyle

long GetTabStyle()

Returns the tab style. See constructor documentation for details of valid styles.

wxTabView::GetTabWidth

int GetTabWidth()

Returns the tab default width.

wxTabView::GetTextColour

wxColour GetTextColour()

Returns the colour used to draw label text. By default, this is black.

wxTabView::GetTopMargin

int GetTopMargin()

Returns the height between the top of the view area and the bottom of the first row of tabs.

wxTabView::GetShadowPen

wxPen * GetShadowPen()

Returns the pen used to draw 3D effect shadows. This is set when `SetShadowColour` is called.

wxTabView::GetViewRect

wxRectangle GetViewRect()

Returns the rectangle specifying the view area (above which tabs are placed).

wxTabView::GetVerticalTabTextSpacing

int GetVerticalTabTextSpacing()

Returns the vertical spacing between the top of an unselected tab, and the tab label.

wxTabView::GetWindow

wxWindow * GetWindow()

Returns the window for the view.

wxTabView::OnCreateTabControl

wxTabControl * OnCreateTabControl()

Creates a new tab control. By default, this returns a `wxTabControl` object, but the application may wish to define a derived class, in which case the tab view should be subclassed and this function overridden.

wxTabView::LayoutTabs

void LayoutTabs()

Recalculates the positions of the tabs, and adjusts the layer of the selected tab if necessary.

You may want to call this function if the view width has changed (for example, from an `OnSize` handler).

wxTabView::OnEvent

bool OnEvent(wxMouseEvent& event)

Processes mouse events sent from the panel or dialog. Returns `TRUE` if the event was processed, `FALSE` otherwise.

wxTabView::OnTabActivate

void OnTabActivate(int activateId, int deactivateId)

Called when a tab is activated, with the new active tab id, and the former active tab id.

wxTabView::OnTabPreActivate

bool OnTabPreActivate(int activateId, int deactivateId)

Called just before a tab is activated, with the new active tab id, and the former active tab id.

If the function returns `FALSE`, the tab is not activated.

wxTabView::SetBackgroundColour

void SetBackgroundColour(const wxColour& col)

Sets the colour to be used for each tab background. By default, this is light grey. To ensure a match with the dialog or panel background, omit the `wxTAB_STYLE_COLOUR_INTERIOR` flag from the `wxTabView` constructor.

wxTabView::SetHighlightColour

void SetHighlightColour(const wxColour& col)

Sets the colour to be used for bright highlights on the left side of '3D' surfaces. By default, this is white.

wxTabView::SetHorizontalTabOffset

void SetHorizontalTabOffset(int *offset*)

Sets the horizontal spacing by which each tab layer is offset from the one below.

wxTabView::SetSelectedTabFont

void SetSelectedTabFont(wxFont **font*)

Sets the font to be used for the selected tab label.

wxTabView::SetShadowColour

void SetShadowColour(const wxColour& *col*)

Sets the colour to be used for shadows on the right-hand side of '3D' surfaces. By default, this is dark grey.

wxTabView::SetTabFont

void SetTabFont(wxFont **font*)

Sets the tab label font.

wxTabView::SetTabStyle

void SetTabStyle(long *tabStyle*)

Sets the tab style. See constructor documentation for details of valid styles.

wxTabView::SetTabSize

void SetTabSize(int *width*, int *height*)

Sets the tab default width and height.

wxTabView::SetTabSelectionHeight

void SetTabSelectionHeight(int *height*)

Sets the height to be used for the currently selected tab; normally a few pixels higher than the other tabs.

wxTabView::SetTabSelection

void SetTabSelection(int sel, bool activateTool=TRUE)

Sets the selected tab, calling the application's OnTabActivate function.

If *activateTool* is FALSE, OnTabActivate will not be called.

wxTabView::SetTextColour

void SetTextColour(const wxColour& col)

Sets the colour to be used to draw label text. By default, this is black.

wxTabView::SetTopMargin

void SetTopMargin(int margin)

Sets the height between the top of the view area and the bottom of the first row of tabs.

wxTabView::SetVerticalTabTextSpacing

void SetVerticalTabTextSpacing(int spacing)

Sets the vertical spacing between the top of an unselected tab, and the tab label.

wxTabView::SetViewRect

void SetViewRect(const wxRectangle& rect)

Sets the rectangle specifying the view area (above which tabs are placed). This must be set by the application.

wxTabView::SetWindow

void SetWindow(wxWindow *window)

Set the window that the tab view will use for drawing onto.

wxTabCtrl

This class represents a tab control, which manages multiple tabs.

Derived from

wxControl (p. 176)
wxWindow (p. 1184)
wxEvtHandler (p. 378)
wxObject (p. 746)

Include files

<wx/tabctrl.h>

See also

wxTabEvent (p. 1058), *wxImageList* (p. 584), *wxNotebook* (p. 736)

wxTabCtrl::wxTabCtrl

wxTabCtrl()

Default constructor.

wxTabCtrl(wxWindow* parent, wxWindowID id, const wxPoint& pos = wxDefaultPosition, const wxSize& size, long style = 0, const wxString& name = "tabCtrl")

Constructs a tab control.

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. Its value is a bit list of zero or more of **wxTC_MULTILINE**, **wxTC_RIGHTJUSTIFY**, **wxTC_FIXEDWIDTH** and **wxTC_OWNERDRAW**.

wxTabCtrl::~~wxTabCtrl

~wxTabCtrl()

Destroys the wxTabCtrl object.

wxTabCtrl::Create

bool Create(wxWindow* parent, wxWindowID id, const wxPoint& pos = wxDefaultPosition, const wxSize& size, long style = 0, const wxString& name = "tabCtrl")

Creates a tab control. See *wxTabCtrl::wxTabCtrl* (p. 1053) for a description of the parameters.

wxTabCtrl::DeleteAllItems

bool DeleteAllItems()

Deletes all tab items.

wxTabCtrl::DeleteItem

bool DeleteItem(int item)

Deletes the specified tab item.

wxTabCtrl::GetCurFocus

int GetCurFocus() const

Returns the index for the tab with the focus, or -1 if none has the focus.

wxTabCtrl::GetImageList

wxImageList* GetImageList() const

Returns the associated image list.

See also

wxImageList (p. 584), *wxTabCtrl::SetImageList* (p. 1057)

wxTabCtrl::GetItemCount

int GetItemCount() const

Returns the number of tabs in the tab control.

wxTabCtrl::GetItemData

void* GetItemData() const

Returns the client data for the given tab.

wxTabCtrl::GetItemImage

int GetItemImage() const

Returns the image index for the given tab.

wxTabCtrl::GetItemRect

bool GetItemRect(int *item*, wxRect& *rect*) const

Returns the rectangle bounding the given tab.

See also

wxRect (p. 868)

wxTabCtrl::GetItemText

wxString GetItemText() const

Returns the string for the given tab.

wxTabCtrl::GetRowCount

int GetRowCount() const

Returns the number of rows in the tab control.

wxTabCtrl::GetSelection

int GetSelection() const

Returns the index for the currently selected tab.

See also

wxTabCtrl::SetSelection (p. 1058)

wxTabCtrl::HitTest

int HitTest(const wxPoint& *pt*, long& *flags*)

Tests whether a tab is at the specified position.

Parameters

pt

Specifies the point for the hit test.

flags

Return value for detailed information. One of the following values:

wxTAB_HITTEST_NOWHERE

There was no tab under this point.

wxTAB_HITTEST_ONICON

The point was over an icon.

wxTAB_HITTEST_ONLABEL

The point was over a label.

wxTAB_HITTEST_ONITEM

The point was over an item, but not on the label or icon.

Return value

Returns the zero-based tab index or -1 if no tab is at the specified position.

wxTabCtrl::InsertItem

void InsertItem(int *item*, const wxString& *text*, int *imageId* = -1, void* *clientData* = NULL)

Inserts a new tab.

Parameters

item

Specifies the index for the new item.

text

Specifies the text for the new item.

imageId

Specifies the optional image index for the new item.

clientData

Specifies the optional client data for the new item.

Return value

TRUE if successful, FALSE otherwise.

wxTabCtrl::SetItemData

bool SetItemData(int item, void* data)

Sets the client data for a tab.

wxTabCtrl::SetItemImage

bool SetItemImage(int item, int image)

Sets the image index for the given tab. *image* is an index into the image list which was set with *wxTabCtrl::SetImageList* (p. 1057).

wxTabCtrl::SetImageList

void SetImageList(wxImageList* imageList)

Sets the image list for the tab control.

See also

wxImageList (p. 584)

wxTabCtrl::SetItemSize

void SetItemSize(const wxSize& size)

Sets the width and height of the tabs.

wxTabCtrl::SetItemText

bool SetItemText(int item, const wxString& text)

Sets the text for the given tab.

wxTabCtrl::SetPadding

void SetPadding(const wxSize& padding)

Sets the amount of space around each tab's icon and label.

wxTabCtrl::SetSelection

int SetSelection(int item)

Sets the selection for the given tab, returning the index of the previously selected tab. Returns -1 if the call was unsuccessful.

See also

wxTabCtrl::GetSelection (p. 1055)

wxTabEvent

This class represents the events generated by a tab control.

Derived from

wxCommandEvent (p. 152)

wxEvent (p. 375)

wxEvtHandler (p. 378)

wxObject (p. 746)

Include files

<wx/tabctrl.h>

Event table macros

To process a tab event, use these event handler macros to direct input to member functions that take a *wxTabEvent* argument.

EVT_TAB_SEL_CHANGED(id, func) Process a *wxEVT_TAB_SEL_CHANGED* event, indicating that the tab selection has changed.

EVT_TAB_SEL_CHANGING(id, func) Process a *wxEVT_TAB_SEL_CHANGING* event, indicating that the tab selection is changing.

See also

wxTabCtrl (p. 1052)

wxTabEvent::wxTabEvent

wxTabEvent(WXTYPE *commandType* = 0, int *id* = 0)

Constructor.

wxTaskBarIcon

This class represents a Windows 95 taskbar icon, appearing in the 'system tray' and responding to mouse clicks. An icon has an optional tooltip. This class is only supported for Windows 95/NT.

Derived from

wxObject (p. 746)

Include files

<wx/taskbar.h>

wxTaskBarIcon::wxTaskBarIcon

wxTaskBarIcon()

Default constructor.

wxTaskBarIcon::~~wxTaskBarIcon

~wxTaskBarIcon()

Destroys the wxTaskBarIcon object, removing the icon if not already removed.

wxTaskBarIcon::IsIconInstalled

bool IsIconInstalled()

Returns TRUE if *SetIcon* (p. 1061) was called with no subsequent *RemoveIcon* (p. 1061).

wxTaskBarIcon::IsOK

bool IsOK()

Returns TRUE if the object initialized successfully.

wxTaskBarIcon::OnLButtonDown

virtual void OnLButtonDown()

Override this function to intercept left mouse button down events.

wxTaskBarIcon::OnLButtonDClick

virtual void OnLButtonDClick()

Override this function to intercept left mouse button double-click events.

wxTaskBarIcon::OnLButtonUp

virtual void OnLButtonUp()

Override this function to intercept left mouse button up events.

wxTaskBarIcon::OnRButtonDown

virtual void OnRButtonDown()

Override this function to intercept right mouse button down events.

wxTaskBarIcon::OnRButtonDClick

virtual void OnRButtonDClick()

Override this function to intercept right mouse button double-click events.

wxTaskBarIcon::OnRButtonUp

virtual void OnRButtonUp()

Override this function to intercept right mouse button up events.

wxTaskBarIcon::OnMouseMove

virtual void OnMouseMove()

Override this function to intercept mouse move events.

wxTaskBarIcon::Removelcon

bool Removelcon()

Removes the icon previously set with *Setlcon* (p. 1061).

wxTaskBarIcon::Setlcon

bool Setlcon(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. 298).

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 wxTCPClient is required so that a user-overridden *wxTCPClient::OnMakeConnection* (p. 1062) member can return a wxTCPConnection of the required class, when a connection is made.

Derived from

wxClientBase
wxObject (p. 746)

Include files

<wx/sckipc.h>

See also

wxTCPServer (p. 1067), *wxTCPConnection* (p. 1063), *Interprocess communications overview* (p. 1428)

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. 1062) member to return your own derived connection object.

wxTCPClient::OnMakeConnection

wxConnectionBase * OnMakeConnection()

The type of *wxTCPConnection* (p. 1063) returned from a *wxTCPClient::MakeConnection* (p. 1062) 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. 1065). 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. 299).

A wxTCPConnection object can be created by making a connection using a *wxTCPClient* (p. 1061) object, or by the acceptance of a connection by a *wxTCPServer* (p. 1067) 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. 746)

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,
};
```

See also

wxTCPClient (p. 1061), *wxTCPServer* (p. 1067), *Interprocess communications overview* (p. 1428)

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. 1067) and/or *wxTCPClient::OnMakeConnection* (p. 1062) 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. 1065) 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. 1066) in that respect). Causes the server connection's *wxTCPConnection::OnExecute* (p. 1065) 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. 1065) 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. 1066). 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. 1065) 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. 1065) 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. 1065) 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. 1066) 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. 303).

Derived from

`wxServerBase`
`wxObject` (p. 746)

Include files

<wx/sckipc.h>

See also

`wxTCPClient` (p. 1061), `wxTCPConnection` (p. 1063), *IPC overview* (p. 1428)

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. 1069) to do it. Calling *Discard* (p. 1070) 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. 1069)). Then you can *write* (p. 1069) to `wxTempFile` using *wxFile* (p. 395)-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. 395)

wxTempFile::wxTempFile

wxTempFile()

Default constructor - *Open* (p. 1069) 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. 1069) to verify if the constructor succeeded.

wxTempFile::Open

bool Open(const wxString& strName)

Open the temporary file (strName is the name of file to be replaced), returns TRUE on success, FALSE if an error occurred.

wxTempFile::IsOpened

bool IsOpened() const

Returns TRUE if the file was successfully opened.

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)

Write to the file, return TRUE on success, FALSE on failure.

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. 1070) if temporary file is still opened.

wxTextCtrl

A text control allows text to be displayed and edited. It may be single line or multi-line.

Derived from

streambuf
wxControl (p. 176)
wxWindow (p. 1184)
wxEvtHandler (p. 378)
wxObject (p. 746)

Include files

<wx/textctrl.h>

Window styles

wxTE_PROCESS_ENTER The control will generate the message `wxEVENT_TYPE_TEXT_ENTER_COMMAND` (otherwise pressing <Enter> is either processed internally by the control or used for navigation between dialog controls).

wxTE_PROCESS_TAB The control will receive `EVT_CHAR` messages 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	create rich edit control instead of a normal one under Windows, does nothing elsewhere
wxHSCROLL	A horizontal scrollbar will be created. No effect under GTK+.

See also *window styles overview* (p. 1371) and *wxTextCtrl::wxTextCtrl* (p. 1072).

Remarks

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 any use of C++ iostreams (including this one) deprecated and might get completely removed in the future.

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. 152) argument.

EVT_TEXT(id, func)	Respond to a <i>wxEVT_COMMAND_TEXT_UPDATED</i> 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 <i>SetValue()</i> is called)
EVT_TEXT_ENTER(id, func)	Respond to a <i>wxEVT_COMMAND_TEXT_ENTER</i> event, generated when enter is pressed in a single-line text control.

wxTextCtrl::wxTextCtrl

wxTextCtrl()

Default constructor.

wxTextCtrl(wxWindow* parent, wxWindowID id, const wxString& value = "", const wxPoint& pos, const wxSize& size = wxDefaultSize, long style = 0, const wxValidator& validator, const wxString& name = "text")

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. 1070).

validator

Window validator.

name

Window name.

Remarks

The horizontal scrollbar (**wxTE_HSCROLL** 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. 1075) is always visible.

Under Windows, if the **wxTE_RICH** style is used, the window is implemented as a Windows rich text control with (practically) unlimited capacity. Otherwise, normal edit control limits apply (only about 32Kb of total data under Windows 9x).

See also

wxTextCtrl::Create (p. 1074), *wxValidator* (p. 1166)

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. 1075) and *SetInsertionPoint* (p. 1080).

See also

wxTextCtrl::WriteText (p. 1082)

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.

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, const wxSize& size = wxDefaultSize, long style = 0, const wxValidator& validator, const wxString& name = "text")

Creates the text control for two-step construction. Derived classes should call or replace this function. See *wxTextCtrl::wxTextCtrl* (p. 1072) 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::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. 1077).Length() and *GetLastPosition()* (p. 1075).

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 long 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::GetSelection

virtual void GetSelection(long* *from*, long* *to*)

Gets the current selection span. If the returned values are equal, there was no selection.

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.

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::IsModified

bool IsModified() const

Returns TRUE if the text has been modified.

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::OnChar

void OnChar(wxKeyEvent& event)

Default handler for character input.

Remarks

It is possible to intercept character input by overriding this member. Call this function to let the default behaviour take place; not calling it results in the character being ignored.

You can replace the *keyCode* member of *event* to translate keystrokes.

Note that Windows and Motif have different ways of implementing the default behaviour. In Windows, calling `wxTextCtrl::OnChar` immediately processes the character. In Motif, calling this function simply sets a flag to let default processing happen. This might affect the way in which you write your `OnChar` function on different platforms.

See also

`wxKeyEvent` (p. 607)

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. 364)

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. 1082)

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.

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::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.

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. 1080)(*GetLastPosition* (p. 1075)()).

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.

Parameters

from

The first position.

to

The last position.

wxTextCtrl::SetValue

virtual void SetValue(const wxString& *value*)

Sets the text value and marks the control as not-modified.

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. 1083) and *wxTextCtrl::AppendText* (p. 1073) 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. 1081) 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 l)

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. 121) or a `wxDropSource` (p. 365). 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. 1084) and `GetText` (p. 1084) 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. 201)

`wxDataObject` (p. 196)

Include files

<wx/dataobj.h>

See also

Clipboard and drag and drop overview (p. 1420), *wxDataObject* (p. 196), *wxDataObjectSimple* (p. 201), *wxFileDataObject* (p. 406), *wxBitmapDataObject* (p. 75)

wxTextDataObject::wxTextDataObject

wxTextDataObject(const wxString& text = wxEmptyString)

Constructor, may be used to initialise the text (otherwise *SetText* (p. 1084) 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. 1084). 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 wxWindows' internals. Use this method to get data in text form from the *wxClipboard* (p. 121).

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.

wxTextInputStream

This class provides functions that read text data 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.

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)

Constructs a text stream object from an input stream. Only read methods will be available.

Parameters

stream

The input stream.

wxTextInputStream::~~wxTextInputStream

~wxTextInputStream()

Destroys the `wxTextInputStream` object.

wxTextInputStream::Read8

wxUint8 Read8()

Reads a single byte from the stream.

wxTextInputStream::Read16

wxUint16 Read16()

Reads a 16 bit integer from the stream.

wxTextInputStream::Read32

wxUint16 Read32()

Reads a 32 bit integer from the stream.

wxTextInputStream::ReadDouble

double ReadDouble()

Reads a double (IEEE encoded) from the stream.

wxTextInputStream::ReadLine

wxString wxTextInputStream::ReadLine()

Reads a line from the input stream and returns it (without the end of line character).

wxTextInputStream::ReadString

wxString wxTextInputStream::ReadString()

NB: This method is deprecated, use *ReadLine* (p. 1086) or *ReadWord* (p. 1086) instead.

Same as *ReadLine* (p. 1086).

wxTextInputStream::ReadWord

wxString wxTextInputStream::ReadWord()

Reads a word (a sequence of characters until the next separator) from the input stream.

See also

SetStringSeparators (p. 1087)

wxTextInputStream::SetStringSeparators

void SetStringSeparators(const wxString& sep)

Sets the characters which are used to define the word boundaries in *ReadWord* (p. 1086).

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:

```
wxFileOutputStream 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).

wxTextOutputStream::wxTextOutputStream

wxTextOutputStream(wxOutputStream& stream, wxEOL mode = wxEOL_NATIVE)

Constructs a text stream object from an output stream. Only write methods will be available.

Parameters

stream

The output stream.

mode

The end-of-line mode. One of **wxEOL_NATIVE**, **wxEOL_DOS**, **wxEOL_MAC** and **wxEOL_UNIX**.

wxTextOutputStream::~~wxTextOutputStream

~wxTextOutputStream()

Destroys the `wxTextOutputStream` object.

wxTextOutputStream::GetMode

wxEOL wxTextOutputStream::GetMode()

Returns the end-of-line mode. One of **wxEOL_DOS**, **wxEOL_MAC** and **wxEOL_UNIX**.

wxTextOutputStream::SetMode

void wxTextOutputStream::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 wxTextOutputStream::Write8(wxUint8 i8)

Writes the single byte *i8* to the stream.

wxTextOutputStream::Write16

void wxTextOutputStream::Write16(wxUint16 i16)

Writes the 16 bit integer *i16* to the stream.

wxTextOutputStream::Write32

void wxTextOutputStream::Write32(wxUint32 i32)

Writes the 32 bit integer *i32* to the stream.

wxTextOutputStream::WriteDouble

virtual void wxTextOutputStream::WriteDouble(double f)

Writes the double *f* to the stream using the IEEE format.

wxTextOutputStream::WriteString

virtual void wxTextOutputStream::WriteString(const wxString& string)

Writes *string* as a line. Depending on the end-of-line mode, it adds `\n`, `\r` or `\r\n`.

wxTextEntryDialog

This class represents a dialog that requests a one-line text string from the user. It is implemented as a generic `wxWindows` dialog.

Derived from

wxDialog (p. 310)

wxWindow (p. 1184)

wxEvtHandler (p. 378)

wxObject (p. 746)

Include files

<wx/textdlg.h>

See also

wxTextEntryDialog overview (p. 1401)

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. 1090) 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 wxOK_CANCEL otherwise.

wxTextDropTarget

A predefined drop target for dealing with text data.

Derived from

wxDropTarget (p. 368)

Include files

<wx/dnd.h>

See also

Drag and drop overview (p. 1420), *wxDropSource* (p. 365), *wxDropTarget* (p. 368), *wxFileDropTarget* (p. 412)

wxTextDropTarget::wxTextDropTarget

wxTextDropTarget()

Constructor.

wxTextDropTarget::OnDrop

virtual bool OnDrop(long x, long y, const void *data, size_t size)

See *wxDropTarget::OnDrop* (p. 369). This function is implemented appropriately for text, and calls *wxTextDropTarget::OnDropText* (p. 1091).

wxTextDropTarget::OnDropText

virtual bool OnDropText(long x, long y, const char *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 NULL-terminated string.

Return value

Return TRUE to accept the data, FALSE to veto the operation.

wxTimeSpan

TODO

wxTextValidator

wxTextValidator validates text controls, providing a variety of filtering behaviours.

For more information, please see *Validator overview* (p. 1374).

Derived from

wxValidator (p. 1166)
wxEvtHandler (p. 378)
wxObject (p. 746)

Include files

<wx/valtext.h>

See also

Validator overview (p. 1374), *wxValidator* (p. 1166), *wxGenericValidator* (p. 477)

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.
wxFILTER_EXCLUDE_LIST	Use an exclude list. The validator checks if the user input is on the list, complaining if it is.

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). If NULL, the validator uses its own internal storage for the value.

wxTextValidator::~~wxTextValidator

~wxTextValidator()

Destructor.

wxTextValidator::Clone

virtual wxValidator* Clone() const

Clones the text validator using the copy constructor.

wxTextValidator::GetExcludeList

wxStringList& GetExcludeList() const

Returns a reference to the exclude list (the list of invalid values).

wxTextValidator::GetIncludeList

wxStringList& GetIncludeList() 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::SetExcludeList

void SetExcludeList(const wxStringList& stringList)

Sets the exclude list (invalid values for the user input).

wxTextValidator::SetIncludeList

void SetIncludeList(const wxStringList& 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 string value to the window.

wxTextValidator::TransferToWindow

virtual bool TransferToWindow()

Transfers the window value to the string.

wxTextValidator::Validate

virtual bool Validate(wxWindow* parent)

Validates the window contents against the include or exclude lists, depending on the validator style.

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 so 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 the small files like configuration files or programs 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. 1097) or *Open* (p. 1100) 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. 1097) and *GetLine* (p. 1097) (*operator[]* does exactly the same but looks more like array addressing) or with "sequential access" functions which include *GetFirstLine* (p. 1098)/*GetNextLine* (p. 1099) and also *GetLastLine* (p. 1099)/*GetPrevLine* (p. 1099). For the sequential access functions the current line number is maintained: it is returned by *GetCurrentLine* (p. 1098) and may be changed with *GoToLine* (p. 1098).
- Add/remove lines to the file: *AddLine* (p. 1096) and *InsertLine* (p. 1100) add new lines while *RemoveLine* (p. 1100) deletes the existing ones.
- Save your changes: notice that the changes you make to the file will **not** be saved automatically; calling *Close* (p. 1097) or doing nothing discards them! To save the changes you must explicitly call *Write* (p. 1100) - 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. 395)

wxTextFile::wxTextFile

wxTextFile() const

Default constructor, use *Create* (p. 1097) or *Open* (p. 1100) 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. 1100) 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. 1096). The array of file lines is initially empty.

It will fail if the file already exists, *Open* (p. 1100) 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. 1097).

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. 1098) and used by *GetFirstLine()* (p. 1098)/*GetNextLine()* (p. 1099).

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 and *wxTextFileType_Mac* under Mac.

wxTextFile::GetFirstLine

wxString& GetFirstLine() const

This method together with *GetNextLine()* (p. 1099) 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. 1098) 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. 1099) 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. 1098))

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() const

bool Open(const wxString& strFile) const

`Open()` opens the file with the given name or the name which was given in the *constructor* (p. 1096) and also loads file in memory on success. It will fail if the file does not exist, *Create* (p. 1097) should be used in this case.

wxTextFile::RemoveLine

void RemoveLine(size_t n) const

Delete line number *n* from the file.

wxTextFile::Write

bool Write(wxTextFileType typeNew = wxTextFileType_None) 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.

Returns TRUE if operation succeeded, FALSE if it failed.

wxThread

A thread is basically a path of execution through a program. Threads are also 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 much easier to shoot oneself in the foot, so careful use of synchronization objects such as *mutexes* (p. 730) and/or *critical sections* (p. 178) is recommended.

There are two types of threads in wxWindows: *detached* and *joinable* ones, just as in 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 - it is returned by `Wait()` function. The detached threads (default) can not 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 of the system resources used by it will never be freed and you also must delete the corresponding `wxThread` object yourself, while 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. The joinable threads may be created on stack (don't create global thread objects because they allocate memory in their constructor which is a bad thing to do), although usually they will be created on the heap as well.

Derived from

None.

Include files

<wx/thread.h>

See also

wxMutex (p. 730), *wxCondition* (p. 160), *wxCriticalSection* (p. 178)

wxThread::wxThread

wxThread(`wxThreadKind kind = wxTHREAD_DETACHED`)

Constructor creates a new detached (default) or joinable C++ thread object. It does not create (or starts execution of) the real thread - for this you should use *Create* (p. 1102) and *Run* (p. 1106) methods.

The possible values for *kind* parameters are: **wxTHREAD_DETACHED** Create a detached thread.
wxTHREAD_JOINABLE Create a joinable thread

wxThread::~~wxThread

~wxThread()

Destructor frees the resources associated with the thread. Notice that you should never delete a detached thread - you may only call *Delete* (p. 1102) 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.

The joinable threads, however, may and should be deleted explicitly and *Delete* (p. 1102) and *Kill* (p. 1105) functions will not delete the C++ thread object. It is also safe to allocate them on stack.

wxThread::Create

wxThreadError Create()

Creates a new thread. The thread object is created in the suspended state, you should call *Run* (p. 1106) to start running it.

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. 1102) 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. 1107) which should happen quiet soon.

However, if the thread doesn't call *TestDestroy* (p. 1107) often enough (or at all), the function will not return immediately, but wait until the thread terminates. As it may take a long time, the message processing is not stopped during this function execution, so the

message handlers may be called from inside it!

Delete() may be called for thread in any state: running, paused or even not yet created. Moreover, it must be called if *Create* (p. 1102) or *Run* (p. 1106) failed for a detached thread to free the memory occupied by the thread object (it will be done in the destructor for joinable threads).

Delete() may be called for thread in any state: running, paused or even not yet created. Moreover, it must be called if *Create* (p. 1102) or *Run* (p. 1106) fail 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 the joinable threads and is the value returned by *Wait* (p. 1107).

This function is called by wxWindows itself and should never be called directly.

wxThread::Exit

void Exit(ExitCode exitcode = 0)

This is a protected function of wxThread class and thus can be called only from a derived class. It also can be called only 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. 1105) 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. 1107)

wxThread::GetId

unsigned long GetId() const

Gets the thread identifier: this is a platform dependent number which 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).

wxThread::IsDetached

bool IsDetached() const

Returns TRUE if the thread is of 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.

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. 1102) 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. 1105) 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 can not 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, `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. 1105).

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. 1107) 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. 1102).

This function can only be called from another thread context.

wxThread::SetPriority

void SetPriority(int *priority*)

Sets the priority of the thread, between zero and 100. This must be set before the thread is created.

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. 1282) by all worker (i.e. all except the main one) threads.

wxThread::Resume

wxThreadError Resume()

Resumes a thread suspended by the call to *Pause* (p. 1105).

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

bool TestDestroy()

This function should be periodically called by the thread to ensure that calls to *Pause* (p. 1105) and *Delete* (p. 1102) will work. If it returns TRUE, the thread should exit as soon as possible.

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. 1104) 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 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. 1106).

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.

wxTime

Representation of time and date.

NOTE: this class is retained only for compatibility, and has been replaced by *wxDateTime* (p. 215). *wxTime* may be withdrawn in future versions of wxWindows.

Derived from

wxObject (p. 746)

Include files

<wx/time.h>

Data structures

```
typedef unsigned short hourTy;
typedef unsigned short minuteTy;
typedef unsigned short secondTy;
typedef unsigned long clockTy;
enum tFormat { wx12h, wx24h };
enum tPrecision { wxStdMinSec, wxStdMin };
```

See also

wxDate (p. 206)

wxTime::wxTime

wxTime()

Initialize the object using the current time.

wxTime(clockTy s)

Initialize the object using the number of seconds that have elapsed since ???.

wxTime(const wxTime& time)

Copy constructor.

wxTime(hourTy h, minuteTy m, secondTy s = 0, bool dst = FALSE)

Initialize using hours, minutes, seconds, and whether DST time.

wxTime(const wxDate& date, hourTy h = 0, minuteTy m = 0, secondTy s = 0, bool

dst = FALSE)

Initialize using a *wxDate* (p. 206) object, hours, minutes, seconds, and whether DST time.

wxTime::GetDay

int GetDay() const

Returns the day of the month.

wxTime::GetDayOfWeek

int GetDayOfWeek() const

Returns the day of the week, a number from 0 to 6 where 0 is Sunday and 6 is Saturday.

wxTime::GetHour

hourTy GetHour() const

Returns the hour in local time.

wxTime::GetHourGMT

hourTy GetHourGMT() const

Returns the hour in GMT.

wxTime::GetMinute

minuteTy GetMinute() const

Returns the minute in local time.

wxTime::GetMinuteGMT

minuteTy GetMinuteGMT() const

Returns the minute in GMT.

wxTime::GetMonth

int GetMonth() const

Returns the month.

wxTime::GetSecond

secondTy GetSecond() const

Returns the second in local time or GMT.

wxTime::GetSecondGMT

secondTy GetSecondGMT() const

Returns the second in GMT.

wxTime::GetSeconds

clockTy GetSeconds() const

Returns the number of seconds since ???.

wxTime::GetYear

int GetYear() const

Returns the year.

wxTime::FormatTime

char* FormatTime() const

Formats the time according to the current formatting options: see *wxTime::SetFormat* (p. 1111).

wxTime::IsBetween

bool IsBetween(const wxTime& a, const wxTime& b) const

Returns TRUE if this time is between the two given times.

wxTime::Max

wxTime Max(const wxTime& *time*) const

Returns the maximum of the two times.

wxTime::Min

wxTime Min(const wxTime& *time*) const

Returns the minimum of the two times.

wxTime::SetFormat

static void SetFormat(const tFormat *format* = wx12h, const tPrecision *precision* = wxStdMinSec)

Sets the format and precision.

wxTime::operator char*

operator char*()

Returns a pointer to a static char* containing the formatted time.

wxTime::operator wxDate

operator wxDate() const

Converts the wxTime into a wxDate.

wxTime::operator =

void operator =(const wxTime& *t*)

Assignment operator.

wxTime::operator <

bool operator <(const wxTime& *t*) const

Less than operator.

wxTime::operator <=

bool operator <=(const wxTime& t) const

Less than or equal to operator.

wxTime::operator >

bool operator >(const wxTime& t) const

Greater than operator.

wxTime::operator >=

bool operator >=(const wxTime& t) const

Greater than or equal to operator.

wxTime::operator ==

bool operator ==(const wxTime& t) const

Equality operator.

wxTime::operator !=

bool operator !=(const wxTime& t) const

Inequality operator.

wxTime::operator +

bool operator +(long sec) const

Addition operator.

wxTime::operator -

bool operator -(long sec) const

Subtraction operator.

wxTime::operator +=

bool operator +=(long sec) const

Increment operator.

wxTime::operator -=

bool operator -=(long sec) const

Decrement operator.

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. 1114) member to perform the required action.
2. Or you may redirect the notifications to any *wxEvtHandler* (p. 378) derived object by using the non default constructor or *SetOwner* (p. 1115). Then use `EVT_TIMER` macro to connect it to the event handler which will receive *wxTimerEvent* (p. 1115) notifications.

In any case, you must start the timer with *Start* (p. 1115) after constructing it before it actually starts sending notifications. It can be stopped later with *Stop* (p. 1115).

Derived from*wxObject* (p. 746)**Include files**

<wx/timer.h>

See also

::wxStartTimer (p. 1302), ::wxGetElapsedTime (p. 1300), wxStopWatch (p. 997)

wxTimer::wxTimer

wxTimer()

Default constructor. If you use it to construct the object and don't call *SetOwner* (p. 1115) later, you must override *Notify* (p. 1114) method to process the notifications.

wxTimer(wxEvtHandler *owner, int id = -1)

Creates a timer and associates it with *owner*. Please see *SetOwner* (p. 1115) for the description of parameters.

wxTimer::~~wxTimer

~wxTimer()

Destructor. Stops the timer if it is running.

wxTimer::GetInterval

wxtimergetinterval

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. 1115) 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. 1115) 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. 1114) function will be called repeatedly until the timer is stopped. If TRUE, it will be called only once and the timer will stop automatically.

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. 1113)

wxTimerEvent::GetInterval

int GetInterval() const

Returns the interval of the timer which generated this event.

wxTipProvider

This is the class used together with *wxShowTip* (p. 1261) function. It must implement *GetTip* (p. 1117) 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. 1256).

Derived from

None.

Include files

<wx/tipdlg.h>

See also

Startup tips overview (p. 1418), *::wxShowTip* (p. 1261)

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.

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. 1261). 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.

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.
- **wxToolBarSimple** A simple implementation, with scrolling. Used on platforms with no native toolbar control, or where scrolling is required.

Note that the base class **wxToolBarBase** defines automatic scrolling management

functionality which is similar to *wxScrolledWindow* (p. 911), so please refer to this class also. Not all toolbars support scrolling, but *wxToolBarSimple* does.

Derived from

wxToolBarBase
wxControl (p. 176)
wxWindow (p. 1184)
wxEvtHandler (p. 378)
wxObject (p. 746)

Include files

<wx/toolbar.h> (to allow *wxWindows* to select an appropriate toolbar class)
 <wx/tbarbase.h> (the base class)
 <wx/tbarmsw.h> (the non-Windows 95 Windows toolbar class)
 <wx/tbar95.h> (the Windows 95/98 toolbar class)
 <wx/tbarsmpl.h> (the generic simple toolbar class)

Remarks

You may also create a toolbar that is managed by the frame, by calling *wxFrame::CreateToolBar* (p. 456).

Due to the use of native toolbars on the various platforms, certain adaptations will often have to be made in order to get optimal look on all platforms as some platforms ignore the values for explicit placement and use their own layout and the meaning of a "separator" is a vertical line under Windows95 vs. simple space under GTK etc.

wxToolBar95: Note that this toolbar paints tools to reflect user-selected colours. The toolbar orientation must always be **wxHORIZONTAL**.

wxToolBarGtk: The toolbar orientation is ignored and is always **wxHORIZONTAL**.

Window styles

wxTB_FLAT	Gives the toolbar a flat look ('coolbar' or 'flatbar' style). Windows 95 and GTK 1.2 only.
wxTB_DOCKABLE	Makes the toolbar floatable and dockable. GTK only.
wxTB_HORIZONTAL	Specifies horizontal layout.
wxTB_VERTICAL	Specifies vertical layout (not available for the GTK and Windows 95 toolbar).
wxTB_3DBUTTONS	Gives <i>wxToolBarSimple</i> a mild 3D look to its buttons.

See also *window styles overview* (p. 1371).

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 is passed and the tool id is retrieved from the `wxCommandEvent`. This is because the id may be -1 when the mouse moves off a tool, and -1 is not allowed as an identifier in the event system.

Note that tool commands (and UI update events for tools) are first sent to the focus window within the frame that contains the toolbar. If no window within the frame has the focus, then the events are sent directly to the toolbar (and up the hierarchy to the frame, depending on where the application has put its event handlers). This allows command and UI update handling to be processed by specific windows and controls, and not necessarily by the application frame.

EVT_TOOL(id, func)	Process a <code>wxEVT_COMMAND_TOOL_CLICKED</code> event (a synonym for <code>wxEVT_COMMAND_MENU_SELECTED</code>). Pass the id of the tool.
EVT_MENU(id, func)	The same as <code>EVT_TOOL</code> .
EVT_TOOL_RANGE(id1, id2, func)	Process a <code>wxEVT_COMMAND_TOOL_CLICKED</code> event for a range id identifiers. Pass the ids of the tools.
EVT_MENU_RANGE(id1, id2, func)	The same as <code>EVT_TOOL_RANGE</code> .
EVT_TOOL_RCLICKED(id, func)	Process a <code>wxEVT_COMMAND_TOOL_RCLICKED</code> event. Pass the id of the tool.
EVT_TOOL_RCLICKED_RANGE(id1, id2, func)	Process a <code>wxEVT_COMMAND_TOOL_RCLICKED</code> event for a range of ids. Pass the ids of the tools.
EVT_TOOL_ENTER(id, func)	Process a <code>wxEVT_COMMAND_TOOL_ENTER</code> event. Pass the id of the toolbar itself. The value of <code>wxCommandEvent::GetSelection</code> is the tool id, or -1 if the mouse cursor has moved off a tool.

See also

Toolbar overview (p. 1412), *wxScrolledWindow* (p. 911)

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 wxWindows 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 wxWindows should generate a default size for the window.

style

Window style. See *wxToolBar* (p. 1117) for details.

name

Window name.

Remarks

After a toolbar is created, you use *wxToolBar::AddTool* (p. 1121) and perhaps *wxToolBar::AddSeparator* (p. 1121), and then you must call *wxToolBar::Realize* (p. 1129) to construct and display the toolbar tools.

You may also create a toolbar that is managed by the frame, by calling *wxFrame::CreateToolBar* (p. 456).

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. 1121), *wxToolBar::SetToolSeparation* (p. 1132)

wxToolBar::AddTool

wxToolBarTool* AddTool(int toolId, const wxBitmap& bitmap1, const wxString& shortHelpString = "", const wxString& longHelpString = "")

wxToolBarTool* AddTool(int toolId, const wxBitmap& bitmap1, const wxBitmap& bitmap2 = wxNullBitmap, bool isToggle = FALSE, long xPos = -1, long yPos = -1, wxObject* clientData = NULL, const wxString& shortHelpString = "", const wxString& longHelpString = "")

Adds a tool to the toolbar. The first (short and most commonly used) version adds a normal (and not a togglable) button without any associated client data.

Parameters

toolId

An integer by which the tool may be identified in subsequent operations.

isToggle

Specifies whether the tool is a toggle or not: a toggle tool may be in two states, whereas a non-toggle tool is just a button.

bitmap1

The primary tool bitmap for toggle and button tools.

bitmap2

The second bitmap specifies the on-state bitmap for a toggle tool. If this is `wxNullBitmap`, either an inverted version of the primary bitmap is used for the on-state of a toggle tool (monochrome displays) or a black border is drawn around the tool (colour displays) or the pixmap is shown as a pressed button (GTK).

xPos

Specifies the x position of the tool if automatic layout is not suitable.

yPos

Specifies the y position of the tool if automatic layout is not suitable.

clientData

An optional pointer to client data which can be retrieved later using *wxToolBar::GetToolClientData* (p. 1124).

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

Remarks

After you have added tools to a toolbar, you must call *wxToolBar::Realize* (p. 1129) in order to have the tools appear.

See also

wxToolBar::AddSeparator (p. 1121), *wxToolBar::InsertTool* (p. 1127), *wxToolBar::DeleteTool* (p. 1122), *wxToolBar::Realize* (p. 1129),

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. 1129) instead.

Note that it is unnecessary to call *Realize* (p. 1129) for the change to take place, it will happen immediately.

Returns TRUE if the tool was deleted, FALSE otherwise.

See also

DeleteToolByPos (p. 1122)

wxToolBar::DeleteToolByPos

bool DeleteToolByPos(size_t pos)

This function behaves like *DeleteTool* (p. 1122) 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. 1129).

Remarks

For *wxToolBarSimple*, does nothing. Some other implementations will change the visible state of the tool to indicate that it is disabled.

See also

wxToolBar::GetToolEnabled (p. 1125), *wxToolBar::ToggleTool* (p. 1132)

wxToolBar::FindToolForPosition

wxToolBarTool* 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. 1130), *wxToolBar::GetToolBitmapSize* (p. 1124)

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. 1121), and not the eventual size of the tool button.

See also

wxToolBar::SetToolBitmapSize (p. 1130), *wxToolBar::GetToolSize* (p. 1124)

wxToolBar::GetMargins

wxSize GetMargins() const

Returns the left/right and top/bottom margins, which are also used for inter-tools spacing.

See also

wxToolBar::SetMargins (p. 1130)

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. 1121).

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. 1123)

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. 1131), `wxToolBar::SetToolShortHelp` (p. 1132)

wxToolBar::GetToolPacking

int GetToolPacking() const

Returns the value used for packing tools.

See also

wxToolBar::SetToolPacking (p. 1131)

wxToolBar::GetToolSeparation

int GetToolSeparation() const

Returns the default separator size.

See also

wxToolBar::SetToolSeparation (p. 1132)

wxToolBar::GetToolShortHelp

wxString GetToolShortHelp(int *toolId*) const

Returns the short help for the given tool.

Returns the long help for the given tool.

Parameters

toolId

The tool in question.

See also

wxToolBar::GetToolLongHelp (p. 1125), *wxToolBar::SetToolShortHelp* (p. 1132)

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. 1132)

wxToolBar::InsertControl

wxToolBarTool * InsertControl(size_t pos, wxControl *control)

Inserts the control into the toolbar at the given position.

You must call *Realize* (p. 1129) for the change to take place.

See also

AddControl (p. 1120),
InsertTool (p. 1127)

wxToolBar::InsertSeparator

wxToolBarTool * InsertSeparator(size_t pos)

Inserts the separator into the toolbar at the given position.

You must call *Realize* (p. 1129) for the change to take place.

See also

AddSeparator (p. 1121),
InsertTool (p. 1127)

wxToolBar::InsertTool

wxToolBarTool * 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 = "")

Inserts the tool with the specified attributes into the toolbar at the given position.

You must call *Realize* (p. 1129) for the change to take place.

See also

AddTool (p. 1121),
InsertControl (p. 1127),
InsertSeparator (p. 1127)

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. 1121).

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. 1128), *wxToolBar::OnRightClick* (p. 1128)

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, *wxWindows* 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. 1121).

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. 1128), *wxToolBar::OnLeftClick* (p. 1127)

wxToolBar::Realize

bool Realize()

This function should be called after you have added tools.

If you are using absolute positions for your tools when using a *wxToolBarSimple* object, do not call this function. You must call it at all other times.

wxToolBar::RemoveTool

wxToolBarTool * 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. 1129) for the change to take place, it will happen immediately.

See also

DeleteTool (p. 1122)

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. 1124), *wxSize* (p. 922)

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. 1121), and

not the eventual size of the tool button.

See also

wxToolBar::GetToolBitmapSize (p. 1124), *wxToolBar::GetToolSize* (p. 1124)

wxToolBar::SetToolClientData

void SetToolClientData(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. 1125), *wxToolBar::SetToolShortHelp* (p. 1132),

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. 1125)

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. 1126), *wxToolBar::SetToolLongHelp* (p. 1131)

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. 1121)

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. 1126)

wxToolTip

This class holds information about a tooltip associated with a window (see *wxWindow::SetToolTip* (p. 1231)).

The two static methods, *wxToolTip::Enable* (p. 1133) and *wxToolTip::SetDelay* (p. 1133) can be used to globally alter tooltips behaviour.

Derived from

wxObject (p. 746)

wxToolTip::Enable

static void Enable(bool *flag*)

Enable or disable tooltips globally.

wxToolTip::SetDelay

static void SetDelay(long *msecs*)

Set the delay after which the tooltip appears.

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.

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. The only method of `wxTreeItemId` class which can be used is `IsOk()` which returns `TRUE` if the handle is valid and `FALSE` otherwise.

To intercept events from a tree control, use the event table macros described in `wxTreeEvent` (p. 1151).

Derived from

`wxControl` (p. 176)

`wxWindow` (p. 1184)

`wxEvtHandler` (p. 378)

`wxObject` (p. 746)

Include files

<wx/treectrl.h>

Window styles

wxTR_HAS_BUTTONS	Use this style to show + and - buttons to the left of parent items. Win32 only.
wxTR_EDIT_LABELS	Use this style if you wish the user to be able to edit labels in the tree control.
wxTR_MULTIPLE	Use this style to allow the user to select more than one item in the control - by default, only one item may be selected.

See also *window styles overview* (p. 1371).

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. 1151) 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)	Drag ended (drop).
EVT_TREE_BEGIN_LABEL_EDIT(id, func)	Begin editing a label. This can be prevented by calling <i>Veto()</i> (p. 746).
EVT_TREE_END_LABEL_EDIT(id, func)	Finish editing a label. This can be prevented by calling <i>Veto()</i> (p. 746).
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)	Parent has been collapsed.
EVT_TREE_ITEM_COLLAPSING(id, func)	Parent is being collapsed. This can be prevented by calling <i>Veto()</i> (p. 746).
EVT_TREE_ITEM_EXPANDED(id, func)	Parent has been expanded.
EVT_TREE_ITEM_EXPANDING(id, func)	Parent is being expanded. This can be prevented by calling <i>Veto()</i> (p. 746).
EVT_TREE_SEL_CHANGED(id, func)	Selection has changed.
EVT_TREE_SEL_CHANGING(id, func)	Selection is changing. This can be prevented by calling <i>Veto()</i> (p. 746).
EVT_TREE_KEY_DOWN(id, func)	A key has been pressed.

Note that to allow drag and drop in the tree control you must explicitly call *Allow()* (p. 746) in the event handler, by default dragging is disabled. On the other hand, label editing is allowed by default for the controls with **wxTR_EDIT_LABELS** style but you can call *Veto()* (p. 746) to prevent it from happening.

See also

wxTreeItemData (p. 1149), *wxTreeCtrl* overview (p. 1396), *wxListBox* (p. 621), *wxListCtrl* (p. 630), *wxImageList* (p. 584), *wxTreeEvent* (p. 1151)

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. 1134).

validator

Window validator.

name

Window name.

See also

wxTreeCtrl::Create (p. 1138), *wxValidator* (p. 1166)

wxTreeCtrl::~~wxTreeCtrl

void ~wxTreeCtrl()

Destructor, destroying the list control.

wxTreeCtrl::AddRoot

wxTreeItemId AddRoot(const wxString& text, int image = -1, int sellmage = -1, wxTreeItemData* data = NULL)

Adds the root node to the tree, returning the new item.

If *image* > -1 and *sellmage* 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 sellmage = -1, wxTreeItemData* data = NULL)

Appends an item to the end of the branch identified by *parent*, return a new item id.

If *image* > -1 and *sellmage* is -1, the same image is used for both selected and unselected items.

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. 1136) for further details.

wxTreeCtrl::Delete

```
void Delete(const wxTreeItemId& item)
```

Deletes the specified item.

wxTreeCtrl::DeleteAllItems

```
void DeleteAllItems()
```

Deletes all the items in the control.

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. 1138), *wxTreeEvent* (p. 1151)

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. 1138)

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 items label will be returned, otherwise the items 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 `awxRect` object or `None`.

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 used to edit a label.

wxTreeCtrl::GetFirstChild

wxTreeItemId GetFirstChild(const wxTreeItemId& *item*, long& *cookie*) const

Returns the first child; call *wxTreeCtrl::GetNextChild* (p. 1142) 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.

Returns an invalid tree item if there are no further children.

See also

wxTreeCtrl::GetNextChild (p. 1142)

wxPython note: In wxPython the returned *wxTreeItemId* and the new cookie value are both returned as a tuple containing the two values.

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::GetItemData

wxTreeItemData* GetItemData(const wxTreeItemId& *item*) const

Returns the tree item data associated with the item.

See also

wxTreeItemData (p. 1149)

wxPython note: wxPython provides the following shortcut method:

GetPyData(item)	Returns the Python Object associated with the wxTreeItemData for the given item Id.
------------------------	---

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::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. 1140), *GetLastChild* (p. 1141)

wxTreeCtrl::GetNextChild

wxTreeItemId GetNextChild(const wxTreeItemId& *item*, long& *cookie*) const

Returns the next child; call *wxTreeCtrl::GetFirstChild* (p. 1140) 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. 1140)

wxPython note: In wxPython the returned *wxTreeItemId* and the new cookie value are both returned as a tuple containing the two values.

wxTreeCtrl::GetNextSibling

wxTreeItemId GetNextSibling(const wxTreeItemId& *item*) const

Returns the next sibling of the specified item; call *wxTreeCtrl::GetPrevSibling* (p. 1143) for the previous sibling.

Returns an invalid tree item if there are no further siblings.

See also

wxTreeCtrl::GetPrevSibling (p. 1143)

wxTreeCtrl::GetNextVisible

wxTreeItemId GetNextVisible(const wxTreeItemId& *item*) const

Returns the next visible item.

wxTreeCtrl::GetParent

wxTreeItemId GetParent(const wxTreeItemId& *item*) const

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. 1142) for the next sibling.

Returns an invalid tree item if there are no further children.

See also

`wxTreeCtrl::GetNextSibling` (p. 1142)

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. 1144) for the controls which do have this style.

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*).

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. 1147)

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. 1148)

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.

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*)

Selects the given item.

wxTreeCtrl::SetIndent

void SetIndent(int *indent*)

Sets the indentation for the tree control.

wxTreeCtrl::SetImageList

void SetImageList(wxImageList* imageList)

Sets the normal image list.

wxTreeCtrl::SetItemBackgroundColour

void SetItemBackgroundColour(const wxTreeItemId& item, const wxColour& col)

Sets the colour of the items 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. 1145)

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.
-----------------------------	--

wxTreeCtrl::SetItemFont

void SetItemFont(const wxTreeItemId& item, const wxFont& font)

Sets the items 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. 1147)

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. 1141) 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 items text.

wxTreeCtrl::SetStateImageList

void SetStateImageList(wxImageList* imageList)

Sets the state image list (from which application-defined state images are taken).

wxTreeCtrl::SortChildren

void SortChildren(const wxTreeItemId& item)

Sorts the children of the given item using *OnCompareItems* (p. 1146) method of `wxTreeCtrl`. You should override that method to change the sort order (the default is ascending alphabetical order).

See also

wxTreeItemData (p. 1149), *OnCompareItems* (p. 1146)

`wxTreeCtrl::Toggle`

void Toggle(const wxTreeItemId& item)

Toggles the given item between collapsed and expanded states.

`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. 1149) if the control doesn't have `wxTR_MULTIPLE` style, or removes the selection from all items if it does have this style.

`wxTreeItemData`

`wxTreeItemData` is some (arbitrary) user class associated with some item. The main advantage of having this class (compared to the old untyped interface) is that `wxTreeItemData`'s are destroyed automatically by the tree and, as this class has virtual dtor, it means that the memory will be automatically freed. We don't just use `wxObject` instead of `wxTreeItemData` because the size of this class is critical: in any real application, each tree leaf will have `wxTreeItemData` associated with it and number of leaves may be quite big.

Because the objects of this class are deleted by the tree, they should always be allocated on the heap.

Derived from

`wxTreeItemId`

Include files

<wx/treectrl.h>

See also

wxTreeCtrl (p. 1134)

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 <i>wxTreeItemData</i>

wxTreeItemData::~~wxTreeItemData

void ~wxTreeItemData()

Virtual destructor.

wxTreeItemData::GetId

const wxTreeItem& GetId()

Returns the item associated with this node.

wxTreeItemData::SetId

void SetId(const wxTreeItemId& *id*)

Sets the item associated with this node.

wxTreeEvent

A tree event holds information about events associated with wxTreeCtrl objects.

Derived from

wxNotifyEvent (p. 745)
wxCommandEvent (p. 152)
wxEvent (p. 375)
wxObject (p. 746)

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) 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) Drag ended (drop).
EVT_TREE_BEGIN_LABEL_EDIT(id, func) Begin editing a label. This can be prevented by calling *Veto()* (p. 746).
EVT_TREE_END_LABEL_EDIT(id, func) Finish editing a label. This can be prevented by calling *Veto()* (p. 746).
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_EXPANDED(id, func) Parent has been expanded.
EVT_TREE_ITEM_EXPANDING(id, func) Parent is being expanded. This can be prevented by calling *Veto()* (p. 746).
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. 746).
EVT_TREE_KEY_DOWN(id, func) A key has been pressed.

See also

wxTreeCtrl (p. 1134)

wxTreeEvent::wxTreeEvent

wxTreeEvent(WXTYPE *commandType* = 0, int *id* = 0)

Constructor.

wxTreeEvent::GetItem

wxTreeItemId GetItem() const

Returns the item (valid for all events).

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 or a click event.

wxTreeEvent::GetCode

int GetCode() const

The key code if the event was a key event.

wxTreeEvent::GetLabel

const wxString& GetLabel() const

Returns the label if the event was a begin or end edit label event.

wxTreeLayout

wxTreeLayout provides layout of simple trees with one root node, drawn left-to-right, with user-defined spacing between nodes.

wxTreeLayout is an abstract class that must be subclassed. The programmer defines various member functions which will access whatever data structures are appropriate for

the application, and `wxTreeLayout` uses these when laying out the tree.

Nodes are identified by long integer identifiers. The derived class communicates the actual tree structure to `wxTreeLayout` by defining `wxTreeLayout::GetChildren` (p. 1155) and `wxTreeLayout::GetNodeParent` (p. 1156) functions.

The application should call `wxTreeLayout::DoLayout` (p. 1154) to do the tree layout. Depending on how the derived class has been defined, either `wxTreeLayout::Draw` (p. 1154) must be called (for example by the `OnPaint` member of a `wxScrolledWindow`) or the application-defined drawing code should be called as normal.

For example, if you have an image drawing system already defined, you may want `wxTreeLayout` to position existing node images in that system. So you just need a way for `wxTreeLayout` to set the node image positions according to the layout algorithm, and the rest will be done by your own image drawing system.

The algorithm is due to Gabriel Robins [1], a linear-time algorithm originally implemented in LISP for AI applications.

The original algorithm has been modified so that both X and Y planes are calculated simultaneously, increasing efficiency slightly. The basic code is only a page or so long.

Below is the example tree generated by the program `test.cc`.

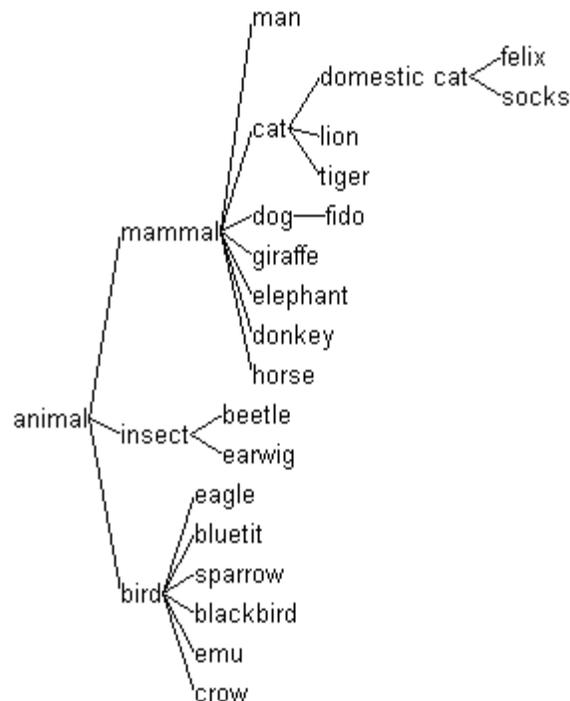


Figure 1: Example tree

Derived from

wxObject

See also

wxTreeLayoutStored (p. 1159)

wxTreeLayout::wxTreeLayout

wxTreeLayout()

Constructor.

wxTreeLayout::ActivateNode

void ActivateNode(long id, bool active)

Define this so wxTreeLayout can turn nodes on and off for drawing purposes (not all nodes may be connected in the tree). See also *wxTreeLayout::NodeActive* (p. 1157).

wxTreeLayout::CalcLayout

void CalcLayout(long id, int level)

Private function for laying out a branch.

wxTreeLayout::DoLayout

void DoLayout(wxDC& dc, long topNode = -1)

Calculates the layout for the tree, optionally specifying the top node.

wxTreeLayout::Draw

void Draw(wxDC& dc)

Call this to let wxTreeLayout draw the tree itself, once the layout has been calculated with *wxTreeLayout::DoLayout* (p. 1154).

wxTreeLayout::DrawBranch

void DrawBranch(long from, long to, wxDC& dc)

Defined by wxTreeLayout to draw an arc between two nodes.

wxTreeLayout::DrawBranches

void DrawBranches(wxDC& dc)

Defined by wxTreeLayout to draw the arcs between nodes.

wxTreeLayout::DrawNode

void DrawNode(long id, wxDC& dc)

Defined by wxTreeLayout to draw a node.

wxTreeLayout::DrawNodes

void DrawNodes(wxDC& dc)

Defined by wxTreeLayout to draw the nodes.

wxTreeLayout::GetChildren

void GetChildren(long id, wxList &list)

Must be defined to return the children of node *id* in the given list of integers.

wxTreeLayout::GetNextNode

long GetNextNode(long id)

Must be defined to return the next node after *id*, so that wxTreeLayout can iterate through all relevant nodes. The ordering is not important. The function should return -1 if there are no more nodes.

wxTreeLayout::GetNodeName

wxString GetNodeName(long id) const

May optionally be defined to get a node's name (for example if leaving the drawing to wxTreeLayout).

wxTreeLayout::GetNodeSize

void GetNodeSize(long id, long* x, long* y) const

Can be defined to indicate a node's size, or left to wxTreeLayout to use the name as an indication of size.

wxTreeLayout::GetNodeParent

long GetNodeParent(long id) const

Must be defined to return the parent node of *id*. The function should return -1 if there is no parent.

wxTreeLayout::GetNodeX

long GetNodeX(long id) const

Must be defined to return the current X position of the node. Note that coordinates are assumed to be at the top-left of the node so some conversion may be necessary for your application.

wxTreeLayout::GetNodeY

long GetNodeY(long id) const

Must be defined to return the current Y position of the node. Note that coordinates are assumed to be at the top-left of the node so some conversion may be necessary for your application.

wxTreeLayout::GetLeftMargin

long GetLeftMargin() const

Gets the left margin set with *wxTreeLayout::SetMargins* (p. 1158).

wxTreeLayout::GetOrientation

bool GetOrientation() const

Gets the orientation: TRUE means top-to-bottom, FALSE means left-to-right (the default).

wxTreeLayout::GetTopMargin

long GetTopMargin() const

Gets the top margin set with *wxTreeLayout::SetMargins* (p. 1158).

wxTreeLayout::GetTopNode

long GetTopNode() const

wxTreeLayout calls this to get the top of the tree. Don't redefine this; call *wxTreeLayout::SetTopNode* (p. 1158) instead before calling *wxTreeLayout::DoLayout* (p. 1154).

wxTreeLayout::GetXSpacing

long GetXSpacing() const

Gets the horizontal spacing between nodes.

wxTreeLayout::GetYSpacing

long GetYSpacing() const

Gets the vertical spacing between nodes.

wxTreeLayout::Initialize

void Initialize()

Initializes *wxTreeLayout*. Call from application or overridden **Initialize** or constructor.

wxTreeLayout::NodeActive

bool NodeActive(long id)

Define this so *wxTreeLayout* can know which nodes are to be drawn (not all nodes may be connected in the tree). See also *wxTreeLayout::ActivateNode* (p. 1154).

wxTreeLayout::SetNodeName

void SetNodeName(long id, const wxString& name)

May optionally be defined to set a node's name.

wxTreeLayout::SetNodeX

void SetNodeX(long id, long x)

Must be defined to set the current X position of the node. Note that coordinates are assumed to be at the top-left of the node so some conversion may be necessary for your application.

wxTreeLayout::SetNodeY

void SetNodeY(long id, long y)

Must be defined to set the current Y position of the node. Note that coordinates are assumed to be at the top-left of the node so some conversion may be necessary for your application.

wxTreeLayout::SetOrientation

void SetOrientation(bool orientation)

Sets the tree orientation: TRUE means top-to-bottom, FALSE means left-to-right (the default).

wxTreeLayout::SetTopNode

void SetTopNode(long id)

Call this to identify the top of the tree to wxTreeLayout.

wxTreeLayout::SetSpacing

void SetSpacing(long x, long y)

Sets the horizontal and vertical spacing between nodes in the tree.

wxTreeLayout::SetMargins

void SetMargins(long x, long y)

Sets the left and top margins of the whole tree.

wxTreeLayoutStored

`wxTreeLayoutStored` provides storage for node labels, position and client data. It also provides hit-testing (which node a mouse event occurred on). It is usually a more convenient class to use than `wxTreeLayout`.

Derived from

`wxTreeLayout` (p. 1152)
`wxObject` (p. 746)

See also

`wxTreeLayout` (p. 1152)

wxTreeLayoutStored::wxTreeLayoutStored

wxTreeLayoutStored(int *noNodes* = 200)

Constructor. Specify the maximum number of nodes to be allocated.

wxTreeLayoutStored::AddChild

long AddChild(const wxString& *name*, const wxString& *parent* = "")

Adds a child with a given parent, returning the node id.

wxTreeLayoutStored::GetClientData

long GetClientData(long *id*) const

Gets the client data for the given node.

wxTreeLayoutStored::GetNode

wxStoredNode* GetNode(long *id*) const

Returns the `wxStoredNode` object for the given node id.

wxTreeLayoutStored::GetNodeCount

int GetNodeCount() const

Returns the current number of nodes.

wxTreeLayoutStored::GetNumNodes

int GetNumNodes() const

Returns the maximum number of nodes.

wxTreeLayoutStored::HitTest

wxString HitTest(wxMouseEvent& *event*, wxDC& *dc*)

Returns a string with the node name corresponding to the position of the mouse event, or the empty string if no node was detected.

wxTreeLayoutStored::NameTold

long NameTold(const wxString& *name*)

Returns the id for the given node name, or -1 if there was no such node.

wxTreeLayoutStored::SetClientData

void SetClientData(long *id*, long *clientData*)

Sets client data for the given node.

wxUpdateUIEvent

This class is used for pseudo-events which are called by wxWindows to give an application the chance to update various user interface elements.

Derived from

wxEvent (p. 375)

wxObject (p. 746)

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 <code>wxEVT_UPDATE_UI</code> event for the command with the given id.
EVT_UPDATE_UI_RANGE(id1, id2, func)	Process a <code>wxEVT_UPDATE_UI</code> 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. `wxWindows` 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. 1162). `wxWindows` 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. 693) is called to process any UI events for the window that owns the menu.

See also

Event handling overview (p. 1364)

wxUpdateUIEvent::wxUpdateUIEvent

`wxUpdateUIEvent(wxWindowID commandId = 0)`

Constructor.

wxUpdateUIEvent::m_checked

bool m_checked

TRUE if the element should be checked, FALSE otherwise.

wxUpdateUIEvent::m_enabled

bool m_checked

TRUE if the element should be enabled, FALSE otherwise.

wxUpdateUIEvent::m_setChecked

bool m_setCheckedTRUE if the application has set the **m_checked** member.**wxUpdateUIEvent::m_setEnabled**

bool m_setEnabledTRUE if the application has set the **m_enabled** member.**wxUpdateUIEvent::m_setText**

bool m_setTextTRUE if the application has set the **m_text** member.**wxUpdateUIEvent::m_text**

wxString m_text

Holds the text with which the the application wishes to update the UI element.

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() constReturns TRUE if the application has called **SetChecked**. For wxWindows internal use only.**wxUpdateUIEvent::GetSetEnabled**

bool GetSetEnabled() constReturns TRUE if the application has called **SetEnabled**. For wxWindows internal use only.**wxUpdateUIEvent::GetSetText**

bool GetSetText() constReturns TRUE if the application has called **SetText**. For wxWindows internal use only.**wxUpdateUIEvent::GetText**

wxString GetText() const

Returns the text that should be set for the UI element.

wxUpdateUIEvent::SetText

void SetText(const wxString& text)

Sets the text for this UI element.

wxURL

Derived from

wxObject (p. 746)

Include files

<wx/url.h>

See also

wxSocketBase (p. 938), *wxProtocol* (p. 849)

Example

```
wxURL url("http://a.host/a.dir/a.file");
wxInputStream *in_stream;

in_stream = url.GetInputStream();
// Then, you can use all IO calls of in_stream (See wxStream)
```

wxURL::wxURL

wxURL(const wxString& url)

Constructs an URL object from the string.

Parameters

url

Url string to parse.

wxURL::~~wxURL

~wxURL()

Destroys the URL object.

wxURL::GetProtocolName

wxString GetProtocolName() const

Returns the name of the protocol which will be used to get the URL.

wxURL::GetProtocol

wxProtocol& GetProtocol()

Returns a reference to the protocol which will be used to get the URL.

wxURL::GetPath

wxString GetPath()

Returns the path of the file to fetch. This path was encoded in 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_SNTAXERR	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 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.

Return value

Returns the initialized stream. You will have to delete it yourself.

See also

wxInputStream (p. 592)

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. 1166)

wxURL::SetProxy

void SetProxy(const wxString& url_proxy)

Sets the proxy to use for this URL.

See also

wxURL::SetDefaultProxy (p. 1166)

wxURL::ConvertToValidURI

static wxString ConvertToValidURI(const wxString& uri)

It converts a non-standardized URI to a valid network URI. It encodes non standard characters.

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. 1374).

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. 378)

wxObject (p. 746)

Include files

<wx/validate.h>

See also

Validator overview (p. 1374), *wxTextValidator* (p. 1092), *wxGenericValidator* (p. 477),

`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

wxvalidatorsetbellonerror

void SetBellOnError(bool *dolt* = 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. 1177) 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.

wxVariant is similar to wxExpr and also to wxPropertyValue. However, wxExpr is efficiency-optimized for a restricted range of data types, whereas wxVariant is less efficient but more extensible. wxPropertyValue may be replaced by wxVariant eventually.

Derived from

wxObject (p. 746)

Include files

<wx/variant.h>

See also

wxVariantData (p. 1177)

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(const wxStringList& *value*, const wxString& *name* = "")

Construction from a list of strings. This constructor copies *value*, the application is still responsible for deleting *value* and its contents.

wxVariant(const wxTime& *value*, const wxString& *name* = "")

Construction from a time.

wxVariant(const wxDate& value, const wxString& name = "")

Construction from a date.

wxVariant(void* value, const wxString& name = "")

Construction from a void pointer.

wxVariant(wxVariantData* data, const wxString& name = "")

Construction from user-defined data. The variant holds on to the *data* pointer.

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::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::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::GetDate

wxDate GetDate() const

Gets 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::GetTime

wxTime GetTime() const

Gets the time 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::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::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 =(const wxDate& value)

void operator =(const wxTime& value)

void operator =(void* value)

void operator =(const wxList& value)

void operator =(const wxStringList& value)

Assignment operators.

wxVariant::operator ==

bool operator ==(const wxVariant& value)

bool operator ==(const wxString& value)

bool operator ==(const char* value)

bool operator ==(char value)

bool operator ==(const long value)

bool operator ==(const bool value)

bool operator ==(const double value)

bool operator ==(const wxDate& value)

bool operator ==(const wxTime& value)

bool operator ==(void* value)

bool operator ==(const wxList& value)

bool operator ==(const wxStringList& value)

Equality test operators.

wxVariant::operator !=

bool operator !=(const wxVariant& value)
bool operator !=(const wxString& value)
bool operator !=(const char* value)
bool operator !=(char value)
bool operator !=(const long value)
bool operator !=(const bool value)
bool operator !=(const double value)
bool operator !=(const wxDate& value)
bool operator !=(const wxTime& value)
bool operator !=(void* value)
bool operator !=(const wxList& value)
bool operator !=(const wxStringList& value)

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. 1172).

wxVariant::operator double

double operator double() const

Operator for implicit conversion to a double, using *wxVariant::GetDouble* (p. 1172).

long operator long() const

Operator for implicit conversion to a long, using *wxVariant::GetLong* (p. 1172).

wxVariant::operator wxDate

wxDate operator wxDate() const

Operator for implicit conversion to a wxDate, using *wxVariant::GetDate* (p. 1172).

wxVariant::operator wxString

wxString operator wxString() const

Operator for implicit conversion to a string, using *wxVariant::MakeString* (p. 1174).

wxVariant::operator wxTime

wxTime operator wxTime() const

Operator for implicit conversion to a wxTime, using *wxVariant::GetTime* (p. 1173).

wxVariant::operator void*

void* operator void*() const

Operator for implicit conversion to a pointer to a void, using *wxVariant::GetVoidPtr* (p. 1173).

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. 746)

Include files

<wx/variant.h>

See also

wxVariant (p. 1169)

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::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*.

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 `wxWindows`, and cooperates with the `wxDocument` (p. 351), `wxDocTemplate` (p. 345) and `wxDocManager` (p. 332) classes.

Derived from

`wxEvtHandler` (p. 378)

`wxObject` (p. 746)

Include files

<wx/docview.h>

See also

`wxView` overview (p. 1404), `wxDocument` (p. 351), `wxDocTemplate` (p. 345), `wxDocManager` (p. 332)

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

wxDocumentManager* GetDocumentManager() const

Returns a pointer to the document manager instance associated with this view.

wxView::GetFrame

wxFrame * GetFrame()

Gets the frame associated with the view (if any).

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 application's 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::OnCreate

virtual bool OnCreate(wxDocument* doc, long flags)

Called just after view construction to give the view a chance to initialize itself based on the passed document and flags (unused). By default, simply returns TRUE. If the function returns FALSE, the view will be deleted.

The predefined document child frame, wxDocChildFrame, calls this function automatically.

wxView::OnCreatePrintout

virtual wxPrintout* OnCreatePrintout()

If the printing framework is enabled in the library, this function returns a *wxPrintout* (p. 807) object for the purposes of printing. It should create a new object everytime 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::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(wxFrame* frame)

Sets the frame associated with this view. The application should call this if possible, to

tell the view about the frame.

wxView::SetViewName

void SetViewName(const wxString& name)

Sets the view type name. Should only be called by the framework.

wxWave

This class represents a short wave file, in Windows WAV format, that can be stored in memory and played. Currently this class is implemented on Windows and GTK (Linux) only.

Derived from

wxObject (p. 746)

Include files

<wx/wave.h>

wxWave::wxWave

wxWave()

Default constructor.

wxWave(const wxString& fileName, bool isResource = FALSE)

Constructs a wave object from a file or resource. Call *wxWave::IsOk* (p. 1184) 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.

wxWave::~~wxWave

~wxWave()

Destroys the wxWave object.

wxWave::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.

wxWave::IsOk

bool IsOk() const

Returns TRUE if the object contains a successfully loaded file or resource, FALSE otherwise.

wxWave::Play

bool Play(bool async = TRUE, bool looped = FALSE) const

Plays the wave file synchronously or asynchronously, looped or single-shot.

wxWindow

wxWindow is the base class for all windows. Any children of the window will be deleted automatically by the destructor before the window itself is deleted.

Please note that we documented a number of handler functions (OnChar(), OnMouse() etc.) in this help text. These must not be called by a user program and are documented only for illustration. On several platforms, only a few of these handlers are actually

written (they are not always needed) and if you are uncertain on how to add a certain behaviour to a window class, intercept the respective event as usual and call `wxEvtHandler::Skip` (p. 378) so that the native platform can implement its native behaviour or just ignore the event if nothing needs to be done.

Derived from

`wxEvtHandler` (p. 378)
`wXObject` (p. 746)

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. <code>wxBORDER</code> is the old name for this style.
wxDOUBLE_BORDER	Displays a double border. Windows only.
wxSUNKEN_BORDER	Displays a sunken border.
wxRAISED_BORDER	Displays a raised border. GTK only.
wxSTATIC_BORDER	Displays a border suitable for a static control. Windows only.
wxTRANSPARENT_WINDOW	The window is transparent, that is, it will not receive paint events. Windows only.
wxNO_3D	Prevents the children of this window taking on 3D styles, even though the application-wide policy is for 3D controls. 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 events - even for keys like TAB or ENTER which are usually used for dialog navigation and which wouldn't be generated without this style
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. (Still used?)
wxHSCROLL	Use this style to enable a horizontal scrollbar. (Still used?)
wxCLIP_CHILDREN	Use this style to eliminate flicker caused by the background being repainted, then children being painted over them. Windows only.

See also *window styles overview* (p. 1371).

See also

Event handling overview (p. 1364)

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 *wxWindows* 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 *wxWindows* 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. 1184).

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. 1192) so that

wxWindows can delete a window only when it is safe to do so, in idle time.

See also

Window deletion overview (p. 1371), *wxWindow::OnCloseWindow* (p. 1208), *wxWindow::Destroy* (p. 1192), *wxCloseEvent* (p. 124)

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.

Parameters

child

Child window to add.

wxWindow::CaptureMouse

virtual void CaptureMouse()

Directs all mouse input to this window. Call *wxWindow::ReleaseMouse* (p. 1219) to release the capture.

See also

wxWindow::ReleaseMouse (p. 1219)

wxWindow::Center

void Center(int direction)

A synonym for *Centre* (p. 1188).

wxWindow::CenterOnParent

void CenterOnParent(int direction)

A synonym for *CentreOnParent* (p. 1188).

wxWindow::CenterOnScreen

void CenterOnScreen(int direction)

A synonym for *CentreOnScreen* (p. 1189).

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. 1187)

wxWindow::CentreOnParent

void CentreOnParent(int *direction* = *wxBOTH*)

Centres the window on its parent. This is a more readable synonym for *Centre* (p. 1188).

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. 1188).

See also

wxWindow::CentreOnScreen (p. 1187)

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. 1187)

wxWindow::Clear

void Clear()

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**

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

virtual bool Close(bool *force* = FALSE)

The purpose of this call is to provide a safer way of destroying a window than using the *delete* operator.

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. 124) for the window, providing an opportunity for the window to choose whether to destroy the window.

The close handler should check whether the window is being deleted forcibly, using *wxCloseEvent::GetForce* (p. 126), in which case it should destroy the window using *wxWindow::Destroy* (p. 1192).

Applies to managed windows (wxFrame and wxDialog classes) only.

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 itself only hides the dialog.

To guarantee that the window will be destroyed, call *wxWindow::Destroy* (p. 1192) instead.

See also

Window deletion overview (p. 1371), *wxWindow::OnCloseWindow* (p. 1208), *wxWindow::Destroy* (p. 1192), *wxCloseEvent* (p. 124)

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. Dialogs created using Dialog Editor optionally use dialog units.

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. 1191)

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. Dialogs created using Dialog Editor optionally use dialog units.

See also

wxWindow::ConvertDialogToPixels (p. 1190)

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::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.

See also

wxWindow::OnDropFiles (p. 1208)

wxWindow::Enable

virtual void Enable(bool *enable*)

Enable or disable the window for user input.

Parameters

enable

If TRUE, enables the window for input. If FALSE, disables the window.

See also

wxWindow::IsEnabled (p. 1202)

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. 1225)

wxWindow::FindWindow

wxWindow* FindWindow(long *id*)

Find a child of this window, by identifier.

wxWindow* FindWindow(const wxString& name)

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::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.

wxWindow::GetBackgroundColour

virtual wxColour GetBackgroundColour() const

Returns the background colour of the window.

See also

wxWindow::SetBackgroundColour (p. 1221), *wxWindow::SetForegroundColour* (p. 1225), *wxWindow::GetForegroundColour* (p. 1196), *wxWindow::OnEraseBackground* (p. 1209)

wxWindow::GetBestSize

virtual 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. 764)), the size returned by this function will be the same as the size the window would have had after calling *Fit* (p. 1194).

wxWindow::GetCaret

wxCaret * GetCaret() const

Returns the *caret* (p. 105) associated with the window.

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::GetClientSize

virtual void GetClientSize(int* width, int* height) const**virtual 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 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:

wxGetClientSizeTuple()	Returns a 2-tuple of (width, height)
wxGetClientSize()	Returns a wxSize object

wxWindow::GetConstraints

wxLayoutConstraints* GetConstraints() const

Returns a pointer to the window's layout constraints, or NULL if there are none.

wxWindow::GetDropTarget

wxDropTarget* GetDropTarget() const

Returns the associated drop target, which may be NULL.

See also

wxWindow::SetDropTarget (p. 1224), *Drag and drop overview* (p. 1420)

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. 1224), *wxWindow::PushEventHandler* (p. 1218), *wxWindow::PopEventHandler* (p. 1218), *wxEvtHandler::ProcessEvent* (p. 382), *wxEvtHandler* (p. 378)

wxWindow::GetExtraStyle

long GetExtraStyle() const

Returns the extra style bits for the window.

wxWindow::GetFont

wxFont& GetFont() const

Returns a reference to the font for this window.

See also

wxWindow::SetFont (p. 1225)

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. 1225), *wxWindow::SetBackgroundColour* (p. 1221), *wxWindow::GetBackgroundColour* (p. 1194)

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 or **GtkWidget** for GTK.

wxPython note: This method will return an integer in wxPython.

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. 1226), *Window identifiers* (p. 1368)

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::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. 1226).

See also

wxWindow::SetName (p. 1226)

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 or if no parent, relative to the whole display.

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)

wxWindow::GetRect

virtual wxRect GetRect() const

Returns the size and position of the window as a *wxRect* (p. 868) object.

wxWindow::GetScrollThumb

virtual int GetScrollThumb(int orientation)

Returns the built-in scrollbar thumb size.

See also

wxWindow::SetScrollbar (p. 1227)

wxWindow::GetScrollPos

virtual int GetScrollPos(int orientation)

Returns the built-in scrollbar position.

See also

See *wxWindow::SetScrollbar* (p. 1227)

wxWindow::GetScrollRange

virtual int GetScrollRange(int orientation)

Returns the built-in scrollbar range.

See also

wxWindow::SetScrollbar (p. 1227)

wxWindow::GetSize

virtual void GetSize(int* width, int* height) const

virtual wxSize GetSize() const

This gets the size of the entire window in pixels.

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)

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)

wxWindow::GetTitle

virtual wxString GetTitle()

Gets the window's title. Applicable only to frames and dialogs.

See also

wxWindow::SetTitle (p. 1231)

wxWindow::GetUpdateRegion

virtual wxRegion GetUpdateRegion() const

Returns the region specifying which parts of the window have been damaged. Should only be called within an *OnPaint* (p. 1214) event handler.

See also

wxRegion (p. 885), *wxRegionIterator* (p. 889), *wxWindow::OnPaint* (p. 1214)

wxWindow::GetValidator

wxValidator* GetValidator() const

Returns a pointer to the current validator for the window, or NULL if there is none.

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::InitDialog

void InitDialog()

Sends an *wxWindow::OnInitDialog* (p. 1211) event, which in turn transfers data to the dialog via validators.

See also

wxWindow::OnInitDialog (p. 1211)

wxWindow::IsEnabled

virtual bool IsEnabled() const

Returns TRUE if the window is enabled for input, FALSE otherwise.

See also

wxWindow::Enable (p. 1193)

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 a 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. 1221) on when this function gets called automatically using auto layout.

wxWindow::LoadFromResource

virtual bool LoadFromResource(wxWindow* parent, const wxString& resourceName, const wxResourceTable* resourceTable = NULL)

Loads a panel or dialog from a resource file.

Parameters

parent
Parent window.

resourceName
The name of the resource to load.

resourceTable

The resource table to load it from. If this is NULL, the default resource table will be used.

Return value

TRUE if the operation succeeded, otherwise FALSE.

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. (This function is not implemented anywhere).

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. 785) 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. 1228)

wxPython note: In place of a single overloaded method name, wxPython implements the following methods:

Move(point)	Accepts a <code>wxPoint</code>
MoveXY(x, y)	Accepts a pair of integers

wxWindow::OnActivate

void OnActivate(wxActivateEvent& event)

Called when a window is activated or deactivated.

Parameters

event
Object containing activation information.

Remarks

If the window is being activated, `wxActivateEvent::GetActive` (p. 21) returns `TRUE`, otherwise it returns `FALSE` (it is being deactivated).

See also

`wxActivateEvent` (p. 20), *Event handling overview* (p. 1364)

wxWindow::OnChar

void OnChar(wxKeyEvent& event)

Called when the user has pressed a key that is not a modifier (`SHIFT`, `CONTROL` or `ALT`).

Parameters

event
Object containing keypress information. See `wxKeyEvent` (p. 607) for details about

this class.

Remarks

This member function is called in response to a keypress. To intercept this event, use the `EVT_CHAR` macro in an event table definition. Your **OnChar** handler may call this default function to achieve default keypress functionality.

Note that the ASCII values do not have explicit key codes: they are passed as ASCII values.

Note that not all keypresses can be intercepted this way. If you wish to intercept modifier keypresses, then you will need to use `wxWindow::OnKeyDown` (p. 1209) or `wxWindow::OnKeyUp` (p. 1210).

Most, but not all, windows allow keypresses to be intercepted.

See also

`wxWindow::OnKeyDown` (p. 1209), `wxWindow::OnKeyUp` (p. 1210), `wxKeyEvent` (p. 607), `wxWindow::OnCharHook` (p. 1206), *Event handling overview* (p. 1364)

wxWindow::OnCharHook

void OnCharHook(wxKeyEvent& event)

This member is called to allow the window to intercept keyboard events before they are processed by child windows.

Parameters

event

Object containing keypress information. See `wxKeyEvent` (p. 607) for details about this class.

Remarks

This member function is called in response to a keypress, if the window is active. To intercept this event, use the `EVT_CHAR_HOOK` macro in an event table definition. If you do not process a particular keypress, call `wxEvent::Skip` (p. 378) to allow default processing.

An example of using this function is in the implementation of escape-character processing for `wxDialog`, where pressing ESC dismisses the dialog by **OnCharHook** 'forging' a cancel button press event.

Note that the ASCII values do not have explicit key codes: they are passed as ASCII values.

This function is only relevant to top-level windows (frames and dialogs), and under

Windows only. Under GTK the normal `EVT_CHAR_` event has the functionality, i.e. you can intercept it and if you don't call `wxEvent::Skip` (p. 378) the window won't get the event.

See also

`wxKeyEvent` (p. 607), `wxWindow::OnCharHook` (p. 1206), `wxApp::OnCharHook` (p. 26), *Event handling overview* (p. 1364)

wxWindow::OnCommand

virtual void OnCommand(`wxEvtHandler& object`, `wxCommandEvent& event`)

This virtual member function is called if the control does not handle the command event.

Parameters

object
Object receiving the command event.

event
Command event

Remarks

This virtual function is provided mainly for backward compatibility. You can also intercept commands from child controls by using an event table, with identifiers or identifier ranges to identify the control(s) in question.

See also

`wxCommandEvent` (p. 152), *Event handling overview* (p. 1364)

wxWindow::OnClose

virtual bool OnClose()

Called when the user has tried to close a a frame or dialog box using the window manager (X) or system menu (Windows).

Note: This is an obsolete function. It is superceded by the `wxWindow::OnCloseWindow` (p. 1208) event handler.

Return value

If `TRUE` is returned by `OnClose`, the window will be deleted by the system, otherwise the attempt will be ignored. Do not delete the window from within this handler, although you may delete other windows.

See also

Window deletion overview (p. 1371), *wxWindow::Close* (p. 1190), *wxWindow::OnCloseWindow* (p. 1208), *wxCloseEvent* (p. 124)

wxWindow::OnCloseWindow

void OnCloseWindow(wxCloseEvent& event)

This is an event handler function called when the user has tried to close a frame or dialog box using the window manager (X) or system menu (Windows). It is called via the *wxWindow::Close* (p. 1190) function, so that the application can also invoke the handler programmatically.

Use the `EVT_CLOSE` event table macro to handle close events.

You should check whether the application is forcing the deletion of the window using *wxCloseEvent::GetForce* (p. 126). If this is `TRUE`, destroy the window using *wxWindow::Destroy* (p. 1192). If not, it is up to you whether you respond by destroying the window.

(Note: `GetForce` is now superseded by `CanVeto`. So to test whether forced destruction of the window is required, test for the negative of `CanVeto`. If `CanVeto` returns `FALSE`, it is not possible to skip window deletion.)

If you don't destroy the window, you should call *wxCloseEvent::Veto* (p. 126) to let the calling code know that you did not destroy the window. This allows the *wxWindow::Close* (p. 1190) function to return `TRUE` or `FALSE` depending on whether the close instruction was honoured or not.

Remarks

The *wxWindow::OnClose* (p. 1207) virtual function remains for backward compatibility with earlier versions of wxWindows. The default **OnCloseWindow** handler for `wxFrame` and `wxDialog` will call **OnClose**, destroying the window if it returns `TRUE` or if the close is being forced.

See also

Window deletion overview (p. 1371), *wxWindow::Close* (p. 1190), *wxWindow::OnClose* (p. 1207), *wxWindow::Destroy* (p. 1192), *wxCloseEvent* (p. 124), *wxApp::OnQueryEndSession* (p. 28), *wxApp::OnEndSession* (p. 27)

wxWindow::OnDropFiles

void OnDropFiles(wxDropFilesEvent& event)

Called when files have been dragged from the file manager to the window.

Parameters

event

Drop files event. For more information, see *wxDropFilesEvent* (p. 364).

Remarks

The window must have previously been enabled for dropping by calling *wxWindow::DragAcceptFiles* (p. 1192).

This event is only generated under Windows.

To intercept this event, use the `EVT_DROP_FILES` macro in an event table definition.

See also

wxDropFilesEvent (p. 364), *wxWindow::DragAcceptFiles* (p. 1192), *Event handling overview* (p. 1364)

wxWindow::OnEraseBackground

void OnEraseBackground(wxEraseEvent& *event*)

Called when the background of the window needs to be erased.

Parameters

event

Erase background event. For more information, see *wxEraseEvent* (p. 374).

Remarks

Under non-Windows platforms, 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.

See also

wxEraseEvent (p. 374), *Event handling overview* (p. 1364)

wxWindow::OnKeyDown

void OnKeyDown(wxKeyEvent& event)

Called when the user has pressed a key, before it is translated into an ASCII value using other modifier keys that might be pressed at the same time.

Parameters

event

Object containing keypress information. See *wxKeyEvent* (p. 607) for details about this class.

Remarks

This member function is called in response to a key down event. To intercept this event, use the `EVT_KEY_DOWN` macro in an event table definition. Your **OnKeyDown** handler may call this default function to achieve default keypress functionality.

Note that not all keypresses can be intercepted this way. If you wish to intercept special keys, such as shift, control, and function keys, then you will need to use *wxWindow::OnKeyDown* (p. 1209) or *wxWindow::OnKeyUp* (p. 1210).

Most, but not all, windows allow keypresses to be intercepted.

See also

wxWindow::OnChar (p. 1205), *wxWindow::OnKeyUp* (p. 1210), *wxKeyEvent* (p. 607), *wxWindow::OnCharHook* (p. 1206), *Event handling overview* (p. 1364)

wxWindow::OnKeyUp

void OnKeyUp(wxKeyEvent& event)

Called when the user has released a key.

Parameters

event

Object containing keypress information. See *wxKeyEvent* (p. 607) for details about this class.

Remarks

This member function is called in response to a key up event. To intercept this event, use the `EVT_KEY_UP` macro in an event table definition. Your **OnKeyUp** handler may call this default function to achieve default keypress functionality.

Note that not all keypresses can be intercepted this way. If you wish to intercept special keys, such as shift, control, and function keys, then you will need to use *wxWindow::OnKeyDown* (p. 1209) or *wxWindow::OnKeyUp* (p. 1210).

Most, but not all, windows allow key up events to be intercepted.

See also

wxWindow::OnChar (p. 1205), *wxWindow::OnKeyDown* (p. 1209), *wxKeyEvent* (p. 607), *wxWindow::OnCharHook* (p. 1206), *Event handling overview* (p. 1364)

wxWindow::OnKillFocus

void OnKillFocus(wxFocusEvent& event)

Called when a window's focus is being killed.

Parameters

event

The focus event. For more information, see *wxFocusEvent* (p. 433).

Remarks

To intercept this event, use the macro `EVT_KILL_FOCUS` in an event table definition.

Most, but not all, windows respond to this event.

See also

wxFocusEvent (p. 433), *wxWindow::OnSetFocus* (p. 1216), *Event handling overview* (p. 1364)

wxWindow::OnIdle

void OnIdle(wxIdleEvent& event)

Provide this member function for any processing which needs to be done when the application is idle.

See also

wxApp::OnIdle (p. 27), *wxIdleEvent* (p. 557)

wxWindow::OnInitDialog

void OnInitDialog(wxInitDialogEvent& event)

Default handler for the `wxEVT_INIT_DIALOG` event. Calls *wxWindow::TransferDataToWindow* (p. 1233).

Parameters

event
Dialog initialisation event.

Remarks

Gives the window the default behaviour of transferring data to child controls via the validator that each control has.

See also

wxValidator (p. 1166), *wxWindow::TransferDataToWindow* (p. 1233)

wxWindow::OnMenuCommand

void OnMenuCommand(wxCommandEvent& *event*)

Called when a menu command is received from a menu bar.

Parameters

event
The menu command event. For more information, see *wxCommandEvent* (p. 152).

Remarks

A function with this name doesn't actually exist; you can choose any member function to receive menu command events, using the EVT_COMMAND macro for individual commands or EVT_COMMAND_RANGE for a range of commands.

See also

wxCommandEvent (p. 152), *wxWindow::OnMenuHighlight* (p. 1212), *Event handling overview* (p. 1364)

wxWindow::OnMenuHighlight

void OnMenuHighlight(wxMenuEvent& *event*)

Called when a menu select is received from a menu bar: that is, the mouse cursor is over a menu item, but the left mouse button has not been pressed.

Parameters

event
The menu highlight event. For more information, see *wxMenuEvent* (p. 707).

Remarks

You can choose any member function to receive menu select events, using the `EVT_MENU_HIGHLIGHT` macro for individual menu items or `EVT_MENU_HIGHLIGHT_ALL` macro for all menu items.

The default implementation for `wxFrame::OnMenuHighlight` (p. 460) displays help text in the first field of the status bar.

This function was known as **OnMenuSelect** in earlier versions of wxWindows, but this was confusing since a selection is normally a left-click action.

See also

`wxMenuEvent` (p. 707), `wxWindow::OnMenuCommand` (p. 1212), *Event handling overview* (p. 1364)

wxWindow::OnMouseEvent

void OnMouseEvent(wxMouseEvent& event)

Called when the user has initiated an event with the mouse.

Parameters

event

The mouse event. See `wxMouseEvent` (p. 721) for more details.

Remarks

Most, but not all, windows respond to this event.

To intercept this event, use the `EVT_MOUSE_EVENTS` macro in an event table definition, or individual mouse event macros such as `EVT_LEFT_DOWN`.

See also

`wxMouseEvent` (p. 721), *Event handling overview* (p. 1364)

wxWindow::OnMove

void OnMove(wxMoveEvent& event)

Called when a window is moved.

Parameters

event

The move event. For more information, see *wxMoveEvent* (p. 729).

Remarks

Use the `EVT_MOVE` macro to intercept move events.

Remarks

Not currently implemented.

See also

wxMoveEvent (p. 729), *wxFrame::OnSize* (p. 460), *Event handling overview* (p. 1364)

wxWindow::OnPaint

void OnPaint(wxPaintEvent& event)

Sent to the event handler when the window must be refreshed.

Parameters

event

Paint event. For more information, see *wxPaintEvent* (p. 760).

Remarks

Use the `EVT_PAINT` macro in an event table definition to intercept paint events.

Note that In a paint event handler, the application must *always* create a *wxPaintDC* (p. 759) 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. 889) 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(&rect);

        // Repaint this rectangle
        ...some code...

        upd ++ ;
    }
}

```

See also

wxPaintEvent (p. 760), *wxPaintDC* (p. 759), *Event handling overview* (p. 1364)

wxWindow::OnScroll

```
void OnScroll(wxScrollWinEvent& event)
```

Called when a scroll window event is received from one of the window's built-in scrollbars.

Parameters

event

Command event. Retrieve the new scroll position by calling *wxScrollEvent::GetPosition* (p. 911), and the scrollbar orientation by calling *wxScrollEvent::GetOrientation* (p. 911).

Remarks

Note that it is not possible to distinguish between horizontal and vertical scrollbars until the function is executing (you can't have one function for vertical, another for horizontal events).

See also

wxScrollWinEvent (p. 908), *Event handling overview* (p. 1364)

wxWindow::OnSetFocus

void OnSetFocus(wxFocusEvent& event)

Called when a window's focus is being set.

Parameters

event

The focus event. For more information, see *wxFocusEvent* (p. 433).

Remarks

To intercept this event, use the macro `EVT_SET_FOCUS` in an event table definition.

Most, but not all, windows respond to this event.

See also

wxFocusEvent (p. 433), *wxWindow::OnKillFocus* (p. 1211), *Event handling overview* (p. 1364)

wxWindow::OnSize

void OnSize(wxSizeEvent& event)

Called when the window has been resized.

Parameters

event

Size event. For more information, see *wxSizeEvent* (p. 923).

Remarks

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. 1195) 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. 1219) to invalidate the entire window.

See also

wxSizeEvent (p. 923), *Event handling overview* (p. 1364)

wxWindow::OnSysColourChanged

void OnSysColourChanged(wxOnSysColourChangedEvent& event)

Called when the user has changed the system colours. Windows only.

Parameters

event

System colour change event. For more information, see *wxSysColourChangedEvent* (p. 1034).

See also

wxSysColourChangedEvent (p. 1034), *Event handling overview* (p. 1364)

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. 1224), *wxWindow::GetEventHandler* (p. 1196), *wxWindow::PushEventHandler* (p. 1218), *wxEvtHandler::ProcessEvent* (p. 382), *wxEvtHandler* (p. 378)

wxWindow::PopupMenu

bool PopupMenu(wxMenu* menu, const wxPoint& pos)

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.

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. 683)

Remarks

Just before the menu is popped up, *wxMenu::UpdateUI* (p. 693) is called to ensure that the menu items are in the correct state. The menu does not get deleted by the window.

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. 1218) 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. 1217) to remove the event handler.

See also

`wxWindow::SetEventHandler` (p. 1224), `wxWindow::GetEventHandler` (p. 1196),
`wxWindow::PopEventHandler` (p. 1218), `wxEvtHandler::ProcessEvent` (p. 382),
`wxEvtHandler` (p. 378)

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 a message or event to be generated to repaint the window.

Parameters

eraseBackground

If TRUE, the background will be erased.

rect

If non-NULL, only the given rectangle will be treated as damaged.

wxWindow::ReleaseMouse

virtual void ReleaseMouse()

Releases mouse input captured with `wxWindow::CaptureMouse` (p. 1187).

See also

`wxWindow::CaptureMouse` (p. 1187)

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.

Parameters

child

Child window to remove.

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::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. 17).

wxWindow::SetAutoLayout

void SetAutoLayout(bool *autoLayout*)

Determines whether the *wxWindow::Layout* (p. 1203) function will be called automatically when the window is resized. Use in connection with *wxWindow::SetSizer* (p. 1230) and *wxWindow::SetConstraints* (p. 1223) for laying out subwindows.

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. 1223)

wxWindow::SetBackgroundColour

virtual void SetBackgroundColour(const wxColour& colour)

Sets the background colour of the window.

Parameters

colour

The colour to be used as the background colour.

Remarks

The background colour is usually painted by the default *wxWindow::OnEraseBackground* (p. 1209) 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::Clear* (p. 1189) or *wxWindow::Refresh* (p. 1219) after calling this function.

Use this function with care under GTK as the new appearance of the window might not look equally well when used with "Themes", i.e GTK's ability to change its look as the user wishes with run-time loadable modules.

See also

wxWindow::GetBackgroundColour (p. 1194), *wxWindow::SetForegroundColour* (p. 1225), *wxWindow::GetForegroundColour* (p. 1196), *wxWindow::Clear* (p. 1189), *wxWindow::Refresh* (p. 1219), *wxWindow::OnEraseBackground* (p. 1209)

wxWindow::SetCaret

void SetCaret(wxCaret *caret) const

Sets the *caret* (p. 105) 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. 1228), 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::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. 1263), *wxCursor* (p. 184)

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. 1221) 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. 924), only the sizer will have effect.

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. 1196), *Drag and drop overview* (p. 1420)

wxWindow::SetEventHandler

void SetEventHandler(wxEventHandler* 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. 1218) 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. 1196), `wxWindow::PushEventHandler` (p. 1218), `wxWindow::PopEventHandler` (p. 1218), `wxEvtHandler::ProcessEvent` (p. 382), `wxEvtHandler` (p. 378)

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.

wxWindow::SetFocus

virtual void SetFocus()

This sets the window to receive keyboard input.

wxWindow::SetFont

void SetFont(const wxFont& *font*)

Sets the font for this window.

Parameters

font

Font to associate with this window.

See also

wxWindow::GetFont (p. 1196)

wxWindow::SetForegroundColour

virtual void SetForegroundColour(const wxColour& *colour*)

Sets the foreground colour of the window.

Parameters

colour

The colour to be used as the foreground 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.

Note that when using this functions under GTK, you will disable the so called "themes", i.e. the user chosen apperance of windows and controls, including the themes of their parent windows.

See also

wxWindow::GetForegroundColour (p. 1196), *wxWindow::SetBackgroundColour* (p. 1221), *wxWindow::GetBackgroundColour* (p. 1194)

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. 1197), *Window identifiers* (p. 1368)

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. 1198)

wxWindow::SetPalette

virtual void SetPalette(wxPalette* *palette*)

Obsolete - use `wxDC::SetPalette` (p. 294) 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 `wxWindow::OnSize` (p. 1216) event handler function.

See also

Scrolling overview (p. 1386), *wxScrollBar* (p. 903), *wxScrolledWindow* (p. 911)

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. 1227), *wxWindow::GetScrollPos* (p. 1228), *wxWindow::GetScrollThumb* (p. 1199), *wxScrollBar* (p. 903), *wxScrolledWindow* (p. 911)

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. 922) object for setting the size.
- rect*
wxRect (p. 868) 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 *wxWindows*-supplied default width.
 - wxSIZE_AUTO_HEIGHT**: a -1 height value is taken to indicate a *wxWindows*-supplied default width.
 - wxSIZE_AUTO**: -1 size values are taken to indicate a *wxWindows*-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 *wxWindows*, or that the current value of the dimension should be used.

See also

wxWindow::Move (p. 1204)

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)

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).

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)

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.

Parameters

sizer

The sizer to set. Pass NULL to disassociate and delete the window's sizer.

Remarks

You must call *wxWindow::SetAutoLayout* (p. 1221) 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. 613), only the sizer will have effect.

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. 1201)

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::SetToolTip

void SetToolTip(const wxString& tip)

void SetToolTip(wxToolTip* tip)

Attach a tooltip to the window.

See also: *GetToolTip* (p. 1231), *wxToolTip* (p. 1133)

wxWindow::GetToolTip

wxToolTip* GetToolTip() const

Get the associated tooltip or NULL if none.

wxWindow::SetWindowStyle

void SetWindowStyle(long style)

Identical to *SetWindowStyleFlag* (p. 1232).

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. 1219) might be called after changing the others for the change to take place immediately.

See *Window styles* (p. 1371) for more information about flags.

See also

GetWindowStyleFlag (p. 1201)

wxWindow::Show

virtual bool Show(bool show)

Shows or hides the window.

Parameters

show

If TRUE, displays the window and brings it to the front. Otherwise, hides the window.

See also

wxWindow::IsShown (p. 1203)

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. 1233), *wxValidator* (p. 1166),
wxWindow::Validate (p. 1233)

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. 1232), *wxValidator* (p. 1166),
wxWindow::Validate (p. 1233)

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::TransferDataFromWindow (p. 1232), *wxWindow::TransferDataFromWindow*
(p. 1232), *wxValidator* (p. 1166)

wxWindow::WarpPointer

void WarpPointer(int x, int y)

Moves the pointer to the given position on the window.

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. 759) object.

To draw on the client area of a window from outside **OnPaint**, construct a `wxClientDC` (p. 120) object.

To draw on the whole window including decorations, construct a `wxWindowDC` (p. 1234) object (Windows only).

Derived from

`wxDC` (p. 280)

Include files

<wx/dcclient.h>

See also

`wxDC` (p. 280), `wxMemoryDC` (p. 678), `wxPaintDC` (p. 759), `wxClientDC` (p. 120), `wxScreenDC` (p. 901)

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. 87)

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

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 else but a sequence of 'pages' each of them displayed inside a dialog which has the buttons to pas 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 object of `wxWizard` class using `Create` (p. 1236) function. Then you should add all pages you want the wizard to show and call `RunWizard` (p. 1237). Finally, don't forget to call `wizard->Destroy()`.

Derived from

`wxDialog` (p. 310)
`wxPanel` (p. 764)
`wxWindow` (p. 1184)
`wxEvtHandler` (p. 378)
`wxObject` (p. 746)

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. 1238) argument. For some events, `Veto()` (p. 746) can be called to prevent the event from happening.

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).

See also

`wxWizardEvent` (p. 1238), `wxWizardPage` (p. 1239), `wxWizard sample` (p. 1328)

wxWizard::Create

```
static wxWizard* Create(wxWindow* parent, int id = -1, const wxString& title =  
wxEmptyString, const wxBitmap& bitmap = wxNullBitmap, const wxPoint& pos =  
wxDefaultPosition)
```

Creates the wizard dialog. The returned pointer should not be deleted directly, you should rather call `Destroy()` on it and `wxWindows` will delete it itself.

Notice that unlike almost all other `wxWindows` classes, there is no `size` parameter in `wxWizard` constructor because the wizard will have a predefined default size by default. If you want to change this, you should use the `SetPageSize` (p. 1237) 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. 1240).

pos

The position of the dialog, it will be centered on the screen by default.

`wxWizard::RunWizard`

`bool RunWizard(wxWizardPage* firstPage)`

Executes the wizard starting from the given page, returns `TRUE` if it was successfully finished or `FALSE` if user cancelled it. The *firstPage* can not be `NULL`.

`wxWizard::GetCurrentPage`

`wxWizardPage* GetCurrentPage() const`

Get the current page while the wizard is running. `NULL` is returned if `RunWizard()` (p. 1237) is not being executed now.

`wxWizard::GetPageSize`

`wxSize GetPageSize() const`

Returns the size available for the pages.

`wxWizard::SetPageSize`

`void SetPageSize(const wxSize& sizePage)`

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 layout all wizard pages using the sizers (even though the wizard is not resizable) and then use `wxSizer::CalcMin` (p. 926) in a loop to calculate the maximum of minimal sizes of the pages and pass it to `SetPageSize()`.

wxWizardEvent

`wxWizardEvent` class represents an event generated by the *wizard* (p. 1235): 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. 745)
`wxCommandEvent` (p. 152)
`wxEvent` (p. 375)
`wxObject` (p. 746)

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).

See also

`wxWizard` (p. 1235), `wxWizard sample` (p. 1328)

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.

wxWizardPage

wxWizardPage is one of the screens in *wxWizard* (p. 1235): 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 to decide what is 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. 1241) class is enough and it is simpler to use.

Virtual functions to override

To use this class, you must override *GetPrev* (p. 1240) and *GetNext* (p. 1240) pure virtual functions (or you may use *wxWizardPageSimple* (p. 1241) instead).

GetBitmap (p. 1240) can also be overridden, but this should be very rarely needed.

Derived from

wxPanel (p. 764)
wxWindow (p. 1184)
wxEvtHandler (p. 378)
wxObject (p. 746)

Include files

<wx/wizard.h>

See also

wxWizard (p. 1235), *wxWizard sample* (p. 1328)

wxWizardPage::wxWizardPage

wxWizardPage(wxWizard* parent, const wxBitmap& bitmap = wxNullBitmap)

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.

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. 1240)

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.

See also

GetPrev (p. 1240)

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. 1240).

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. 1239) 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. 1239) instead.

Derived from

`wxWizardPage` (p. 1239)

`wxPanel` (p. 764)

`wxWindow` (p. 1184)

`wxEvtHandler` (p. 378)

`wxObject` (p. 746)

Include files

<wx/wizard.h>

See also

`wxWizard` (p. 1235), `wxWizard` sample (p. 1328)

wxWizardPageSimple::wxWizardPageSimple

wxWizardPageSimple(`wxWizard*` parent = *NULL*, `wxWizardPage*` prev = *NULL*, `wxWizardPage*` next = *NULL*)

Constructor takes the previous and next pages. They may be modified later by `SetPrev()` (p. 1241) or `SetNext()` (p. 1242).

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);
```

wxZipInputStream

This class is input stream from ZIP archive. The archive must be local file (accessible via FILE*). It has all features including GetSize and seeking.

Note

If you need to enumerate files in ZIP archive, you can use *wxFileSystem* (p. 422) together with *wxZipFSHandler* (see *the overview* (p. 1362)).

Derived from

wxInputStream (p. 592)

Include files

<wx/zipstrm.h>

wxZipInputStream::wxZipInputStream

wxZipInputStream(const wxString& archive, const wxString& file)

Constructor.

Parameters

archive

name of ZIP file

file

name of file stored in the archive

wxZlibInputStream

This stream uncompresses all data read from it. It uses the "filtered" stream to get new compressed data.

Derived from

wxFilterInputStream (p. 432)

Include files

<wx/zstream.h>

See also

wxInputStream (p. 592)

wxZlibOutputStream

This stream compresses all data written to it, and passes the compressed data to the "filtered" stream.

Derived from

wxFilterOutputStream (p. 432)

Include files

<wx/zstream.h>

See also

wxOutputStream (p. 751)

wxZlibOutputStream::wxZlibOutputStream

wxZlibOutputStream(wxOutputStream& *stream*,int *level* = -1)

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).

Chapter 6 Functions

The functions and macros defined in wxWindows are described here.

Version macros

The following constants are defined in wxWindows:

- wxMAJOR_VERSION is the major version of wxWindows
- wxMINOR_VERSION is the minor version of wxWindows
- wxRELEASE_NUMBER is the release number

For example, the values of these constants for wxWindows 2.1.15 are 2, 1 and 15.

Additionally, wxVERSION_STRING is a user-readable string containing the full wxWindows 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 wxWindows 2.2.

Include files

<wx/version.h> or <wx/defs.h>

wxCHECK_VERSION

bool wxCHECK_VERSION(*major, minor, release*)

This is a macro which evaluates to true if the current wxWindows version is at least major.minor.release.

For example, to test if the program is compiled with wxWindows 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 ( strcmp(s, "foo", 3) == 0 )  
#endif  
    {  
        ...  
    }
```

Thread functions

Include files

<wx/thread.h>

See also

wxThread (p. 1101), *wxMutex* (p. 730), *Multithreading overview* (p. 1420)

::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. 1246).

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. 1246).

This function is only defined on platforms which support preemptive threads.

File functions

Include files

<wx/utils.h>

See also

wxPathList (p. 769), *wxDir* (p. 323), *wxFile* (p. 395)

::wxDirExists

bool wxDirExists(const wxString& *dirname*)

Returns TRUE if the directory exists.

::wxDos2UnixFilename

void Dos2UnixFilename(const wxString& *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. It also returns TRUE if the file is a directory.

::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*)

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. 1248) to get the next matching file. Neither will report the current directory "." or the parent directory "..".

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.IsEmpty() )
{
    ...
    f = wxFindNextFile();
}
```

::wxFindNextFile

wxString wxFindNextFile()

Returns the next file that matches the path passed to *wxFindFirstFile* (p. 1248).

See *wxFindFirstFile* (p. 1248) for an example.

::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.

::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.

::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)

Copies *file1* to *file2*, returning TRUE if successful.

::wxGetCwd

wxString wxGetCwd()

Returns a string containing the current (or working) directory.

::wxGetWorkingDirectory

wxString wxGetWorkingDirectory(char* buf=NULL, int sz=1000)

This function is obsolete: use *wxGetCwd* (p. 1249) 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)

Makes a temporary filename based on *prefix*, opens and closes the file, and places the name in *buf*. If *buf* is NULL, new store is allocated for the temporary filename using *new*.

Under Windows, the filename will include the drive and name of the directory allocated for temporary files (usually the contents of the TEMP variable). Under Unix, the /tmp directory is used.

It is the application's responsibility to create and delete the file.

::wxIsWild

bool wxIsWild(const wxString& pattern)

Returns TRUE if the pattern contains wildcards. See *wxMatchWild* (p. 1250).

::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. 1250).

::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.

::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)

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).

Use of this function requires the file `wx_doc.h` to be included.

::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).

Use of this function requires the file `wx_doc.h` to be included.

Network functions

::wxGetFullHostName

wxString wxGetFullHostName()

Returns the FQDN (fully qualified domain host name) or an empty string on error.

See also

wxGetHostName (p. 1252)

Include files

<wx/utils.h>

::wxGetEmailAddress

bool wxGetEmailAddress(const wxString& *buf*, int *sz*)

Copies the user's email address into the supplied buffer, by concatenating the values returned by *wxGetFullHostName* (p. 1252) and *wxGetUserId* (p. 1253).

Returns TRUE if successful, FALSE otherwise.

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 **wxWindows** 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. 1252)

Include files

<wx/utils.h>

User identification

::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 **wxWindows** 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. 1254)

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 **wxWindows** 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. 1253)

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.

::wxStringMatch

**bool wxStringMatch(const wxString& s1, const wxString& s2,
bool subString = TRUE, bool exact = FALSE)**

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)

A macro defined as:

```
#define wxStringEq(s1, s2) (s1 && s2 && (strcmp(s1, s2) == 0))
```

::IsEmpty

bool IsEmpty(const char * p)

Returns TRUE if the string is empty, FALSE otherwise. It is safe to pass NULL pointer to this function and it will return TRUE for it.

::Stricmp

int Stricmp(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.

::Strlen

size_t Strlen(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.

::wxGetTranslation

const char * wxGetTranslation(const char * str)

This function returns the translation of string *str* in the current *locale* (p. 648). If the string is not found in any of the loaded message catalogs (see *internationalization overview* (p. 1346)), 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 syntax is provided: the *_()* macro is defined as *wxGetTranslation()*.

::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. 1256), `wxString::Printf` (p. 1022)

::wxVsnprintf

int wxVsnprintf(wxChar *buf, size_t len, const wxChar *format, va_list argptr)

The same as `wxSnprintf` (p. 1255) but takes a `va_list` argument instead of arbitrary number of parameters.

See also

`wxSnprintf` (p. 1255), `wxString::PrintV` (p. 1023)

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.

::wxCreateFileTipProvider

wxTipProvider * wxCreateFileTipProvider(const wxString& filename, size_t currentTip)

This function creates a `wxTipProvider` (p. 1116) which may be used with `wxShowTip` (p. 1261).

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. 1418)

Include files

<wx/tipdlg.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 = "",
    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, wxHIDE_READONLY, 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:

```
const wxString& s = wxFileSelector("Choose a file to open");
if (s)
{
    ...
}
```

Include files

<wx/filedlg.h>

::wxGetColourFromUser

```
wxColour wxGetColourFromUser(wxWindow *parent, const wxColour& collnit)
```

Shows the colour selection dialog and returns the colour selected by user or invalid colour (use `wxColour::Ok` (p. 137) to test whether a colour is valid) if the dialog was cancelled.

Parameters

parent

The parent window for the colour selection dialog

collnit

If given, this will be the colour initially selected in the dialog.

Include files

<wx/colordlg.h>

::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 dialog's 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/textdlg.h>

::wxGetPasswordFromUser

wxString wxGetTextFromUser(const wxString& message, const wxString& caption = "Input text", const wxString& default_value = "", wxWindow *parent = NULL)

Similar to `wxGetTextFromUser` (p. 1259) 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 = -1, int y = -1, 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,
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.

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>
```

::wxGetSingleChoiceIndex

```
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>
```

::wxGetSingleChoiceData

```
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.

Include files

```
<wx/choicdlg.h>
```

::wxMessageBox

```
int wxMessageBox(const wxString& message, const wxString& caption =
"Message", int style = wxOK | wxCENTRE,
wxWindow *parent = NULL, int x = -1, int y = -1)
```

General purpose message dialog. *style* may be a bit list of the following identifiers:

<code>wxYES_NO</code>	Puts Yes and No buttons on the message box. May be combined with <code>wxCANCEL</code> .
<code>wxCANCEL</code>	Puts a Cancel button on the message box. May be combined with <code>wxYES_NO</code> or <code>wxOK</code> .
<code>wxOK</code>	Puts an Ok button on the message box. May be combined with <code>wxCANCEL</code> .
<code>wxCENTRE</code>	Centres the text.
<code>wxICON_EXCLAMATION</code>	Displays an exclamation mark symbol.
<code>wxICON_HAND</code>	Displays a hand symbol.
<code>wxICON_QUESTION</code>	Displays a question mark symbol.
<code>wxICON_INFORMATION</code>	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)
    delete main_frame;
...
```

message may contain newline characters, in which case the message will be split into separate lines, to cater for large messages.

Under Windows, the native `MessageBox` function is used unless `wxCENTRE` is specified in the style, in which case a generic function is used. This is because the native `MessageBox` function cannot centre text. The symbols are not shown when the generic function is used.

Include files

<wx/msgdlg.h>

::wxShowTip

```
bool wxShowTip(wxWindow *parent, wxTipProvider *tipProvider, bool showAtStartup
= TRUE)
```

This function shows a "startup tip" to the user.

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. 1256) 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. 1418)

Include files

<wx/tipdlg.h>

GDI functions

The following are relevant to the GDI (Graphics Device Interface).

Include files

<wx/gdicmn.h>

::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).

::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. 712)) 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 *wxWindows* 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. 184), *wxWindow::SetCursor* (p. 1223).

Printer settings

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. 121) 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

::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. 366).

Include files

<wx/dnd.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>

::wxBeginBusyCursor

void wxBeginBusyCursor(wxCursor *cursor = wxHOURLASS_CURSOR)

Changes the cursor to the given cursor for all windows in the application. Use *wxEndBusyCursor* (p. 1272) 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. 1279), *wxBusyCursor* (p. 87).

Include files

<wx/utils.h>

::wxBell

void wxBell()

Ring the system bell.

Include files

<wx/utils.h>

::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.

::wxDDECleanUp

void wxDDECleanUp()

Called when wxWindows exits, to clean up the DDE system. This no longer needs to be called by the application.

See also *wxDDEInitialize* (p. 1270).

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 wxWindows if necessary.

See also *wxDDEServer* (p. 303), *wxDDEClient* (p. 298), *wxDDEConnection* (p. 299), *wxDDECleanUp* (p. 1270).

Include files

<wx/dde.h>

::wxDebugMsg

void wxDebugMsg(const wxString& fmt, ...)

This function is deprecated, use *wxLogDebug* (p. 1299) instead!

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.

This function is now obsolete, replaced by *Log functions* (p. 1297).

Include files

<wx/utils.h>

::wxDisplaySize

void wxDisplaySize(int *width, int *height)

Gets the physical size of the display in pixels.

Include files

<wx/gdicmn.h>

::wxEnableTopLevelWindows

void wxEnableTopLevelWindow(bool enable = TRUE)

This function enables or disables all top level windows. It is used by `::wxSafeYield` (p. 1281).

Include files

<wx/utils.h>

::wxEntry

This initializes `wxWindows` in a platform-dependent way. Use this if you are not using the default `wxWindows` entry code (e.g. `main` or `WinMain`). For example, you can initialize `wxWindows` from an Microsoft Foundation Classes application using this function.

void wxEntry(HANDLE hInstance, HANDLE hPrevInstance, const wxString& commandLine, int cmdShow, bool enterLoop = TRUE)

`wxWindows` initialization under Windows (non-DLL). If `enterLoop` is `FALSE`, the function will return immediately after calling `wxApp::OnInit`. Otherwise, the `wxWindows` message loop will be entered.

```
void wxEntry(HANDLE hInstance, HANDLE hPrevInstance, WORD wDataSegment,
WORD wHeapSize, const wxString& commandLine)
```

wxWindows initialization under Windows (for applications constructed as a DLL).

```
int wxEntry(int argc, const wxString& *argv)
```

wxWindows initialization under Unix.

Remarks

To clean up wxWindows, call `wxApp::OnExit` followed by the static function `wxApp::CleanUp`. For example, if exiting from an MFC application that also uses wxWindows:

```
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>
```

::wxEndBusyCursor

```
void wxEndBusyCursor()
```

Changes the cursor back to the original cursor, for all windows in the application. Use with `wxBeginBusyCursor` (p. 1269).

See also `wxIsBusy` (p. 1279), `wxBusyCursor` (p. 87).

Include files

```
<wx/utils.h>
```

::wxError

```
void wxError(const wxString& msg, const wxString& title = "wxWindows Internal Error")
```

Displays *msg* and continues. This writes to standard error under Unix, and pops up a message box under Windows. Used for internal wxWindows errors. See also `wxFatalError` (p. 1274).

Include files

```
<wx/utils.h>
```

::wxExecute

```
long wxExecute(const wxString& command, bool sync = FALSE, wxProcess  
*callback = NULL)
```

```
long wxExecute(char **argv, bool sync = FALSE, wxProcess *callback = NULL)
```

```
long wxExecute(const wxString& command, wxArrayString& output)
```

```
long wxExecute(const wxString& command, wxArrayString& output,  
wxArrayString& errors)
```

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 *sync* is FALSE (the default), flow of control immediately returns. If TRUE, 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. 1285). The caller should ensure that this can cause no recursion, in the simplest case by calling *wxEnableTopLevelWindows(FALSE)* (p. 1271).

For asynchronous execution, however, the return value is the process id and zero value indicates that the command could not be executed.

If *callback* isn't NULL and if execution is asynchronous (note that *callback* parameter can not be non-NULL for synchronous execution), *wxProcess::OnTerminate* (p. 817) will be called when the process finishes.

Finally, you may use the third overloaded version of this function to execute a process (always synchronously) 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.

See also *wxShell* (p. 1282), *wxProcess* (p. 815), *Exec sample* (p. 1323).

Include files

<wx/utils.h>

::wxExit

void wxExit()

Exits application after calling *wxApp::OnExit* (p. 26). 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 *wxWindow::OnCloseWindow* (p. 1208) and *wxApp* (p. 21).

Include files

<wx/app.h>

::wxFatalError

void wxFatalError(const wxString& msg, const wxString& title = "wxWindows Fatal Error")

Displays *msg* and exits. This writes to standard error under Unix, and pops up a message box under Windows. Used for fatal internal wxWindows errors. See also *wxError* (p. 1272).

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)

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)

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>

::wxGetActiveWindow

wxWindow * wxGetActiveWindow()

Gets the currently active window (Windows only).

Include files

<wx/windows.h>

::wxGetDisplayName

wxString wxGetDisplayName()

Under X only, returns the current display name. See also *wxSetDisplayName* (p. 1282).

Include files

<wx/utils.h>

::wxGetHomeDir

wxString wxGetHomeDir()

Return the (current) user's home directory.

See also

wxGetUserHome (p. 1278)

Include files

<wx/utils.h>

::wxGetFreeMemory

long wxGetFreeMemory()

Returns the amount of free memory in bytes under environments which support it, and -1 if not supported. Currently, it is supported only under Windows, Linux and Solaris.

Include files

<wx/utils.h>

::wxGetMousePosition

void wxGetMousePosition(int* x, int* y)

Returns the mouse position in screen coordinates.

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. 1277)

Include files

<wx/utils.h>

::wxGetOsVersion

int wxGetOsVersion(int *major = NULL, int *minor = NULL)

Gets operating system version information.

Platform	Return types
Macintosh	Return value is wxMACINTOSH.
GTK	Return value is wxGTK, For GTK 1.0, <i>major</i> is 1, <i>minor</i> is 0.
Motif	Return value is wxMOTIF_X, <i>major</i> is X version, <i>minor</i> is X revision.
OS/2	Return value is wxOS2_PM.
Windows 3.1	Return value is wxWINDOWS, <i>major</i> is 3, <i>minor</i> is 1.
Windows NT/2000	Return value is wxWINDOWS_NT, version is returned in <i>major</i> and <i>minor</i>
Windows 98	Return value is wxWIN95, <i>major</i> is 4, <i>minor</i> is 1 or greater.
Windows 95	Return value is wxWIN95, <i>major</i> is 4, <i>minor</i> is 0.
Win32s (Windows 3.1)	Return value is wxWIN32S, <i>major</i> is 3, <i>minor</i> is 1.
Watcom C++ 386 supervisor mode (Windows 3.1)	Return value is wxWIN386, <i>major</i> is 3, <i>minor</i> is 1.

See also

::wxGetOsDescription (p. 1276)

Include files

<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. 1284), `wxConfigBase` (p. 162).

Include files

```
<wx/utils.h>
```

::wxGetUserId

```
bool wxGetUserId(const wxString& buf, int bufSize)
```

Copies the user's login identity (such as "jacs") into the buffer *buf*, of maximum size *bufSize*, returning TRUE if successful. Under Windows, this returns "user".

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. 1275).

Include files

```
<wx/utils.h>
```

::wxGetUserName

```
bool wxGetUserName(const wxString& buf, int bufSize)
```

Copies the user's name (such as "Julian Smart") into the buffer *buf*, of maximum size *bufSize*, returning TRUE if successful. Under Windows, this returns "unknown".

Include files

<wx/utils.h>

::wxHandleFatalExceptions

bool wxHandleFatalExceptions(bool *dolt* = TRUE)

If *dolt* 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. 26). 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 *dolt* equal to FALSE will restore this default behaviour.

::wxKill

int wxKill(long *pid*, int *sig*)

Under Unix (the only supported platform), equivalent to the Unix kill function. Returns 0 on success, -1 on failure.

Tip: sending a signal of 0 to a process returns -1 if the process does not exist. It does not raise a signal in the receiving process.

Include files

<wx/utils.h>

::wxInitAllImageHandlers

void wxInitAllImageHandlers()

Initializes all available image handlers. For a list of available handlers, see *wxImage* (p. 565).

See also

wxImage (p. 565), *wxImageHandler* (p. 580)

::wxIsBusy

bool wxIsBusy()

Returns TRUE if between two *wxBeginBusyCursor* (p. 1269) and *wxEndBusyCursor* (p. 1272) calls.

See also *wxBusyCursor* (p. 87).

Include files

<wx/utils.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.

One use of this is to store `.wxr` files instead of including the data in the C++ file; some compilers cannot cope with the long strings in a `.wxr` file. The resource data can then be parsed using *wxResourceParseString* (p. 1296).

This function is available under Windows only.

Include files

<wx/utils.h>

::wxNow

wxString wxNow()

Returns a string representing the current date and time.

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. 1190) instead.

Include files

<wx/utils.h>

::wxPostEvent

void wxPostEvent(wxEvtHandler *dest, wxEvent& event)

This function posts the event to the specified *dest* object. The difference between sending an event and posting it is that in the first case the event is processed before the function returns (in `wxWindows`, event sending is done with `ProcessEvent` (p. 382) function), but in the second, the function returns immediately and the event will be processed sometime later - usually during the next even loop iteration.

Note that a copy of the *event* is made by the function, so the original copy can be deleted as soon as function returns. This function can also be used to send events between different threads safely. As this function makes a copy of the event, the event needs to have a fully implemented `Clone()` method, which may not be the case for all event in `wxWindows`.

See also `AddPendingEvent` (p. 379) (which this function uses internally).

Include files

<wx/app.h>

::wxSafeYield

bool wxSafeYield(wxWindow* win = NULL)

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. 1285).

Include files

<wx/utils.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. 1275).

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. 1273), *Exec sample* (p. 1323).

Include files

<wx/utils.h>

::wxSleep

void wxSleep(int *secs*)

Sleeps for the specified number of seconds.

Include files

<wx/utils.h>

::wxStripMenuCodes

wxString wxStripMenuCodes(const wxString& *in*)

void wxStripMenuCodes(char* *in*, char* *out*)

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 shortcut in Windows and Motif) and \t (tab in Windows).

Include files

<wx/utils.h>

::wxToLower

char wxToLower(char *ch*)

Converts the character to lower case. This is implemented as a macro for efficiency.

Include files

<wx/utils.h>

::wxToUpper

char wxToUpper(char *ch*)

Converts the character to upper case. This is implemented as a macro for efficiency.

Include files

<wx/utils.h>

::wxTrace

void wxTrace(const wxString& *fmt*, ...)

Takes printf-style variable argument syntax. Output is directed to the current output stream (see *wxDebugContext* (p. 1357)).

This function is now obsolete, replaced by *Log functions* (p. 1297).

Include files

<wx/memory.h>

::wxTraceLevel

```
void wxTraceLevel(int level, const wxString& fmt, ...)
```

Takes printf-style variable argument syntax. Output is directed to the current output stream (see *wxDebugContext* (p. 1357)). 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.

This function is now obsolete, replaced by *Log functions* (p. 1297).

Include files

```
<wx/memory.h>
```

::wxUsleep

```
void wxUsleep(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>
```

::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. 1277), *wxConfigBase* (p. 162).

Include files

<wx/utils.h>

::wxYield

bool wxYield()

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. 1281) for a better function.

Note that *wxYield* will not flush the message logs. This is intentional as calling *wxYield* 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. 655).

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. 1281) and *wxEvtHandler::AddPendingEvent* (p. 379).

Include files

<wx/app.h>

Macros

These macros are defined in wxWindows.

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)

This macro will swap the bytes of the *value* variable from little endian to big endian or vice versa.

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 (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.

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 function implemented by IMPLEMENT_APP. 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`.

Include files

```
<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 `wxWindows` for dynamic construction. You use this instead of

Old form:

```
MyApp myApp;
```

New form:

```
IMPLEMENT_APP(MyApp)
```

See also *DECLARE_APP* (p. 1287).

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>

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 `#ifdefs` when creating bitmaps.

See also

Bitmaps and icons overview (p. 1388), *wxICON* (p. 1292)

Include files

<wx/gdicmn.h>

wxConstCast

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

wxDynamicCast (p. 1291)

wxStaticCast (p. 1292)

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

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) or NULL 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. 1329)

wxConstCast (p. 1291)

wxStaticCast (p. 1292)

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 `#ifdefs` when creating icons.

See also

Bitmaps and icons overview (p. 1388), *wxBITMAP* (p. 1291)

Include files

<wx/gdicmn.h>

wxStaticCast**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)`.

wxDynamicCast (p. 1291)
wxConstCast (p. 1291)

WXTRACE

Include files

<wx/object.h>

WXTRACE(formatString, ...)

Calls `wxTrace` with printf-style variable argument syntax. Output is directed to the current output stream (see *wxDebugContext* (p. 1357)).

This macro is now obsolete, replaced by *Log functions* (p. 1297).

Include files

<wx/memory.h>

WXTRACELEVEL

WXTRACELEVEL(level, formatString, ...)

Calls `wxTraceLevel` with printf-style variable argument syntax. Output is directed to the current output stream (see *wxDebugContext* (p. 1357)). 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.

This function is now obsolete, replaced by *Log functions* (p. 1297).

Include files

<wx/memory.h>

wxWindows resource functions

wxWindows resource system (p. 1379)

This section details functions for manipulating wxWindows (.WXR) resource files and loading user interface elements from resources.

Please note that this use of the word 'resource' is different from that used when talking

about initialisation file resource reading and writing, using such functions as `wxWriteResource` and `wxGetResource`. It is just an unfortunate clash of terminology.

For an overview of the `wxWindows` resource mechanism, see *the wxWindows resource system* (p. 1379).

See also `wxWindow::LoadFromResource` (p. 1203) for loading from resource data.

::wxResourceAddIdentifier

bool wxResourceAddIdentifier(const wxString& name, int value)

Used for associating a name with an integer identifier (equivalent to dynamically #defining a name to an integer). Unlikely to be used by an application except perhaps for implementing resource functionality for interpreted languages.

::wxResourceClear

void wxResourceClear()

Clears the `wxWindows` resource table.

::wxResourceCreateBitmap

wxBitmap * wxResourceCreateBitmap(const wxString& resource)

Creates a new bitmap from a file, static data, or Windows resource, given a valid `wxWindows` bitmap resource identifier. For example, if the `.WXR` file contains the following:

```
static const wxString& project_resource = "bitmap(name =
'project_resource',\
  bitmap = ['project', wxBITMAP_TYPE_BMP_RESOURCE, 'WINDOWS'],\
  bitmap = ['project.xpm', wxBITMAP_TYPE_XPM, 'X']).";
```

then this function can be called as follows:

```
wxBitmap *bitmap = wxResourceCreateBitmap("project_resource");
```

::wxResourceCreateIcon

wxIcon * wxResourceCreateIcon(const wxString& resource)

Creates a new icon from a file, static data, or Windows resource, given a valid `wxWindows` icon resource identifier. For example, if the `.WXR` file contains the following:

```
static const wxString& project_resource = "icon(name =
```

```
'project_resource',\
  icon = ['project', wxBITMAP_TYPE_ICO_RESOURCE, 'WINDOWS'],\
  icon = ['project', wxBITMAP_TYPE_XBM_DATA, 'X'])).";
```

then this function can be called as follows:

```
wxIcon *icon = wxResourceCreateIcon("project_resource");
```

::wxResourceCreateMenuBar

wxMenuBar * wxResourceCreateMenuBar(const wxString& resource)

Creates a new menu bar given a valid wxWindows menubar resource identifier. For example, if the .WXR file contains the following:

```
static const wxString& menuBar11 = "menu(name = 'menuBar11',\
  menu = \
  [\
    ['&File', 1, '', \
      ['&Open File', 2, 'Open a file'],\
      ['&Save File', 3, 'Save a file'],\
      [],\
      ['E&xit', 4, 'Exit program']\
    ],\
    ['&Help', 5, '', \
      ['&About', 6, 'About this program']\
    ]\
  ]).\
);
```

then this function can be called as follows:

```
wxMenuBar *menuBar = wxResourceCreateMenuBar("menuBar11");
```

::wxResourceGetIdentifier

int wxResourceGetIdentifier(const wxString& name)

Used for retrieving the integer value associated with an identifier. A zero value indicates that the identifier was not found.

See *wxResourceAddIdentifier* (p. 1294).

::wxResourceParseData

bool wxResourceParseData(const wxString& resource, wxResourceTable *table = NULL)

Parses a string containing one or more wxWindows resource objects. If the resource objects are global static data that are included into the C++ program, then this function must be called for each variable containing the resource data, to make it known to wxWindows.

resource should contain data in the following form:

```
dialog(name = 'dialog1',
       style = 'wxCAPTION | wxDEFAULT_DIALOG_STYLE',
       title = 'Test dialog box',
       x = 312, y = 234, width = 400, height = 300,
       modal = 0,
       control = [1000, wxStaticBox, 'Groupbox', '0', 'group6', 5, 4, 380,
                 262,
                 [11, 'wxSWISS', 'wxNORMAL', 'wxNORMAL', 0]],
       control = [1001, wxTextCtrl, '', 'wxTE_MULTILINE', 'text3',
                 156, 126, 200, 70, 'wxWindows is a multi-platform, GUI toolkit.',
                 [11, 'wxSWISS', 'wxNORMAL', 'wxNORMAL', 0],
                 [11, 'wxSWISS', 'wxNORMAL', 'wxNORMAL', 0]]).
```

This function will typically be used after including a `.wxr` file into a C++ program as follows:

```
#include "dialog1.wxr"
```

Each of the contained resources will declare a new C++ variable, and each of these variables should be passed to `wxResourceParseData`.

::wxResourceParseFile

```
bool wxResourceParseFile(const wxString& filename, wxResourceTable *table = NULL)
```

Parses a file containing one or more `wxWindows` resource objects in C++-compatible syntax. Use this function to dynamically load `wxWindows` resource data.

::wxResourceParseString

```
bool wxResourceParseString(char* s, wxResourceTable *table = NULL)
```

Parses a string containing one or more `wxWindows` resource objects. If the resource objects are global static data that are included into the C++ program, then this function must be called for each variable containing the resource data, to make it known to `wxWindows`.

resource should contain data with the following form:

```
dialog(name = 'dialog1',
       style = 'wxCAPTION | wxDEFAULT_DIALOG_STYLE',
       title = 'Test dialog box',
       x = 312, y = 234, width = 400, height = 300,
       modal = 0,
       control = [1000, wxStaticBox, 'Groupbox', '0', 'group6', 5, 4, 380,
                 262,
                 [11, 'wxSWISS', 'wxNORMAL', 'wxNORMAL', 0]],
       control = [1001, wxTextCtrl, '', 'wxTE_MULTILINE', 'text3',
                 156, 126, 200, 70, 'wxWindows is a multi-platform, GUI toolkit.',
```

```
[11, 'wxSWISS', 'wxNORMAL', 'wxNORMAL', 0],
[11, 'wxSWISS', 'wxNORMAL', 'wxNORMAL', 0]]).
```

This function will typically be used after calling *wxLoadUserResource* (p. 1280) to load an entire *.wxr* file into a string.

::wxResourceRegisterBitmapData

bool wxResourceRegisterBitmapData(const wxString& name, char* xbm_data, int width, int height, wxResourceTable *table = NULL)

bool wxResourceRegisterBitmapData(const wxString& name, char xpm_data)**

Makes #included XBM or XPM bitmap data known to the wxWindows resource system. This is required if other resources will use the bitmap data, since otherwise there is no connection between names used in resources, and the global bitmap data.

::wxResourceRegisterIconData

Another name for *wxResourceRegisterBitmapData* (p. 1297).

Log functions

These functions provide a variety of logging functions: see *Log classes overview* (p. 1353) 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 wxWindows in the beginning of the program).

Include files

<wx/log.h>

::wxLogError

void wxLogError(const char* formatString, ...)

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.

::wxLogFatalError

void wxLogFatalError(const char* formatString, ...)

Like `wxLogError` (p. 1297), 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, ...)

For warnings - they are also normally shown to the user, but don't interrupt the program work.

::wxLogMessage

void wxLogMessage(const char* formatString, ...)

for all normal, informational messages. They also appear in a message box by default (but it can be changed). Notice that the standard behaviour is to not show informational messages if there are any errors later - the logic being that the later error messages make the informational messages preceding them meaningless.

::wxLogVerbose

void wxLogVerbose(const char* formatString, ...)

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 wxLogStatus(const char* formatString, ...)

Messages logged by this function 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 function).

If the target frame doesn't have a statusbar, the message will be lost.

::wxLogSysError

void wxLogSysError(const char* formatString, ...)

Mostly used by `wxWindows` 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. 1300), *wxSysErrorMsg* (p. 1300)

::wxLogDebug

void wxLogDebug(const char* *formatString*, ...)

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).

::wxLogTrace

void wxLogTrace(const char* *formatString*, ...)

void wxLogTrace(const char **mask*, const char **formatString*, ...)

void wxLogTrace(wxTraceMask *mask*, const char **formatString*, ...)

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. 651). 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. 654). The predefined string trace masks used by *wxWindows* 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)

The third version of the function only logs the message if all the bit corresponding to the *mask* are set in the *wxLog* trace mask which can be set by *SetTraceMask* (p. 656). 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)

::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. 1300), `wxLogSysError` (p. 1298)

::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. 1300)) is used.

See also

`wxSysErrorCode` (p. 1300), `wxLogSysError` (p. 1298)

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. 1113) and/or `wxStopWatch` (p. 997) classes should be used instead. For retrieving the current time, you may also use `wxDateTime::Now` (p. 225) or `wxDateTime::UNow` (p. 226) methods.

::wxGetElapsedTime

long wxGetElapsedTime(bool *resetTimer* = TRUE)

Gets the time in milliseconds since the last `::wxStartTimer` (p. 1302).

If *resetTimer* is TRUE (the default), the timer is reset to zero by this call.

See also *wxTimer* (p. 1113).

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. 225)

Include files

<wx/timer.h>

::wxGetLocalTimeMillis

wxLongLone wxGetLocalTimeMillis()

Returns the number of milliseconds since local time 00:00:00 Jan 1st 1970.

See also

wxDateTime::Now (p. 225),
wxLongLone (p. 656)

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. 225)

Include files

<wx/timer.h>

::wxStartTimer

void wxStartTimer()

Starts a stopwatch; use `::wxGetElapsedTime` (p. 1300) to get the elapsed time.

See also `wxTimer` (p. 1113).

Include files

<wx/timer.h>

Debugging macros and functions

Useful macros and functions for error checking and defensive programming. ASSERTs are only compiled if `__WXDEBUG__` is defined, whereas CHECK macros stay in release builds.

Include files

<wx/debug.h>

::wxOnAssert

void wxOnAssert(const char* fileName, int lineNumber, const char* msg = NULL)

This function may be redefined to do something non trivial and is called whenever one of debugging macros fails (i.e. condition is false in an assertion).

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. 1303)

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. 1302)

wxFAIL

wxFAIL()

Will always generate an assert error if this code is reached (in debug mode).

See also: *wxFAIL_MSG* (p. 1303)

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. 1303)

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. 1304).

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. 1303).

wxCHECK2

wxCHECK2(*condition, operation*)

Checks that the condition is true and *wxFail* (p. 1303) and execute *operation* if it is not. This is a generalisation of *wxCHECK* (p. 1303) 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. 1304), but *wxFail_MSG* (p. 1303) with the specified *msg* is called instead of *wxFail*() if the *condition* is false.

Keycodes

Keypresses are represented by an enumerated type, *wxKeyCode*. The possible values are the ASCII character codes, plus the following:

WXX_BACK	= 8
WXX_TAB	= 9
WXX_RETURN	= 13
WXX_ESCAPE	= 27
WXX_SPACE	= 32
WXX_DELETE	= 127
WXX_START	= 300
WXX_LBUTTON	
WXX_RBUTTON	
WXX_CANCEL	
WXX_MBUTTON	
WXX_CLEAR	
WXX_SHIFT	
WXX_CONTROL	
WXX_MENU	

WXX_PAUSE
WXX_CAPITAL
WXX_PRIOR
WXX_NEXT
WXX_END
WXX_HOME
WXX_LEFT
WXX_UP
WXX_RIGHT
WXX_DOWN
WXX_SELECT
WXX_PRINT
WXX_EXECUTE
WXX_SNAPSHOT
WXX_INSERT
WXX_HELP
WXX_NUMPAD0
WXX_NUMPAD1
WXX_NUMPAD2
WXX_NUMPAD3
WXX_NUMPAD4
WXX_NUMPAD5
WXX_NUMPAD6
WXX_NUMPAD7
WXX_NUMPAD8
WXX_NUMPAD9
WXX_MULTIPLY
WXX_ADD
WXX_SEPARATOR
WXX_SUBTRACT
WXX_DECIMAL
WXX_DIVIDE
WXX_F1
WXX_F2
WXX_F3
WXX_F4
WXX_F5
WXX_F6
WXX_F7
WXX_F8
WXX_F9
WXX_F10
WXX_F11
WXX_F12
WXX_F13
WXX_F14
WXX_F15
WXX_F16
WXX_F17
WXX_F18
WXX_F19
WXX_F20
WXX_F21
WXX_F22
WXX_F23
WXX_F24
WXX_NUMLOCK
WXX_SCROLL

Chapter 7 Classes by category

A classification of wxWindows 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 may contain windows, and dialog boxes may directly contain controls.

<i>wxDialog</i> (p. 310)	Dialog box
<i>wxFrame</i> (p. 452)	Normal frame
<i>wxMDIParentFrame</i> (p. 671)	MDI parent frame
<i>wxMDIChildFrame</i> (p. 666)	MDI child frame
<i>wxMiniFrame</i> (p. 716)	A frame with a small title bar
<i>wxWizard</i> (p. 1235)	A wizard dialog.
<i>wxTabbedDialog</i> (p. 1038)	Tabbed dialog (deprecated, use <i>wxNotebook</i> instead)

See also **Common dialogs**.

Miscellaneous windows

The following are a variety of classes that are derived from *wxWindow*.

<i>wxPanel</i> (p. 764)	A window whose colour changes according to current user settings
<i>wxScrolledWindow</i> (p. 911)	Window with automatically managed scrollbars
<i>wxGrid</i> (p. 479)	A grid (table) window
<i>wxSplitterWindow</i> (p. 973)	Window which can be split vertically or horizontally
<i>wxStatusBar</i> (p. 991)	Implements the status bar on a frame
<i>wxToolBar</i> (p. 1117)	Toolbar class
<i>wxNotebook</i> (p. 736)	Notebook class
<i>wxPlotWindow</i> (p. 782)	A class to display data.
<i>wxSashWindow</i> (p. 897)	Window with four optional sashes that can be dragged
<i>wxSashLayoutWindow</i> (p. 894)	Window that can be involved in an IDE-like layout arrangement
<i>wxWizardPage</i> (p. 1239)	A base class for the page in wizard dialog.
<i>wxWizardPageSimple</i> (p. 1241)	A page in wizard dialog.

Common dialogs

Overview (p. 1398)

Common dialogs are ready-made dialog classes which are frequently used in an application.

<i>wxDialog</i> (p. 310)	Base class for common dialogs
<i>wxColourDialog</i> (p. 142)	Colour chooser dialog
<i>wxDirDialog</i> (p. 325)	Directory selector dialog
<i>wxFileDialog</i> (p. 407)	File selector dialog
<i>wxMultipleChoiceDialog</i> (p. 730)	Dialog to get one or more selections from a list
<i>wxSingleChoiceDialog</i> (p. 919)	Dialog to get a single selection from a list and return the string
<i>wxTextEntryDialog</i> (p. 1089)	Dialog to get a single line of text from the user
<i>wxFontDialog</i> (p. 444)	Font chooser dialog
<i>wxPageSetupDialog</i> (p. 758)	Standard page setup dialog
<i>wxPrintDialog</i> (p. 797)	Standard print dialog
<i>wxPageSetupDialog</i> (p. 758)	Standard page setup dialog
<i>wxMessageDialog</i> (p. 709)	Simple message box dialog
<i>wxWizard</i> (p. 1235)	A wizard dialog.

Controls

Typically, these are small windows which provide interaction with the user. Controls that are not static can have *validators* (p. 1166) associated with them.

<i>wxControl</i> (p. 176)	The base class for controls
<i>wxButton</i> (p. 89)	Push button control, displaying text
<i>wxBitmapButton</i> (p. 70)	Push button control, displaying a bitmap
<i>wxCalendarCtrl</i> (p. 95)	Date picker control
<i>wxCheckBox</i> (p. 108)	Checkbox control
<i>wxCheckListBox</i> (p. 111)	A listbox with a checkbox to the left of each item
<i>wxChoice</i> (p. 113)	Choice control (a combobox without the editable area)
<i>wxComboBox</i> (p. 143)	A choice with an editable area
<i>wxGauge</i> (p. 470)	A control to represent a varying quantity, such as time remaining
<i>wxStaticBox</i> (p. 985)	A static, or group box for visually grouping related controls
<i>wxListBox</i> (p. 621)	A list of strings for single or multiple selection
<i>wxListCtrl</i> (p. 630)	A control for displaying lists of strings and/or icons, plus a multicolumn report view
<i>wxTabCtrl</i> (p. 1052)	Manages several tabs
<i>wxTextCtrl</i> (p. 1070)	Single or multiline text editing control
<i>wxTreeCtrl</i> (p. 1134)	Tree (hierarchy) control
<i>wxScrollBar</i> (p. 903)	Scrollbar control
<i>wxSpinButton</i> (p. 963)	A spin or 'up-down' control
<i>wxSpinCtrl</i> (p. 966)	A spin control - i.e. spin button and text control
<i>wxStaticText</i> (p. 989)	One or more lines of non-editable text
<i>wxStaticBitmap</i> (p. 982)	A control to display a bitmap
<i>wxRadioBox</i> (p. 858)	A group of radio buttons
<i>wxRadioButton</i> (p. 864)	A round button to be used with others in a mutually exclusive way
<i>wxSlider</i> (p. 929)	A slider that can be dragged by the user

Menus

<i>wxMenu</i> (p. 683)	Displays a series of menu items for selection
<i>wxMenuBar</i> (p. 693)	Contains a series of menus for use with a frame
<i>wxMenuItem</i> (p. 702)	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 allows for more detailed layouts.

These are the classes relevant to the sizer-based layout.

<i>wxSizer</i> (p. 924)	Abstract base class
<i>wxGridSizer</i> (p. 492)	A sizer for laying out windows in a grid with all fields having the same size
<i>wxFlexGridSizer</i> (p. 431)	A sizer for laying out windows in a flexible grid
<i>wxBoxSizer</i> (p. 77)	A sizer for laying out windows in a row or column
<i>wxStaticBoxSizer</i> (p. 986)	Same as <i>wxBoxSizer</i> , but with surrounding static box
<i>wxNotebookSizer</i> (p. 734)	Sizer to use with the <i>wxNotebook</i> control.

Overview (p. 1376) over the constraints-based layout.

These are the classes relevant to constraints-based window layout.

<i>wxIndividualLayoutConstraint</i> (p. 588)	Represents a single constraint dimension
<i>wxLayoutConstraints</i> (p. 613)	Represents the constraints for a window class

Device contexts

Overview (p. 1391)

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.

<i>wxClientDC</i> (p. 120)	A device context to access the client area outside OnPaint events
<i>wxPaintDC</i> (p. 759)	A device context to access the client area inside OnPaint events
<i>wxWindowDC</i> (p. 1234)	A device context to access the non-client area
<i>wxScreenDC</i> (p. 901)	A device context to access the entire screen
<i>wxDC</i> (p. 280)	The device context base class
<i>wxMemoryDC</i> (p. 678)	A device context for drawing into bitmaps
<i>wxMetafileDC</i> (p. 712)	A device context for drawing into metafiles
<i>wxPostScriptDC</i> (p. 786)	A device context for drawing into PostScript files
<i>wxPrinterDC</i> (p. 806)	A device context for drawing to printers

Graphics device interface

Bitmaps overview (p. 1388)

These classes are related to drawing on device contexts and windows.

<i>wxColour</i> (p. 135)	Represents the red, blue and green elements of a colour
<i>wxBitmap</i> (p. 54)	Represents a bitmap
<i>wxBrush</i> (p. 80)	Used for filling areas on a device context
<i>wxBrushList</i> (p. 85)	The list of previously-created brushes
<i>wxCursor</i> (p. 184)	A small, transparent bitmap representing the cursor
<i>wxFont</i> (p. 434)	Represents fonts
<i>wxFontList</i> (p. 447)	The list of previously-created fonts
<i>wxIcon</i> (p. 558)	A small, transparent bitmap for assigning to frames and drawing on device contexts
<i>wxImage</i> (p. 565)	A platform-independent image class
<i>wxImageList</i> (p. 584)	A list of images, used with some controls
<i>wxMask</i> (p. 659)	Represents a mask to be used with a bitmap for transparent drawing
<i>wxPen</i> (p. 771)	Used for drawing lines on a device context
<i>wxPenList</i> (p. 778)	The list of previously-created pens
<i>wxPalette</i> (p. 761)	Represents a table of indices into RGB values
<i>wxRegion</i> (p. 885)	Represents a simple or complex region on a window or device context

Events

Overview (p. 1364)

An event object contains information about a specific event. Event handlers (usually member functions) have a single, event argument.

<i>wxActivateEvent</i> (p. 20)	A window or application activation event
<i>wxCalendarEvent</i> (p. 104)	Used with <i>wxCalendarCtrl</i> (p. 95)
<i>wxCalculateLayoutEvent</i> (p. 94)	Used to calculate window layout
<i>wxCloseEvent</i> (p. 124)	A close window or end session event
<i>wxCommandEvent</i> (p. 152)	An event from a variety of standard controls
<i>wxDialUpEvent</i> (p. 318)	Event send by <i>wxDialUpManager</i> (p. 319)
<i>wxDropFilesEvent</i> (p. 364)	A drop files event
<i>wxEraseEvent</i> (p. 374)	An erase background event
<i>wxEvent</i> (p. 375)	The event base class
<i>wxFocusEvent</i> (p. 433)	A window focus event
<i>wxKeyEvent</i> (p. 607)	A keypress event
<i>wxIdleEvent</i> (p. 557)	An idle event
<i>wxInitDialogEvent</i> (p. 591)	A dialog initialisation event
<i>wxJoystickEvent</i> (p. 604)	A joystick event
<i>wxListEvent</i> (p. 644)	A list control event

<i>wxMenuEvent</i> (p. 707)	A menu event
<i>wxMouseEvent</i> (p. 721)	A mouse event
<i>wxMoveEvent</i> (p. 729)	A move event
<i>wxNotebookEvent</i> (p. 743)	A notebook control event
<i>wxNotifyEvent</i> (p. 745)	A notification event, which can be vetoed
<i>wxPaintEvent</i> (p. 760)	A paint event
<i>wxProcessEvent</i> (p. 820)	A process ending event
<i>wxQueryLayoutInfoEvent</i> (p. 856)	Used to query layout information
<i>wxScrollEvent</i> (p. 909)	A scroll event from sliders, stand-alone scrollbars and spin buttons
<i>wxScrollWinEvent</i> (p. 908)	A scroll event from scrolled windows
<i>wxSizeEvent</i> (p. 923)	A size event
<i>wxSocketEvent</i> (p. 958)	A socket event
<i>wxSpinEvent</i> (p. 969)	An event from <i>wxSpinButton</i> (p. 963)
<i>wxSysColourChangedEvent</i> (p. 1034)	A system colour change event
<i>wxTabEvent</i> (p. 1058)	A tab control event
<i>wxTreeEvent</i> (p. 1151)	A tree control event
<i>wxUpdateUIEvent</i> (p. 1160)	A user interface update event
<i>wxWizardEvent</i> (p. 1238)	A wizard event

Validators

Overview (p. 1374)

These are the window validators, used for filtering and validating user input.

<i>wxValidator</i> (p. 1166)	Base validator class
<i>wxTextValidator</i> (p. 1092)	Text control validator class
<i>wxGenericValidator</i> (p. 477)	Generic control validator class

Data structures

These are the data structure classes supported by *wxWindows*.

<i>wxCmdLineParser</i> (p. 126)	Command line parser class
<i>wxDate</i> (p. 206)	A class for date manipulation (deprecated in favour of <i>wxDateTime</i>)
<i>wxDateSpan</i> (p. 214)	A logical time interval.
<i>wxDateTime</i> (p. 215)	A class for date/time manipulations
<i>wxExpr</i> (p. 385)	A class for flexible I/O
<i>wxExprDatabase</i> (p. 392)	A class for flexible I/O
<i>wxHashTable</i> (p. 493)	A simple hash table implementation
<i>wxList</i> (p. 615)	A simple linked list implementation
<i>wxLongLong</i> (p. 656)	A portable 64 bit integer type
<i>wxNode</i> (p. 735)	Represents a node in the <i>wxList</i> implementation
<i>wxObject</i> (p. 746)	The root class for most <i>wxWindows</i> classes
<i>wxPathList</i> (p. 769)	A class to help search multiple paths
<i>wxPoint</i> (p. 785)	Representation of a point

<i>wxRect</i> (p. 868)	A class representing a rectangle
<i>wxRegion</i> (p. 885)	A class representing a region
<i>wxString</i> (p. 1007)	A string class
<i>wxStringList</i> (p. 1029)	A class representing a list of strings
<i>wxStringTokenizer</i> (p. 1032)	A class for interpreting a string as a list of tokens or words
<i>wxRealPoint</i> (p. 868)	Representation of a point using floating point numbers
<i>wxSize</i> (p. 922)	Representation of a size
<i>wxTime</i> (p. 1108)	A class for time manipulation (deprecated in favour of <i>wxDateTime</i>)
<i>wxTimeSpan</i> (p. 1092)	A time interval.
<i>wxVariant</i> (p. 1169)	A class for storing arbitrary types that may change at run-time

Run-time class information system

Overview (p. 1329)

wxWindows supports run-time manipulation of class information, and dynamic creation of objects given class names.

<i>wxClassInfo</i> (p. 119)	Holds run-time class information
<i>wxObject</i> (p. 746)	Root class for classes with run-time information
<i>Macros</i> (p. 1285)	Macros for manipulating run-time information

Debugging features

Overview (p. 1356)

wxWindows supports some aspects of debugging an application through classes, functions and macros.

<i>wxDebugContext</i> (p. 304)	Provides memory-checking facilities
<i>wxLog</i> (p. 651)	Logging facility
<i>Log functions</i> (p. 1297)	Error and warning logging functions
<i>Debugging macros</i> (p. 1302)	Debug macros for assertion and checking
<i>WXDEBUG_NEW</i> (p. 1291)	Use this macro to give further debugging information

Networking classes

wxWindows provides its own classes for socket based networking.

<i>wxDialUpManager</i> (p. 319)	Provides functions to check the status of network connection and to establish one
<i>wxIPv4address</i> (p. 595)	Represents an Internet address
<i>wxSocketBase</i> (p. 938)	Represents a socket base object
<i>wxSocketClient</i> (p. 956)	Represents a socket client
<i>wxSocketServer</i> (p. 959)	Represents a socket server

<i>wxSocketEvent</i> (p. 958)	A socket event
<i>wxFTP</i> (p. 466)	FTP protocol class
<i>wxHTTP</i> (p. 555)	HTTP protocol class
<i>wxURL</i> (p. 1164)	Represents a Universal Resource Locator

Interprocess communication

Overview (p. 1428)

wxWindows provides a simple interprocess communications facilities based on DDE.

<i>wxDDEClient</i> (p. 298)	Represents a client
<i>wxDDEConnection</i> (p. 299)	Represents the connection between a client and a server
<i>wxDDEServer</i> (p. 303)	Represents a server
<i>wxTCPClient</i> (p. 1061)	Represents a client
<i>wxTCPConnection</i> (p. 1063)	Represents the connection between a client and a server
<i>wxTCPServer</i> (p. 1067)	Represents a server

Document/view framework

Overview (p. 1402)

wxWindows supports a document/view framework which provides housekeeping for a document-centric application.

<i>wxDocument</i> (p. 351)	Represents a document
<i>wxView</i> (p. 1179)	Represents a view
<i>wxDocTemplate</i> (p. 345)	Manages the relationship between a document class and a view class
<i>wxDocManager</i> (p. 332)	Manages the documents and views in an application
<i>wxDocChildFrame</i> (p. 330)	A child frame for showing a document view
<i>wxDocParentFrame</i> (p. 344)	A parent frame to contain views

Printing framework

Overview (p. 1419)

A printing and previewing framework is implemented to make it relatively straightforward to provide document printing facilities.

<i>wxPreviewFrame</i> (p. 790)	Frame for displaying a print preview
<i>wxPreviewCanvas</i> (p. 787)	Canvas for displaying a print preview
<i>wxPreviewControlBar</i> (p. 788)	Standard control bar for a print preview
<i>wxPrintDialog</i> (p. 797)	Standard print dialog
<i>wxPageSetupDialog</i> (p. 758)	Standard page setup dialog
<i>wxPrinter</i> (p. 803)	Class representing the printer
<i>wxPrinterDC</i> (p. 806)	Printer device context

<code>wxPrintout</code> (p. 807)	Class representing a particular printout
<code>wxPrintPreview</code> (p. 810)	Class representing a print preview
<code>wxPrintData</code> (p. 792)	Represents information about the document being printed
<code>wxPrintDialogData</code> (p. 799)	Represents information about the print dialog
<code>wxPageSetupDialogData</code> (p. 752)	Represents information about the page setup dialog

Drag and drop and clipboard classes

Drag and drop and clipboard overview (p. 1420)

<code>wxDataObject</code> (p. 196)	Data object class
<code>wxDataFormat</code> (p. 194)	Represents a data format
<code>wxTextDataObject</code> (p. 1083)	Text data object class
<code>wxFileDataObject</code> (p. 1083)	File data object class
<code>wxBitmapDataObject</code> (p. 75)	Bitmap data object class
<code>wxCustomDataObject</code> (p. 181)	Custom data object class
<code>wxClipboard</code> (p. 121)	Clipboard class
<code>wxDropTarget</code> (p. 368)	Drop target class
<code>wxFileDropTarget</code> (p. 412)	File drop target class
<code>wxTextDropTarget</code> (p. 1090)	Text drop target class
<code>wxDropSource</code> (p. 365)	Drop source class

File related classes

`wxWindows` has several small classes to work with disk files, see *file classes overview* (p. 1350) for more details.

<code>wxDir</code> (p. 323)	Class for enumerating files/subdirectories.
<code>wxFile</code> (p. 395)	Low-level file input/output class.
<code>wxFFile</code> (p. 402)	Another low-level file input/output class.
<code>wxTempFile</code> (p. 1068)	Class to safely replace an existing file
<code>wxTextFile</code> (p. 1095)	Class for working with text files as with arrays of lines

Stream classes

`wxWindows` has its own set of stream classes, as an alternative to often buggy standard stream libraries, and to provide enhanced functionality.

<code>wxStreamBase</code> (p. 998)	Stream base class
<code>wxStreamBuffer</code> (p. 1000)	Stream buffer class
<code>wxInputStream</code> (p. 592)	Input stream class
<code>wxOutputStream</code> (p. 751)	Output stream class
<code>wxCountingOutputStream</code> (p. 177)	Stream class for querying what size a stream would have.
<code>wxFilterInputStream</code> (p. 432)	Filtered input stream class
<code>wxFilterOutputStream</code> (p. 432)	Filtered output stream class

<i>wxBufferedInputStream</i> (p. 92)	Buffered input stream class
<i>wxBufferedOutputStream</i> (p. 93)	Buffered output stream class
<i>wxMemoryInputStream</i> (p. 681)	Memory input stream class
<i>wxMemoryOutputStream</i> (p. 682)	Memory output stream class
<i>wxDataInputStream</i> (p. 203)	Platform-independent binary data input stream class
<i>wxDataOutputStream</i> (p. 205)	Platform-independent binary data output stream class
<i>wxTextInputStream</i> (p. 1084)	Platform-independent text data input stream class
<i>wxTextOutputStream</i> (p. 1087)	Platform-independent text data output stream class
<i>wxFileInputStream</i> (p. 416)	File input stream class
<i>wxFileOutputStream</i> (p. 417)	File output stream class
<i>wxFileInputStream</i> (p. 419)	Another file input stream class
<i>wxFileOutputStream</i> (p. 420)	Another file output stream class
<i>wxZlibInputStream</i> (p. 1243)	Zlib (compression) input stream class
<i>wxZlibOutputStream</i> (p. 1243)	Zlib (compression) output stream class
<i>wxZipInputStream</i> (p. 1242)	Input stream for reading from ZIP archives
<i>wxSocketInputStream</i> (p. 962)	Socket input stream class
<i>wxSocketOutputStream</i> (p. 962)	Socket output stream class

Threading classes

Multithreading overview (p. 1420)

wxWindows provides a set of classes to make use of the native thread capabilities of the various platforms.

<i>wxThread</i> (p. 1101)	Thread class
<i>wxMutex</i> (p. 730)	Mutex class
<i>wxMutexLocker</i> (p. 733)	Mutex locker utility class
<i>wxCriticalSection</i> (p. 178)	Critical section class
<i>wxCriticalSectionLocker</i> (p. 179)	Critical section locker utility class
<i>wxCondition</i> (p. 160)	Condition class

HTML classes

wxWindows provides a set of classes to display text in HTML format. These class include a help system based on the HTML widget.

<i>wxHtmlHelpController</i> (p. 519)	HTML help controller class
<i>wxHtmlWindow</i> (p. 542)	HTML window class
<i>wxHtmlEasyPrinting</i> (p. 515)	Simple class for printing HTML
<i>wxHtmlPrintout</i> (p. 534)	Generic HTML wxPrintout class
<i>wxHtmlParser</i> (p. 530)	Generic HTML parser class
<i>wxHtmlTagHandler</i> (p. 539)	HTML tag handler, pluginable into wxHtmlParser
<i>wxHtmlWinParser</i> (p. 548)	HTML parser class for wxHtmlWindow
<i>wxHtmlWinTagHandler</i> (p. 555)	HTML tag handler, pluginable into wxHtmlWinParser

Virtual file system classes

wxWindows provides a set of classes that implement an extensible virtual file system, used internally by the HTML classes.

<i>wxFSFile</i> (p. 464)	Represents a file in the virtual file system
<i>wxFileSystem</i> (p. 422)	Main interface for the virtual file system
<i>wxFileSystemHandler</i> (p. 424)	Class used to announce file system type

Miscellaneous

<i>wxApp</i> (p. 21)	Application class
<i>wxCaret</i> (p. 105)	A caret (cursor) object
<i>wxCmdLineParser</i> (p. 126)	Command line parser class
<i>wxConfig</i> (p. 162)	Classes for configuration reading/writing (using either INI files or registry)
<i>wxDllLoader</i> (p. 327)	Class to work with shared libraries.
<i>wxHelpController</i> (p. 495)	Family of classes for controlling help windows
<i>wxLayoutAlgorithm</i> (p. 610)	An alternative window layout facility
<i>wxProcess</i> (p. 815)	Process class
<i>wxTimer</i> (p. 1113)	Timer class
<i>wxStopWatch</i> (p. 997)	Stop watch class
<i>wxMimeTypesManager</i> (p. 713)	MIME-types manager class
<i>wxSystemSettings</i> (p. 1035)	System settings class
<i>wxAcceleratorTable</i> (p. 17)	Accelerator table
<i>wxAutomationObject</i> (p. 49)	OLE automation class
<i>wxFontMapper</i> (p. 448)	Font mapping, finding suitable font for given encoding
<i>wxEncodingConverter</i> (p. 371)	Encoding conversions
<i>wxCalendarDateAttr</i> (p. 101)	Used with <i>wxCalendarCtrl</i> (p. 95)

Database classes

wxWindows provides a set of classes for accessing Microsoft's ODBC (Open Database Connectivity) product, also available for Unix and Linux. The new version by Remstar is documented here:

<i>wxDb</i> (p. 242)	Database class
<i>wxDbColFor</i> (p. 265)	International formatting
<i>wxDbCollnf</i> (p. 265)	Information about a columns definition
<i>wxDbTable</i> (p. 266)	Access to rows of data in a table
<i>wxDbTableInf</i> (p. 280)	Information describing the table
<i>wxDbInf</i> (p. 266)	Database connection

Chapter 8 Topic overviews

This chapter contains a selection of topic overviews, first things first:

Notes on using the reference

In the descriptions of the wxWindows 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 wxWindows 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 wxWindows. Therefore, return values should always be copied for long-term use, especially since the same buffer is often used by wxWindows.

The member functions are given in alphabetical order except for constructors and destructors which appear first.

Writing a wxWindows application: a rough guide

To set a wxWindows application going, you will need to derive a wxApp (p. 21) class and override wxApp::OnInit (p. 28).

An application must have a top-level wxFrame (p. 452) or wxDialog (p. 310) window. Each frame may contain one or more instances of classes such as wxPanel (p. 764), wxSplitterWindow (p. 973) or other windows and controls.

A frame can have a wxMenuBar (p. 693), a wxToolBar (p. 1117), a status line, and a wxIcon (p. 558) for when the frame is iconized.

A wxPanel (p. 764) is used to place controls (classes derived from wxControl (p. 176)) which are used for user interaction. Examples of controls are wxButton (p. 89), wxCheckBox (p. 108), wxChoice (p. 113), wxListBox (p. 621), wxRadioBox (p. 858), wxSlider (p. 929).

Instances of wxDialog (p. 310) 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. 709) and *wxFileDialog* (p. 407).

You never draw directly onto a window - you use a *device context* (DC). *wxDC* (p. 280) is the base for *wxClientDC* (p. 120), *wxPaintDC* (p. 759), *wxMemoryDC* (p. 678), *wxPostScriptDC* (p. 786), *wxMemoryDC* (p. 678), *wxMetafileDC* (p. 712) and *wxPrinterDC* (p. 806). 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. 285) and *wxDC::DrawText* (p. 287). Control colour on a window (*wxColour* (p. 135)) with brushes (*wxBrush* (p. 80)) and pens (*wxPen* (p. 771)).

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 *wxWindow::OnChar* (p. 1205) and *wxWindow::OnMouseEvent* (p. 1213).

Most modern applications will have an on-line, hypertext help system; for this, you need *wxHelp* and the *wxHelpController* (p. 495) class to control *wxHelp*.

GUI applications aren't all graphical wizardry. List and hash table needs are catered for by *wxList* (p. 615), *wxStringList* (p. 1029) and *wxHashTable* (p. 493). You will undoubtedly need some platform-independent *file functions* (p. 1247), and you may find it handy to maintain and search a list of paths using *wxPathList* (p. 769). There's a *miscellany* (p. 1268) of operating system and other functions.

See also *Classes by Category* (p. 1306) for a list of classes.

wxWindows "Hello World"

As many people have requested a mini-sample to be published here so that some quick judgments concerning syntax and basic principles can be made, you can now look at wxWindows' "Hello World":

You have to include wxWindows' 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: wxWindows "Hello world"
//
```

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

#ifdef __BORLANDC__
    #pragma hdrstop
#endif

#ifdef 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 wxWindows 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 wxWindows!" );
}

```

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 wxWindows' Hello world sample",
                  "About Hello World", wxOK | wxICON_INFORMATION );
}

```

wxWindows samples

Probably the best way to learn wxWindows is by reading the source of some 50+ samples provided with it. Many aspects of wxWindows 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 wxWindows are they supposed to test.

There are currently more than 50 different samples as part of wxWindows and this list is not complete. You should start your tour of wxWindows with the *minimal sample* (p. 1320) which is the wxWindows version of "Hello, world!". It shows the basic structure of wxWindows 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. 1321) one which shows many of wxWindows 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:

<i>wxCalendarCtrl</i> (p. 1321)	Calendar a.k.a. date picker control
<i>wxListCtrl</i> (p. 1325)	List view control
<i>wxTreeCtrl</i> (p. 1328)	Tree view control
<i>wxGrid</i> (p. 1324)	Grid control

Finally, it might be helpful to do a search in the entire sample directory if you can't find the sample you showing the control you are interested in by name. Most of wxWindows classes, 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 `wxWindows`.

Calendar sample

This font shows the *calendar control* (p. 95) 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 the use of the *wxCheckListBox* (p. 111) class intercepting check, select and double click events. It also tests the 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. 621) so the sample tests that class as well). The layout of the dialog is created using a *wxBoxSizer* (p. 77) demonstrating a simple dynamic layout.

Config sample

This sample demonstrates the *wxConfig* (p. 162) 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. 1358) 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 `wxWindows`. 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 item to a list box etc. Apart from that, the sample uses a *wxNotebook* (p. 736) and tests most features of this special control (using bitmap in the tabs, using *wxSizers* (p. 924) and *constraints* (p. 613) within notebook pages, advancing pages programmatically and vetoing a page change by intercepting the *wxNotebookEvent* (p. 743).

The various controls tested are listed here:

<i>wxButton</i> (p. 89)	Push button control, displaying text
<i>wxBitmapButton</i> (p. 70)	Push button control, displaying a bitmap
<i>wxCheckBox</i> (p. 108)	Checkbox control
<i>wxChoice</i> (p. 113)	Choice control (a combobox without the editable area)
<i>wxComboBox</i> (p. 143)	A choice with an editable area
<i>wxGauge</i> (p. 470)	A control to represent a varying quantity, such as time

	remaining
<i>wxStaticBox</i> (p. 985)	A static, or group box for visually grouping related controls
<i>wxListBox</i> (p. 621)	A list of strings for single or multiple selection
<i>wxSpinCtrl</i>	A spin ctrl with a text field and a 'up-down' control
<i>wxSpinButton</i> (p. 963)	A spin or 'up-down' control
<i>wxStaticText</i> (p. 989)	One or more lines of non-editable text
<i>wxStaticBitmap</i> (p. 982)	A control to display a bitmap
<i>wxRadioBox</i> (p. 858)	A group of radio buttons
<i>wxRadioButton</i> (p. 864)	A round button to be used with others in a mutually exclusive way
<i>wxSlider</i> (p. 929)	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. These classes are documented in a separate manual available from the *wxWindows* homepage. Obviously, this sample requires a database with ODBC support to be correctly installed on your system.

Dialogs sample

This sample shows how to use the common dialogs available from *wxWindows*. These dialogs are described in details in the *Common dialogs overview* (p. 1398).

Dialup sample

This sample shows *wxDialUpManager* (p. 319) class. It displays in the status bar 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 have 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.

Dynamic sample

This sample is a very small sample that demonstrates the use of the *wxEvtHandler::Connect* (p. 379) 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 *wxWindows* or programs where forms or similar datagrams can be created by the uses.

Exec sample

The exec sample demonstrates the *wxExecute* (p. 1273) and *wxShell* (p. 1282) 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.

Scroll subwindow sample

This sample demonstrates the use of the *wxScrolledWindow* (p. 911) class including placing subwindows into it and drawing simple graphics. It uses the *SetTargetWindow* (p. 918) 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 *wxWindows*, in particular using the *wxWindow::IsExposed* (p. 1202) method with the aim to prevent unnecessary drawing in the window and thus reducing or removing flicker on screen.

Rotate sample

This is a simple example which demonstrates how to rotate an image with the *wxImage::Rotate* (p. 577) 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.

Font sample

The font sample demonstrates *wxFont* (p. 434), *wxFontEnumerator* (p. 445) and *wxFontMapper* (p. 448) classes. It allows you to see the fonts available (to *wxWindows*) on the computer and shows all characters of the chosen font as well.

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. 368): *wxTextDropTarget* (p. 1090) and *wxFileDropTarget* (p. 412).

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 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 `wxWindows` 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. 194) 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. 196) to achieve this.

Grid sample

TODO.

HTML samples

Eight HTML samples (you can find them in directory `samples/html`) cover all features of HTML sub-library.

Test demonstrates how to create `wxHtmlWindow` (p. 542) and also shows most of supported HTML tags.

Widget shows how you can embed ordinary controls or windows within 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 `wxWindows`) allows you to access HTML pages stored in 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. 515) 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. 519) in your application while *Helpview* is simple tool that only pops up help window and displays help books given at command line.

Image sample

The image sample demonstrates the use of the *wxImage* (p. 565) 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. 54), 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 save PNG files are tested.

At the bottom of the main frame is a test for using a monochrome bitmap by drawing into a *wxMemoryDC* (p. 678). The bitmap is then drawn specifying the foreground and background colours with *wxDC::SetTextForeground* (p. 297) and *wxDC::SetTextBackground* (p. 297) (on the left). The bitmap is then converted to a *wxImage* and the foreground colour (black) is replaced with red using *wxImage::Replace* (p. 576).

Layout sample

The layout sample demonstrates the two different layout systems offered by *wxWindows*. 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. 613) class. See also the *overview* (p. 1376) and the *wxIndividualLayoutConstraint* (p. 588) class for further information.

The menu in this sample offers two more tests, one showing how to use a *wxBoxSizer* (p. 77) in a simple dialog and the other one showing how to use sizers in connection with a *wxNotebook* (p. 736) class. See also *wxNotebookSizer* (p. 734) and *wxSizer* (p. 924).

Listctrl sample

This sample shows *wxListCtrl* (p. 630) 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) controls into the control.

Sockets sample

The sockets sample demonstrates how to use the communication facilities provided by *wxSocket* (p. 938). There are two different applications in this sample: a server, which is implemented using a *wxSocketServer* (p. 959) object, and a client, which is implemented as a *wxSocketClient* (p. 956).

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 stands there, waiting for clients (*listening* in the socket parlance). For each accepted connection, a new *wxSocketBase* (p. 938) 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. 959) 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. 938), such as *Read* (p. 950), *Write* (p. 954), *ReadMsg* (p. 951) and *WriteMsg* (p. 955), 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 an URL test which shows how to use the *wxURL* (p. 1164) 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).

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. 1070) class (single line, multi line, read only, password, ignoring TAB, ignoring ENTER).

Secondly it shows how to intercept a *wxKeyEvent* (p. 607) 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. 121) 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 the 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, *wxWindows* offers the *wxPostEvent* (p. 1281) 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. 730) class) that prevent threads from accessing the GUI classes as long as any other thread accesses them. For this, *wxWindows* has the *wxMutexGuiEnter* (p. 1246) and *wxMutexGuiLeave* (p. 1246) functions, both of which are used and tested in the sample as well.

See also *Multithreading overview* (p. 1420) and *wxThread* (p. 1101).

Toolbar sample

The toolbar sample shows the *wxToolBar* (p. 1117) class in action.

The following things are demonstrated:

- Creating the toolbar using `wxToolBar::AddTool` (p. 1121) and `wxToolBar::AddControl` (p. 1120): 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. 1122) and `wxToolBar::InsertTool` (p. 1127) to dynamically update the toolbar.

Treectrl sample

This sample demonstrates using `wxTreeCtrl` (p. 1134) 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 so-called wizard dialog (implemented using `wxWizard` (p. 1235) 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. 1232) 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 `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. 1239))

wxApp overview

Classes: `wxApp` (p. 21)

A `wxWindows` application does not have a *main* procedure; the equivalent is the `OnInit` (p. 28) member defined for a class derived from `wxApp`. `OnInit` will usually create a top

window as a bare minimum.

Unlike in earlier versions of `wxWindows`, `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. 31) to let `wxWindows` 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 `wxWindow::OnCloseWindow` (p. 1208) handler.

In emergencies the `wxExit` (p. 1274) function can be called to kill the application.

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, argv[0]);
    ...
    SetTopWindow(the_frame);

    return TRUE;
}
```

Note the use of `IMPLEMENT_APP(appClass)`, which allows `wxWindows` to dynamically create an instance of the application object at the appropriate point in `wxWindows` initialization. Previous versions of `wxWindows` 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.

Run time class information overview

Classes: `wxObject` (p. 746), `wxClassInfo` (p. 119).

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 `wxWindows` is no exception. As it originated before the addition of RTTI to the standard C++ and as support for it still missing from some (albeit old) compilers, `wxWindows` 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 `wxDynamicCast()` (p. 1291) macro which, for the implementations that support it, is defined just as `dynamic_cast<>` and uses `wxWindows` RTTI for all the others. This macro is limited to `wxWindows` classes only and only works with pointers (unlike the real `dynamic_cast<>` which also accepts references).

Each class that you wish to be known 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. 1285) 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. 748) 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. 1270) 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`.

wxClassInfo

Run time class information overview (p. 1329)

Class: `wxClassInfo` (p. 119)

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. 746) and *wxCreateDynamicObject* (p. 1270).

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. 1007), *wxArrayString* (p. 44), *wxStringTokenizer* (p. 1032)

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, although care should be taken when passing strings containing it to other functions.

`wxString` only works with ASCII (8 bit characters) strings as of this release, but support for UNICODE (16 bit characters) is planned for the next one.

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. 1022) and stream-like insertion functions as well as much more - see `wxString` (p. 1007) 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` object takes exactly the same space as a `char *` pointer, see *reference counting* (p. 1334)) and speed. It also provides performance *statistics gathering code* (p. 1335) 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 `wxWindows 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. 1015), *BeforeLast* (p. 1016), *operator<<* (p. 1028) or *Printf* (p. 1022). Of course, all the standard string operations are supported as well.
4. **UNICODE** In this release, `wxString` only supports *construction* from a UNICODE string, but in the next one it will be capable of also storing its internal data in either ASCII or UNICODE format.
5. **Used by `wxWindows`** And, of course, this class is used everywhere inside `wxWindows` 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 `wxWindows`.

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. 1021) or `Length()` (p. 1021) 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 `wxWindows` 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 `wxWindows` and other programs (by just typedefing `wxString` as `std::string` when used outside `wxWindows`) and by staying compatible with future versions of `wxWindows` 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 `wxWindows` 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. 1016) 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. 1016): 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. 1334)) 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: `IsEmpty()` (p. 1255) verifies whether the string is empty (returning TRUE for NULL pointers), `Strlen()` (p. 1255) also handles NULLs correctly and returns 0 for them and `Stricmp()` (p. 1255) 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 `wxSprintf` (p. 1255) and `wxVsprintf` (p. 1256) 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. 1022) which is also safe.

There is another class which might be useful when working with `wxString`: `wxStringTokenizer` (p. 1032). 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. 44): 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. 1018)) cannot be chosen and the "read/write" *operator[]* (the same as *GetWritableChar()* (p. 1019)) 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. 1018) instead of the array subscript operator for this reasons. Please note that *at()* (p. 1011) 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. 1333)) 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. 1334).

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. 1014) 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.

Date and time classes overview

Classes: *wxDateTime* (p. 215), *wxDateSpan* (p. 214), *wxTimeSpan* (p. 1092), *wxCalendarCtrl* (p. 95)

Introduction

wxWindows provides a set of powerful classes to work with dates and times. Some of the supported features of *wxDateTime* (p. 215) 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

Many features	from rounding errors. 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 <code>wxDateTime</code> 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. 215) itself which represents an absolute moment in the time, there are also two classes - `wxTimeSpan` (p. 1092) and `wxDateSpan` (p. 214) which represent the intervals of time.

There are also helper classes which are used together with `wxDateTime`: `wxDateTimeHolidayAuthority` (p. 242) which is used to determine whether a given date is a holiday or not and `wxDateTimeWorkDays` (p. 242) 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. 1340).

Finally, in other parts of this manual you may find mentions of `wxDate` and `wxTime` classes. *These classes* (p. 1340) are obsolete and superseded by `wxDateTime`.

wxDateTime characteristics

`wxDateTime` (p. 215) 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 approximately 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. 1339) 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. 1092): it is just a difference in milliseconds between two moments in the time. Adding and subtracting such 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. 214). It handles this sort of operations in the most natural way possible, but note that manipulating with these 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-existing 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 does 'in a month' mean (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 date by a number is an invalid operation, even though multiplying either of time span classes by a number is perfectly valid.

Here is what can be done:

Addition 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. 241) or *MakeTimezone* (p. 241) methods to adjust the date for the target time zone. There are also special versions of these functions *ToGMT* (p. 241) and *MakeGMT* (p. 242) 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. 215) 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. 1339) 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 wxWindows 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 wxWindows

This section briefly describes the state of the Unicode support in wxWindows. 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 wxWindows 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 16 bit for encoding each character. This allows to have 65536 characters 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 wxWindows 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 wxWindows. 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()`, ...)

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 on 2 bytes instead of 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.

Finally, the 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").

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);
#else // ANSI
    char ch = '*';
    const char *s = "Hello, world!";
    int len = strlen(s);
#endif // Unicode/ANSI
```

Of course, it would be nearly impossible 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 wxWindows

In wxWindows, 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 `#ifdefs` at all. Instead, we define some types and macros which behave differently in the Unicode and ANSI builds and allows 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. 1007) supports Unicode, i.e. it stores either ANSI or Unicode strings depending on the compile mode.

Finally, there is a special `wxT()` 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()` 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()` macro unless they're already converted to the right representation (another standard wxWindows macro `_()` does it, 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 wxWindows 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).

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. 1016) 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 that some parts of `wxWindows` are not Unicode-compliant yet (ODBC classes, for example). 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 (`wxEncodingConverter` (p. 371) depends on this partially) and construct `wxString` objects from Unicode strings (presumably read from some external file or elsewhere).

wxMBConv classes overview

Classes: `wxMBConv` (p. 661), `wxMBConvFile` (p. 664), `wxMBConvUTF7` (p. 665), `wxMBConvUTF8` (p. 665), `wxCSCConv` (p. 180)

The `wxMBConv` classes in `wxWindows` enables 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 wxWindows 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 wxMBConvFile, wxMBConvUTF7, wxMBConvUTF8, 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

In C++, for a class to be useful and possible to pass around, it needs to be instantiated. All of the wxWindows-provided wxMBConv classes have predefined instances (wxConvLibc, wxConvFile, 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.

wxCSSConv

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 `wxWindows`, 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 `wxCSCConv` instance on the fly.

```
wxString str(input_data, wxCSCConv(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 `wxCSCConv` instance.

```
wxCSCConv 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 `wxWindows`; 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 `wxWindows`, 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. `wxWindows` provides facilities for message translation with its `wxLocale` (p. 648) class and is itself fully translated into several languages. Please consult `wxWindows` 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 the future versions of the library!

The `wxWindows` approach to i18n closely follows GNU `gettext` package. `wxWindows` 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 the 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. 1255) or equivalently the `_()` 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 may be told (using its `-k` option) to recognise `_()` and *wxGetTranslation* and extract 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. 648).

See also the GNU *gettext* documentation linked from `docs/html/index.htm` in your *wxWindows* distribution.

See also *Writing non-English applications* (p. 1347). It focuses on handling charsets related problems.

Writing non-English applications

This article describes how to write applications that communicate with user in language other than English. Unfortunately many languages use different charsets under Unix and Windows (and other platforms, to make situation even more complicated). These charsets usually differ in so many characters it is impossible to use same texts under all platforms.

wxWindows library provides 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. 1346) 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. 1346)).

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"
```

Notice 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*.)

wxWindows 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 correct header, it checks the charset. If the charset is "alien" on the platform the program is currently running (e.g. any of ISO encodings under Windows or CP12XX under Unix) it uses *wxEncodingConverter::GetPlatformEquivalents* (p. 372) to obtain an encoding that is more common on this platform and converts the message catalog to this encoding. Note that it does **not** check for presence of fonts in the "platform" encoding! It only assumes that it is always better to have strings in platform native encoding than in an encoding that is rarely (if ever) used.

The behaviour described above is disabled by default. You must set *bConvertEncoding* to TRUE in *wxLocale constructor* (p. 648) in order to enable runtime encoding conversion.

Font mapping

You can use *wxEncodingConverter* (p. 371) and *wxFontMapper* (p. 448) to display text:

```
if (!wxTheFontMapper->IsEncodingAvailable(enc, facename))
{
    wxFontEncoding alternative;
    if (wxTheFontMapper->GetAltForEncoding(enc, &alternative,
                                           facename, FALSE))
    {
        wxEncodingConverted encconv;
        if (!encconv.Init(enc, alternative))
            ...failure...
        else
            text = encconv.Convert(text);
    }
    else
        ...failure...
}
...display text...
```

Converting data

You may want to store all program data (created documents etc.) in the same encoding, let's say windows1250. Obviously, the best way would be to use *wxEncodingConverter* (p. 371).

Help files

If you're using *wxHtmlHelpController* (p. 519) there is no problem at all. You must only 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. 615), *wxArray* (p. 32)

wxWindows 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 *wxWindows* does not use STL which provides the standard implementation of many container classes in C++. First of all, *wxWindows* 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 wxWindows on STL - but this is not yet the case.

wxWindows 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 wxWindows, but may, of course, be used in your programs as well if you wish.

The list classes in wxWindows 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. 618) method.

Dynamic arrays resemble C arrays but with two important differences: they provide run-time range checking in debug builds and they expand automatically 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 wxWindows 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. 615) and *wxArray* (p. 32) documentation.

Finally, wxWindows 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. 1007) reference counting schema.

File classes and functions overview

Classes: *wxFile* (p. 395), *wxDir* (p. 323), *wxTempFile* (p. 1068), *wxTextFile* (p. 1095)

Functions: see *file functions* (p. 1247).

`wxWindows` 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. 1095) 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. 651) 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. 1068) 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. 998), `wxStreamBuffer` (p. 1000), `wxInputStream` (p. 592), `wxOutputStream` (p. 751), `wxFilterInputStream` (p. 432), `wxFilterOutputStream` (p. 432)

Purpose of wxStream

We had troubles with standard C++ streams on several platforms: they react 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 `wxWindows` because an application should compile and run on all supported platforms and we don't want users to depend on release X.XX of `libg++` or some other compiler to run the program.

`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 the destruction.
```

Compatibility with C++ streams

As I said previously, we could add a filter stream so it takes an `istream` argument and builds a `wxInputStream` from it: I don't think it should be difficult to implement it and it may be available in the fix of `wxWindows 2.0`.

wxLog classes overview

Classes: `wxLog` (p. 651), `wxLogStderr`, `wxLogOstream`, `wxLogTextCtrl`, `wxLogWindow`, `wxLogGui`, `wxLogNull`

This is a general overview of logging classes provided by `wxWindows`. The word logging here has a broad sense, including all of the program output, not only non interactive messages. The logging facilities included in `wxWindows` 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()`, i.e. they take the format string as the first argument and a variable number of arguments. Here are all of them:

- **wxLogFatalError** which is like `wxLogError`, but also terminates the program with the exit code 3 (using `abort()` standard function also terminates the program with this exit code).
- **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). Notice that the standard behaviour is to not show informational messages if there are any errors later - the logic being that the later error messages make the informational messages preceding them meaningless.
- **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. 452) if it has one.
- **wxLogSysError** is mostly used by `wxWindows` 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 `wxWindows`, while the log classes are. Some of advantages in using `wxWindows` 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()`.

Moreover `wxMSW` doesn't have a **console** as you may have with `wxGTK`. Under `wxMSW`, a call using `cout` just goes nowhere. To cope with this problem, `wxWindows` provides a way to redirect `cout` calls to `wxTextCtrl` (p. 1070), i.e.:

```
wxLogWindow *logger=new wxLogWindow(your_frame,"Logger");
cout=*new ostream(logger->GetTextCtrl());
wxLog::SetActiveTarget(logger);
```

On the opposite, if you like your `wxLogXXX` calls to behave as a `cout` call does, just write :

```
wxLog *logger=new wxLogStream(&cout);
wxLog::SetActiveTarget(logger);
```

- **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 `wxWindows` 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 `wxWindows` 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.

wxWindows has the notion of a *log target*: it is just a class deriving from *wxLog* (p. 651). 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 *wxWindows* 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
```

Debugging overview

Classes, functions and macros: *wxDebugContext* (p. 304), *wxObject* (p. 746), *wxLog* (p. 651), *Log functions* (p. 1297), *Debug macros* (p. 1302)

Various classes, functions and macros are provided in wxWindows to help you debug your application. Most of these are only available if you compile both wxWindows, your application and *all* libraries that use wxWindows 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. 304) 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. 747) member function for each class you derive from a wxWindows class, so that *wxDebugContext::Dump* (p. 305) 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. 305) and *wxDebugContext::PrintStatistics* (p. 307) 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 wxWindows 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 symbol `wxUSE_GLOBAL_MEMORY_OPERATORS` is set to 1 in `setup.h`, 'new' is defined to be:

```
#define new new(__FILE__, __LINE__)
```

All occurrences of 'new' in wxWindows 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. 1302) as part of a 'defensive programming' strategy, scattering `wxASSERT`s 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. 1302) is used to pop up an error message box when a condition is not true. You can also use `wxASSERT_MSG` (p. 1303) 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. 1299) and `wxLogTrace` (p. 1299) functions to output debugging information in debug mode; it will do nothing for non-debugging code.

wxDebugContext overview

Debugging overview (p. 1356)

Class: `wxDebugContext` (p. 304)

`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 `wxWindows` 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. 162)

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. 162) 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 the limitations of the underlying physical storage as in the case of `wxIniConfig`, 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. 162).

wxExpr overview

`wxExpr` is a C++ class reading and writing a subset of Prolog-like syntax, supporting objects attribute/value pairs.

`wxExpr` can be used to develop programs with readable and robust data files. Within `wxWindows` itself, it is used to parse the `.wxr` dialog resource files.

History of wxExpr

During the development of the tool Hardy within the AIAI, a need arose for a data file format for C++ that was easy for both humans and programs to read, was robust in the face of fast-moving software development, and that provided some compatibility with AI languages such as Prolog and LISP.

The result was the `wxExpr` library (formerly called PrologIO), which is able to read and write a Prolog-like attribute-value syntax, and is additionally capable of writing LISP syntax for no extra programming effort. The advantages of such a library are as follows:

1. The data files are readable by humans;
2. I/O routines are easier to write and debug compared with using binary files;
3. the files are robust: unrecognised data will just be ignored by the application
4. Inbuilt hashing gives a random access capability, useful for when linking up C++ objects as data is read in;
5. Prolog and LISP programs can load the files using a single command.

The library was extended to use the ability to read and write Prolog-like structures for remote procedure call (RPC) communication. The next two sections outline the two main ways the library can be used.

wxExpr for data file manipulation

The fact that the output is in Prolog syntax is irrelevant for most programmers, who just need a reasonable I/O facility. Typical output looks like this:

```

diagram_definition(type = "Spirit Belief Network").

node_definition(type = "Model",
  image_type = "Diamond",
  attribute_for_label = "name",
  attribute_for_status_line = "label",
  colour = "CYAN",
  default_width = 120,
  default_height = 80,
  text_size = 10,
  can_resize = 1,
  has_hypertext_item = 1,
  attributes = ["name", "combining_function", "level_of_belief"]).

arc_definition(type = "Potentially Confirming",
  image_type = "Spline",
  arrow_type = "End",
  line_style = "Solid",
  width = 1,
  segmentable = 0,
  attribute_for_label = "label",
  attribute_for_status_line = "label",
  colour = "BLACK",
  text_size = 10,
  has_hypertext_item = 1,
  can_connect_to = ["Evidence", "Cluster", "Model", "Evidence",
"Evidence", "Cluster"],
  can_connect_from = ["Data", "Evidence", "Cluster", "Evidence", "Data",
"Cluster"]).

```

This is substantially easier to read and debug than a series of numbers and strings.

Note the object-oriented style: a file comprises a series of *clauses*. Each clause is an object with a *functor* or object name, followed by a list of attribute-value pairs enclosed in parentheses, and finished with a full stop. Each attribute value may be a string, a word (no quotes), an integer, a real number, or a list with potentially recursive elements.

The way that the facility is used by an application to read in a file is as follows:

1. The application creates a wxExprDatabase instance.
2. The application tells the database to read in the entire file.
3. The application searches the database for objects it requires, decomposing the objects using the wxExpr API. The database may be hashed, allowing rapid linking-up of application data.
4. The application deletes or clears the wxExprDatabase.

Writing a file is just as easy:

1. The application creates a wxExprDatabase instance.
2. The application adds objects to the database using the API.
3. The application tells the database to write out the entire database, in Prolog or LISP notation.
4. The application deletes or clears the wxExprDatabase.

To use the library, include "wxexpr.h".

wxExpr compilation

For UNIX compilation, ensure that YACC and LEX or FLEX are on your system. Check that the makefile uses the correct programs: a common error is to compile `y_tab.c` with a C++ compiler. Edit the `CCLEX` variable in `make.env` to specify a C compiler. Also, do not attempt to compile `lex_yy.c` since it is included by `y_tab.c`.

For DOS compilation, the simplest thing is to copy `dosyacc.c` to `y_tab.c`, and `doslex.c` to `lex_yy.c`. It is `y_tab.c` that must be compiled (`lex_yy.c` is included by `y_tab.c`) so if adding source files to a project file, ONLY add `y_tab.c` plus the `.cc` files. If you wish to alter the parser, you will need YACC and FLEX on DOS.

The DOS tools are available at the AIAI ftp site, in the tools directory. Note that for FLEX installation, you need to copy `flex.skl` into the directory `c:/lib`.

If you are using Borland C++ and wish to regenerate `lex_yy.c` and `y_tab.c` you need to generate `lex_yy.c` with FLEX and then comment out the 'malloc' and 'free' prototypes in `lex_yy.c`. It will compile with lots of warnings. If you get an undefined `_PROIO_YYWRAP` symbol when you link, you need to remove `USE_DEFINE` from the makefile and recompile. This is because the `parser.y` file has a choice of defining this symbol as a function or as a define, depending on what the version of FLEX expects. See the bottom of `parser.y`, and if necessary edit it to make it compile in the opposite way to the current compilation.

Bugs

These are the known bugs:

1. Functors are permissible only in the main clause (object). Therefore nesting of structures must be done using lists, not predicates as in Prolog.
2. There is a limit to the size of strings read in (about 5000 bytes).

Using wxExpr

This section is a brief introduction to using the `wxExpr` package.

First, some terminology. A `wxExprDatabase` is a list of *clauses*, each of which represents an object or record which needs to be saved to a file. A clause has a *functor* (name), and a list of attributes, each of which has a value. Attributes may take the following types of value: string, word, integer, floating point number, and list. A list can itself contain any type, allowing for nested data structures.

Consider the following code.

```
wxExprDatabase db;

wxExpr *my_clause = new wxExpr("object");
my_clause->AddAttributeValue("id", (long)1);
my_clause->AddAttributeValueString("name", "Julian Smart");
db.Append(my_clause);

ofstream file("my_file");
db.Write(file);
```

This creates a database, constructs a clause, adds it to the database, and writes the whole database to a file. The file it produces looks like this:

```
object(id = 1,
       name = "Julian Smart").
```

To read the database back in, the following will work:

```
wxExprDatabase db;
db.Read("my_file");

db.BeginFind();

wxExpr *my_clause = db.FindClauseByFunctor("object");
int id = 0;
wxString name = "None found";

my_clause->GetAttributeValue("id", id);
my_clause->GetAttributeValue("name", name);

cout << "Id is " << id << ", name is " << name << "\n";
```

Note the setting of defaults before attempting to retrieve attribute values, since they may not be found.

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. 464) class provides information about opened file (name, input stream, mime type and anchor).
- The *wxFileSystem* (p. 422) class is the interface. Its main methods are *ChangePathTo()* and *OpenFile()*. This class is most often used by the end user.
- The *wxFileSystemHandler* (p. 424) is the core of virtual file systems mechanism. You can derive your own handler and pass it to of 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.wxwindows.org/index.html" the right location is "//www.wxwindows.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", the left location is "reference/fopen.htm", the anchor is "syntax" and the right 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 `wxWindows` so far:

wxInternetFSHandler	A handler for accessing documents via HTTP or FTP protocols. Include file is <code><wx/fs_inet.h></code> .
wxZipFSHandler	A handler for ZIP archives. Include file is <code><wx/fs_zip.h></code> . 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 <i>wxMemoryFSHandler documentation</i> (p. 680) for details. Include file is <code><wx/fs_mem.h></code> . UURL is prefixed with <code>memory:</code> , e.g. "memory:myfile.htm"

In addition, `wxFileSystem` itself can access local files.

Initializing file system handlers

Use `wxFileSystem::AddHandler` (p. 423) to initialize a handler, for example:

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

...

bool MyApp::OnInit()
{
    wxFileSystem::AddHandler(new wxMemoryFSHandler);
    ...
}
```

Event handling overview

Classes: `wxEvtHandler` (p. 378), `wxWindow` (p. 1184), `wxEvent` (p. 375)

Introduction

Before version 2.0 of `wxWindows`, events were handled by the application either by supplying callback functions, or by overriding virtual member functions such as **OnSize**.

From `wxWindows 2.0`, *event tables* are used instead, with a few exceptions.

An event table is placed in an implementation file to tell `wxWindows` how to map events to member functions. These member functions are not virtual functions, but they 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. (In fact you could intercept a particular window's size event by using `EVT_CUSTOM(wxEVT_SIZE, id, func)`.)

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. 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. 923) is used. For menu commands and most control commands (such as button presses), *wxCommandEvent* (p. 152) is used. When controls get more complicated, then specific event classes are used, such as *wxTreeEvent* (p. 1151) for events from *wxTreeCtrl* (p. 1134) windows.

As well as the event table in the implementation file, there must be a `DECLARE_EVENT_TABLE` macro in the class definition. For example:

```
class MyFrame: public wxFrame {
    DECLARE_DYNAMIC_CLASS(MyFrame)

public:
    ...
    void OnExit(wxCommandEvent& event);
    void OnSize(wxSizeEvent& event);
protected:
    int     m_count;
    ...
    DECLARE_EVENT_TABLE()
};
```

How events are processed

When an event is received from the windowing system, *wxWindows* calls *wxEvtHandler::ProcessEvent* (p. 382) on the first event handler object belonging to the window generating the event.

It may be noted that *wxWindows*' 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. 378).

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 wxWindows 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. 384)) the function skips to step (6).
2. If the object is a `wxWindow`, **ProcessEvent** is recursively called on the window's `wxValidator` (p. 1166). 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.

Pay close attention to Step 5. People often overlook or get confused by this powerful feature of the `wxWindows` event processing system. To put it a different way, events derived either directly or indirectly from `wxCommandEvent` will travel up the containment hierarchy from child to parent until an event handler is found that doesn't call `event.Skip()`. Events not derived from `wxCommandEvent` are sent only to the window they occurred in and then stop.

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 `wxEvtHandler::IsCommandEvent` 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. 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:

<i>wxEvtHandler</i> (p. 375)	The event base class
<i>wxActivateEvent</i> (p. 20)	A window or application activation event
<i>wxCloseEvent</i> (p. 124)	A close window or end session event
<i>wxEraseEvent</i> (p. 374)	An erase background event
<i>wxFocusEvent</i> (p. 433)	A window focus event
<i>wxKeyEvent</i> (p. 607)	A keypress event
<i>wxIdleEvent</i> (p. 557)	An idle event
<i>wxInitDialogEvent</i> (p. 591)	A dialog initialisation event
<i>wxJoystickEvent</i> (p. 604)	A joystick event
<i>wxMenuEvent</i> (p. 707)	A menu event
<i>wxMouseEvent</i> (p. 721)	A mouse event
<i>wxMoveEvent</i> (p. 729)	A move event
<i>wxPaintEvent</i> (p. 760)	A paint event
<i>wxQueryLayoutInfoEvent</i> (p. 856)	Used to query layout information
<i>wxSizeEvent</i> (p. 923)	A size event
<i>wxScrollWinEvent</i> (p. 908)	A scroll event sent by a scrolled window (not a scroll bar)
<i>wxSysColourChangedEvent</i> (p. 1034)	A system colour change event
<i>wxUpdateUIEvent</i> (p. 1160)	A user interface update 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. 1224) (or, preferably, `wxWindow::PushEventHandler` (p. 1218)) 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 series 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 `-1` to a window constructor, an identifier will be generated for you, but beware: if things don't respond in the way they should, it could be because of an id conflict. It is safer to supply window ids at all times. Automatic generation of identifiers starts at `1` so may well conflict with your own identifiers.

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_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_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

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 <code>wxEVT_SIZE</code>), the window identifier, and a member function to call.
EVT_CUSTOM_RANGE(event, id1, id2, func)	The same as <code>EVT_CUSTOM</code> , but responds to a range of window identifiers.
EVT_COMMAND(id, event, func)	The same as <code>EVT_CUSTOM</code> , but expects a member function with a <code>wxCommandEvent</code> argument.
EVT_COMMAND_RANGE(id1, id2, event, func)	The same as <code>EVT_CUSTOM_RANGE</code> , but expects a member function with a <code>wxCommandEvent</code> argument.

Macros listed by event class

The documentation for specific event macros is organised by event class. Please refer to these sections for details.

<i>wxActivateEvent</i> (p. 20)	The <code>EVT_ACTIVATE</code> and <code>EVT_ACTIVATE_APP</code> macros intercept activation and deactivation events.
<i>wxCommandEvent</i> (p. 152)	A range of commonly-used control events.
<i>wxCloseEvent</i> (p. 124)	The <code>EVT_CLOSE</code> macro handles window closure called via <code>wxWindow::Close</code> (p. 1190).
<i>wxDropFilesEvent</i> (p. 364)	The <code>EVT_DROP_FILES</code> macros handles file drop events.
<i>wxEraseEvent</i> (p. 374)	The <code>EVT_ERASE_BACKGROUND</code>

<i>wxFocusEvent</i> (p. 433)	macro is used to handle window erase requests.
<i>wxKeyEvent</i> (p. 607)	The EVT_SET_FOCUS and EVT_KILL_FOCUS macros are used to handle keyboard focus events.
<i>wxIdleEvent</i> (p. 557)	EVT_CHAR and EVT_CHAR_HOOK macros handle keyboard input for any window.
<i>wxInitDialogEvent</i> (p. 591)	The EVT_IDLE macro handle application idle events (to process background tasks, for example).
<i>wxListEvent</i> (p. 644)	The EVT_INIT_DIALOG macro is used to handle dialog initialisation.
<i>wxMenuEvent</i> (p. 707)	These macros handle <i>wxListCtrl</i> (p. 630) events.
<i>wxMouseEvent</i> (p. 721)	These macros handle special menu events (not menu commands).
<i>wxMoveEvent</i> (p. 729)	Mouse event macros can handle either individual mouse events or all mouse events.
<i>wxPaintEvent</i> (p. 760)	The EVT_MOVE macro is used to handle a window move.
<i>wxScrollEvent</i> (p. 909)	The EVT_PAINT macro is used to handle window paint requests.
<i>wxSizeEvent</i> (p. 923)	These macros are used to handle scroll events from <i>wxScrollBar</i> (p. 903), <i>wxSlider</i> (p. 929), and <i>wxSpinButton</i> (p. 963).
<i>wxSplitterEvent</i> (p. 970)	The EVT_SIZE macro is used to handle a window resize.
<i>wxSysColourChangedEvent</i> (p. 1034)	The EVT_SPLITTER_SASH_POS_CHANGE D, EVT_SPLITTER_UNSPPLIT and EVT_SPLITTER_DOUBLECLICKED macros are used to handle the various splitter window events.
<i>wxTreeEvent</i> (p. 1151)	The EVT_SYS_COLOUR_CHANGED macro is used to handle events informing the application that the user has changed the system colours (Windows only).
<i>wxUpdateUIEvent</i> (p. 1160)	These macros handle <i>wxTreeCtrl</i> (p. 1134) events.
	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.

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. 1184) in addition to their own styles.

Window deletion overview

Classes: *wxCloseEvent* (p. 124), *wxWindow* (p. 1184)

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, *wxWindows* calls *wxWindow::Close* (p. 1190). This in turn generates an EVT_CLOSE event: see *wxWindow::OnCloseWindow* (p. 1208).

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. 125) 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. 126). Calling this provides useful information to the calling code.

The *wxCloseEvent* handler should only call *wxWindow::Destroy* (p. 1192) to delete the window, and not use the **delete** operator. This is because for some window classes, *wxWindows* 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. 1192) 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. 1190) event just as the framework

does, or it can call `wxWindow::Destroy` (p. 1192) 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()`.

Under Windows, `wxDialog` defines a handler for `wxWindow::OnCharHook` (p. 1206) that generates a Cancel event if the Escape key has been pressed.

What should I do when the user calls up Exit from a menu?

You can simply call `wxWindow::Close` (p. 1190) 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 `wxWindows 1.xx`, the **OnClose** function did not actually delete 'this', but signaled to the calling function (either **Close**, or the `wxWindows` 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 wxWindows 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. 26) (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. 1192) function instead of the **delete** operator when deleting these kinds of windows explicitly.

wxDialog overview

Classes: *wxDialog* (p. 310)

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 wxWindows, 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. 1256).

See also *wxPanel* (p. 764) and *wxWindow* (p. 1184) for inherited member functions. Validation of data in controls is covered in *Validator overview* (p. 1374).

wxValidator overview

Classes: *wxValidator* (p. 1166), *wxTextValidator* (p. 1092), *wxGenericValidator* (p. 477)

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. 1092) (which does text control data transfer, validation and filtering) and *wxGenericValidator* (p. 477) (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, VALIDATE_TEXT, "",
    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. 1169) 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. 1168) member function should transfer the data from the validator or associated C++ variable to the control.

The *wxValidator::TransferFromWindow* (p. 1168) 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. 1168) 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 *wxWindows* 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. 317) is called (for a modeless dialog) or *wxDialog::ShowModal* (p. 317) is called (for a modal dialog), the function *wxWindow::InitDialog* (p. 1202) 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. 1233). 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 <i>wxWindow::InitDialog</i> (p. 1202) 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. 1233), 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. 1232) 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. 613), `wxIndividualLayoutConstraint` (p. 588).

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 constraints calculation is done in `wxWindow::Layout` (p. 1203) function which evaluates constraints. To call it you can either call `wxWindow::SetAutoLayout` (p. 1221) 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. 1203) yourself if the parent window is not a frame, panel or dialog).

Constraint layout: more detail

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 `wxWindows`, 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 `wxWindows` 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);
```

The wxWindows resource system

wxWindows has an optional *resource file* facility, which allows separation of dialog, menu, bitmap and icon specifications from the application code.

It is similar in principle to the Windows resource file (whose ASCII form is suffixed .RC and whose binary form is suffixed .RES). The wxWindows resource file is currently ASCII-only, suffixed .WXR. Note that under Windows, the .WXR file does not *replace* the native Windows resource file, it merely supplements it. There is no existing native resource format in X (except for the defaults file, which has limited expressive power).

For details of functions for manipulating resource files and loading user interface elements, see *wxWindows resource functions* (p. 1293).

You can use Dialog Editor to create resource files. Unfortunately neither Dialog Editor nor the .WXR format currently cover all wxWindows controls; some are missing, such as wxSpinCtrl, wxSpinButton, wxListCtrl, wxTreeCtrl and others.

Note that in later versions of wxWindows, this resource format will be replaced by XML specifications that can also include sizers.

The format of a .WXR file

A wxWindows resource file may look a little odd at first. It is C++ compatible, comprising mostly of static string variable declarations with wxExpr syntax within the string.

Here's a sample .WXR file:

```

/*
 * wxWindows Resource File
 *
 */

#include "noname.ids"

static char *my_resource = "bitmap(name = 'my_resource',\
    bitmap = ['myproject', wxBITMAP_TYPE_BMP_RESOURCE, 'WINDOWS'],\
    bitmap = ['myproject.xpm', wxBITMAP_TYPE_XPM, 'X']).";

static char *menuBar11 = "menu(name = 'menuBar11',\
    menu = \
    [\
        ['&File', 1, '', \
            ['&Open File', 2, 'Open a file'],\
            ['&Save File', 3, 'Save a file'],\
            [],\
            ['E&xit', 4, 'Exit program']\
        ],\
        ['&Help', 5, '', \
            ['&About', 6, 'About this program']\
        ]\
    ]).\
    ]).";

static char *project_resource = "icon(name = 'project_resource',\
    icon = ['project', wxBITMAP_TYPE_ICO_RESOURCE, 'WINDOWS'],\
    icon = ['project_data', wxBITMAP_TYPE_XBM, 'X']).";

static char *panel3 = "dialog(name = 'panel3',\
    style = '',\
    title = 'untitled',\
    button_font = [14, 'wxSWISS', 'wxNORMAL', 'wxBOLD', 0],\
    label_font = [10, 'wxSWISS', 'wxNORMAL', 'wxNORMAL', 0],\
    x = 0, y = 37, width = 292, height = 164,\
    control = [1000, wxButton, 'OK', '', 'button5', 23, 34, -1, -1,\
    'my_resource'],\
    control = [1001, wxStaticText, 'A Label', '', 'message7', 166, 61, -1,\
    -1, 'my_resource'],\
    control = [1002, wxTextCtrl, 'Text', 'wxTE_MULTITEXT', 'text8', 24,
```

```
110, -1, -1])).";
```

As you can see, C++-style comments are allowed, and apparently include files are supported too: but this is a special case, where the included file is a file of defines shared by the C++ application code and resource file to relate identifiers (such as `FILE_OPEN`) to integers.

Each *resource object* is of standard *wxExpr* (p. 385) syntax, that is, an object name such as **dialog** or **icon**, then an open parenthesis, a list of comma-delimited attribute/value pairs, a closing parenthesis, and a full stop. Backslashes are required to escape newlines, for the benefit of C++ syntax. If double quotation marks are used to delimit strings, they need to be escaped with backslash within a C++ string (so it is easier to use single quotation marks instead).

A note on string syntax: A string that begins with an alphabetic character, and contains only alphanumeric characters, hyphens and underscores, need not be quoted at all. Single quotes and double quotes may be used to delimit more complex strings. In fact, single-quoted and no-quoted strings are actually called *words*, but are treated as strings for the purpose of the resource system.

A resource file like this is typically included in the application main file, as if it were a normal C++ file. This eliminates the need for a separate resource file to be distributed alongside the executable. However, the resource file can be dynamically loaded if desired (useful for non-C++ languages such as Python).

Once included, the resources need to be 'parsed' (interpreted), because so far the data is just a number of static string variables. The function `::wxResourceParseData` is called early on in initialization of the application (usually in `wxApp::OnInit`) with a variable as argument. This may need to be called a number of times, one for each variable. However, more than one resource 'object' can be stored in one string variable at a time, so you can get all your resources into one variable if you want to.

`::wxResourceParseData` parses the contents of the resource, ready for use by functions such as `::wxResourceCreateBitmap` and `wxPanel::LoadFromResource`.

If a `wxWindows` resource object (such as a bitmap resource) refers to a C++ data structure, such as static XPM data, a further call (`::wxResourceRegisterBitmapData`) needs to be made on initialization to tell `wxWindows` about this data. The `wxWindows` resource object will refer to a string identifier, such as 'project_data' in the example file above. This identifier will be looked up in a table to get the C++ static data to use for the bitmap or icon.

In the C++ fragment below, the `WXR` resource file is included, and appropriate resource initialization is carried out in `OnInit`. Note that at this stage, no actual `wxWindows` dialogs, menus, bitmaps or icons are created; their 'templates' are merely being set up for later use.

```
/*
 * File:      project.cpp
 * Purpose:  main application module
 */
```

```

#include "wx/wx.h"
#include "project.h"

// Includes the dialog, menu etc. resources
#include "project.wxr"

// Includes XPM data
#include "project.xpm"

IMPLEMENT_APP(AppClass)

// Called to initialize the program
bool AppClass::OnInit()
{
    wxResourceRegisterBitmapData("project_data", project_bits,
project_width, project_height);

    wxResourceParseData(menuBar11);
    wxResourceParseData(my_resource);
    wxResourceParseData(project_resource);
    wxResourceParseData(panel3);
    ...

    return TRUE;
}

```

The following code shows a dialog:

```

// project.wxr contains dialog1
MyDialog *dialog = new MyDialog;
if (dialog->LoadFromResource(this, "dialog1"))
{
    wxTextCtrl *text = (wxTextCtrl *)wxFindWindowByName("text3",
dialog);
    if (text)
        text->SetValue("wxWindows resource demo");
    dialog->ShowModal();
}
dialog->Destroy();

```

Please see also the resource sample.

Dialog resource format

A dialog resource object may be used for either panels or dialog boxes, and consists of the following attributes. In the following, a *font specification* is a list consisting of point size, family, style, weight, underlined, optional facename.

Attribute	Value
id	The integer identifier of the resource.
name	The name of the resource.
style	Optional dialog box or panel window style.
title	The title of the dialog box (unused if a panel).
.modal	Whether modal: 1 if modal, 0 if modeless, absent if a panel resource.

<code>use_dialog_units</code>	If 1, use dialog units (dependent on the dialog font size) for control sizes and positions.
<code>use_system_defaults</code>	If 1, override colours and fonts to use system settings instead.
<code>button_font</code>	The font used for control buttons: a list comprising point size (integer), family (string), font style (string), font weight (string) and underlining (0 or 1).
<code>label_font</code>	The font used for control labels: a list comprising point size (integer), family (string), font style (string), font weight (string) and underlining (0 or 1). Now obsolete; use <code>button_font</code> instead.
<code>x</code>	The x position of the dialog or panel.
<code>y</code>	The y position of the dialog or panel.
<code>width</code>	The width of the dialog or panel.
<code>height</code>	The height of the dialog or panel.
<code>background_colour</code>	The background colour of the dialog or panel.
<code>label_colour</code>	The default label colour for the children of the dialog or panel. Now obsolete; use <code>button_colour</code> instead.
<code>button_colour</code>	The default button text colour for the children of the dialog or panel.

Then comes zero or more attributes named 'control' for each control (panel item) on the dialog or panel. The value is a list of further elements. In the table below, the names in the first column correspond to the first element of the value list, and the second column details the remaining elements of the list. Note that titles for some controls are obsolete (they don't have titles), but the syntax is retained for backward compatibility.

Control	Values
<code>wxButton</code>	id (integer), title (string), window style (string), name (string), x, y, width, height, button bitmap resource (optional string), button font spec
<code>wxCheckBox</code>	id (integer), title (string), window style (string), name (string), x, y, width, height, default value (optional integer, 1 or 0), label font spec
<code>wxChoice</code>	id (integer), title (string), window style (string), name (string), x, y, width, height, values (optional list of strings), label font spec, button font spec
<code>wxComboBox</code>	id (integer), title (string), window style (string), name (string), x, y, width, height, default text value, values (optional list of strings), label font spec, button font spec
<code>wxGauge</code>	id (integer), title (string), window style

wxStaticBox	(string), name (string), x, y, width, height, value (optional integer), range (optional integer), label font spec, button font spec
wxListBox	id (integer), title (string), window style (string), name (string), x, y, width, height, values (optional list of strings), multiple (optional string, wxSINGLE or wxMULTIPLE), label font spec, button font spec
wxStaticText	id (integer), title (string), window style (string), name (string), x, y, width, height, message bitmap resource (optional string), label font spec
wxRadioBox	id (integer), title (string), window style (string), name (string), x, y, width, height, values (optional list of strings), number of rows or cols, label font spec, button font spec
wxRadioButton	id (integer), title (string), window style (string), name (string), x, y, width, height, default value (optional integer, 1 or 0), label font spec
wxScrollBar	id (integer), title (string), window style (string), name (string), x, y, width, height, value (optional integer), page length (optional integer), object length (optional integer), view length (optional integer)
wxSlider	id (integer), title (string), window style (string), name (string), x, y, width, height, value (optional integer), minimum (optional integer), maximum (optional integer), label font spec, button font spec
wxTextCtrl	id (integer), title (string), window style (string), name (string), x, y, width, height, default value (optional string), label font spec, button font spec

Menubar resource format

A menubar resource object consists of the following attributes.

Attribute	Value
name	The name of the menubar resource.
menu	A list containing all the menus, as detailed below.

The value of the **menu** attribute is a list of menu item specifications, where each menu item specification is itself a list comprising:

- title (a string)
- menu item identifier (a string or non-zero integer, see below)
- help string (optional)
- 0 or 1 for the 'checkable' parameter (optional)
- optionally, further menu item specifications if this item is a pulldown menu.

If the menu item specification is the empty list ([]), this is interpreted as a menu separator.

If further (optional) information is associated with each menu item in a future release of wxWindows, it will be placed after the help string and before the optional pulldown menu specifications.

Note that the menu item identifier must be an integer if the resource is being included as C++ code and then parsed on initialisation. Unfortunately, #define substitution is not performed inside strings, and therefore the program cannot know the mapping. However, if the .WXR file is being loaded dynamically, wxWindows will attempt to replace string identifiers with #defined integers, because it is able to parse the included #defines.

Bitmap resource format

A bitmap resource object consists of a name attribute, and one or more **bitmap** attributes. There can be more than one of these to allow specification of bitmaps that are optimum for the platform and display.

- Bitmap name or filename.
- Type of bitmap; for example, wxBITMAP_TYPE_BMP_RESOURCE. See class reference under **wxBitmap** for a full list).
- Platform this bitmap is valid for; one of WINDOWS, X, MAC and ANY.
- Number of colours (optional).
- X resolution (optional).
- Y resolution (optional).

Icon resource format

An icon resource object consists of a name attribute, and one or more **icon** attributes. There can be more than one of these to allow specification of icons that are optimum for the platform and display.

- Icon name or filename.
- Type of icon; for example, wxBITMAP_TYPE_ICO_RESOURCE. See class

- reference under **wxBitmap** for a full list).
- Platform this bitmap is valid for; one of WINDOWS, X, MAC and ANY.
- Number of colours (optional).
- X resolution (optional).
- Y resolution (optional).

Resource format design issues

The .WXR file format is a recent addition and subject to change. The use of an ASCII resource file format may seem rather inefficient, but this choice has a number of advantages:

- Since it is C++ compatible, it can be included into an application's source code, eliminating the problems associated with distributing a separate resource file with the executable. However, it can also be loaded dynamically from a file, which will be required for non-C++ programs that use wxWindows.
- No extra binary file format and separate converter need be maintained for the wxWindows project (although others are welcome to add the equivalent of the Windows 'rc' resource parser and a binary format).
- It would be difficult to append a binary resource component onto an executable in a portable way.
- The file format is essentially the *wxExpr* (p. 385) object format, for which a parser already exists, so parsing is easy. For those programs that use wxExpr anyway, the size overhead of the parser is minimal.

The disadvantages of the approach include:

- Parsing adds a small execution overhead to program initialization.
- Under 16-bit Windows especially, global data is at a premium. Using a .RC resource table for some wxWindows resource data may be a partial solution, although .RC strings are limited to 255 characters.
- Without a resource preprocessor, it is not possible to substitute integers for identifiers (so menu identifiers have to be written as integers in the resource object, in addition to providing #defines for application code convenience).

Compiling the resource system

To enable the resource system, set **wxUSE_WX_RESOURCES** to 1 in setup.h.

Scrolling overview

Classes: *wxWindow* (p. 1184), *wxScrolledWindow* (p. 911), *wxIcon* (p. 558), *wxScrollBar* (p. 903).

Scrollbars come in various guises in `wxWindows`. 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. 1184) 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. 911) (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. 479) 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. 1227) gives a clue about the way a scrollbar is modeled. This function takes the following arguments:

orientation	Which scrollbar: <code>wxVERTICAL</code> or <code>wxHORIZONTAL</code> .
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 `wxWindow::OnSize` (p. 1216) event handler function.

Bitmaps and icons overview

Classes: *wxBitmap* (p. 54), *wxBitmapHandler* (p. 66), *wxIcon* (p. 558), *wxCursor* (p. 184).

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. 678)). This enables the bitmap to be copied to a window or memory device context using *wxDC::Blit* (p. 281), or to be used as a drawing surface. The **wxToolBarSimple** class is implemented using bitmaps, and the toolbar demo shows one of the toolbar bitmaps being used for drawing a miniature version of the graphic which appears on the main window.

See *wxMemoryDC* (p. 678) for an example of drawing onto a bitmap.

The following shows the conditional compilation required to load a bitmap under Unix and in Windows. The alternative is to use the string version of the bitmap constructor, which loads a file under Unix and a resource or file under Windows, but has the disadvantage of requiring the XPM icon file to be available at run-time.

```
#if defined(__WXGTK__) || defined(__WXMOTIF__)
#include "mondrian.xpm"
#endif
```

A macro, *wxICON* (p. 1292), 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. 1291) macro which allows to create the bitmaps in much the same way as *wxICON* (p. 1292) 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 *_xpmsuffix*). 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*, *wxBe*, ...) and are more short and clear than versions with *#ifdefs*.

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. 565) to load the data, and then converting it to *wxBitmap* form. Note that using *wxImage* is the preferred way to load images in *wxWindows*, 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. 565) 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. 565) class.

Under wxMotif, 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. 565) 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. 565) class.

Under wxMotif, 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. 565) 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 addition to stock cursors):

- None (stock cursors only).

Under wxMotif, 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. 57). For example:

```
#include <wx/xpmhand.h>
...
// Initialisation
wxBitmap::AddHandler(new wxXPMFileHandler);
wxBitmap::AddHandler(new wxXPMDDataHandler);
...
```

Assuming the handlers have been written correctly, you should now be able to load and save XPM files using the usual wxBitmap API.

Note: bitmap handlers are not implemented on all platforms. Currently, the above is only necessary on Windows, to save the extra overhead of formats that may not be necessary (if you don't use them, they are not linked into the executable). Unix platforms have XPM capability built-in (where supported).

Also, just because a handler (such as a PNG handler) is not present does not mean that wxBitmap does not support that file format. If wxBitmap fails to find a suitable handler, the file-loading capabilities of wxImage are used instead.

Device context overview

Classes: *wxDC* (p. 280), *wxPostScriptDC* (p. 786), *wxMetafileDC* (p. 712), *wxMemoryDC* (p. 678), *wxPrinterDC* (p. 806), *wxScreenDC* (p. 901), *wxClientDC* (p.

120), *wxPaintDC* (p. 759), *wxWindowDC* (p. 1234).

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. 901), *wxClientDC* (p. 120), *wxPaintDC* (p. 759), and *wxWindowDC* (p. 1234). 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 *wxWindow::OnPaint* (p. 1214) event.
- **wxPaintDC.** Use this to paint on the client area of a window, but *only* from within an *wxWindow::OnPaint* (p. 1214) event.
- **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. 434)

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 <i>font overview</i> (p. 1393) 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.

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 id 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.

Font encoding overview

`wxWindows` 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. 1340).

Font encoding support is assured by several classes: `wxFont` (p. 434) itself, but also `wxFontEnumerator` (p. 445) and `wxFontMapper` (p. 448). `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. 436). 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).
<code>wxFONTENCODING_ISO8859_1..15</code>	ISO8859 family encodings which are usually used by all non-Microsoft operating systems
<code>wxFONTENCODING_KOI8</code>	Standard Cyrillic encoding for the Internet (but see also <code>wxFONTENCODING_ISO8859_5</code> and <code>wxFONTENCODING_CP1251</code>)
<code>wxFONTENCODING_CP1250</code>	Microsoft analogue of ISO8859-2
<code>wxFONTENCODING_CP1251</code>	Microsoft analogue of ISO8859-5
<code>wxFONTENCODING_CP1252</code>	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) encoding:ws 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, not Win32, systems).

To allow to see clearer in this, `wxFontEnumerator` (p. 445) 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 (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. 445) will never return one, even if the user has installed a KOI8 font on his system.

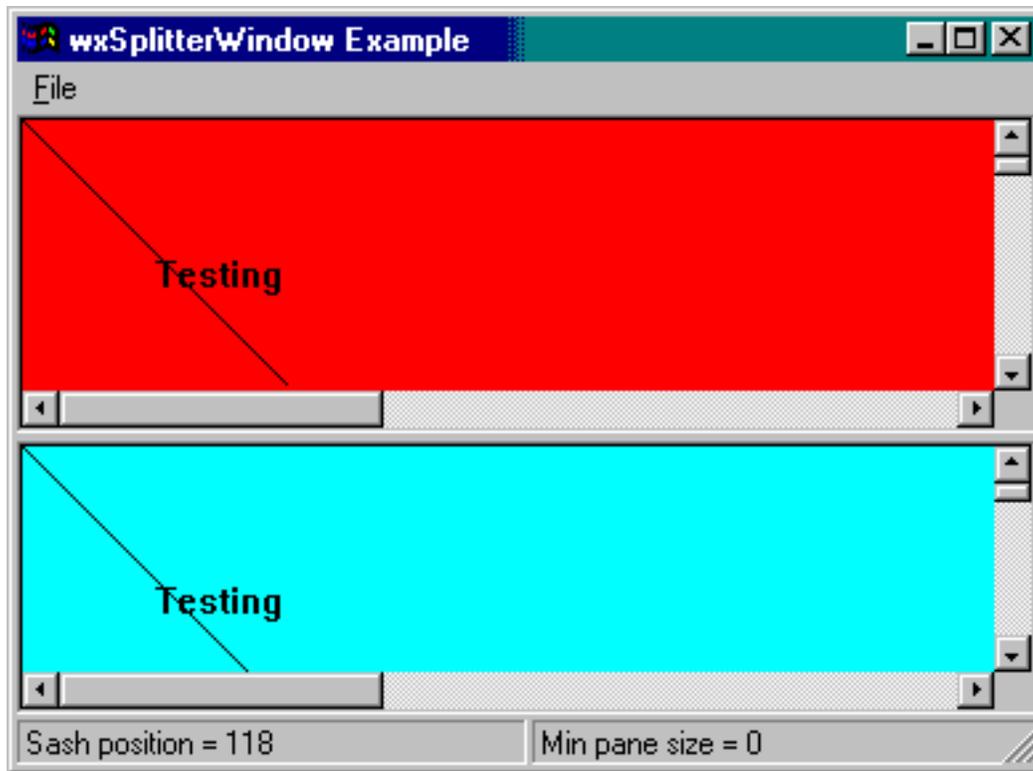
To solve this problem, a `wxFontMapper` (p. 448) class is provided. This class stores the mapping between the encodings and the font face names which support them in `wxConfig` (p. 1358) object. Of course, it would be fairly useless if it tried to determine these mappings by itself, so, instead, it (optionally) ask the user and remember 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. 1323), please refer to it and the documentation of the classes mentioned here for further explanations.

wxSplitterWindow overview

Classes: *wxSplitterWindow* (p. 973)

The following screenshot shows the appearance of a splitter window with a vertical split.



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. 1134), *wxImageList* (p. 584)

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*.

The items text and image may be retrieved and changed with *GetItemText* (p. 1141)/*SetItemText* (p. 1148) and *GetItemImage* (p. 1141)/*SetItemImage* (p. 1148). 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. 1148)/*GetItemSelectedImage* (p. 1143) 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. 1145), *IsVisible* (p. 1145), *IsBold* (p. 1145) and *IsExpanded* (p. 1145). Only one item at a time may be selected, selecting another one (with *SelectItem* (p. 1146)) 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. 1147) to prevent the tree from deleting the pointer second time). The associated data may be retrieved with *GetItemData()* (p. 1141) 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. 1137)). 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. 1140) and to *GetNextChild* (p. 1142) 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. 1148) which uses the user-defined comparison function *OnCompareItems* (p. 1146) (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. 1144) and editing of the tree item labels in place (see *EditLabel* (p. 1138)).

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 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. 630), *wxImageList* (p. 584)

Sorry, this topic has yet to be written.

wxImageList overview

Classes: *wxImageList* (p. 584)

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. 736), *wxListCtrl* (p. 630), *wxTreeCtrl* (p. 630) and some other control classes.

Common dialogs overview

Classes: *wxColourDialog* (p. 142), *wxFontDialog* (p. 444), *wxPrintDialog* (p. 797), *wxFileDialog* (p. 407), *wxDirDialog* (p. 325), *wxTextEntryDialog* (p. 1089), *wxMessageDialog* (p. 709), *wxSingleChoiceDialog* (p. 919), *wxMultipleChoiceDialog* (p. 730)

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 that 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. 142), *wxColourData* (p. 138)

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. 444), *wxFontData* (p. 441)

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 wxWindows 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 full range of Windows fonts cannot be used in wxWindows at present.

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`.

In both cases, the application is responsible for deleting the new font returned from calling `wxFontDialog::Show` (if any). This returned font is guaranteed to be a new object and not one currently in use in the application.

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. 797), *wxPrintData* (p. 792)

This class represents the print and print setup common dialogs. You may obtain a *wxPrinterDC* (p. 806) device context from a successfully dismissed print dialog.

The `samples/printing` example shows how to use it: see *Printing overview* (p. 1419) for an excerpt from this example.

wxFileDialog overview

Classes: *wxFileDialog* (p. 407)

Pops up a file selector box. In Windows, 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 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. 325)

This dialog shows a directory selector dialog, allowing the user to select a single directory.

wxTextEntryDialog overview

Classes: *wxTextEntryDialog* (p. 1089)

This is a dialog with a text entry field. The value that the user entered is obtained using *wxTextEntryDialog::GetValue* (p. 1090).

wxMessageDialog overview

Classes: *wxMessageDialog* (p. 709)

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. 710) indicates which button the user pressed.

wxSingleChoiceDialog overview

Classes: *wxSingleChoiceDialog* (p. 919)

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.

wxMultipleChoiceDialog overview

Classes: *wxMultipleChoiceDialog* (p. 730)

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. 351), *wxView* (p. 1179), *wxDocTemplate* (p. 345), *wxDocManager* (p. 332), *wxDocParentFrame* (p. 344), *wxDocChildFrame* (p. 330), *wxDocMDIParentFrame* (p. 343), *wxDocMDIChildFrame* (p. 341), *wxCommand* (p. 151), *wxCommandProcessor* (p. 158)

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`), routing non-application-specific identifiers to the base frame's `OnMenuCommand`.
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. 807) 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 `wxWindows` *predefined command identifiers* (p. 1407) in your menus.

wxDocument overview

Document/view framework overview (p. 1402)

Class: *wxDocument* (p. 351)

The `wxDocument` class can be used to model an application's file-based data. It is part of the document/view framework supported by `wxWindows`, and cooperates with the `wxView` (p. 1179), `wxDocTemplate` (p. 345) and `wxDocManager` (p. 332) 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. 345) 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 `wxWindows` 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. 1402)

Class: `wxView` (p. 1179)

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 `wxWindows`, and cooperates with the `wxDocument` (p. 351), `wxDocTemplate` (p. 345) and `wxDocManager` (p. 332) 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. 345) 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 `wxWindows` 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. 1402)

Class: *wxDocTemplate* (p. 345)

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. Under non-Windows platforms, the user will be prompted for a list of templates before the file selector is shown, since most file selectors do not allow a choice of file filters.

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 *wxWindows*, and cooperates with the *wxView* (p. 1179), *wxDocument* (p. 351) and *wxDocManager* (p. 332) 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 *wxWindows* 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. C++ templates are not used anywhere in *wxWindows*.

wxDocManager overview

Document/view framework overview (p. 1402)

Class: *wxDocManager* (p. 332)

The *wxDocManager* class is part of the document/view framework supported by *wxWindows*, and cooperates with the *wxView* (p. 1179), *wxDocument* (p. 351) and *wxDocTemplate* (p. 345) 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. 1402)

Classes: *wxCommand* (p. 151), *wxCommandProcessor* (p. 158)

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. 1406) object to execute and store.

The *wxWindows* 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. 1402)

Classes: *wxCommandProcessor* (p. 158), *wxCommand* (p. 151)

`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. 1402)

Classes: *wxFileHistory* (p. 413), *wxDocManager* (p. 332)

`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.GetSelection() -
wxID_FILE1));
    if (f != "")
        (void)m_docManager->CreateDocument(f, wxDOC_SILENT);
}
```

wxWindows 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)

wxTab classes overview

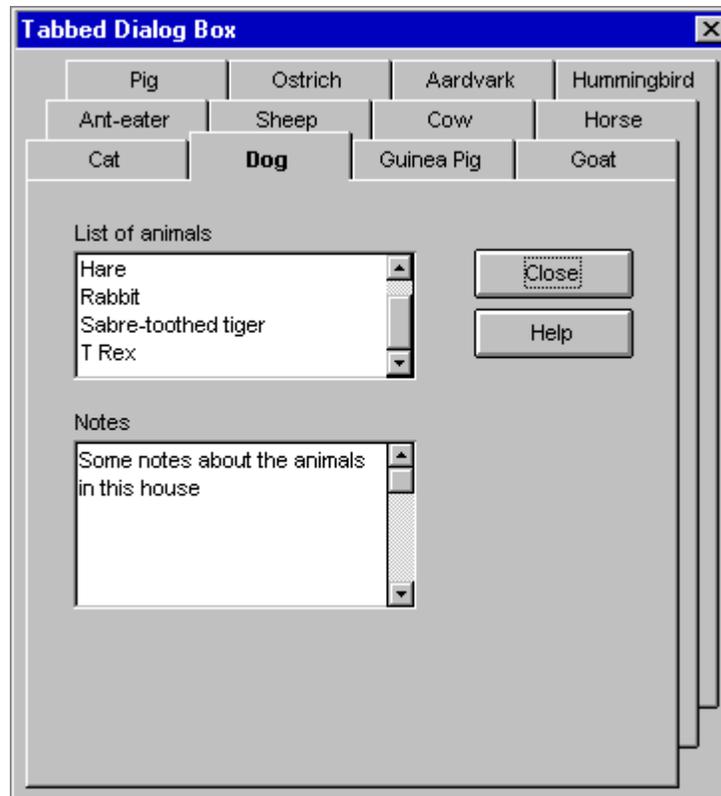
Classes: *wxTabView* (p. 1044), *wxPanelTabView* (p. 767), *wxTabbedPanel* (p. 1040), *wxTabbedDialog* (p. 1038), *wxTabControl* (p. 1041)

The tab classes provides a way to display rows of tabs (like file divider tabs), which can be used to switch between panels or other information. Tabs are most commonly used in dialog boxes where the number of options is too great to fit on one dialog.

Please note that the preferred class for programming tabbed windows is *wxNotebook* (p. 736). The old tab classes are retained for backward compatibility and also to implement *wxNotebook* on platforms that don't have native tab controls.

The appearance and behaviour of a wxTabbedDialog

The following screenshot shows the appearance of the sample tabbed dialog application.



By clicking on the tabs, the user can display a different set of controls. In the example, the Close and Help buttons remain constant. These two buttons are children of the main dialog box, whereas the other controls are children of panels which are shown and hidden according to which tab is active.

A tabbed dialog may have several layers (rows) of tabs, each being offset vertically and horizontally from the previous. Tabs work in columns, in that when a tab is pressed, it swaps place with the tab on the first row of the same column, in order to give the effect of displaying that tab. All tabs must be of the same width. This is a constraint of the implementation, but it also means that the user will find it easier to find tabs since there are distinct tab columns. On some tabbed dialog implementations, tabs jump around seemingly randomly because tabs have different widths. In this implementation, a tab can always be found on the same column.

Tabs are always drawn along the top of the view area; the implementation does not allow for vertical tabs or any other configuration.

Using tabs

The tab classes provide facilities for switching between contexts by means of 'tabs', which look like file divider tabs.

You must create both a *view* to handle the tabs, and a *window* to display the tabs and related information. The `wxTabbedDialog` and `wxTabbedPanel` classes are provided for convenience, but you could equally well construct your own window class and derived tab view.

If you wish to display a tabbed dialog - the most common use - you should follow these steps.

1. Create a new `wxTabbedDialog` class, and any buttons you wish always to be displayed (regardless of which tab is active).
2. Create a new `wxPanelTabView`, passing the dialog as the first argument.
3. Set the view rectangle with `wxTabView::SetViewRect` (p. 1052), to specify the area in which child panels will be shown. The tabs will sit on top of this view rectangle.
4. Call `wxTabView::CalculateTabWidth` (p. 1045) to calculate the width of the tabs based on the view area. This is optional if, for example, you have one row of tabs which does not extend the full width of the view area.
5. Call `wxTabView::AddTab` (p. 1045) for each of the tabs you wish to create, passing a unique identifier and a tab label.
6. Construct a number of windows, one for each tab, and call `wxPanelTabView::AddTabWindow` (p. 768) for each of these, passing a tab identifier and the window.
7. Set the tab selection.
8. Show the dialog.

Under Motif, you may also need to size the dialog just before setting the tab selection, for unknown reasons.

Some constraints you need to be aware of:

- All tabs must be of the same width.
- Omit the `wxTAB_STYLE_COLOUR_INTERIOR` flag to ensure that the dialog background and tab backgrounds match.

Example

The following fragment is taken from the file `test.cpp`.

```
void MyDialog::Init(void)
{
    int dialogWidth = 365;
    int dialogHeight = 390;

    wxButton *okButton = new wxButton(this, wxID_OK, "Close", wxPoint(100,
330), wxSize(80, 25));
    wxButton *cancelButton = new wxButton(this, wxID_CANCEL, "Cancel",
wxPoint(185, 330), wxSize(80, 25));
    wxButton *helpButton = new wxButton(this, wxID_HELP, "Help",
wxPoint(270, 330), wxSize(80, 25));
    okButton->SetDefault();

    // Note, omit the wxTAB_STYLE_COLOUR_INTERIOR, so we will guarantee a
match
    // with the panel background, and save a bit of time.
    wxPanelTabView *view = new wxPanelTabView(this, wxTAB_STYLE_DRAW_BOX);

    wxRectangle rect;
```

```

    rect.x = 5;
    rect.y = 70;
    // Could calculate the view width from the tab width and spacing,
    // as below, but let's assume we have a fixed view width.
    // rect.width = view->GetTabWidth()*4 + 3*view-
>GetHorizontalTabSpacing();
    rect.width = 326;
    rect.height = 250;

    view->SetViewRect(rect);

    // Calculate the tab width for 4 tabs, based on a view width of 326
and
    // the current horizontal spacing. Adjust the view width to exactly
fit
    // the tabs.
    view->CalculateTabWidth(4, TRUE);

    if (!view->AddTab(TEST_TAB_CAT,          wxString("Cat")))
        return;

    if (!view->AddTab(TEST_TAB_DOG,          wxString("Dog")))
        return;
    if (!view->AddTab(TEST_TAB_GUINEAPIG,    wxString("Guinea Pig")))
        return;
    if (!view->AddTab(TEST_TAB_GOAT,         wxString("Goat")))
        return;
    if (!view->AddTab(TEST_TAB_ANTEATER,     wxString("Ant-eater")))
        return;
    if (!view->AddTab(TEST_TAB_SHEEP,        wxString("Sheep")))
        return;
    if (!view->AddTab(TEST_TAB_COW,          wxString("Cow")))
        return;
    if (!view->AddTab(TEST_TAB_HORSE,        wxString("Horse")))
        return;
    if (!view->AddTab(TEST_TAB_PIG,          wxString("Pig")))
        return;
    if (!view->AddTab(TEST_TAB_OSTRICH,      wxString("Ostrich")))
        return;
    if (!view->AddTab(TEST_TAB_AARDVARK,     wxString("Aardvark")))
        return;
    if (!view->AddTab(TEST_TAB_HUMMINGBIRD, wxString("Hummingbird")))
        return;

    // Add some panels
    wxPanel *panell = new wxPanel(this, -1, wxPoint(rect.x + 20, rect.y +
10), wxSize(290, 220), wxTAB_TRAVERSAL);
    (void)new wxButton(panell, -1, "Press me", wxPoint(10, 10));
    (void)new wxTextCtrl(panell, -1, "1234", wxPoint(10, 40), wxSize(120,
150));

    view->AddTabWindow(TEST_TAB_CAT, panell);

    wxPanel *panel2 = new wxPanel(this, -1, wxPoint(rect.x + 20, rect.y +
10), wxSize(290, 220));

    wxString animals[] = { "Fox", "Hare", "Rabbit", "Sabre-toothed tiger",
"T Rex" };
    (void)new wxListBox(panel2, -1, wxPoint(5, 5), wxSize(170, 80), 5,
animals);

    (void)new wxTextCtrl(panel2, -1, "Some notes about the animals in this
house", wxPoint(5, 100), wxSize(170, 100)),
        wxTE_MULTILINE;

```

```
view->AddTabWindow(TEST_TAB_DOG, panel2);

// Don't know why this is necessary under Motif...
#ifdef wx_motif
    this->SetSize(dialogWidth, dialogHeight-20);
#endif

view->SetTabSelection(TEST_TAB_CAT);

this->Centre(wxBOTH);
}
```

wxTabView overview

Classes: *wxTabView* (p. 1044), *wxPanelTabView* (p. 767)

A *wxTabView* manages and draws a number of tabs. Because it is separate from the tabbed window implementation, it can be reused in a number of contexts. This library provides tabbed dialog and panel classes to use with the *wxPanelTabView* class, but an application could derive other kinds of view from *wxTabView*.

For example, a help application might draw a representation of a book on a window, with a row of tabs along the top. The new tab view class might be called *wxCanvasTabView*, for example, with the *wxBookCanvas* posting the *OnEvent* function to the *wxCanvasTabView* before processing further, application-specific event processing.

A window class designed to work with a view class must call the view's *OnEvent* and *Draw* functions at appropriate times.

Toolbar overview

Classes: *wxToolBar* (p. 1117)

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, *wxWindows* 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 wxWindows 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 1.6.2.2 2000/04/27 14:32:59 VS 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, wxWindows");

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("wxWindows 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

wxGrid is a class for displaying and editing tabular information.

To use wxGrid, include the wxgrid.h header file and link with the wxGrid library. Create a wxGrid object, or, if you need to override some default behaviour, create an object of a class derived from wxGrid. You need to call CreateGrid before there are any cells in the grid.

All row and column positions start from zero, and dimensions are in pixels.

If you make changes to row or column dimensions, call UpdateDimensions and then AdjustScrollbars. If you make changes to the grid appearance (such as a change of cell background colour or font), call Refresh for the changes to be shown.

Example

The following fragment is taken from the file `samples/grid/test.cpp`. Note the call to `UpdateDimensions`, which is required if the application has changed any dimensions such as column width or row height. You may also need to call `AdjustScrollbars`. In this case, `AdjustScrollbars` isn't necessary because it will be called by `wxGrid::OnSize` which is invoked when the window is first displayed.

```
// Make a grid
frame->grid = new wxGrid(frame, 0, 0, 400, 400);

frame->grid->CreateGrid(10, 8);
frame->grid->SetColumnWidth(3, 200);
frame->grid->SetRowHeight(4, 45);
frame->grid->SetCellValue("First cell", 0, 0);
frame->grid->SetCellValue("Another cell", 1, 1);
frame->grid->SetCellValue("Yet another cell", 2, 2);
frame->grid->SetCellTextFont(wxTheFontList->FindOrCreateFont(12,
wxROMAN, wxITALIC, wxNORMAL), 0, 0);
frame->grid->SetCellTextColour(*wxRED, 1, 1);
frame->grid->SetCellBackgroundColour(*wxCYAN, 2, 2);
frame->grid->UpdateDimensions();
```

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 wxWindows programmer, implementing this feature is extremely easy. To show a tip, it is enough to just call `wxShowTip` (p. 1261) 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. 1116). It has to implement one pure virtual function of the base class: `GetTip` (p. 1117). In the case of the tip provider created by `wxCreateFileTipProvider` (p. 1256), 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.

Finally, 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. 1117) and the flag telling whether to show the tips at startup at all.

Printing overview

Classes: *wxPrintout* (p. 807), *wxPrinter* (p. 803), *wxPrintPreview* (p. 810), *wxPrinterDC* (p. 806), *wxPrintDialog* (p. 797), *wxPrintData* (p. 792), *wxPrintDialogData* (p. 799), *wxPageSetupDialog* (p. 758), *wxPageSetupDialogData* (p. 752)

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 *wxWindows* 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. 1402) 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 *wxWindows*. 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", 100, 100, 600, 650);
    frame->Centre(wxBOTH);
    frame->Initialize();
    frame->Show(TRUE);
    break;
}

```

```
case WXPRIINT_PRINT_SETUP:
{
    wxPrintDialog printerDialog(this);
    printerDialog.GetPrintData().SetSetupDialog(TRUE);
    printerDialog.Show(TRUE);
    break;
}
```

Multithreading overview

Classes: *wxThread* (p. 1101), *wxMutex* (p. 730), *wxCriticalSection* (p. 178), *wxCondition* (p. 160)

wxWindows provides a complete set of classes encapsulating objects necessary in multithreaded (MT) programs: the *thread* (p. 1101) class itself and different synchronization objects: *mutexes* (p. 730) and *critical sections* (p. 178) with *conditions* (p. 160). The thread API in *wxWindows* 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 a 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. 557) or call *wxYield()* (p. 1285) 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 threads, use *wxEvtHandler::AddPendingEvent* (p. 379) or its short version *wxPostEvent* (p. 1281). These functions have thread safe implementation so that they can be used as they are for sending event from one thread to another.

Drag and drop overview

Classes: *wxDataObject* (p. 196), *wxTextDataObject* (p. 1083), *wxDropSource* (p. 365), *wxDropTarget* (p. 368), *wxTextDropTarget* (p. 1090), *wxFileDropTarget* (p. 412)

Note that `wxUSE_DRAG_AND_DROP` must be defined in `setup.h` in order to use drag and drop in `wxWindows`.

See also: *wxDataObject overview* (p. 1422) and *DnD sample* (p. 1323)

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. 196) and identify its format(s) using the *wxDataFormat* (p. 194) class.

To be a *drag source*, i.e. to provide the data which may be dragged by 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 dragging process (typically in response to a mouse click) you must call *wxDropSource::DoDragDrop* (p. 367) 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 release the mouse button (unless you override *GiveFeedback* (p. 367) 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 XnD protocol under X Windows), the corresponding *wxDropTarget* (p. 368) 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. 368)):

```
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 user you should follow the instructions below:

- **Initialization:** For a window to be drop target, it needs to have an associated *wxDropTarget* (p. 368) object. Normally, you will call *wxWindow::SetDropTarget* (p. 1224) during window creation associating you drop target with it. You must derive a class from *wxDropTarget* and override its pure virtual methods. Alternatively, you may derive from *wxTextDropTarget* (p. 1090) or *wxFileDropTarget* (p. 412) and override their *OnDropText()* or *OnDropFiles()* method.
- **Drop:** When the user releases the mouse over a window, *wxWindows* queries the associated *wxDropTarget* object if it accepts the data. For this, a *wxDataObject* (p. 196) 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. 369) 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 drop. There is currently no way for the drop target to change this return code.

wxDataObject overview

Classes: *wxDataObject* (p. 196), *wxClipboard* (p. 121), *wxDataFormat* (p. 194), *wxDropSource* (p. 365), *wxDropTarget* (p. 368)

See also: *Drag and drop overview* (p. 1420) and *DnD sample* (p. 1323)

This overview discusses data transfer through clipboard or drag and drop. In *wxWindows*, 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. 196) 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. 196) containing the data to be transferred. Then it should either pass it to the clipboard using *SetData* (p. 124) function or to *wxDropSource* (p. 365) and call *DoDragDrop* (p. 367) 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. 367) 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. 367) 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. 196) derived class which supports the data formats you need and pass it as argument to *wxClipboard::GetData* (p. 123). 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. 369) virtual function will be called when a data object is dropped, from which the data itself may be requested by calling *wxDropTarget::GetData* (p. 369) method which fills the data object.

Database classes overview

Classes: *wxDatabase* (p. 188), *wxRecordSet* (p. 872), *wxQueryCol* (p. 851), *wxQueryField* (p. 854)

Note that more sophisticated ODBC classes are provided by the Remstar database classes: please see the separate HTML and Word documentation.

wxWindows provides a set of classes for accessing a subset of Microsoft's ODBC (Open Database Connectivity) product. Currently, this wrapper is available under MS Windows only, although ODBC may appear on other platforms, and a generic or product-specific SQL emulator for the ODBC classes may be provided in *wxWindows* at a later date.

ODBC presents a unified API (Application Programmer's Interface) to a wide variety of databases, by interfacing indirectly to each database or file via an ODBC driver. The language for most of the database operations is SQL, so you need to learn a small amount of SQL as well as the *wxWindows* ODBC wrapper API. Even though the

databases may not be SQL-based, the ODBC drivers translate SQL into appropriate operations for the database or file: even text files have rudimentary ODBC support, along with dBASE, Access, Excel and other file formats.

The run-time files for ODBC are bundled with many existing database packages, including MS Office. The required header files, `sql.h` and `sql.h`, are bundled with several compilers including MS VC++ and Watcom C++. The only other way to obtain these header files is from the ODBC SDK, which is only available with the MS Developer Network CD-ROMs -- at great expense. If you have `odbc.dll`, you can make the required import library `odbc.lib` using the tool 'implib'. You need to have `odbc.lib` in your compiler library path.

The minimum you need to distribute with your application is `odbc.dll`, which must go in the Windows system directory. For the application to function correctly, ODBC drivers must be installed on the user's machine. If you do not use the database classes, `odbc.dll` will be loaded but not called (so ODBC does not need to be setup fully if no ODBC calls will be made).

A sample is distributed with `wxWindows` in `samples/odbc`. You will need to install the sample dbf file as a data source using the ODBC setup utility, available from the control panel if ODBC has been fully installed.

Procedures for writing an ODBC application

You first need to create a `wxDatabase` object. If you want to get information from the ODBC manager instead of from a particular database (for example using `wxRecordSet::GetDataSources` (p. 877)), then you do not need to call `wxDatabase::Open` (p. 192). If you do wish to connect to a datasource, then call `wxDatabase::Open`. You can reuse your `wxDatabase` object, calling `wxDatabase::Close` and `wxDatabase::Open` multiple times.

Then, create a `wxRecordSet` object for retrieving or sending information. For ODBC manager information retrieval, you can create it as a dynaset (retrieve the information as needed) or a snapshot (get all the data at once). If you are going to call `wxRecordSet::ExecuteSQL` (p. 875), you need to create it as a snapshot. Dynaset mode is not yet implemented for user data.

Having called a function such as `wxRecordSet::ExecuteSQL` or `wxRecordSet::GetDataSources`, you may have a number of records associated with the recordset, if appropriate to the operation. You can now retrieve information such as the number of records retrieved and the actual data itself. Use `wxRecordSet::GetFieldData` (p. 878) or `wxRecordSet::GetFieldDataPtr` (p. 878) to get the data or a pointer to it, passing a column index or name. The data returned will be for the current record. To move around the records, use `wxRecordSet::MoveNext` (p. 883), `wxRecordSet::MovePrev` (p. 883) and associated functions.

You can use the same recordset for multiple operations, or delete the recordset and create a new one.

Note that when you delete a `wxDatabase`, any associated recordsets also get deleted,

so beware of holding onto invalid pointers.

wxDatabase overview

Class: *wxDatabase* (p. 188)

Every database object represents an ODBC connection. To do anything useful with a database object you need to bind a *wxRecordSet* object to it. All you can do with *wxDatabase* is opening/closing connections and getting some info about it (users, passwords, and so on).

See also

Database classes overview (p. 1423)

wxQueryCol overview

Class: *wxQueryCol* (p. 851)

Every data column is represented by an instance of this class. It contains the name and type of a column and a list of *wxQueryFields* where the real data is stored. The links to user-defined variables are stored here, as well.

See also

Database classes overview (p. 1423)

wxQueryField overview

Class: *wxQueryField* (p. 854)

As every data column is represented by an instance of the class *wxQueryCol*, every data item of a specific column is represented by an instance of *wxQueryField*. Each column contains a list of *wxQueryFields*. If *wxRecordSet* is of the type *wxOPEN_TYPE_DYNASET*, there will be only one field for each column, which will be updated every time you call functions like *wxRecordSet::Move* or *wxRecordSet::GoTo*. If *wxRecordSet* is of the type *wxOPEN_TYPE_SNAPSHOT*, all data returned by an ODBC function will be loaded at once and the number of *wxQueryField* instances for each column will depend on the number of records.

See also

Database classes overview (p. 1423)

wxRecordSet overview

Class: *wxRecordSet* (p. 872)

Each *wxRecordSet* represents a database query. You can make multiple queries at a time by using multiple *wxRecordSets* with a *wxDatabase* or you can make your queries in sequential order using the same *wxRecordSet*.

See also

Database classes overview (p. 1423)

ODBC SQL data types

These are the data types supported in ODBC SQL. Note that there are other, extended level conformance types, not currently supported in *wxWindows*.

CHAR(<i>n</i>)	A character string of fixed length <i>n</i> .
VARCHAR(<i>n</i>)	A varying length character string of maximum length <i>n</i> .
LONG VARCHAR(<i>n</i>)	A varying length character string: equivalent to VARCHAR for the purposes of ODBC.
DECIMAL(<i>p</i> , <i>s</i>)	An exact numeric of precision <i>p</i> and scale <i>s</i> .
NUMERIC(<i>p</i> , <i>s</i>)	Same as DECIMAL.
SMALLINT	A 2 byte integer.
INTEGER	A 4 byte integer.
REAL	A 4 byte floating point number.
FLOAT	An 8 byte floating point number.
DOUBLE PRECISION	Same as FLOAT.

These data types correspond to the following ODBC identifiers:

SQL_CHAR	A character string of fixed length.
SQL_VARCHAR	A varying length character string.
SQL_DECIMAL	An exact numeric.
SQL_NUMERIC	Same as SQL_DECIMAL.
SQL_SMALLINT	A 2 byte integer.
SQL_INTEGER	A 4 byte integer.
SQL_REAL	A 4 byte floating point number.
SQL_FLOAT	An 8 byte floating point number.
SQL_DOUBLE	Same as SQL_FLOAT.

See also

Database classes overview (p. 1423)

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. 1423)

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: *wxDDEServer* (p. 303), *wxDDEConnection* (p. 299), *wxDDEClient* (p. 298), *wxTCPServer* (p. 1067), *wxTCPConnection* (p. 1063), *wxTCPClient* (p. 1061)

`wxWindows` 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. 958), *wxSocketBase* (p. 938), *wxSocketClient* (p. 956), *wxSocketServer* (p. 959): classes for the low-level TCP/IP API.
- *wxProtocol* (p. 849), *wxURL* (p. 1164), *wxFTP* (p. 466), *wxHTTP*: classes for programming popular Internet protocols.

Further information on these classes will be available in due course.

`wxWindows` has 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 is the same apart from the names of the classes, you should find it easy to switch between the

two implementations.

The following description refers to 'DDE' but remember that the equivalent wxTCP... classes can be used in much the same way.

Three classes are central to the DDE API:

1. `wxDDEClient`. This represents the client application, and is used only within a client program.
2. `wxDDEServer`. This represents the server application, and is used only within a server program.
3. `wxDDEConnection`. This represents the connection from the current client or server to the other application (server or client), and can be used in both server and client programs. 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 sends the message `MakeConnection` to the client 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 must contain an integer port identifier.

The server then responds and either vetoes the connection or allows it. If allowed, a connection object is created which persists until the connection is closed. The connection object is then used for subsequent messages between client and server.

To create a working server, the programmer must:

1. Derive a class from `wxDDEServer`.
2. Override the handler `OnAcceptConnection` for accepting or rejecting a connection, on the basis of the topic argument. This member must create and return a connection object if the connection is accepted.
3. Create an instance of your server object, and call `Create` to activate it, giving it a service name.
4. Derive a class from `wxDDEConnection`.
5. Provide handlers for various messages that are sent to the server side of a `wxDDEConnection`.

To create a working client, the programmer must:

1. Derive a class from `wxDDEClient`.
2. Override the handler `OnMakeConnection` to create and return an appropriate connection object.
3. Create an instance of your client object.
4. Derive a class from `wxDDEConnection`.
5. Provide handlers for various messages that are sent to the client side of a `wxDDEConnection`.
6. When appropriate, create a new connection by sending a `MakeConnection` message to the client object, with arguments host name (processed in Unix only), service name, and topic name for this connection. The client object will call `OnMakeConnection` to create a connection object of the desired type.

7. Use the `wxDDEConnection` member functions to send messages to the server.

Data transfer

These are the ways that data can be transferred from one application to another.

- **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 `wxDDEClient` object represents the client part of a client-server DDE (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 `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. 299) member can return a `wxDDEConnection` of the required class, when a connection is made.

For example:

```
class MyConnection: public wxDDEConnection
{
public:
    MyConnection(void)::wxDDEConnection(ipc_buffer, 3999) {}
```

```

    ~MyConnection(void) { }
    bool OnAdvise(const wxString& topic, const wxString& item, char *data,
int size, wxIPCFormat format)
    { wxMessageBox(topic, data); }
};

class MyClient: public wxDDEClient
{
public:
    MyClient(void) {}
    wxConnectionBase *OnMakeConnection(void) { return new MyConnection; }
};

```

Here, **MyConnection** will respond to *OnAdvise* (p. 301) messages sent by the server.

When the client application starts, it must create an instance of the derived `wxDDEClient`. 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 `wxDDEClient::MakeConnection` (p. 298) 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, 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");

```

Note that it is no longer necessary to call `wxDDEInitialize` or `wxDDECleanUp`, since `wxWindows` will do this itself if necessary.

Chapter 9 wxHTML Notes

This addendum is 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. 422) 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 `lib/mod_*.cpp` files for details.

There is a generic (i.e. independent on wxHtmlWindow) wxHtmlParser class.

wxHTML quick start

Displaying HTML

First of all, you must include `<wx/wxhtml.h>`.

Class *wxHtmlWindow* (p. 542) (derived from *wxScrolledWindow*) is used to display HTML documents. It has two important methods: *LoadPage* (p. 545) and *SetPage* (p. 547). *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></html>");
```

I think the difference is quite clear.

Displaying Help

See *wxHtmlHelpController* (p. 519).

Setting up wxHtmlWindow

Because *wxHtmlWindow* is derived from *wxScrolledWindow* and not from *wxFrame*, it doesn't have visible frame. But the user usually want to see the title of HTML page

displayed somewhere and frame's titlebar is ideal place for it.

`wxHtmlWindow` provides 2 methods in order to handle this: *SetRelatedFrame* (p. 548) and *SetRelatedStatusBar* (p. 548). See the example:

```
html = new wxHtmlWindow(this);
html -> SetRelatedFrame(this, "HTML : %s");
html -> SetRelatedStatusBar(0);
```

The first command associates `html` object with its parent frame (this points to `wxFrame` object there) and sets format of 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:

- *SetFont*s (p. 547)
- *SetBorders* (p. 546)
- *ReadCustomization* (p. 546)
- *WriteCustomization* (p. 548)

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. 515). 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. 534), normal `wxWindows` printout class.

And finally there is the low level class *wxHtmlDCRenderer* (p. 512) 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. 519) 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. 1347).

Contents file (.hhc)

Contents file has HTML syntax and it can be parsed by regular HTML parser. It contains exactly one list (`...` statement):

```
<ul>
  <li> <object>
    <param name="Name" value="@topic name@">
    <param name="ID" value=@numeric_id@>
    <param name="Local" value="@filename.htm@">
  </object>
  <li> <object>
    <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. 521)

Items in the list may be nested - one `` statement may contain a `` sub-statement:

```
<ul>
  <li> <object>
    <param name="Name" value="Top node">
    <param name="Local" value="top.htm">
  </object>
  <ul>
    <li> <object>
      <param name="Name" value="subnode in topnode">
      <param name="Local" value="subnode1.htm">
    </object>
    ...
  </ul>
  <li> <object>
    <param name="Name" value="Another Top">
    <param name="Local" value="top2.htm">
  </object>
  ...
</ul>
```

Index file (.hbk)

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. 545) can load not only HTML files but any known file. To make a file type known to *wxHtmlWindow* you must create a *wxHtmlFilter* (p. 518) filter and register it using *wxHtmlWindow::AddFilter* (p. 543).

Cells and Containers

This article describes mechanism used by *wxHtmlWinParser* (p. 548) and *wxHtmlWindow* (p. 542) 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. 501).

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. 507), *wxHtmlCell::Layout* (p. 504).

Using Containers in Tag Handler

wxHtmlWinParser (p. 548) provides a user-friendly way of managing containers. It is based on the idea of opening and closing containers.

Use *OpenContainer* (p. 553) 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. 549) 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. 540)'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. 548) 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. 540) 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. 548) 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_tmpl.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. 555)(or directly from *wxHtmlTagHandler* (p. 539))

You can use set of macros to define the handler (see *src/mod_*.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_tmpl.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. 540) 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. 540) 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. 541) 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]	#nnnnnn where n is hexadecimal digit
[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

P	ALIGN=[alignment]
BR	ALIGN=[alignment]
DIV	ALIGN=[alignment]
CENTER	
BLOCKQUOTE	
TITLE	
BODY	TEXT=[color] LINK=[color] BGCOLOR=[color]
HR	ALIGN=[alignment] SIZE=[pixels] WIDTH=[percent] WIDTH=[pixels]
FONT	COLOR=[color] SIZE=[fontsize] FACE=[comma-separated list of facenames]
U	
B	
I	
EM	
STRONG	
CITE	
ADDRESS	
CODE	
KBD	

SAMP

TT

H1

H2

H3

H4

H5

H6

A NAME=[string]
 href=[url]

PRE

LI

UL

OL

DL

DT

DD

TABLE ALIGN=[alignment]
 width=[percent]
 width=[pixels]
 border=[pixels]
 valign=[v_alignment]
 bgcolor=[color]
 cellspacing=[pixels]
 cellpadding=[pixels]

TR ALIGN=[alignment]
 valign=[v_alignment]
 bgcolor=[color]

TH ALIGN=[alignment]
 valign=[v_alignment]
 bgcolor=[color]
 width=[percent]
 width=[pixels]
 colspan=[pixels]
 rowspan=[pixels]

TD ALIGN=[alignment]
 valign=[v_alignment]
 bgcolor=[color]
 width=[percent]
 width=[pixels]
 colspan=[pixels]
 rowspan=[pixels]

IMG src=[url]

```
        WIDTH=[pixels]
        HEIGHT=[pixels]
        ALIGN=TEXTTOP
        ALIGN=CENTER
        ALIGN=ABSCENTER
        ALIGN=BOTTOM
        USEMAP=[url]

MAP      NAME=[string]

AREA     SHAPE=POLY
        SHAPE=CIRCLE
        SHAPE=RECT
        COORDS=[coords]
        HREF=[url]

META     HTTP-EQUIV="Content-Type"
        CONTENT=[string]
```

Chapter 10 Property sheet classes

Introduction

The Property Sheet Classes help the programmer to specify complex dialogs and their relationship with their associated data. By specifying data as a `wxPropertySheet` containing `wxProperty` objects, the programmer can use a range of available or custom `wxPropertyView` classes to allow the user to edit this data. Classes derived from `wxPropertyView` act as mediators between the `wxPropertySheet` and the actual window (and associated panel items).

For example, the `wxPropertyListView` is a kind of `wxPropertyView` which displays data in a Visual Basic-style property list (see *the next section* (p. 1443) for screen shots). This is a listbox containing names and values, with an edit control and other optional controls via which the user edits the selected data item.

`wxPropertyFormView` is another kind of `wxPropertyView` which mediates between the data and a panel or dialog box which has already been created. This makes it a contender for the replacement of `wxForm`, since programmer-controlled layout is going to be much more satisfactory. If automatic layout is desired, then `wxPropertyListView` could be used instead.

The main intention of this class library was to provide property *list* behaviour, but it has been generalised as much as possible so that the concept of a property sheet and its viewers can reduce programming effort in a range of user interface tasks.

For further details on the classes and how they are used, please see *Property classes overview* (p. 1445).

The appearance and behaviour of a property list view

The property list, as seen in an increasing number of development tools such as Visual Basic and Delphi, is a convenient and compact method for displaying and editing a number of items without the need for one control per item, and without the need for designing a special form. The controls are as follows:

- A listbox showing the properties and their current values, which has double-click properties dependent on the nature of the current property;
- a text editing area at the top of the display, allowing the user to edit the currently selected property if appropriate;
- 'confirm' and 'cancel' buttons to confirm or cancel an edit (for the property, not the whole sheet);
- an optional list that appears when the user can make a choice from several

- known possible values;
- a small Edit button to invoke 'detailed editing' (perhaps showing or hiding the above value list, or maybe invoking a common dialog);
- optional OK/Close, Cancel and Help buttons for the whole dialog.

The concept of 'detailed editing' versus quick editing gives the user a choice of editing mode, so novice and expert behaviour can be catered for, or the user can just use what he feels comfortable with.

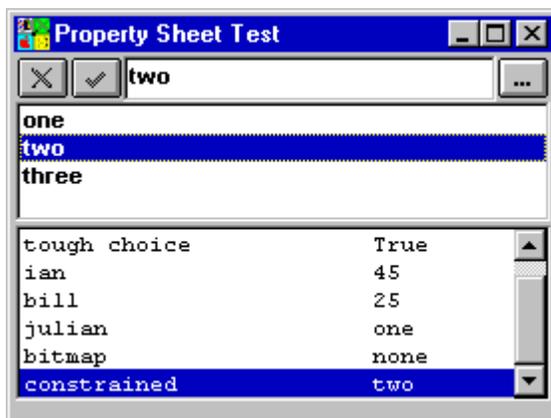
Behaviour alters depending on the kind of property being edited. For example, a boolean value has the following behaviour:

- Double-clicking on the item toggles between TRUE and FALSE.
- Showing the value list enables the user to select TRUE or FALSE.
- The user may be able to type in the word TRUE or FALSE, or the edit control may be read-only to disallow this since it is error-prone.

A list of strings may pop up a dialog for editing them, a simple string just allows text editing, double-clicking a colour property may show a colour selector, double-clicking on a filename property may show a file selector (in addition to being able to type in the name in the edit control), etc.

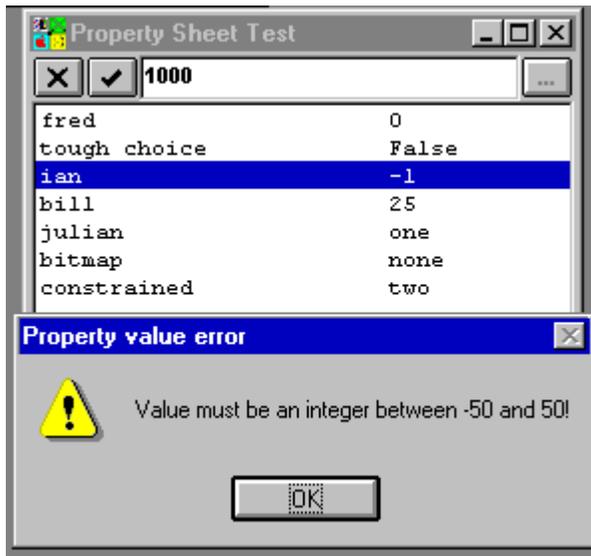
Note that the 'type' of property, such as string or integer, does not necessarily determine the behaviour of the property. The programmer has to be able to specify different behaviours for the same type, depending on the meaning of the property. For example, a colour and a filename may both be strings, but their editing behaviour should be different. This is why objects of type `wxPropertyValidator` need to be used, to define behaviour for a given class of properties or even specific property name. Objects of class `wxPropertyView` contain a list of property registries, which enable reuse of bunches of these validators in different circumstances. Or a `wxProperty` can be explicitly set to use a particular validator object.

The following screen shot of the property classes test program shows the user editing a string, which is constrained to be one of three possible values.



The second picture shows the user having entered a integer that was outside the range specified to the validator. Note that in this picture, the value list is hidden because it is

not used when editing an integer.



Headers

The property class library comprises the following files:

- prop.h: base property class header
- proplist.h: wxPropertyListView and associated classes
- propform.h: wxPropertyListView and associated classes

Topic overviews

This chapter contains a selection of topic overviews.

Property classes overview

The property classes help a programmer to express relationships between data and physical windows, in particular:

- the transfer of data to and from the physical controls;
- the behaviour of various controls and custom windows for particular types of data;
- the validation of data, notifying the user when incorrect data is entered, or even better, constraining the input so only valid data can be entered.

With a consistent framework, the programmer should be able to use existing components and design new ones in a principled manner, to solve many data entry requirements.

Each datum is represented in a *wxProperty* (p. 821), which has a name and a value. Various C++ types are permitted in the value of a property, and the property can store a pointer to the data instead of a copy of the data. A *wxPropertySheet* (p. 837) represents a number of these properties.

These two classes are independent from the way in which the data is visually manipulated. To mediate between property sheets and windows, the abstract class *wxPropertyView* (p. 846) is available for programmers to derive new kinds of view. One kind of view that is available is the *wxPropertyListView* (p. 834), which displays the data in a Visual Basic-style list, with a small number of controls for editing the currently selected property. Another is *wxPropertyFormView* (p. 827) which mediates between an existing dialog or panel and the property sheet.

The hard work of mediation is actually performed by validators, which are instances of classes derived from *wxPropertyValidator* (p. 839). A validator is associated with a particular property and is responsible for responding to user interface events, and displaying, updating and checking the property value. Because a validator's behaviour depends largely on the kind of view being used, there has to be a separate hierarchy of validators for each class of view. So for *wxPropertyListView*, there is an abstract class *wxPropertyListValidator* (p. 832) from which concrete classes are derived, such as *wxRealListValidator* (p. 867) and *wxStringListValidator* (p. 1031).

A validator can be explicitly set for a property, so there is no doubt which validator should be used to edit that property. However, it is also possible to define a registry of validators, and have the validator chosen on the basis of the *role* of the property. So a property with a "filename" role would match the "filename" validator, which pops up a file selector when the user double clicks on the property.

You don't have to define your own frame or window classes: there are some predefined that will work with the property list view. See *Window classes* (p. 1454) for further details.

Example 1: Property list view

The following code fragment shows the essentials of creating a registry of standard validators, a property sheet containing some properties, and a property list view and dialog or frame. *RegisterValidators* will be called on program start, and *PropertySheetTest* is called in response to a menu command.

Note how some properties are created with an explicit reference to a validator, and others are provided with a "role" which can be matched against a validator in the registry.

The interface generated by this test program is shown in the section *Appearance and behaviour of a property list view* (p. 1443).

```

void RegisterValidators(void)
{
    myListValidatorRegistry.RegisterValidator((wxString)"real", new
wxRealListValidator);
    myListValidatorRegistry.RegisterValidator((wxString)"string", new
wxStringListValidator);
    myListValidatorRegistry.RegisterValidator((wxString)"integer", new
wxIntegerListValidator);
    myListValidatorRegistry.RegisterValidator((wxString)"bool", new
wxBoolListValidator);
}

void PropertyListTest(Bool useDialog)
{
    wxPropertySheet *sheet = new wxPropertySheet;

    sheet->AddProperty(new wxProperty("fred", 1.0, "real"));
    sheet->AddProperty(new wxProperty("tough choice", (Bool)TRUE,
"bool"));
    sheet->AddProperty(new wxProperty("ian", (long)45, "integer", new
wxIntegerListValidator(-50, 50));
    sheet->AddProperty(new wxProperty("bill", 25.0, "real", new
wxRealListValidator(0.0, 100.0));
    sheet->AddProperty(new wxProperty("julian", "one", "string"));
    sheet->AddProperty(new wxProperty("bitmap", "none", "string", new
wxFilenameListValidator("Select a bitmap file", "*.bmp"));
    wxStringList *strings = new wxStringList("one", "two", "three", NULL);
    sheet->AddProperty(new wxProperty("constrained", "one", "string", new
wxStringListValidator(strings));

    wxPropertyListView *view =
        new wxPropertyListView(NULL,
wxPROP_BUTTON_CHECK_CROSS|wxPROP_DYNAMIC_VALUE_FIELD|wxPROP_PULLDOWN);

    wxDialogBox *propDialog = NULL;
    wxPropertyListFrame *propFrame = NULL;
    if (useDialog)
    {
        propDialog = new wxPropertyListDialog(view, NULL, "Property Sheet
Test", TRUE, -1, -1, 400, 500);
    }
    else
    {
        propFrame = new wxPropertyListFrame(view, NULL, "Property Sheet
Test", -1, -1, 400, 500);
    }

    view->AddRegistry(&myListValidatorRegistry);

    if (useDialog)
    {
        view->ShowView(sheet, propDialog);
        propDialog->Centre(wxBOTH);
        propDialog->Show(TRUE);
    }
    else
    {
        propFrame->Initialize();
        view->ShowView(sheet, propFrame->GetPropertyPanel());
        propFrame->Centre(wxBOTH);
        propFrame->Show(TRUE);
    }
}

```

Example 2: Property form view

This example is similar to Example 1, but uses a property form view to edit a property sheet using a predefined dialog box.

```
void RegisterValidators(void)
{
    myFormValidatorRegistry.RegisterValidator((wxString)"real", new
wxRealFormValidator);
    myFormValidatorRegistry.RegisterValidator((wxString)"string", new
wxStringFormValidator);
    myFormValidatorRegistry.RegisterValidator((wxString)"integer", new
wxIntegerFormValidator);
    myFormValidatorRegistry.RegisterValidator((wxString)"bool", new
wxBoolFormValidator);
}

void PropertyFormTest(Bool useDialog)
{
    wxPropertySheet *sheet = new wxPropertySheet;

    sheet->AddProperty(new wxProperty("fred", 25.0, "real", new
wxRealFormValidator(0.0, 100.0)));
    sheet->AddProperty(new wxProperty("tough choice", (Bool)TRUE,
"bool"));
    sheet->AddProperty(new wxProperty("ian", (long)45, "integer", new
wxIntegerFormValidator(-50, 50)));
    sheet->AddProperty(new wxProperty("julian", "one", "string"));
    wxStringList *strings = new wxStringList("one", "two", "three", NULL);
    sheet->AddProperty(new wxProperty("constrained", "one", "string", new
wxStringFormValidator(strings)));

    wxPropertyFormView *view = new wxPropertyFormView(NULL);

    wxDialogBox *propDialog = NULL;
    wxPropertyFormFrame *propFrame = NULL;
    if (useDialog)
    {
        propDialog = new wxPropertyFormDialog(view, NULL, "Property Form
Test", TRUE, -1, -1, 400, 300);
    }
    else
    {
        propFrame = new wxPropertyFormFrame(view, NULL, "Property Form
Test", -1, -1, 400, 300);
        propFrame->Initialize();
    }

    wxPanel *panel = propDialog ? propDialog : propFrame-
>GetPropertyPanel();
    panel->SetLabelPosition(wxVERTICAL);

    // Add items to the panel

    (void) new wxButton(panel, (wxFunction)NULL, "OK", -1, -1, -1, -1, 0,
"ok");
    (void) new wxButton(panel, (wxFunction)NULL, "Cancel", -1, -1, 80, -1,
0, "cancel");
    (void) new wxButton(panel, (wxFunction)NULL, "Update", -1, -1, 80, -1,
0, "update");
}
```

```

    (void) new wxButton(panel, (wxFunction)NULL, "Revert", -1, -1, -1, -1,
0, "revert");
    panel->NewLine();

    // The name of this text item matches the "fred" property
    (void) new wxText(panel, (wxFunction)NULL, "Fred", "", -1, -1, 90, -1,
0, "fred");
    (void) new wxCheckBox(panel, (wxFunction)NULL, "Yes or no", -1, -1, -
1, -1, 0, "tough choice");
    (void) new wxSlider(panel, (wxFunction)NULL, "Sliding scale", 0, -50,
50, 100, -1, -1, wxHORIZONTAL, "ian");
    panel->NewLine();
    (void) new wxListBox(panel, (wxFunction)NULL, "Constrained", wxSINGLE,
-1, -1, 100, 90, 0, NULL, 0, "constrained");

    view->AddRegistry(&myFormValidatorRegistry);

    if (useDialog)
    {
        view->ShowView(sheet, propDialog);
        view->AssociateNames();
        view->TransferToDialog();
        propDialog->Centre(wxBOTH);
        propDialog->Show(TRUE);
    }
    else
    {
        view->ShowView(sheet, propFrame->GetPropertyPanel());
        view->AssociateNames();
        view->TransferToDialog();
        propFrame->Centre(wxBOTH);
        propFrame->Show(TRUE);
    }
}

```

Validator classes overview

Classes: *Validator classes* (p. 1453)

The validator classes provide functionality for mediating between a `wxProperty` and the actual display. There is a separate family of validator classes for each class of view, since the differences in user interface for these views implies that little common functionality can be shared amongst validators.

wxPropertyValidator overview

Class: *wxPropertyValidator* (p. 839)

This class is the root of all property validator classes. It contains a small amount of common functionality, including functions to convert between strings and C++ values.

A validator is notionally an object which sits between a property and its displayed value, and checks that the value the user enters is correct, giving an error message if the validation fails. In fact, the validator does more than that, and is akin to a view class but at a finer level of detail. It is also responsible for loading the dialog box control with the value from the property, putting it back into the property, preparing special controls for

editing the value, and may even invoke special dialogs for editing the value in a convenient way.

In a property list dialog, there is quite a lot of scope for supplying custom dialogs, such as file or colour selectors. For a form dialog, there is less scope because there is no concept of 'detailed editing' of a value: one control is associated with one property, and there is no provision for invoking further dialogs. The reader may like to work out how the form view could be extended to provide some of the functionality of the property list!

Validator objects may be associated explicitly with a `wxProperty`, or they may be indirectly associated by virtue of a property 'kind' that matches validators having that kind. In the latter case, such validators are stored in a validator registry which is passed to the view before the dialog is shown. If the validator takes arguments, such as minimum and maximum values in the case of a `wxIntegerListValidator`, then the validator must be associated explicitly with the property. The validator will be deleted when the property is deleted.

wxPropertyListValidator overview

Class: `wxPropertyListValidator` (p. 832)

This class is the abstract base class for property list view validators. The list view acts upon a user interface containing a list of properties, a text item for direct property value editing, confirm/cancel buttons for the value, a pulldown list for making a choice between values, and OK/Cancel/Help buttons for the dialog (see *property list appearance* (p. 1443)).

By overriding virtual functions, the programmer can create custom behaviour for different kinds of property. Custom behaviour can use just the available controls on the property list dialog, or the validator can invoke custom editors with quite different controls, which pop up in 'detailed editing' mode.

See the detailed class documentation for the members you should override to give your validator appropriate behaviour.

wxPropertyFormValidator overview

This class is the abstract base class for property form view validators. The form view acts upon an existing dialog box or panel, where either the panel item names correspond to property names, or the programmer has explicitly associated the panel item with the property.

By overriding virtual functions, the programmer determines how values are displayed or retrieved, and the checking that the validator does.

See the detailed class documentation for the members you should override to give your validator appropriate behaviour.

View classes overview

Classes: *View classes* (p. 1453)

An instance of a view class relates a property sheet with an actual window. Currently, there are two classes of view: `wxPropertyListView` and `wxPropertyFormView`.

wxPropertyView overview

Class: *wxPropertyView* (p. 846)

This is the abstract base class for property views.

wxPropertyListView overview

Class: *wxPropertyListView* (p. 834)

The property list view defines the relationship between a property sheet and a property list dialog or panel. It manages user interface events such as clicking on a property, pressing return in the text edit field, and clicking on Confirm or Cancel. These events cause member functions of the class to be called, and these in turn may call member functions of the appropriate validator to be called, to prepare controls, check the property value, invoke detailed editing, etc.

wxPropertyFormView overview

Class: *wxPropertyFormView* (p. 827)

The property form view manages the relationship between a property sheet and an existing dialog or panel.

You must first create a panel or dialog box for the view to work on. The panel should contain panel items with names that correspond to properties in your property sheet; or you can explicitly set the panel item for each property.

Apart from any custom panel items that you wish to control independently of the property-editing items, `wxPropertyFormView` takes over the processing of item events. It can also control normal dialog behaviour such as OK, Cancel, so you should also create some standard buttons that the property view can recognise. Just create the buttons with standard names and the view will do the rest. The following button names are recognised:

- **ok**: indicates the OK button. Calls `wxPropertyFormView::OnOk`. By default, checks and updates the form values, closes and deletes the frame or dialog, then deletes the view.
- **cancel**: indicates the Cancel button. Calls `wxPropertyFormView::OnCancel`. By default, closes and deletes the frame or dialog, then deletes the view.

- **help**: indicates the Help button. Calls `wxPropertyFormView::OnHelp`. This needs to be overridden by the application for anything interesting to happen.
- **revert**: indicates the Revert button. Calls `wxPropertyFormView::OnRevert`, which by default transfers the `wxProperty` values to the panel items (in effect undoing any unsaved changes in the items).
- **update**: indicates the Revert button. Calls `wxPropertyFormView::OnUpdate`, which by defaults transfers the displayed values to the `wxProperty` objects.

wxPropertySheet overview

Classes: `wxPropertySheet` (p. 837), `wxProperty` (p. 821), `wxPropertyValue` (p. 841)

A property sheet defines zero or more properties. This is a bit like an explicit representation of a C++ object. `wxProperty` objects can have values which are pointers to C++ values, or they can allocate their own storage for values.

Because the property sheet representation is explicit and can be manipulated by a program, it is a convenient form to be used for a variety of editing purposes. `wxPropertyListView` and `wxPropertyFormView` are two classes that specify the relationship between a property sheet and a user interface. You could imagine other uses for `wxPropertySheet`, for example to generate a form-like user interface without the need for GUI programming. Or for storing the names and values of command-line switches, with the option to subsequently edit these values using a `wxPropertyListView`.

A typical use for a property sheet is to represent values of an object which are only implicit in the current representation of it. For example, in Visual Basic and similar programming environments, you can 'edit a button', or rather, edit the button's properties. One of the properties you can edit is *width* - but there is no explicit representation of width in a `wxWindows` button; instead, you call `SetSize` and `GetSize` members. To translate this into a consistent, property-oriented scheme, we could derive a new class `wxButtonWithProperties`, which has two new functions: `SetProperty` and `GetProperty`. `SetProperty` accepts a property name and a value, and calls an appropriate function for the property that is being passed. `GetProperty` accepts a property name, returning a property value. So instead of having to use the usual arbitrary set of C++ member functions to set or access attributes of a window, programmer deals merely with `SetValue/GetValue`, and property names and values. We now have a single point at which we can modify or query an object by specifying names and values at run-time. (The implementation of `SetProperty` and `GetProperty` is probably quite messy and involves a large if-then-else statement to test the property name and act accordingly.)

When the user invokes the property editor for a `wxButtonWithProperties`, the system creates a `wxPropertySheet` with 'imaginary' properties such as width, height, font size and so on. For each property, `wxButtonWithProperties::GetProperty` is called, and the result is passed to the corresponding `wxProperty`. The `wxPropertySheet` is passed to a `wxPropertyListView` as described elsewhere, and the user edits away. When the user has finished editing, the system calls `wxButtonWithProperties::SetProperty` to transfer the `wxProperty` value back into the button by way of an appropriate call, `wxWindow::SetSize` in the case of width and height properties.

Classes by category

A classification of property sheet classes by category.

Data classes

- *wxProperty* (p. 821)
- *wxPropertyValue* (p. 841)
- *wxPropertySheet* (p. 837)

Validator classes

Validators check that the values the user has entered for a property are valid. They can also define specific ways of entering data, such as a file selector for a filename, and they are responsible for transferring values between the *wxProperty* and the physical display.

Base classes:

- *wxPropertyValidator* (p. 821)
- *wxPropertyListValidator* (p. 832)
- *wxPropertyFormValidator* (p. 826)

List view validators:

- *wxBoolListValidator* (p. 77)
- *wxFilenameListValidator* (p. 422)
- *wxIntegerListValidator* (p. 595)
- *wxListOfStringsListValidator* (p. 647)
- *wxRealListValidator* (p. 867)
- *wxStringListValidator* (p. 1031)

Form view validators:

- *wxBoolFormValidator* (p. 76)
- *wxIntegerFormValidator* (p. 594)
- *wxRealFormValidator* (p. 867)
- *wxStringFormValidator* (p. 1029)

View classes

View classes mediate between a property sheet and a physical window.

- *wxPropertyView* (p. 846)
- *wxPropertyListView* (p. 834)
- *wxPropertyFormView* (p. 827)

Window classes

The class library defines some window classes that can be used as-is with a suitable view class and property sheet.

- *wxPropertyFormFrame* (p. 824)
- *wxPropertyFormDialog* (p. 824)
- *wxPropertyFormPanel* (p. 825)
- *wxPropertyListFrame* (p. 830)
- *wxPropertyListDialog* (p. 830)
- *wxPropertyListPanel* (p. 831)

Registry classes

A validator registry is a list of validators that can be applied to properties in a property sheet. There may be one or more registries per property view.

- *wxPropertyValidatorRegistry* (p. 840)

Chapter 11 wxPython Notes

This addendum is written by Robin Dunn, author of the wxPython wrapper

What is wxPython?

wxPython is a blending of the wxWindows GUI classes and the Python (<http://www.python.org/>) programming language.

Python

So what is Python? Go to <http://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 wxWindows GUI classes. This extension module attempts to mirror the class heirarchy of wxWindows 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 wxWindows? Personally I prefer using Python for everything. I only use C++ when I absolutely have to eek 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 wxWindows apps. With C++ you have to continuously go through 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++/wxWindows app should be a straight forward task.

Other Python GUIs

There are other GUI solutions out there for Python.

Tkinter

Tkinter is the defacto 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 wxWindows 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, -
025:         1)),
026:         wxTE_READONLY)
027:     self.posCtrl = wxTextCtrl(panel, -1, "",
028:         wxDLG_PNT(panel, wxPoint(24,
029:         14)),
030:         wxDLG_SZE(panel, wxSize(36, -1)),
031:         wxTE_READONLY)
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
039:     # resized,
040:     # because of the association above.
041:     def OnSize(self, event):
042:         size = event.GetSize()
043:         self.sizeCtrl.SetValue("%s, %s" % (size.width,
044:         size.height))
045:         # tell the event system to continue looking for an event
046:         # handler,
047:         # so the default handler will get called.
048:         event.Skip()
049:
050:     # This method is called by the system when the window is moved,
051:     # because of the association above.
052:     def OnMove(self, event):
053:         pos = event.GetPosition()
054:         self.posCtrl.SetValue("%s, %s" % (pos.x, pos.y))
055:
056:     # Every wxWindows application must have a class derived from wxApp
057:     class MyApp(wxApp):
058:         # wxWindows 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 wxWindows 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 wxWindows 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_SIZE` 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 wxWindows 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 wxWindows 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.

wxWindows 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. 16)
- *wxAcceleratorTable* (p. 17)
- *wxActivateEvent* (p. 20)
- *wxBitmap* (p. 54)
- *wxBitmapButton* (p. 70)
- *wxBitmapDataObject* (p. 75)
- *wxBMPHandler*
- *wxBoxSizer* (p. 77)
- *wxBrush* (p. 80)
- *wxBusyInfo* (p. 88)
- *wxBusyCursor* (p. 87)
- *wxButton* (p. 89)
- *wxCalculateLayoutEvent* (p. 94)
- *wxCalendarCtrl* (p. 95)
- *wxCaret*
- *wxCheckBox* (p. 108)
- *wxCheckListBox* (p. 111)
- *wxChoice* (p. 113)
- *wxClientDC* (p. 120)
- *wxClipboard* (p. 121)
- *wxCloseEvent* (p. 124)
- *wxColourData* (p. 138)
- *wxColourDialog* (p. 142)
- *wxColour* (p. 135)
- *wxComboBox* (p. 143)
- *wxCommandEvent* (p. 152)
- *wxConfig* (p. 162)
- *wxControl* (p. 176)
- *wxCursor* (p. 184)
- *wxCustomDataObject* (p. 181)
- *wxDataFormat* (p. 194)
- *wxDataObject* (p. 196)
- *wxDataObjectComposite* (p. 200)
- *wxDataObjectSimple* (p. 201)
- *wxDateTime* (p. 215)
- *wxDateSpan* (p. 214)
- *wxDC* (p. 280)
- *wxDialog* (p. 310)
- *wxDirDialog* (p. 325)
- *wxDragImage* (p. 359)
- *wxDropFilesEvent* (p. 364)
- *wxDropSource* (p. 365)
- *wxDropTarget* (p. 368)

- *wxEraseEvent* (p. 374)
- *wxEvent* (p. 375)
- *wxEvtHandler* (p. 378)
- *wxFileConfig*
- *wxFileDataObject* (p. 406)
- *wxFileDialog* (p. 407)
- *wxFileDropTarget* (p. 412)
- *wxFileSystem* (p. 422)
- *wxFileSystemHandler* (p. 424)
- *wxFocusEvent* (p. 433)
- *wxFontData* (p. 441)
- *wxFontDialog* (p. 444)
- *wxFont* (p. 434)
- *wxFrame* (p. 452)
- *wxFSFile* (p. 464)
- *wxGauge* (p. 470)
- *wxGIFHandler*
- *wxGLCanvas*
- *wxHtmlCell* (p. 501)
- *wxHtmlContainerCell* (p. 507)
- *wxHtmlDCRenderer* (p. 512)
- *wxHtmlEasyPrinting* (p. 515)
- *wxHtmlParser* (p. 530)
- *wxHtmlTagHandler* (p. 539)
- *wxHtmlTag* (p. 536)
- *wxHtmlWinParser* (p. 548)
- *wxHtmlPrintout* (p. 534)
- *wxHtmlWinTagHandler* (p. 555)
- *wxHtmlWindow* (p. 542)
- *wxIconizeEvent*
- *wxIcon* (p. 558)
- *wxIdleEvent* (p. 557)
- *wxImage* (p. 565)
- *wxImageHandler* (p. 580)
- *wxImageList* (p. 584)
- *wxIndividualLayoutConstraint* (p. 588)
- *wxInitDialogEvent* (p. 591)
- *wxInputStream* (p. 592)
- *wxInternetFSHandler*
- *wxJoystickEvent* (p. 604)
- *wxJPEGHandler*
- *wxKeyEvent* (p. 607)
- *wxLayoutAlgorithm* (p. 610)
- *wxLayoutConstraints* (p. 613)
- *wxListBox* (p. 621)
- *wxListCtrl* (p. 630)
- *wxListEvent* (p. 644)
- *wxListItem* (p. 641)

- *wxMask* (p. 659)
- *wxMaximizeEvent*
- *wxMDIChildFrame* (p. 666)
- *wxMDIClientWindow* (p. 670)
- *wxMDIParentFrame* (p. 671)
- *wxMemoryDC* (p. 678)
- *wxMemoryFSHandler* (p. 680)
- *wxMenuBar* (p. 693)
- *wxMenuEvent* (p. 707)
- *wxMenuItem* (p. 702)
- *wxMenu* (p. 683)
- *wxMessageDialog* (p. 709)
- *wxMetaFileDC* (p. 712)
- *wxMiniFrame* (p. 716)
- *wxMouseEvent* (p. 721)
- *wxMoveEvent* (p. 729)
- *wxNotebookEvent* (p. 743)
- *wxNotebook* (p. 736)
- *wxPageSetupDialogData* (p. 752)
- *wxPageSetupDialog* (p. 758)
- *wxPaintDC* (p. 759)
- *wxPaintEvent* (p. 760)
- *wxPalette* (p. 761)
- *wxPanel* (p. 764)
- *wxPen* (p. 771)
- *wxPNGHandler*
- *wxPoint* (p. 785)
- *wxPostScriptDC* (p. 786)
- *wxPreviewFrame* (p. 790)
- *wxPrintData* (p. 792)
- *wxPrintDialogData* (p. 799)
- *wxPrintDialog* (p. 797)
- *wxPrinter* (p. 803)
- *wxPrintPreview* (p. 810)
- *wxPrinterDC* (p. 806)
- *wxPrintout* (p. 807)
- *wxProcess* (p. 815)
- *wxQueryLayoutInfoEvent* (p. 856)
- *wxRadioBox* (p. 858)
- *wxRadioButton* (p. 864)
- *wxRealPoint* (p. 868)
- *wxRect* (p. 868)
- *wxRegionIterator* (p. 889)
- *wxRegion* (p. 885)
- *wxSashEvent* (p. 892)
- *wxSashLayoutWindow* (p. 894)
- *wxSashWindow* (p. 897)
- *wxScreenDC* (p. 901)

- *wxScrollBar* (p. 903)
- *wxScrollEvent* (p. 909)
- *wxScrolledWindow* (p. 911)
- *wxScrollWinEvent* (p. 908)
- *wxShowEvent*
- *wxSingleChoiceDialog* (p. 919)
- *wxSizeEvent* (p. 923)
- *wxSize* (p. 922)
- *wxSizer* (p. 924)
- *wxSizerItem*
- *wxSlider* (p. 929)
- *wxSpinButton* (p. 963)
- *wxSpinEvent*
- *wxSplitterWindow* (p. 973)
- *wxStaticBitmap* (p. 982)
- *wxStaticBox* (p. 985)
- *wxStaticBoxSizer* (p. 986)
- *wxStaticLine* (p. 987)
- *wxStaticText* (p. 989)
- *wxStatusBar* (p. 991)
- *wxSysColourChangedEvent* (p. 1034)
- *wxTaskBarIcon* (p. 1059)
- *wxTextCtrl* (p. 1070)
- *wxTextDataObject* (p. 1083)
- *wxTextDropTarget* (p. 1090)
- *wxTextEntryDialog* (p. 1089)
- *wxTimer* (p. 1113)
- *wxTimerEvent* (p. 1115)
- *wxTimeSpan* (p. 1092)
- *wxTipProvider* (p. 1116)
- *wxToolBarTool*
- *wxToolBar* (p. 1117)
- *wxToolTip*
- *wxTreeCtrl* (p. 1134)
- *wxTreeEvent* (p. 1151)
- *wxTreeItemData* (p. 1149)
- *wxTreeItemId*
- *wxUpdateUIEvent* (p. 1160)
- *wxValidator* (p. 1166)
- *wxWindowDC* (p. 1234)
- *wxWindow* (p. 1184)
- *wxZipFSHandler*

Where to go for help

Since wxPython is a blending of multiple technologies, help comes from multiple sources. See <http://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.sourceforge.net/mailman/listinfo/wxpython-users>
(`http://lists.sourceforge.net/mailman/listinfo/wxpython-users`)

Or you can send mail directly to the list using this address:

`wxpython-users@lists.sourceforge.net`

Chapter 12 Porting from wxWindows 1.xx

This addendum gives guidelines and tips for porting applications from version 1.xx of wxWindows to version 2.0.

The first section offers tips for writing 1.xx applications in a way to minimize porting time. The following sections detail the changes and how you can modify your application to be 2.0-compliant.

You may be worrying that porting to 2.0 will be a lot of work, particularly if you have only recently started using 1.xx. In fact, the wxWindows 2.0 API has far more in common with 1.xx than it has differences. The main challenges are using the new event system, doing without the default panel item layout, and the lack of automatic labels in some controls.

Please don't be freaked out by the jump to 2.0! For one thing, 1.xx is still available and will be supported by the user community for some time. And when you have changed to 2.0, we hope that you will appreciate the benefits in terms of greater flexibility, better user interface aesthetics, improved C++ conformance, improved compilation speed, and many other enhancements. The revised architecture of 2.0 will ensure that wxWindows can continue to evolve for the foreseeable future.

Please note that this document is a work in progress.

Preparing for version 2.0

Even before compiling with version 2.0, there's also a lot you can do right now to make porting relatively simple. Here are a few tips.

- **Use constraints or .wxr resources** for layout, rather than the default layout scheme. Constraints should be the same in 2.0, and resources will be translated.
- **Use separate wxMessage items** instead of labels for wxText, wxMultiText, wxChoice, wxComboBox. These labels will disappear in 2.0. Use separate wxMessages whether you're creating controls programmatically or using the dialog editor. The future dialog editor will be able to translate from old to new more accurately if labels are separated out.
- **Parameterise functions that use wxDC** or derivatives, i.e. make the wxDC an argument to all functions that do drawing. Minimise the use of wxWindow::GetDC and definitely don't store wxDCs long-term because in 2.0, you can't use GetDC() and wxDCs are not persistent. You will use wxClientDC, wxPaintDC stack objects instead. Minimising the use of GetDC() will ensure that there are very few places you have to change drawing code for 2.0.

- **Don't set GDI objects** (wxPen, wxBrush etc.) in windows or wxCanvasDCs before they're needed (e.g. in constructors) - do so within your drawing routine instead. In 2.0, these settings will only take effect between the construction and destruction of temporary wxClient/PaintDC objects.
- **Don't rely** on arguments to wxDC functions being floating point - they will be 32-bit integers in 2.0.
- **Don't use the wxCanvas member functions** that duplicate wxDC functions, such as SetPen and DrawLine, since they are going.
- **Using member callbacks** called from global callback functions will make the transition easier - see the FAQ for some notes on using member functions for callbacks. wxWindows 2.0 will banish global callback functions (and OnMenuCommand), and nearly all event handling will be done by functions taking a single event argument. So in future you will have code like:

```
void MyFrame::OnOK(wxCommandEvent&event)
```

```
...
```

You may find that writing the extra code to call a member function isn't worth it at this stage, but the option is there.

- **Use wxString wherever possible.** 2.0 replaces char * with wxString in most cases, and if you use wxString to receive strings returned from wxWindows functions (except when you need to save the pointer if deallocation is required), there should be no conversion problems later on.
- Be aware that under Windows, **font sizes will change** to match standard Windows font sizes (for example, a 12-point font will appear bigger than before). Write your application to be flexible where fonts are concerned. Don't rely on fonts being similarly-sized across platforms, as they were (by chance) between Windows and X under wxWindows 1.66. Yes, this is not easy... but I think it is better to conform to the standards of each platform, and currently the size difference makes it difficult to conform to Windows UI standards. You may eventually wish to build in a global 'fudge-factor' to compensate for size differences. The old font sizing will still be available via wx_setup.h, so do not panic...
- **Consider dropping wxForm usage:** wxPropertyFormView can be used in a wxForm-like way, except that you specify a pre-constructed panel or dialog; or you can use a wxPropertyListView to show attributes in a scrolling list - you don't even need to lay panel items out.

Because wxForm uses a number of features to be dropped in wxWindows 2.0, it cannot be supported in the future, at least in its present state.

- **When creating a wxListBox,** put the wxLB_SINGLE, wxLB_MULTIPLE,

wxB_EXTENDED styles in the window style parameter, and put zero in the *multiple* parameter. The *multiple* parameter will be removed in 2.0.

- **For MDI applications**, don't rely on MDI being run-time-switchable in the way that the MDI sample is. In wxWindows 2.0, MDI functionality is separated into distinct classes.

The new event system

The way that events are handled has been radically changed in wxWindows 2.0. Please read the topic 'Event handling overview' in the wxWindows 2.0 manual for background on this.

Callbacks

Instead of callbacks for panel items, menu command events, control commands and other events are directed to the originating window, or an ancestor, or an event handler that has been plugged into the window or its ancestor. Event handlers always have one argument, a derivative of wxEvent.

For menubar commands, the **OnMenuCommand** member function will be replaced by a series of separate member functions, each of which responds to a particular command. You need to add these (non-virtual) functions to your frame class, add a DECLARE_EVENT_TABLE entry to the class, and then add an event table to your implementation file, as a BEGIN_EVENT_TABLE and END_EVENT_TABLE block. The individual event mapping macros will be of the form:

```
BEGIN_EVENT_TABLE(MyFrame, wxFrame)
    EVT_MENU(MYAPP_NEW, MyFrame::OnNew)
    EVT_MENU(wxID_EXIT, MyFrame::OnExit)
END_EVENT_TABLE()
```

Control commands, such as button commands, can be routed to a derived button class, the parent window, or even the frame. Here, you use a function of the form EVT_BUTTON(id, func). Similar macros exist for other control commands.

Other events

To intercept other events, you used to override virtual functions, such as OnSize. Now, while you can use the OnSize name for such event handlers (or any other name of your choice), it has only a single argument (wxSizeEvent) and must again be 'mapped' using the EVT_SIZE macro. The same goes for all other events, including OnClose (although in fact you can still use the old, virtual form of OnClose for the time being).

Class hierarchy

The class hierarchy has changed somewhat. `wxToolBar` and `wxButtonBar` classes have been split into several classes, and are derived from `wxControl` (which was called `wxItem`). `wxPanel` derives from `wxWindow` instead of from `wxCanvas`, which has disappeared in favour of `wxScrolledWindow` (since all windows are now effectively canvases which can be drawn into). The status bar has become a class in its own right, `wxStatusBar`.

There are new MDI classes so that `wxFrame` does not have to be overloaded with this functionality.

There are new device context classes, with `wxPanelDC` and `wxCanvasDC` disappearing. See *Device contexts and painting* (p. 1470).

GDI objects

These objects - instances of classes such as `wxPen`, `wxBrush`, `wxBitmap` (but not `wxColour`) - are now implemented with reference-counting. This makes assignment a very cheap operation, and also means that management of the resource is largely automatic. You now pass *references* to objects to functions such as `wxDC::SetPen`, not pointers, so you will need to dereference your pointers. The device context does not store a copy of the pen itself, but takes a copy of it (via reference counting), and the object's data gets freed up when the reference count goes to zero. The application does not have to worry so much about who the object belongs to: it can pass the reference, then destroy the object without leaving a dangling pointer inside the device context.

For the purposes of code migration, you can use the old style of object management - maintaining pointers to GDI objects, and using the `FindOrCreate...` functions. However, it is preferable to keep this explicit management to a minimum, instead creating objects on the fly as needed, on the stack, unless this causes too much of an overhead in your application.

At a minimum, you will have to make sure that calls to `SetPen`, `SetBrush` etc. work. Also, where you pass `NULL` to these functions, you will need to use an identifier such as `wxNullPen` or `wxNullBrush`.

Dialogs and controls

Labels

Most controls no longer have labels and values as they used to in 1.xx. Instead, labels

should be created separately using `wxStaticText` (the new name for `wxMessage`). This will need some reworking of dialogs, unfortunately; programmatic dialog creation that doesn't use constraints will be especially hard-hit. Perhaps take this opportunity to make more use of dialog resources or constraints. Or consider using the `wxPropertyListView` class which can do away with dialog layout issues altogether by presenting a list of editable properties.

Constructors

All window constructors have two main changes, apart from the label issue mentioned above. Windows now have integer identifiers; and position and size are now passed as `wxPoint` and `wxSize` objects. In addition, some windows have a `wxValidator` argument.

Show versus ShowModal

If you have used or overridden the `wxDialog::Show` function in the past, you may find that modal dialogs no longer work as expected. This is because the function for modal showing is now `wxDialog::ShowModal`. This is part of a more fundamental change in which a control may tell the dialog that it caused the dismissal of a dialog, by calling `wxDialog::EndModal` or `wxWindow::SetReturnCode`. Using this information, `ShowModal` now returns the id of the control that caused dismissal, giving greater feedback to the application than just `TRUE` or `FALSE`.

If you overrode or called `wxDialog::Show`, use `ShowModal` and test for a returned identifier, commonly `wxID_OK` or `wxID_CANCEL`.

wxItem

This is renamed `wxControl`.

wxText, wxMultiText and wxTextWindow

These classes no longer exist and are replaced by the single class `wxTextCtrl`. Multi-line text items are created using the `wxTE_MULTILINE` style.

wxButton

Bitmap buttons are now a separate class, instead of being part of `wxBitmap`.

wxMessage

Bitmap messages are now a separate class, `wxStaticBitmap`, and `wxMessage` is renamed `wxStaticText`.

wxGroupBox

`wxGroupBox` is renamed `wxStaticBox`.

wxForm

Note that `wxForm` is no longer supported in `wxWindows 2.0`. Consider using the

`wxPropertyFormView` class instead, which takes standard dialogs and panels and associates controls with property objects. You may also find that the new validation method, combined with dialog resources, is easier and more flexible than using `wxForm`.

Device contexts and painting

In `wxWindows 2.0`, device contexts are used for drawing into, as per 1.xx, but the way they are accessed and constructed is a bit different.

You no longer use **GetDC** to access device contexts for panels, dialogs and canvases. Instead, you create a temporary device context, which means that any window or control can be drawn into. The sort of device context you create depends on where your code is called from. If painting within an **OnPaint** handler, you create a `wxPaintDC`. If not within an **OnPaint** handler, you use a `wxClientDC` or `wxWindowDC`. You can still parameterise your drawing code so that it doesn't have to worry about what sort of device context to create - it uses the DC it is passed from other parts of the program.

You **must** create a `wxPaintDC` if you define an `OnPaint` handler, even if you do not actually use this device context, or painting will not work correctly under Windows.

If you used device context functions with `wxPoint` or `wxIntPoint` before, please note that `wxPoint` now contains integer members, and there is a new class `wxRealPoint`. `wxIntPoint` no longer exists.

`wxMetaFile` and `wxMetaFileDC` have been renamed to `wxMetafile` and `wxMetafileDC`.

Miscellaneous

Strings

`wxString` has replaced `char*` in the majority of cases. For passing strings into functions, this should not normally require you to change your code if the syntax is otherwise the same. This is because C++ will automatically convert a `char*` or `const char*` to a `wxString` by virtue of appropriate `wxString` constructors.

However, when a `wxString` is returned from a function in `wxWindows 2.0` where a `char*` was returned in `wxWindows 1.xx`, your application will need to be changed. Usually you can simplify your application's allocation and deallocation of memory for the returned string, and simply assign the result to a `wxString` object. For example, replace this:

```
char* s = wxFunctionThatReturnsString();
```

```
s = copystring(s); // Take a copy in case it is temporary
.... // Do something with it
delete[] s;
```

with this:

```
wxString s = wxFunctionThatReturnsString();
.... // Do something with it
```

To indicate an empty return value or a problem, a function may return either the empty string ("") or a null string. You can check for a null string with `wxString::IsNull()`.

Use of const

The **const** keyword is now used to denote constant functions that do not affect the object, and for function arguments to denote that the object passed cannot be changed.

This should not affect your application except for where you are overriding virtual functions which now have a different signature. If functions are not being called which were previously, check whether there is a parameter mismatch (or function type mismatch) involving consts.

Try to use the **const** keyword in your own code where possible.

Backward compatibility

Some `wxWindows 1.xx` functionality has been left to ease the transition to 2.0. This functionality (usually) only works if you compile with `WXWIN_COMPATIBILITY` set to 1 in `setup.h`.

Mostly this defines old names to be the new names (e.g. `wxRectangle` is defined to be `wxRect`).

Quick reference

This section allows you to quickly find features that need to be converted.

Include files

Use the form:

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

For precompiled header support, use this form:

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

#ifdef __BORLANDC__
    #pragma hdrstop
#endif

// Any files you want to include if not precompiling by including
// the whole of <wx/wx.h>
#ifndef WX_PRECOMP
    #include <stdio.h>
    #include <wx/setup.h>
    #include <wx/bitmap.h>
    #include <wx/brush.h>
#endif

// Any files you want to include regardless of precompiled headers
#include <wx/toolbar.h>
```

IPC classes

These are now separated out into wxDDEServer/Client/Connection (Windows only) and wxTCPClient/Server/Connection (Windows and Unix). Take care to use wxString for your overridden function arguments, instead of char*, as per the documentation.

MDI style frames

MDI is now implemented as a family of separate classes, so you can't switch to MDI just by using a different frame style. Please see the documentation for the MDI frame classes, and the MDI sample may be helpful too.

OnActivate

Replace the arguments with one wxActivateEvent& argument, make sure the function isn't virtual, and add an EVT_ACTIVATE event table entry.

OnChar

This is now a non-virtual function, with the same wxKeyEvent& argument as before. Add an EVT_CHAR macro to the event table for your window, and the implementation of your function will need very few changes.

OnClose

The old virtual function `OnClose` is now obsolete. Add an `OnCloseWindow` event handler using an `EVT_CLOSE` event table entry. For details about window destruction, see the *Windows Deletion Overview* in the manual. This is a subtle topic so please read it very carefully. Basically, `OnCloseWindow` is now responsible for destroying a window with `Destroy()`, but the default implementation (for example for `wxDialog`) may not destroy the window, so to be sure, always provide this event handler so it is obvious what's going on.

OnEvent

This is now a non-virtual function, with the same `wxMouseEvent&` argument as before. However you may wish to rename it `OnMouseEvent`. Add an `EVT_MOUSE_EVENTS` macro to the event table for your window, and the implementation of your function will need very few changes. However, if you wish to intercept different events using different functions, you can specify specific events in your event table, such as `EVT_LEFT_DOWN`.

Your `OnEvent` function is likely to have references to `GetDC()`, so make sure you create a `wxClientDC` instead. See *Device contexts* (p. 1470).

If you are using a `wxScrolledWindow` (formerly `wxCanvas`), you should call `PrepareDC(dc)` to set the correct translation for the current scroll position.

OnMenuCommand

You need to replace this virtual function with a series of non-virtual functions, one for each case of your old switch statement. Each function takes a `wxCommandEvent&` argument. Create an event table for your frame containing `EVT_MENU` macros, and insert `DECLARE_EVENT_TABLE()` in your frame class, as per the samples.

OnPaint

This is now a non-virtual function, with a `wxPaintEvent&` argument. Add an `EVT_PAINT` macro to the event table for your window.

Your function *must* create a `wxPaintDC` object, instead of using `GetDC` to obtain the device context.

If you are using a `wxScrolledWindow` (formerly `wxCanvas`), you should call `PrepareDC(dc)` to set the correct translation for the current scroll position.

OnSize

Replace the arguments with one `wxSizeEvent&` argument, make it non-virtual, and add

to your event table using EVT_SIZE.

wxApp definition

The definition of OnInit has changed. Return a bool value, not a wxFrame.

Also, do *not* declare a global application object. Instead, use the macros DECLARE_APP and IMPLEMENT_APP as per the samples. Remove any occurrences of IMPLEMENT_WXWIN_MAIN: this is subsumed in IMPLEMENT_APP.

wxButton

For bitmap buttons, use wxBitmapButton.

wxCanvas

Change the name to wxScrolledWindow.

wxDialogBox

Change the name to wxDialog, and for modal dialogs, use ShowModal instead of Show.

wxDialog::Show

If you used **Show** to show a modal dialog or to override the standard modal dialog **Show**, use **ShowModal** instead.

See also

Dialogs and controls (p. 1468)

wxForm

Sorry, this class is no longer available. Try using the wxPropertyListView or wxPropertyFormView class instead, or use .wxr files and validators.

wxPoint

The old wxPoint is called wxRealPoint, and wxPoint now uses integers.

wxRectangle

This is now called `wxRect`.

wxScrollBar

The function names have changed for this class: please refer to the documentation for `wxScrollBar`. Instead of setting properties individually, you will call `SetScrollbar` with several parameters.

wxText, wxMultiText, wxTextWindow

Change all these to `wxTextCtrl`. Add the window style `wxTE_MULTILINE` if you wish to have a multi-line text control.

wxToolBar

This name is an alias for the most popular form of toolbar for your platform. There is now a family of toolbar classes, with for example `wxToolBar95`, `wxToolBarMSW` and `wxToolBarSimple` classes existing under Windows 95.

Toolbar management is supported by frames, so calling `wxFrame::CreateToolBar` and adding tools is usually enough, and the SDI or MDI frame will manage the positioning for you. The client area of the frame is the space left over when the menu bar, toolbar and status bar have been taken into account.

Chapter 13 References

- [1] **Robins, Gabriel.** 1987 (September). *The ISI grapher: a portable tool for displaying graphs pictorially (ISI/RS-87-196)*. Technical report. University of South California.

Chapter 14 Index

PRESS F9 TO REFORMAT INDEX