

wxWidgets 2.6.2: A portable C++ and Python GUI toolkit

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

September, 2005

Contents

Copyright notice.....	ix
Introduction	1
What is wxWidgets?	1
Why another cross-platform development tool?	1
wxWidgets requirements	3
Availability and location of wxWidgets	3
Acknowledgements	4
Multi-platform development with wxWidgets.....	5
Include files	5
Libraries	5
Configuration.....	5
Makefiles.....	6
Windows-specific files	6
Allocating and deleting wxWidgets objects.....	7
Architecture dependency	7
Conditional compilation	8
C++ issues	8
File handling	9
Utilities and libraries supplied with wxWidgets.....	10
Programming strategies	12
Strategies for reducing programming errors.....	12
Strategies for portability.....	12
Strategies for debugging	12
Libraries list	15
Alphabetical class reference	18
wxAcceleratorEntry	18
wxAcceleratorTable.....	19
wxAccessible	22
wxActivateEvent.....	29
wxActiveXContainer	31
wxActiveXEvent	35
Returns the dispatch id of this activex event. This is the numeric value from the .idl file specified by the id().wxApp.....	36
wxArchiveClassFactory	47

wxArchiveEntry	48
wxArchiveInputStream	51
wxArchiverIterator	52
wxArchiveNotifier	55
wxArchiveOutputStream.....	55
wxArray.....	57
wxSortedArray doesn't have this function because it is always sorted.wxArrayString	70
wxArtProvider.....	75
wxAutomationObject	80
wxBitmap	84
wxBitmapButton	96
wxBitmapDataObject.....	103
wxBitmapHandler	104
wxBoxSizer	107
wxBrush	108
wxBrushList	114
wxBufferedDC.....	116
wxBufferedPaintDC.....	118
wxBufferedInputStream	118
wxBufferedOutputStream	119
wxBusyCursor.....	120
wxBusyInfo	121
wxButton	122
wxCalculateLayoutEvent	125
wxCalendarCtrl	127
wxCalendarDateAttr	132
wxCalendarEvent	135
wxCaret	136
wxCheckBox	138
wxCheckListBox.....	142
wxChoice	145
This is implemented for Motif only and doesn't do anything under other platforms.wxChoicebook	148
wxClassInfo	148
wxClient	150
wxClientDC	151
wxClientData.....	152
wxClientDataContainer.....	153
wxClipboard	154
wxCloseEvent	157
wxCmdLineParser.....	159

wxColour	168
wxColourData	172
wxColourDatabase	173
wxColourDialog	175
wxComboBox	176
wxCommand	182
wxCommandEvent	184
wxCommandProcessor	189
wxCondition	193
wxConfigBase	196
wxConnection	210
wxContextMenuEvent	214
wxContextHelp	215
wxContextHelpButton	216
wxControl	218
wxControlWithItems	219
SetSelection (p. ??)wxCountingOutputStream	226
wxCriticalSection	227
wxCriticalSectionLocker	228
wxCSSConv	229
wxCursor	230
wxCustomDataObject	235
wxDataFormat	237
wxDataInputStream	239
wxDataObject	242
wxDataObjectComposite	246
wxDataObjectSimple	247
wxDataOutputStream	248
wxDateEvent	251
wxDatePickerCtrl	251
wxDateSpan	255
wxDateTime	260
wxDateTimeHolidayAuthority	287
wxDateTimeWorkDays	287
wxDb	287
wxDbColDataPtr	320
wxDbColDef	320
wxDbColFor	321
wxDbColInf	322
wxDbConnectInf	323

wxDblIdxDef	328
wxDblInf	329
wxDbTable	329
wxDbTableInf	367
wxDbGridCollInfo	367
wxDbGridTableBase	369
wxDC	372
wxDCClipper	391
wxDDEClient	392
wxDDEConnection	393
wxDDEServer	397
wxDebugContext	398
wxDebugStreamBuf	403
wxDebugReport	404
wxDebugReportCompress	408
wxDebugReportPreview	409
wxDebugReportPreviewStd	409
wxDebugReportUpload	410
wxDelegateRendererNative	411
wxDialog	412
<i>wxDialog::EndModal</i> (p. ??), <i>wxDialog::GetReturnCode</i> (p. ??), <i>wxDialog::SetReturnCode</i> (p. ??) <i>wxDialUpEvent</i>	422
wxDialUpManager	422
wxDir	426
wxDirDialog	429
wxDirTraverser	432
wxDisplay	433
wxDllLoader	436
wxDocChildFrame	439
wxDocManager	441
wxDocMDIChildFrame	449
wxDocMDIParentFrame	451
wxDocParentFrame	452
wxDocTemplate	454
wxDocument	459
wxDragImage	466
wxDropFilesEvent	470
wxDropSource	472
wxDropTarget	475
wxDynamicLibrary	478
wxDynamicLibraryDetails	481

wxEncodingConverter	482
wxEraseEvent	486
wxEvent	487
wxEvtHandler	490
wxFFFile	499
wxFFFileInputStream	504
wxFFFileOutputStream	505
wxFFFileStream	506
wxFile	506
wxFileConfig	513
wxFileDataObject	514
wxFileDialog	515
wxFileDropTarget	519
wxFileHistory	520
wxFileInputStream	523
wxFileName	524
wxFileOutputStream	541
wxFileStream	542
wxFileSystem	542
wxFileSystemHandler	545
wxFileType	547
wxFilterInputStream	551
wxFilterOutputStream	552
wxFindDialogEvent	553
wxFindReplaceData	554
wxFindReplaceDialog	556
wxFlexGridSizer	557
Note that this method does not trigger relayayout.wxFocusEvent	560
wxFont	561
wxFontData	571
wxFontDialog	574
wxFontEnumerator	575
wxFontList	577
wxFontMapper	578
wxFrame	582
wxFSFile	593
wxFTP	595
wxGauge	601
wxGBPosition	605
wxGBSizerItem	607

wxGBSpan	608
wxGDIObject	609
wxGenericDirCtrl	610
wxGenericValidator	614
wxGLCanvas	616
wxGLContext	619
wxGrid	621
wxGridCellAttr	654
wxGridBagSizer	657
wxGridCellBoolEditor	660
wxGridCellChoiceEditor	660
wxGridCellEditor	661
wxGridCellFloatEditor	663
wxGridCellNumberEditor	664
wxGridCellTextEditor	665
wxGridEditorCreatedEvent	666
wxGridEvent	667
wxGridRangeSelectEvent	671
wxGridSizeEvent	673
wxGridCellBoolRenderer	674
wxGridCellFloatRenderer	675
wxGridCellNumberRenderer	676
wxGridCellRenderer	677
wxGridCellStringRenderer	678
wxGridTableBase	678
wxGridSizer	682
wxHashMap	684
wxHashSet	688
wxHashTable	692
wxHelpController	694
wxHelpControllerHelpProvider	700
wxHelpEvent	701
wxHelpProvider	702
wxHtmlCell	704
wxHtmlColourCell	709
wxHtmlContainerCell	710
wxHtmlDCRenderer	714
wxHtmlEasyPrinting	717
wxHtmlFilter	720
wxHtmlHelpController	721

wxHtmlHelpData	726
wxHtmlHelpDialog.....	727
wxHtmlHelpFrame.....	729
wxHtmlHelpWindow	730
wxHtmlModalHelp	734
wxHtmlLinkInfo.....	735
wxHtmlListBox	736
wxHtmlParser.....	738
wxHtmlPrintout	743
wxHtmlTag.....	745
wxHtmlTagHandler.....	748
wxHtmlTagsModule.....	750
wxHtmlWidgetCell	750
wxHtmlWindow	751
wxHtmlWinParser.....	760
wxHtmlWinTagHandler.....	766
wxHTTP	766
wxHVScrolledWindow	768
Set the number of rows and columns the window contains. The derived class must provide the heights for all rows and the widths for all columns with indices up to the respective values given here in its <i>OnGetRowHeight()</i> (p. ??) and <i>OnGetColumnWidth()</i> (p. ??) implementations.	
wxIcon	778
wxIconBundle.....	785
wxIconLocation	786
wxIconizeEvent	787
wxIdleEvent	788
wxImage	790
wxImageHandler	814
wxImageList.....	818
wxIndividualLayoutConstraint	823
wxInitDialogEvent.....	825
wxInputStream	826
wxIPAddress	829
wxIPv4address	831
wxJoystick.....	832
wxJoystickEvent.....	838
wxKeyEvent	841
wxLayoutAlgorithm	846
wxLayoutConstraints	849
wxList.....	851
wxListbook	858

wxListBox.....	858
wxListCtrl	864
wxListEvent.....	884
wxListItem.....	887
wxListItemAttr	891
wxListView	893
wxLocale.....	895
wxLog	903
wxLogChain	909
wxLogGui.....	911
wxLogNull	911
wxLogPassThrough	913
wxLogStderr.....	913
wxLogStream	914
wxLogTextCtrl.....	914
wxLogWindow.....	915
wxLongLong	916
wxMask.....	920
wxMaximizeEvent	922
wxMBConv.....	923

Copyright notice

Copyright (c) 1992-2006 Julian Smart, Robert Roebling, Vadim Zeitlin and other
members of the wxWidgets team
Portions (c) 1996 Artificial Intelligence Applications Institute

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

wxWindows Library License, Version 3.1

Copyright (c) 1998-2005 Julian Smart, Robert Roebling et al

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

WXWINDOWS LIBRARY LICENCE 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 Licence as published by the Free Software Foundation; either version 2 of the Licence, 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 Licence for more details.

You should have received a copy of the GNU Library General Public Licence 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 licenced under the wxWindows Library Licence, applying either version 3.1 of the Licence, or (at your option) any later version of the Licence as published by the copyright holders of version 3.1 of the Licence document.
2. The exception is that you may use, copy, link, modify and distribute under your own terms, binary object code versions of works based on the Library.
3. If you copy code from files distributed under the terms of the GNU General Public Licence or the GNU Library General Public Licence into a copy of this library, as this licence 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!

Introduction

What is wxWidgets?

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

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

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

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

Why another cross-platform development tool?

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

1. low price;
2. source availability;
3. simplicity of programming;
4. support for a wide range of compilers.

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

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

Version 2 goes much further than previous versions in terms of generality and features, allowing applications to be produced that are often indistinguishable from those

produced using single-platform toolkits such as Motif, GTK+ and MFC.

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

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

- Low cost (free, in fact!)
- You get the source.
- Available on a variety of popular platforms.
- Works with almost all popular C++ compilers and Python.
- Over 50 example programs.
- Over 1000 pages of printable and on-line documentation.
- Includes Tex2RTF, to allow you to produce your own documentation in Windows Help, HTML and Word RTF formats.
- Simple-to-use, object-oriented API.
- Flexible event system.
- Graphics calls include lines, rounded rectangles, splines, polylines, etc.
- Constraint-based and sizer-based layouts.
- Print/preview and document/view architectures.
- Toolbar, notebook, tree control, advanced list control classes.
- PostScript generation under Unix, normal MS Windows printing on the PC.
- MDI (Multiple Document Interface) support.
- Can be used to create DLLs under Windows, dynamic libraries on Unix.
- Common dialogs for file browsing, printing, colour selection, etc.
- Under MS Windows, support for creating metafiles and copying them to the clipboard.

- An API for invoking help from applications.
- Ready-to-use HTML window (supporting a subset of HTML).
- Network support via a family of socket and protocol classes.
- Support for platform independent image processing.
- Built-in support for many file formats (BMP, PNG, JPEG, GIF, XPM, PNM, PCX).

wxWidgets requirements

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

(a) MS-Windows:

1. A 32-bit or 64-bit PC running MS Windows.
2. A Windows compiler: MS Visual C++ (embedded Visual C++ for wxWinCE port), Borland C++, Watcom C++, Cygwin, MinGW, Metrowerks CodeWarrior, Digital Mars C++. See `install.txt` for details about compiler version supported.
3. At least 100 MB of disk space for source tree and additional space for libraries and application building (depends on compiler and build settings).

(b) Unix:

1. Almost any C++ compiler, including GNU C++ (EGCS 1.1.1 or above).
2. Almost any Unix workstation, and one of: GTK+ 1.2, GTK+ 2.0, Motif 1.2 or higher, Lesstif. If using the wxX11 port, no such widget set is required.
3. At least 100 MB of disk space for source tree and additional space for libraries and application building (depends on compiler and build settings).

(c) Mac OS/Mac OS X:

1. A PowerPC Mac running Mac OS 8.6/9.x (eg. Classic) or Mac OS X 10.x.
2. CodeWarrior 5.3, 6 or 7 for Classic Mac OS.
3. The Apple Developer Tools (eg. GNU C++), CodeWarrior 7 or above for Mac OS X.
4. At least 100 MB of disk space for source tree and additional space for libraries and application building (depends on compiler and build settings).

Availability and location of wxWidgets

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

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

Acknowledgements

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

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

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

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

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

Multi-platform development with wxWidgets

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

Include files

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

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

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

#ifdef __BORLANDC__
#pragma hdrstop
#endif

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

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

The file `"wx/wxprec.h"` includes `"wx/wx.h"`. Although this incantation may seem quirky, it is in fact the end result of a lot of experimentation, and several Windows compilers to use precompilation which is largely automatic for compilers with necessary support. Currently it is used for Visual C++ (including embedded Visual C++), Borland C++, Open Watcom C++, Digital Mars C++ and newer versions of GCC. Some compilers might need extra work from the application developer to set the build environment up as necessary for the support.

Libraries

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

Configuration

When using project files and makefiles directly to build wxWidgets, options are configurable in the file `"wx/XXX/setup.h"` where XXX is the required platform (such as `msw`, `motif`, `gtk`, `mac`). Some settings are a matter of taste, some help with platform-specific problems, and others can be set to minimize the size of the library. Please see

the `setup.h` file and `install.txt` files for details on configuration.

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

Makefiles

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

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

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

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

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

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

Windows-specific files

wxWidgets application compilation under MS Windows requires at least one extra file: a resource file.

Resource file

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

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

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

```
wxicon icon wx.ico
```

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

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

Allocating and deleting wxWidgets objects

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

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

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

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

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

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

Architecture dependency

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

`wxInt32`, `wxInt16`, `wxInt8`, `wxUInt32`, `wxUInt16` = `wxWord`, `wxUInt8` = `wxByte`

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

The macros handling bit-swapping with respect to the applications endianness are described in the *Byte order macros* (p. **Error! Bookmark not defined.**) section.

Conditional compilation

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

C++ issues

The following documents some miscellaneous C++ issues.

Templates

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

RTTI

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

Type of NULL

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

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

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

Precompiled headers

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

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

A related problem is that for compilers that don't have precompiled headers, including a

lot of header files slows down compilation considerably. For this reason, you will find (in the common X and Windows parts of the library) conditional compilation that under Unix, includes a minimal set of headers; and when using Visual C++, includes `wx.h`. This should help provide the optimal compilation for each compiler, although it is biased towards the precompiled headers facility available in Microsoft C++.

File handling

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

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

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

Utilities and libraries supplied with wxWidgets

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

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

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

Helpview Helpview is a program for displaying wxWidgets HTML Help files. In many cases, you may wish to use the wxWidgets HTML Help classes from within your application, but this provides a handy stand-alone viewer. See *wxHTML Notes* (p. **Error! Bookmark not defined.**) for more details. You can find it in `samples/html/helpview`.

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

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

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

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

XRC resource system This is the sizer-aware resource system, and uses XML-based resource specifications that can be generated by tools such as wxDesigner (<http://www.roebling.de>). You can find this in `src/xrc`, `include/wx/xrc`, `samples/xrc`. For more information, see the *XML-based resource system overview* (p. **Error! Bookmark not defined.**).

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

and `contrib/samples/ogl`.

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

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

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

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

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

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

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

Programming strategies

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

Strategies for reducing programming errors

Use ASSERT

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

Use wxString in preference to character arrays

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

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

Strategies for portability

Use relative positioning or constraints

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

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

Use wxWidgets resource files

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

Strategies for debugging

Positive thinking

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

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

Simplify the problem

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

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

Use a debugger

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

Use logging functions

There is a variety of logging functions that you can use in your program: see *Logging functions* (p. **Error! Bookmark not defined.**).

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

Use the wxWidgets debugging facilities

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

You should also use *debug macros* (p. **Error! Bookmark not defined.**) 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. **Error! Bookmark not defined.**) for further information.

Libraries list

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

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



wxBase

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

wxNet

Classes for network access:

- `wxSocket` classes (`wxSocketClient` (p. **Error! Bookmark not defined.**), `wxSocketServer` (p. **Error! Bookmark not defined.**) and related classes)
- `wxSocketOutputStream` (p. **Error! Bookmark not defined.**) and `wxSocketInputStream` (p. **Error! Bookmark not defined.**)

- sockets-based IPC classes (*wxTCPServer* (p. 397), *wxTCPClient* (p. 392) and *wxTCPConnection* (p. 393))
- *wxURL* (p. **Error! Bookmark not defined.**)
- *wxInternetFSHandler* (a *wxFileSystem handler* (p. **Error! Bookmark not defined.**)) Requires *wxBase*.

wxXML

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

wxCore

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

wxAdvanced

Advanced or rarely used GUI classes:

- *wxBufferedDC*
- *wxCalendarCtrl* (p. 127)
- *wxGrid classes* (p. **Error! Bookmark not defined.**)
- *wxJoystick* (p. 832)
- *wxLayoutAlgorithm* (p. 846)
- *wxSplashScreen* (p. **Error! Bookmark not defined.**)
- *wxTaskBarIcon* (p. **Error! Bookmark not defined.**)
- *wxSound* (p. **Error! Bookmark not defined.**)
- *wxWizard* (p. **Error! Bookmark not defined.**)
- *wxSashLayoutWindow* (p. **Error! Bookmark not defined.**)
- *wxSashWindow* (p. **Error! Bookmark not defined.**)

Requires *wxCore* and *wxBase*.

wxMedia

Miscellaneous classes related to multimedia. Currently this library only contains *wxMediaCtrl* (p. **Error! Bookmark not defined.**) but more classes will be added in the future.

Requires wxCore and wxBase.

wxGL

This library contains *wxGLCanvas* (p. 616) class for integrating OpenGL library with wxWidgets. Unlike all others, this library is *not* part of the monolithic library, it is always built as separate library. Requires wxCore and wxBase.

wxHTML

Simple HTML renderer and other *HTML rendering classes* (p. **Error! Bookmark not defined.**) are contained in this library, as well as *wxHtmlHelpController* (p. 721), *wxBestHelpController* (p. 694) and *wxHtmlListBox* (p. 736). Requires wxCore and wxBase.

wxODBC

Database classes (p. **Error! Bookmark not defined.**). Requires wxBase.

wxQA

This is the library containing extra classes for quality assurance. Currently it only contains *wxDebugReport* (p. 404) and related classes, but more will be added to it in the future.

Requires wxCore, wxBase and wxXML.

wxDbGrid

wxDbGridTableBase (p. 369) class which combines *wxGrid* (p. 621) and *wxDbTable* (p. 329). Requires wxODBC and wxAdvanced.

wxXRC

This library contains *wxXmlResource* (p. **Error! Bookmark not defined.**) class that provides access to XML resource files in XRC format. Requires wxXML, wxCore, wxAdvanced and wxHTML.

Alphabetical class reference

wxAcceleratorEntry

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

Derived from

None

Include files

<wx/accel.h>

See also

wxAcceleratorTable (p. 19), *wxWindow::SetAcceleratorTable* (p. **Error! Bookmark not defined.**)

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. **Error! Bookmark not defined.**) 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. **Error! Bookmark not defined.**) 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. **Error! Bookmark not defined.**)

Include files

<wx/accel.h>

Example

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

Remarks

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

See also

wxAcceleratorEntry (p. 18), *wxWindow::SetAcceleratorTable* (p. **Error! Bookmark not defined.**)

wxAcceleratorTable::wxAcceleratorTable

wxAcceleratorTable()

Default constructor.

wxAcceleratorTable(const wxAcceleratorTable& *bitmap*)

Copy constructor.

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

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

wxAcceleratorTable(const wxString& *resource*)

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

Parameters

n

Number of accelerator entries.

entries

The array of entries.

resource

Name of a Windows accelerator.

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

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

wxAcceleratorTable::~~wxAcceleratorTable

~wxAcceleratorTable()

Destroys the wxAcceleratorTable object.

wxAcceleratorTable::Ok

bool Ok() const

Returns true if the accelerator table is valid.

wxAcceleratorTable::operator =

wxAcceleratorTable& operator =(const wxAcceleratorTable& accel)

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

Parameters

accel

Accelerator table to assign.

Return value

Returns reference to this object.

wxAcceleratorTable::operator ==

bool operator ==(const wxAcceleratorTable& accel)

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

Parameters

accel

Accelerator table to compare with

Return value

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

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

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

Parameters*accel*

Accelerator table to compare with

Return value

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

wxAccessible

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

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

To use this class, derive from wxAccessible, implement appropriate functions, and associate an object of the class with a window using *wxWindow::SetAccessible* (p. **Error! Bookmark not defined.**).

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

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

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

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

This class is compiled into wxWidgets only if the wxUSE_ACCESSIBILITY setup symbol is set to 1.

Derived from*wxObject* (p. **Error! Bookmark not defined.**)**Include files**`<wx/access.h>`

Data structures

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

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

Directions of navigation are represented by the following:

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

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

```
typedef enum {
    wxROLE_NONE,
    wxROLE_SYSTEM_ALERT,
    wxROLE_SYSTEM_ANIMATION,
    wxROLE_SYSTEM_APPLICATION,
    wxROLE_SYSTEM_BORDER,
    wxROLE_SYSTEM_BUTTONDROPDOWN,
    wxROLE_SYSTEM_BUTTONDROPDOWNGRID,
    wxROLE_SYSTEM_BUTTONMENU,
    wxROLE_SYSTEM_CARET,
    wxROLE_SYSTEM_CELL,
    wxROLE_SYSTEM_CHARACTER,
    wxROLE_SYSTEM_CHART,
    wxROLE_SYSTEM_CHECKBUTTON,
    wxROLE_SYSTEM_CLIENT,
    wxROLE_SYSTEM_CLOCK,
    wxROLE_SYSTEM_COLUMN,
    wxROLE_SYSTEM_COLUMNHEADER,
    wxROLE_SYSTEM_COMBOBOX,
    wxROLE_SYSTEM_CURSOR,
    wxROLE_SYSTEM_DIAGRAM,
    wxROLE_SYSTEM_DIAL,
    wxROLE_SYSTEM_DIALOG,
    wxROLE_SYSTEM_DOCUMENT,
    wxROLE_SYSTEM_DROPLIST,
    wxROLE_SYSTEM_EQUATION,
    wxROLE_SYSTEM_GRAPHIC,
    wxROLE_SYSTEM_GRIP,
    wxROLE_SYSTEM_GROUPING,
```

```
wxROLE_SYSTEM_HELPBALLOON,  
wxROLE_SYSTEM_HOTKEYFIELD,  
wxROLE_SYSTEM_INDICATOR,  
wxROLE_SYSTEM_LINK,  
wxROLE_SYSTEM_LIST,  
wxROLE_SYSTEM_LISTITEM,  
wxROLE_SYSTEM_MENUBAR,  
wxROLE_SYSTEM_MENUITEM,  
wxROLE_SYSTEM_MENUPOPUP,  
wxROLE_SYSTEM_OUTLINE,  
wxROLE_SYSTEM_OUTLINEITEM,  
wxROLE_SYSTEM_PAGETAB,  
wxROLE_SYSTEM_PAGETABLIST,  
wxROLE_SYSTEM_PANE,  
wxROLE_SYSTEM_PROGRESSBAR,  
wxROLE_SYSTEM_PROPERTYPAGE,  
wxROLE_SYSTEM_PUSHBUTTON,  
wxROLE_SYSTEM_RADIOBUTTON,  
wxROLE_SYSTEM_ROW,  
wxROLE_SYSTEM_ROWHEADER,  
wxROLE_SYSTEM_SCROLLBAR,  
wxROLE_SYSTEM_SEPARATOR,  
wxROLE_SYSTEM_SLIDER,  
wxROLE_SYSTEM_SOUND,  
wxROLE_SYSTEM_SPINBUTTON,  
wxROLE_SYSTEM_STATICTEXT,  
wxROLE_SYSTEM_STATUSBAR,  
wxROLE_SYSTEM_TABLE,  
wxROLE_SYSTEM_TEXT,  
wxROLE_SYSTEM_TITLEBAR,  
wxROLE_SYSTEM_TOOLBAR,  
wxROLE_SYSTEM_TOOLTIP,  
wxROLE_SYSTEM_WHITESPACE,  
wxROLE_SYSTEM_WINDOW  
} wxAccRole
```

Objects are represented by the following type:

```
typedef enum {  
    wxOBJID_WINDOW = 0x00000000,  
    wxOBJID_SYSMENU = 0xFFFFFFFF,  
    wxOBJID_TITLEBAR = 0xFFFFFFFFE,  
    wxOBJID_MENU = 0xFFFFFFFFD,  
    wxOBJID_CLIENT = 0xFFFFFFFFC,  
    wxOBJID_VSCROLL = 0xFFFFFFFFB,  
    wxOBJID_HSCROLL = 0xFFFFFFFFA,  
    wxOBJID_SIZEGRIP = 0xFFFFFFFF9,  
    wxOBJID_CARET = 0xFFFFFFFF8,  
    wxOBJID_CURSOR = 0xFFFFFFFF7,  
    wxOBJID_ALERT = 0xFFFFFFFF6,  
    wxOBJID_SOUND = 0xFFFFFFFF5  
} wxAccObject
```

Selection actions are identified by this type:

```
typedef enum  
{  
    wxACC_SEL_NONE = 0,  
    wxACC_SEL_TAKEFOCUS = 1,  
}
```

```

wxACC_SEL_TAKESELECTION    = 2,
wxACC_SEL_EXTENDSELECTION = 4,
wxACC_SEL_ADDSELECTION     = 8,
wxACC_SEL_REMOVESELECTION = 16
} wxAccSelectionFlags

```

States are represented by the following:

```

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

```

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

```

#define wxACC_EVENT_SYSTEM_SOUND            0x0001
#define wxACC_EVENT_SYSTEM_ALERT            0x0002
#define wxACC_EVENT_SYSTEM_FOREGROUND      0x0003
#define wxACC_EVENT_SYSTEM_MENUSTART       0x0004
#define wxACC_EVENT_SYSTEM_MENUEND         0x0005
#define wxACC_EVENT_SYSTEM_MENUPOPUPSTART  0x0006
#define wxACC_EVENT_SYSTEM_MENUPOPUPEND    0x0007
#define wxACC_EVENT_SYSTEM_CAPTURESTART    0x0008
#define wxACC_EVENT_SYSTEM_CAPTUREEND      0x0009
#define wxACC_EVENT_SYSTEM_MOVESIZESTART   0x000A
#define wxACC_EVENT_SYSTEM_MOVESIZEEND    0x000B
#define wxACC_EVENT_SYSTEM_CONTEXTHELPSTART 0x000C
#define wxACC_EVENT_SYSTEM_CONTEXTHELPEND  0x000D
#define wxACC_EVENT_SYSTEM_DRAGDROPSTART   0x000E
#define wxACC_EVENT_SYSTEM_DRAGDROPEND     0x000F
#define wxACC_EVENT_SYSTEM_DIALOGSTART     0x0010
#define wxACC_EVENT_SYSTEM_DIALOGEND       0x0011
#define wxACC_EVENT_SYSTEM_SCROLLINGSTART  0x0012
#define wxACC_EVENT_SYSTEM_SCROLLINGEND    0x0013
#define wxACC_EVENT_SYSTEM_SWITCHSTART     0x0014
#define wxACC_EVENT_SYSTEM_SWITCHEND       0x0015
#define wxACC_EVENT_SYSTEM_MINIMIZESTART   0x0016

```

```
#define wxACC_EVENT_SYSTEM_MINIMIZEEND      0x0017
#define wxACC_EVENT_OBJECT_CREATE           0x8000
#define wxACC_EVENT_OBJECT_DESTROY         0x8001
#define wxACC_EVENT_OBJECT_SHOW            0x8002
#define wxACC_EVENT_OBJECT_HIDE            0x8003
#define wxACC_EVENT_OBJECT_REORDER         0x8004
#define wxACC_EVENT_OBJECT_FOCUS           0x8005
#define wxACC_EVENT_OBJECT_SELECTION       0x8006
#define wxACC_EVENT_OBJECT_SELECTIONADD    0x8007
#define wxACC_EVENT_OBJECT_SELECTIONREMOVE 0x8008
#define wxACC_EVENT_OBJECT_SELECTIONWITHIN 0x8009
#define wxACC_EVENT_OBJECT_STATECHANGE     0x800A
#define wxACC_EVENT_OBJECT_LOCATIONCHANGE  0x800B
#define wxACC_EVENT_OBJECT_NAMECHANGE      0x800C
#define wxACC_EVENT_OBJECT_DESCRIPTIONCHANGE 0x800D
#define wxACC_EVENT_OBJECT_VALUECHANGE     0x800E
#define wxACC_EVENT_OBJECT_PARENTCHANGE    0x800F
#define wxACC_EVENT_OBJECT_HELPCHANGE      0x8010
#define wxACC_EVENT_OBJECT_DEFACTIONCHANGE 0x8011
#define wxACC_EVENT_OBJECT_ACCELERATORCHANGE 0x8012
```

wxAccessible::wxAccessible

wxAccessible(wxWindow* win = NULL)

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

wxAccessible::~~wxAccessible

~wxAccessible()

Destructor.

wxAccessible::DoDefaultAction

virtual wxAccStatus DoDefaultAction(int childId)

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

wxAccessible::GetChild

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

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

wxAccessible::GetChildCount

virtual wxAccStatus GetChildCount(int* childCount)

Returns the number of children in *childCount*.

wxAcessible::GetDefaultAction

virtual wxAccStatus GetDefaultAction(int childId, wxString* actionName)

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

wxAcessible::GetDescription

virtual wxAccStatus GetDescription(int childId, wxString* description)

Returns the description for this object or a child.

wxAcessible::GetFocus

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

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

wxAcessible::GetHelpText

virtual wxAccStatus GetHelpText(int childId, wxString* helpText)

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

wxAcessible::GetKeyboardShortcut

virtual wxAccStatus GetKeyboardShortcut(int childId, wxString* shortcut)

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

wxAcessible::GetLocation

virtual wxAccStatus GetLocation(wxRect& rect, int elementId)

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

wxAcessible::GetName

virtual wxAccStatus GetName(int childId, wxString* name)

Gets the name of the specified object.

wxAccessible::GetParent

virtual wxAccStatus GetParent(wxAccessible parent)**

Returns the parent of this object, or NULL.

wxAccessible::GetRole

virtual wxAccStatus GetRole(int childId, wxAccRole* role)

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

wxAccessible::GetSelections

virtual wxAccStatus GetSelections(wxVariant* selections)

Gets a variant representing the selected children of this object.

Acceptable values are:

- a null variant (IsNull() returns TRUE)
- a list variant (GetType() == wxT("list"))
- an integer representing the selected child element, or 0 if this object is selected (GetType() == wxT("long"))
- a "void*" pointer to a wxAccessible child object

wxAccessible::GetState

virtual wxAccStatus GetState(int childId, long* state)

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

wxAccessible::GetValue

virtual wxAccStatus GetValue(int childId, wxString* strValue)

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

wxAccessible::GetWindow

wxWindow* GetWindow()

Returns the window associated with this object.

wxAccessible::HitTest

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

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

pt is in screen coordinates.

wxAccessible::Navigate

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

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

wxAccessible::NotifyEvent

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

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

wxAccessible::Select

virtual wxAccStatus Select(int childId, wxAccSelectionFlags selectFlags)

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

wxAccessible::SetWindow

void SetWindow(wxWindow* window)

Sets the window associated with this object.

wxActivateEvent

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

Derived from

wxEvent (p. 487)

wxObject (p. **Error! Bookmark not defined.**)

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

Remarks

A top-level window (a dialog or frame) receives an activate event when it 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 inactivated resulting in all application frames being inactive. (Windows only)

Please note that usually you should call *event.Skip()* (p. 490) in your handlers for these events as not doing so can result in strange effects.

See also

Event handling overview (p. **Error! Bookmark not defined.**), *wxApp::IsActive* (p. 39)

wxActivateEvent::wxActivateEvent

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

Constructor.

wxActivateEvent::GetActive

bool GetActive() const

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

wxActiveXContainer

`wxActiveXContainer` is a host for an activex control on Windows (and as such is a platform-specific class). Note that the `HWND` that the class contains is the actual `HWND`

of the activex control so using dynamic events and connecting to `wxEVT_SIZE`, for example, will receive the actual size message sent to the control.

It is somewhat similar to the ATL class `CXWindow` in operation.

The size of the activex control's content is generally guaranteed to be that of the client size of the parent of this `wxActiveXContainer`.

You can also process activex events through `wxEVT_ACTIVEX` or the corresponding message map macro `EVT_ACTIVEX`.

See also

`wxActiveXEvent` (p. 35)

Derived from

`wxControl` (p. 218)

Include files

<wx/msw/ole/activex.h>

Example

This is an example of how to use the Adobe Acrobat Reader ActiveX control to read PDF files (requires Acrobat Reader 4 and up). Controls like this are typically found and dumped from `OLEVIEW.exe` that is distributed with Microsoft Visual C++. This example also demonstrates how to create a backend for `wxMediaCtrl` (p. **Error! Bookmark not defined.**).

```
//+++++
//
// wxPDFMediaBackend
//
//
http://partners.adobe.com/public/developer/en/acrobat/sdk/pdf/iac/
IACOverview.pdf
//+++++

#include "wx/mediactrl.h" // wxMediaBackendCommonBase
#include "wx/msw/ole/activex.h" // wxActiveXContainer
#include "wx/msw/ole/automtn.h" // wxAutomationObject

const IID DIID__DPdf =
{0xCA8A9781,0x280D,0x11CF,{0xA2,0x4D,0x44,0x45,0x53,0x54,0x00,0x00}};
const IID DIID__DPdfEvents =
{0xCA8A9782,0x280D,0x11CF,{0xA2,0x4D,0x44,0x45,0x53,0x54,0x00,0x00}};
const CLSID CLSID_Pdf =
{0xCA8A9780,0x280D,0x11CF,{0xA2,0x4D,0x44,0x45,0x53,0x54,0x00,0x00}};

class WXDLLIMPEXP_MEDIA wxPDFMediaBackend : public
wxMediaBackendCommonBase
{
```

```

public:
    wxPDFMediaBackend() : m_pAX(NULL) {}
    virtual ~wxPDFMediaBackend()
    {
        if(m_pAX)
        {
            m_pAX->DissociateHandle();
            delete m_pAX;
        }
    }
    virtual bool CreateControl(wxControl* ctrl, wxWindow* parent,
                               wxWindowID id,
                               const wxPoint& pos,
                               const wxSize& size,
                               long style,
                               const wxValidator& validator,
                               const wxString& name)
    {
        IDispatch* pDispatch;
        if( ::CoCreateInstance(CLSID_Pdf, NULL,
                               CLSCTX_INPROC_SERVER,
                               DIID__DPdf, (void**)&pDispatch)
        != 0 )
            return false;

        m_PDF.SetDispatchPtr(pDispatch); // wxAutomationObject
        will release itself

        if ( !ctrl->wxControl::Create(parent, id, pos, size,
                                       (style & ~wxBORDER_MASK) |
wxBORDER_NONE,
                                       validator, name) )
            return false;

        m_ctrl = wxStaticCast(ctrl, wxMediaCtrl);
        m_pAX = new wxActiveXContainer(ctrl,
                                       DIID__DPdf,
                                       pDispatch);

        wxPDFMediaBackend::ShowPlayerControls(wxMEDIACTRLPLAYERCONTROLS_NO
NE);
        return true;
    }

    virtual bool Play()
    {
        return true;
    }
    virtual bool Pause()
    {
        return true;
    }
    virtual bool Stop()
    {
        return true;
    }

    virtual bool Load(const wxString& fileName)
    {
        if(m_PDF.CallMethod(wxT("LoadFile"), fileName).GetBool())
        {
            m_PDF.CallMethod(wxT("setCurrentPage"),
wxVariant((long)0));

```

```
        NotifyMovieLoaded(); // initial refresh
        wxSizeEvent event;
        m_pAX->OnSize(event);
        return true;
    }

    return false;
}
virtual bool Load(const wxURI& location)
{
    return m_PDF.CallMethod(wxT("LoadFile"),
location.BuildUnescapedURI()).GetBool();
}
virtual bool Load(const wxURI& WXUNUSED(location),
                  const wxURI& WXUNUSED(proxy))
{
    return false;
}

virtual wxMediaState GetState()
{
    return wxMEDIASTATE_STOPPED;
}

virtual bool SetPosition(wxLongLong where)
{
    m_PDF.CallMethod(wxT("setCurrentPage"),
wxVariant((long)where.GetValue()));
    return true;
}
virtual wxLongLong GetPosition()
{
    return 0;
}
virtual wxLongLong GetDuration()
{
    return 0;
}

virtual void Move(int WXUNUSED(x), int WXUNUSED(y),
                  int WXUNUSED(w), int WXUNUSED(h))
{
}
wxSize GetVideoSize() const
{
    return wxDefaultSize;
}

virtual double GetPlaybackRate()
{
    return 0;
}
virtual bool SetPlaybackRate(double)
{
    return false;
}

virtual double GetVolume()
{
    return 0;
}
virtual bool SetVolume(double)
{
    return false;
}
```

```
    }

    virtual bool ShowPlayerControls(wxMediaCtrlPlayerControls
flags)
    {
        if(flags)
        {
            m_PDF.CallMethod(wxT("setShowToolbar"), true);
            m_PDF.CallMethod(wxT("setShowScrollbars"), true);
        }
        else
        {
            m_PDF.CallMethod(wxT("setShowToolbar"), false);
            m_PDF.CallMethod(wxT("setShowScrollbars"), false);
        }

        return true;
    }

    wxActiveXContainer* m_pAX;
    wxAutomationObject m_PDF;

    DECLARE_DYNAMIC_CLASS(wxPDFMediaBackend)
};

IMPLEMENT_DYNAMIC_CLASS(wxPDFMediaBackend, wxMediaBackend);
```

Put this in one of your existant source files and then create a `wxMediaCtrl` with `//[this]` is the parent window, "myfile.pdf" is the PDF file to open

```
wxMediaCtrl* mymediactrl = new wxMediaCtrl(this,
wxT("myfile.pdf"), wxID_ANY,
                                wxDefaultPosition,
wxSize(300,300),
                                0,
wxT("wxPDFMediaBackend"));
```

wxActiveXContainer::wxActiveXContainer

```
wxActiveXContainer(    wxWindow* parent,    REFIID iid,    IUnknown* pUnk,
)
```

Creates this activex container.

parent

parent of this control. Must not be NULL.

iid

COM IID of pUnk to query. Must be a valid interface to an activex control.

pUnk

Interface of activex control

wxActiveXEvent

An event class for handling activex events passed from *wxActiveXContainer* (p. 31). ActiveX events are basically a function call with the parameters passed through an array of *wxVariants* along with a return value that is a *wxVariant* itself. What type the parameters or return value are depends on the context (i.e. what the .idl specifies).

Note that unlike the third party *wxActiveX* function names are not supported.

Derived from

wxCommandEvent (p. 184)

Include files

<wx/msw/ole/activex.h>

Event table macros

EVT_ACTIVEX(func)

Sent when the activex control hosted by *wxActiveXContainer* (p. 31) receives an activex event.

wxActiveXEvent::ParamCount

size_t ParamCount() const

Obtains the number of parameters passed through the activex event.

wxActiveXEvent::ParamType

wxString ParamType(size_t idx) const

Obtains the param type of the param number idx specifies as a string.

wxActiveXEvent::ParamName

wxString ParamName(size_t idx) const

Obtains the param name of the param number idx specifies as a string.

wxActiveXEvent::operator[]

wxVariant& operator[](size_t idx)

Obtains the actual parameter value specified by idx.

wxActiveXEvent::GetDispatchId

DISPID GetDispatchId(int idx) const

Returns the dispatch id of this activex event. This is the numeric value from the .idl file specified by the `id().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. 42);
- allow default processing of events not handled by other objects in the application.

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

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

Derived from

`wxEvtHandler` (p. 490)

`wxObject` (p. **Error! Bookmark not defined.**)

Include files

<wx/app.h>

See also

`wxApp overview` (p. **Error! Bookmark not defined.**)

wxApp::wxApp

wxApp()

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

wxApp::~~wxApp

virtual ~wxApp()

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

wxApp::argc

int argc

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

wxApp::argv**wxChar ** argv**

Command line arguments (after environment-specific processing).

wxApp::CreateLogTarget**virtual wxLog* CreateLogTarget()**

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

See also

wxLog (p. 903)

wxApp::Dispatch**virtual void Dispatch()**

Dispatches the next event in the windowing system event queue.

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

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

See also

wxApp::Pending (p. 44)

wxApp::ExitMainLoop**virtual void ExitMainLoop()**

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

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

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

wxApp::GetAppName**wxString GetAppName() const**

Returns the application name.

Remarks

wxWidgets sets this to a reasonable default before calling *wxApp::OnInit* (p. 42), but the application can reset it at will.

wxApp::GetClassName**wxString GetClassName() const**

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

See also

wxApp::SetClassName (p. 44)

wxApp::GetExitOnFrameDelete**bool GetExitOnFrameDelete() const**

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

See also

wxApp::SetExitOnFrameDelete (p. 45),
wxApp shutdown overview (p. **Error! Bookmark not defined.**)

wxApp::GetInstance**static wxAppConsole * GetInstance()**

Returns the one and only global application object. Usually *wxTheApp* is used instead.

See also

wxApp::SetInstance (p. 45)

wxApp::GetTopWindow**virtual wxWindow * GetTopWindow() const**

Returns a pointer to the top window.

Remarks

If the top window hasn't been set using *wxApp::SetTopWindow* (p. 45), this function will

find the first top-level window (frame or dialog) and return that.

See also

SetTopWindow (p. 45)

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. 46)

wxApp::GetVendorName

wxString GetVendorName() const

Returns the application's vendor name.

wxApp::IsActive

bool IsActive() const

Returns `true` if the application is active, i.e. if one of its windows is currently in the foreground. If this function returns `false` and you need to attract users attention to the application, you may use *wxTopLevelWindow::RequestUserAttention* (p. **Error! Bookmark not defined.**) to do it.

wxApp::IsMainLoopRunning

static bool IsMainLoopRunning()

Returns `true` if the main event loop is currently running, i.e. if the application is inside *OnRun* (p. 43).

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

wxApp::MainLoop

virtual int MainLoop()

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

Return value

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

wxApp::OnAssertFailure

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

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

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

Parameters

file

the name of the source file where the assert occurred

line

the line number in this file where the assert occurred

func

the name of the function where the assert occurred, may be empty if the compiler doesn't support C99 `__FUNCTION__`

cond

the condition of the failed assert in text form

msg

the message specified as argument to `wxASSERT_MSG` (p. **Error! Bookmark not defined.**) or `wxFAIL_MSG` (p. **Error! Bookmark not defined.**), will be `NULL` if just `wxASSERT` (p. **Error! Bookmark not defined.**) or `wxFAIL` (p. **Error! Bookmark not defined.**) was used

wxApp::OnCmdLineError

bool OnCmdLineError(wxCmdLineParser& parser)

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

Return `true` to continue normal execution or `false` to return `false` from `OnInit` (p. 42) thus terminating the program.

See also

OnInitCmdLine (p. 43)

wxApp::OnCmdLineHelp

bool OnCmdLineHelp(wxCmdLineParser& parser)

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

Return `true` to continue normal execution or `false` to return `false` from *OnInit* (p. 42) thus terminating the program.

See also

OnInitCmdLine (p. 43)

wxApp::OnCmdLineParsed

bool OnCmdLineParsed(wxCmdLineParser& parser)

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

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

Return `true` to continue normal execution or `false` to return `false` from *OnInit* (p. 42) thus terminating the program.

See also

OnInitCmdLine (p. 43)

wxApp::OnExceptionInMainLoop

virtual bool OnExceptionInMainLoop()

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

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

Finally note that if the exception is rethrown from here, it can be caught in *OnUnhandledException* (p. 43).

wxApp::OnExit

virtual int OnExit()

Override this member function for any processing which needs to be done as the application is about to exit. `OnExit` is called after destroying all application windows and controls, but before `wxWidgets` cleanup. Note that it is not called at all if *OnInit* (p. 42) failed.

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

wxApp::OnFatalException**void OnFatalException()**

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

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

See also

wxHandleFatalExceptions (p. **Error! Bookmark not defined.**)

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. 45). You may use *OnExit* (p. 42) to clean up anything initialized here, provided that the function returns `true`.

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

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

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

Called from *OnInit* (p. 42) and may be used to initialize the parser with the command line options for this application. The base class versions adds support for a few standard options only.

wxApp::OnRun**virtual int OnRun()**

This virtual function is where the execution of a program written in wxWidgets starts. The default implementation just enters the main loop and starts handling the events until it terminates, either because *ExitMainLoop* (p. 37) has been explicitly called or because the last frame has been deleted and *GetExitOnFrameDelete* (p. 38) flag is `true` (this is the default).

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

wxApp::OnUnhandledException

virtual void OnUnhandledException()

This function is called when an unhandled C++ exception occurs inside *OnRun()* (p. 43) (the exceptions which occur during the program startup and shutdown might not be caught at all). Note that the exception type is lost by now, so if you want to really handle the exception you should override *OnRun()* (p. 43) and put a try/catch clause around the call to the base class version there.

wxApp::ProcessMessage

bool ProcessMessage(WXMSG *msg)

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

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

wxApp::Pending

virtual bool Pending()

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

See also

wxApp::Dispatch (p. 37)

wxApp::SendIdleEvents

bool SendIdleEvents(wxWindow* win, wxIdleEvent& event)

Sends idle events to a window and its children.

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

Remarks

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

See also

wxIdleEvent (p. 788)

wxApp::SetAppName

void SetAppName(const wxString& name)

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

See also

wxApp::GetAppName (p. 38)

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. 38)

wxApp::SetExitOnFrameDelete

void SetExitOnFrameDelete(bool flag)

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

Parameters

flag

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

See also

wxApp::GetExitOnFrameDelete (p. 38),

wxApp shutdown overview (p. **Error! Bookmark not defined.**)

wxApp::SetInstance

static void SetInstance(wxAppConsole* *app*)

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

Parameters

app

Replacement for the global application object.

See also

wxApp::GetInstance (p. 38)

wxApp::SetTopWindow

void SetTopWindow(wxWindow* *window*)

Sets the 'top' window. You can call this from within *wxApp::OnInit* (p. 42) to let wxWidgets know which is the main window. You don't have to set the top window; it is only a convenience so that (for example) certain dialogs without parents can use a specific window as the top window. If no top window is specified by the application, wxWidgets just uses the first frame or dialog in its top-level window list, when it needs to use the top window.

Parameters

window

The new top window.

See also

wxApp::GetTopWindow (p. 39), *wxApp::OnInit* (p. 42)

wxApp::SetVendorName

void SetVendorName(const wxString& *name*)

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

See also

wxApp::GetVendorName (p. 39)

wxApp::SetUseBestVisual

void SetUseBestVisual(bool flag)

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

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

This function currently only has effect under GTK.

Parameters

flag

If true, the app will use the best visual.

wxApp::HandleEvent

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

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

wxApp::Yield

bool Yield(bool onlyIfNeeded = false)

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

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

Note that `Yield()` will not flush the message logs. This is intentional as calling `Yield()` is usually done to quickly update the screen and popping up a message box dialog may be undesirable. If you do wish to flush the log messages immediately (otherwise it will be done during the next idle loop iteration), call `wxLog::FlushActive` (p. 908).

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

wxArchiveClassFactory

An abstract base class which serves as a common interface to archive class factories such as *wxZipClassFactory* (p. **Error! Bookmark not defined.**).

For each supported archive type (such as zip) there is a class factory derived from *wxArchiveClassFactory*, which allows archive objects to be created in a generic way, without knowing the particular type of archive being used.

Derived from

wxObject (p. **Error! Bookmark not defined.**)

Include files

<wx/archive.h>

See also

Archive formats such as zip (p. **Error! Bookmark not defined.**)

Generic archive programming (p. **Error! Bookmark not defined.**)

wxArchiveEntry (p. 48)

wxArchiveInputStream (p. 51)

wxArchiveOutputStream (p. 55)

wxArchiveClassFactory::Get/SetConv

wxMBConv& GetConv() const

void SetConv(wxMBConv& conv)

The *wxMBConv* (p. 923) object that the created streams will use when translating meta-data. The initial default, set by the constructor, is *wxConvLocal*.

wxArchiveClassFactory::GetInternalName

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

Calls the static *GetInternalName()* function for the archive entry type, for example *wxZipEntry::GetInternalName()* (p. **Error! Bookmark not defined.**).

wxArchiveClassFactory::NewEntry

wxArchiveEntry* NewEntry() const

Create a new *wxArchiveEntry* (p. 48) object of the appropriate type.

wxArchiveClassFactory::NewStream

wxArchiveInputStream* NewStream(wxInputStream& stream) const

wxArchiveOutputStream* NewStream(wxOutputStream& stream) const

Create a new *wxArchiveInputStream* (p. 51) or *wxArchiveOutputStream* (p. 55) of the appropriate type.

wxArchiveEntry

An abstract base class which serves as a common interface to archive entry classes such as *wxZipEntry* (p. **Error! Bookmark not defined.**). These hold the meta-data (filename, timestamp, etc.), for entries in archive files such as zips and tars.

Derived from

wxObject (p. **Error! Bookmark not defined.**)

Include files

<wx/archive.h>

See also

Archive formats such as zip (p. **Error! Bookmark not defined.**)

Generic archive programming (p. **Error! Bookmark not defined.**)

wxArchiveInputStream (p. 51)

wxArchiveOutputStream (p. 55)

wxArchiveNotifier (p. 55)

Non-seekable streams

This information applies only when reading archives from non-seekable streams. When the stream is seekable *GetNextEntry()* (p. 52) returns a fully populated *wxArchiveEntry* (p. 48). See '*Archives on non-seekable streams* (p. **Error! Bookmark not defined.**)' for more information.

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

GetSize() (p. 50) Guaranteed to be available after the entry has been read to *Eof()* (p. 827), or *CloseEntry()* (p. 52) has been called

IsReadOnly() (p. 51) Guaranteed to be available after the end of the archive has been reached, i.e. after *GetNextEntry()* returns NULL and *Eof()* is true

wxArchiveEntry::Clone

wxArchiveEntry* Clone() const

Returns a copy of this entry object.

wxArchiveEntry::Get/SetDateTime

wxDateTime GetDateTime() const

void SetDateTime(const wxDateTime& dt)

The entry's timestamp.

wxArchiveEntry::GetInternalFormat

wxPathFormat GetInternalFormat() const

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

wxArchiveEntry::GetInternalName

wxString GetInternalName() const

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

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

See also

Looking up an archive entry by name (p. **Error! Bookmark not defined.**)

wxArchiveEntry::Get/SetName

wxString GetName(wxPathFormat format = wxPATH_NATIVE) const

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

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

If this is a directory entry, (i.e. if *IsDir()* (p. 50) is true) then *GetName()* returns the name with a trailing path separator.

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

wxArchiveEntry::GetOffset

off_t GetOffset() const

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

wxArchiveEntry::Get/SetSize

off_t GetSize() const

void SetSize(off_t size)

The size of the entry's data in bytes.

wxArchiveEntry::IsDir/SetIsDir

bool IsDir() const

void SetIsDir(bool isDir = true)

True if this is a directory entry.

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

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

wxArchiveEntry::IsReadOnly/SetIsReadOnly

bool IsReadOnly() const

void SetIsReadOnly(bool isReadOnly = true)

True if the entry is a read-only file.

wxArchiveEntry::Set/UnsetNotifier

void SetNotifier(wxArchiveNotifier& notifier)

void UnsetNotifier()

Sets the *notifier* (p. 55) for this entry. Whenever the *wxArchiveInputStream* (p. 51) updates this entry, it will then invoke the associated notifier's *OnEntryUpdated* (p. 55) method.

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

See also

Archives on non-seekable streams (p. **Error! Bookmark not defined.**)
wxArchiveNotifier (p. 55)

wxArchiveInputStream

An abstract base class which serves as a common interface to archive input streams such as *wxZipInputStream* (p. **Error! Bookmark not defined.**).

GetNextEntry() (p. 52) returns an *wxArchiveEntry* (p. 48) object containing the meta-data for the next entry in the archive (and gives away ownership). Reading from the *wxArchiveInputStream* then returns the entry's data. *Eof()* becomes true after an attempt has been made to read past the end of the entry's data. When there are no more entries, *GetNextEntry()* returns NULL and sets *Eof()*.

Derived from

wxFilterInputStream (p. 551)

Include files

<wx/archive.h>

Data structures `typedef wxArchiveEntry entry_type`

See also

Archive formats such as zip (p. **Error! Bookmark not defined.**)

wxArchiveEntry (p. 48)

wxArchiveOutputStream (p. 55)

wxArchiveInputStream::CloseEntry

bool CloseEntry()

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

wxArchiveInputStream::GetNextEntry

wxArchiveEntry* GetNextEntry()

Closes the current entry if one is open, then reads the meta-data for the next entry and returns it in a *wxArchiveEntry* (p. 48) object, giving away ownership. Reading this *wxArchiveInputStream* then returns the entry's data.

wxArchiveInputStream::OpenEntry

bool OpenEntry(wxArchiveEntry& entry)

Closes the current entry if one is open, then opens the entry specified by the *wxArchiveEntry* (p. 48) object.

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

See also

Looking up an archive entry by name (p. **Error! Bookmark not defined.**)

wxArchivIterator

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

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

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

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

The first template parameter should be the type of archive input stream (e.g. *wxArchiveInputStream* (p. 51)) and the second can either be a pointer to an entry (e.g. *wxArchiveEntry* (p. 48)*), or a string/pointer pair (e.g. `std::pair<wxString, wxArchiveEntry*>`).

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

```
typedef wxArchivIterator<wxArchiveInputStream> wxArchiveIter;

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

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

```
typedef wxArchivIterator<wxZipInputStream> wxZipIter;

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

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

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

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

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

```
typedef std::vector<ZipEntryPtr> ZipCatalog;
```

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

Iterators that return `std::pair` objects can be used to populate a `std::multimap`, to allow entries to be looked up by name. The string is initialised using the `wxArchiveEntry` object's `GetInternalName()` (p. 49) function.

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

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

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

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

Derived from

No base class

Include files

<wx/archive.h>

See also

wxArchiveEntry (p. 48)

wxArchiveInputStream (p. 51)

wxArchiveOutputStream (p. 55)

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

wxArchivelterator::wxArchivelterator

wxArchivelterator()

Construct an 'end of sequence' instance.

wxArchivelterator(Arc& arc)

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

wxArchiveIterator::operator*

const T& operator*() const

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

wxArchiveIterator::operator++

wxArchiveIterator& operator++()

wxArchiveIterator& operator++(int)

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

wxArchiveNotifier

If you need to know when a *wxArchiveInputStream* (p. 51) updates a *wxArchiveEntry* (p. 48) object, you can create a notifier by deriving from this abstract base class, overriding *OnEntryUpdated()* (p. 55). An instance of your notifier class can then be assigned to the *wxArchiveEntry* object using *wxArchiveEntry::SetNotifier()* (p. 51). Your *OnEntryUpdated()* method will then be invoked whenever the input stream updates the entry.

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

Derived from

No base class

Include files

<wx/archive.h>

See also

Archives on non-seekable streams (p. **Error! Bookmark not defined.**)

wxArchiveEntry (p. 48)

wxArchiveInputStream (p. 51)

wxArchiveOutputStream (p. 55)

wxArchiveNotifier::OnEntryUpdated

void OnEntryUpdated(class wxArchiveEntry& entry)

This method must be overridden in your derived class.

wxArchiveOutputStream

An abstract base class which serves as a common interface to archive output streams such as *wxZipOutputStream* (p. **Error! Bookmark not defined.**).

PutNextEntry() (p. 57) is used to create a new entry in the output archive, then the entry's data is written to the *wxArchiveOutputStream*. Another call to *PutNextEntry()* closes the current entry and begins the next.

Derived from

wxFilterOutputStream (p. 552)

Include files

<wx/archive.h>

See also

Archive formats such as zip (p. **Error! Bookmark not defined.**)

wxArchiveEntry (p. 48)

wxArchiveInputStream (p. 51)

wxArchiveOutputStream::~wxArchiveOutputStream

~wxArchiveOutputStream()

Calls *Close()* (p. 56) if it has not already been called.

wxArchiveOutputStream::Close

bool Close()

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

wxArchiveOutputStream::CloseEntry

bool CloseEntry()

Close the current entry. It is called implicitly whenever another new entry is created with *CopyEntry()* (p. 57) or *PutNextEntry()* (p. 57), or when the archive is closed.

wxArchiveOutputStream::CopyArchiveMetaData

bool CopyArchiveMetaData(wxArchiveInputStream& stream)

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

modified copy of an archive.

Since the position of the meta-data can vary between the various archive formats, it is best to call `CopyArchiveMetaData()` before transferring the entries. The `wxArchiveOutputStream` (p. 55) will then hold on to the meta-data and write it at the correct point in the output file.

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

wxArchiveOutputStream::CopyEntry

bool CopyEntry(wxArchiveEntry* entry, wxArchiveInputStream& stream)

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

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

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

wxArchiveOutputStream::PutNextDirEntry

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

Create a new directory entry (see `wxArchiveEntry::IsDir()` (p. 50)) with the given name and timestamp.

`PutNextEntry()` (p. 57) can also be used to create directory entries, by supplying a name with a trailing path separator.

wxArchiveOutputStream::PutNextEntry

bool PutNextEntry(wxArchiveEntry* entry)

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

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

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

wxArray

This section describes the so called *dynamic arrays*. This is a C array-like data structure i.e. the member access time is constant (and not linear according to the number of container elements as for linked lists). However, these arrays are dynamic in the sense that they will automatically allocate more memory if there is not enough of it for adding a new element. They also perform range checking on the index values but in debug mode only, so please be sure to compile your application in debug mode to use it (see *debugging overview* (p. **Error! Bookmark not defined.**) 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. 851)). 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. 61) section you may find some useful hints about optimizing *wxArray* memory usage. As for executable size, all *wxArray* functions are inline, so they do not take *any space at all*.

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

wxArray is suitable for storing integer types and pointers which it does not treat as objects in any way, i.e. the element pointed to by the pointer is not deleted when the element is removed from the array. It should be noted that all of *wxArray*'s functions are inline, so it costs strictly nothing to define as many array types as you want (either in terms of the executable size or the speed) as long as at least one of them is defined and this is always the case because *wxArrays* are used by *wxWidgets* internally. This class has one serious limitation: it can only be used for storing integral types (*bool*, *char*, *short*, *int*, *long* and their unsigned variants) or pointers (of any kind). An attempt to use with objects of *sizeof()* greater than *sizeof(long)* will provoke a runtime assertion failure, however declaring a *wxArray* of floats will not (on the machines where *sizeof(float)* \leq *sizeof(long)*), yet it will **not** work, please use *wxObjArray* for storing floats and doubles (NB: a more efficient *wxArrayDouble* class is scheduled for the next release of *wxWidgets*).

wxSortedArray is a *wxArray* variant which should be used when searching in the array is a frequently used operation. It requires you to define an additional function for comparing two elements of the array element type and always stores its items in the sorted order (according to this function). Thus, it is *Index()* (p. 67) function execution time is $O(\log(N))$ instead of $O(N)$ for the usual arrays but the *Add()* (p. 66) method is slower: it is $O(\log(N))$ instead of constant time (neglecting time spent in memory allocation routine). However, in a usual situation elements are added to an array much less often than searched inside

it, so `wxSortedArray` may lead to huge performance improvements compared to `wxArray`. Finally, it should be noticed that, as `wxArray`, `wxSortedArray` can be only used for storing integral types or pointers.

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

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

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

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

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

...

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

// that's all!
```

It is not as elegant as writing

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

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

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

write

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

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

See also:

Container classes overview (p. **Error! Bookmark not defined.**), *wxList* (p. 851)

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. 62)
`WX_DEFINE_EXPORTED_ARRAY` (p. 62)
`WX_DEFINE_USER_EXPORTED_ARRAY` (p. 62)
`WX_DEFINE_SORTED_ARRAY` (p. 63)
`WX_DEFINE_SORTED_EXPORTED_ARRAY` (p. 63)
`WX_DEFINE_SORTED_USER_EXPORTED_ARRAY` (p. 63)
`WX_DECLARE_EXPORTED_OBJARRAY` (p. 63)
`WX_DECLARE_USER_EXPORTED_OBJARRAY` (p. 63)
`WX_DEFINE_OBJARRAY` (p. 64)
`WX_DEFINE_EXPORTED_OBJARRAY` (p. 64)
`WX_DEFINE_USER_EXPORTED_OBJARRAY` (p. 64)

To slightly complicate the matters even further, the operator `->` defined by default for the array iterators by these macros only makes sense if the array element type is not a pointer itself and, although it still works, this provokes warnings from some compilers and to avoid them you should use the `_PTR` versions of the macros above. For example, to define an array of pointers to `double` you should use:

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

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

The following simple types are supported:

`int`
`long`
`size_t`

double

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

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

WX_DEFINE_ARRAY_INT (p. 62)
WX_DEFINE_EXPORTED_ARRAY_INT (p. 62)
WX_DEFINE_USER_EXPORTED_ARRAY_INT (p. 62)
WX_DEFINE_SORTED_ARRAY_INT (p. 63)
WX_DEFINE_SORTED_EXPORTED_ARRAY_INT (p. 63)
WX_DEFINE_SORTED_USER_EXPORTED_ARRAY_INT (p. 63)

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. 65)
wxArray copy constructors and assignment operators (p. 65)
~wxArray (p. 65)

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. 69) function is provided to deallocate the extra memory. The *Alloc()* (p. 66) 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. 66)
Shrink (p. 69)

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. 68) method.

Count (p. 66)
GetCount (p. 67)
IsEmpty (p. 68)
Item (p. 68)
Last (p. 68)

Adding items

Add (p. 66)
Insert (p. 67)
SetCount (p. 69)
WX_APPEND_ARRAY (p. 64)

Removing items

WX_CLEAR_ARRAY (p. 65)
Empty (p. 67)
Clear (p. 66)
RemoveAt (p. 69)
Remove (p. 68)

Searching and sorting

Index (p. 67)
Sort (p. 69)

WX_DEFINE_ARRAY

WX_DEFINE_ARRAY(*T*, *name*)

WX_DEFINE_EXPORTED_ARRAY(*T*, *name*)

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

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

Example:

```
WX_DEFINE_ARRAY_INT(int, MyArrayInt);

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

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

WX_DEFINE_SORTED_ARRAY**WX_DEFINE_SORTED_ARRAY**(*T*, *name*)**WX_DEFINE_SORTED_EXPORTED_ARRAY**(*T*, *name*)**WX_DEFINE_SORTED_USER_EXPORTED_ARRAY**(*T*, *name*)

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

Example:

```
WX_DEFINE_SORTED_ARRAY_INT(int, MySortedArrayInt);

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

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

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

wxSortedArrayInt sorted(CompareInts);

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

wxArrayOfMyClass another(CompareMyClassObjects);
```

WX_DECLARE_OBJARRAY**WX_DECLARE_OBJARRAY**(*T*, *name*)**WX_DECLARE_EXPORTED_OBJARRAY**(*T*, *name*)**WX_DECLARE_USER_EXPORTED_OBJARRAY**(*T*, *name*)

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

Example:

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

You must use `WX_DEFINE_OBJARRAY()` (p. 64) macro to define the array class - otherwise you would get link errors.

WX_DEFINE_OBJARRAY

WX_DEFINE_OBJARRAY(*name*)

WX_DEFINE_EXPORTED_OBJARRAY(*name*)

WX_DEFINE_USER_EXPORTED_OBJARRAY(*name*)

This macro defines the methods of the array class *name* not defined by the `WX_DECLARE_OBJARRAY()` (p. 63) macro. You must include the file `<wx/arrimpl.cpp>` before using this macro and you must have the full declaration of the class of array elements in scope! If you forget to do the first, the error will be caught by the compiler, but, unfortunately, many compilers will not give any warnings if you forget to do the second - but the objects of the class will not be copied correctly and their real destructor will not be called. The latter two forms are merely aliases of the first to satisfy some people's sense of symmetry when using the exported declarations.

Example of usage:

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

    ...

    virtual ~MyClass();
};

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

WX_APPEND_ARRAY

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

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

WX_CLEAR_ARRAY

void WX_CLEAR_ARRAY(wxArray& *array*)

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

Default constructors

wxArray()

wxObjArray()

Default constructor initializes an empty array object.

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

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

wxArray copy constructor and assignment operator

wxArray(const wxArray& array)

wxSortedArray(const wxSortedArray& array)

wxObjArray(const wxObjArray& array)

wxArray& operator=(const wxArray& array)

wxSortedArray& operator=(const wxSortedArray& array)

wxObjArray& operator=(const wxObjArray& array)

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

wxArray::~~wxArray

~wxArray()

~wxSortedArray()

~wxObjArray()

The wxObjArray destructor deletes all the items owned by the array. This is not done by wxArray and wxSortedArray versions - you may use *WX_CLEAR_ARRAY* (p. 65) macro for this.

wxArray::Add

void Add(T item, size_t copies = 1)

void Add(T *item)

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

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

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

You may also use `WX_APPEND_ARRAY` (p. 64) macro to append all elements of one array to another one but it is more efficient to use *copies* parameter and modify the elements in place later if you plan to append a lot of items.

wxArray::Alloc

void Alloc(size_t count)

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

wxArray::Clear

void Clear()

This function does the same as `Empty()` (p. 67) and additionally frees the memory allocated to the array.

wxArray::Count

size_t Count() const

Same as `GetCount()` (p. 67). 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. 68) 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. 66) for this.

wxArray::GetCount

size_t GetCount() const

Return the number of items in the array.

wxArray::Index

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

int Index(T& item)

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

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

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

NB: even for `wxObjArray` classes, the operator `==()` of the elements in the array is **not** used by this function. It searches exactly the given element in the array and so will only succeed if this element had been previously added to the array, but fail even if another, identical, element is in the array.

wxArray::Insert

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

void Insert(T *item, size_t n)

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

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

Please see `Add()` (p. 66) 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. 67) 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. 65) macro which deletes all elements of a `wxArray` (supposed to contain pointers).

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

Removes *count* elements starting at *index* from the array. When an element is removed from `wxObjArray` it is deleted by the array - use `Detach()` (p. 67) 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. 65) macro which deletes all elements of a `wxArray`

(supposed to contain pointers).

wxArray::SetCount

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

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

See also

GetCount (p. 67)

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. **Error! Bookmark not defined.**) objects. It has the same features as all *wxArray* (p. 57) 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. 57), 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. 74), *Last* (p. 74) or *operator[]* (p. 72) 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. 73) 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. 57) class for the *wxString* member data: it is not implemented like this, but it does have all of the *wxArray* functions.

Include files

<wx/arrstr.h>

See also

wxArray (p. 57), *wxString* (p. **Error! Bookmark not defined.**), *wxString* overview (p. **Error! Bookmark not defined.**)

wxArrayString::wxArrayString

wxArrayString()

Default constructor.

wxArrayString(const wxArrayString& array)

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

wxArrayString(size_t sz, const wxChar** arr)

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

wxArrayString(size_t sz, const wxString* arr)

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

wxArrayString::~~wxArrayString

~wxArrayString()

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

wxArrayString::operator=

wxArrayString & operator =(const wxArrayString& array)

Assignment operator.

wxArrayString::operator==

bool operator ==(const wxArrayString& array) const

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

wxArrayString::operator!=

bool operator !=(const wxArrayString& array) const

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

wxArrayString::operator[]

wxString& operator[](size_t nIndex)

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

This is the operator version of *Item* (p. 74) method.

wxArrayString::Add

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

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

Warning: For sorted arrays, the index of the inserted item will not be, in general, equal to *GetCount()* (p. 73) - 1 because the item is inserted at the correct position to keep the array sorted and not appended.

See also: *Insert* (p. 73)

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. 61)

wxArrayString::Clear

void Clear()

Clears the array contents and frees memory.

See also: *Empty* (p. 73)

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. 73) instead.

wxArrayString::Empty

void Empty()

Empties the array: after a call to this function *GetCount* (p. 73) 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. 72) to empty the array and free memory.

wxArrayString::GetCount

size_t GetCount() const

Returns the number of items in the array.

wxArrayString::Index

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

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

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

case.

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

wxArrayString::Insert

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

Insert the given number of *copies* of the new element in the array before the position *nIndex*. Thus, for example, to insert the string in the beginning of the array you would write

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

If *nIndex* is equal to *GetCount()* this function behaves as *Add* (p. 72).

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. 73) would then not work!

wxArrayString::IsEmpty

bool IsEmpty()

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

wxArrayString::Item

wxString& Item(size_t nIndex) const

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

See also *operator[]* (p. 72) for the operator version.

wxArrayString::Last

wxString& Last()

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

wxArrayString::Remove

void Remove(const char * sz)

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

See also: *Index* (p. 73)

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

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

wxArrayString::Shrink**void Shrink()**

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

See also: *Alloc* (p. 72), *Dynamic array memory management* (p. 61)

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

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

Warning: this function should not be used with sorted array because it could break the order of items and, for example, subsequent calls to *Index()* (p. 73) 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. 73) would then not work!

wxArtProvider

`wxArtProvider` class is used to customize the look of `wxWidgets` application. When `wxWidgets` needs to display an icon or a bitmap (e.g. in the standard file dialog), it does not use a hard-coded resource but asks `wxArtProvider` for it instead. This way users can plug in their own `wxArtProvider` class and easily replace standard art with their own version. All that is needed is to derive a class from `wxArtProvider`, override its `CreateBitmap` (p. 78) method and register the provider with `wxArtProvider::PushProvider` (p. 80):

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

    { ... }
};
...
wxArtProvider::PushProvider(new MyProvider);
```

There's another way of taking advantage of this class: you can use it in your code and use platform native icons as provided by `wxArtProvider::GetBitmap` (p. 79) or `wxArtProvider::GetIcon` (p. 79) (NB: this is not yet really possible as of `wxWidgets` 2.3.3, the set of `wxArtProvider` bitmaps is too small).

Identifying art resources

Every bitmap is known to `wxArtProvider` under an unique ID that is used by when requesting a resource from it. The ID is represented by `wxArtID` type and can have one of these predefined values (you can see bitmaps represented by these constants in the `artprov` (p. **Error! Bookmark not defined.**) sample):

- `wxART_ADD_BOOKMARK`
- `wxART_DEL_BOOKMARK`
- `wxART_HELP_SIDE_PANEL`
- `wxART_HELP_SETTINGS`
- `wxART_HELP_BOOK`
- `wxART_HELP_FOLDER`
- `wxART_HELP_PAGE`
- `wxART_GO_BACK`
- `wxART_GO_FORWARD`
- `wxART_GO_UP`
- `wxART_GO_DOWN`

- wxART_GO_TO_PARENT
- wxART_GO_HOME
- wxART_FILE_OPEN
- wxART_PRINT
- wxART_HELP
- wxART_TIP
- wxART_REPORT_VIEW
- wxART_LIST_VIEW
- wxART_NEW_DIR
- wxART_FOLDER
- wxART_GO_DIR_UP
- wxART_EXECUTABLE_FILE
- wxART_NORMAL_FILE
- wxART_TICK_MARK
- wxART_CROSS_MARK
- wxART_ERROR
- wxART_QUESTION
- wxART_WARNING
- wxART_INFORMATION
- wxART_MISSING_IMAGE

Additionally, any string recognized by custom art providers registered using *PushProvider* (p. 80) may be used.

GTK+ Note

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

Clients

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

- `wxART_TOOLBAR`
- `wxART_MENU`
- `wxART_BUTTON`
- `wxART_FRAME_ICON`
- `wxART_CMN_DIALOG`
- `wxART_HELP_BROWSER`
- `wxART_MESSAGE_BOX`
- `wxART_OTHER` (used for all requests that don't fit into any of the categories above) Client ID serves as a hint to `wxArtProvider` that is supposed to help it to choose the best looking bitmap. For example it is often desirable to use slightly different icons in menus and toolbars even though they represent the same action (e.g. `wx_ART_FILE_OPEN`). Remember that this is really only a hint for `wxArtProvider` -- it is common that `wxArtProvider::GetBitmap` (p. 79) returns identical bitmap for different *client* values!

See also

See the *artprov* (p. **Error! Bookmark not defined.**) sample for an example of `wxArtProvider` usage.

Derived from

`wxObject` (p. **Error! Bookmark not defined.**)

Include files

<wx/artprov.h>

`wxArtProvider::CreateBitmap`

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

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

Parameters

id

wxArtID unique identifier of the bitmap.

client

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

size

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

Note

This is **not** part of wxArtProvider's public API, use `wxArtProvider::GetBitmap` (p. 79) or `wxArtProvider::GetIcon` (p. 79) to query wxArtProvider for a resource.

wxArtProvider::GetBitmap

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

Query registered providers for bitmap with given ID.

Parameters

id

wxArtID unique identifier of the bitmap.

client

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

size

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

Return value

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

wxArtProvider::GetIcon

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

Same as `wxArtProvider::GetBitmap` (p. 79), but return a `wxIcon` object (or `wxNullIcon` on failure).

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


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

wxArtProvider::PopProvider

static bool PopProvider()

Remove latest added provider and delete it.

wxArtProvider::PushProvider

static void PushProvider(wxArtProvider* provider)

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

wxArtProvider::RemoveProvider

static bool RemoveProvider(wxArtProvider* provider)

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

wxAutomationObject

The **wxAutomationObject** class represents an OLE automation object containing a single data member, an `IDispatch` pointer. It contains a number of functions that make it easy to perform automation operations, and set and get properties. The class makes heavy use of the *wxVariant* (p. **Error! Bookmark not defined.**) class.

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

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

Note that this class obviously works under Windows only.

Derived from

wxObject (p. **Error! Bookmark not defined.**)

Include files

<wx/msw/ole/automtn.h>

See also

wxVariant (p. **Error! Bookmark not defined.**)

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. 82) 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. 82)

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

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

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

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

wxAutomationObject::Invoke

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

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

Parameters

member

The member function or property name.

action

Bitlist: may contain DISPATCH_PROPERTYPUT, DISPATCH_PROPERTYPUTREF, DISPATCH_METHOD.

retValue

Return value (ignored if there is no return value)

.

noArgs

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

args

If non-null, contains an array of variants.

ptrArgs

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

Return value

true if the operation was successful, false otherwise.

Remarks

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

wxAutomationObject::PutProperty

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

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

Puts a property value into this object. The first form takes a property name, number of arguments, and an array of variants. The second form takes a property name and zero

to six constant references to variants. Since the variant class has constructors for the basic data types, and C++ provides temporary objects automatically, both of the following lines are syntactically valid:

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

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

wxAutomationObject::SetDispatchPtr

void SetDispatchPtr(WXIDISPATCH* dispatchPtr)

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

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

wxBitmap

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

Derived from

wxGDIObject (p. 609)

wxObject (p. **Error! Bookmark not defined.**)

Include files

<wx/bitmap.h>

Predefined objects

Objects:

wxNullBitmap

See also

wxBitmap overview (p. **Error! Bookmark not defined.**), *supported bitmap file formats* (p. **Error! Bookmark not defined.**), *wxDC::Blit* (p. 373), *wxIcon* (p. 778), *wxCursor* (p. 230), *wxBitmap* (p. 84), *wxMemoryDC* (p. **Error! Bookmark not defined.**)

wxBitmap::wxBitmap

wxBitmap()

Default constructor.

wxBitmap(const wxBitmap& *bitmap*)

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

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

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

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

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

Creates a bitmap from an array of bits.

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

For other bit depths, the behaviour is platform dependent: under Windows, the data is passed without any changes to the underlying `CreateBitmap()` API. Under other platforms, only monochrome bitmaps may be created using this constructor and `wxImage` (p. 790) should be used for creating colour bitmaps from static data.

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

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

wxBitmap(const char *bits*)**

Creates a bitmap from XPM data.

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

Loads a bitmap from a file or resource.

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

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

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

dithering algorithms).

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

Parameters

bits

Specifies an array of pixel values.

width

Specifies the width of the bitmap.

height

Specifies the height of the bitmap.

depth

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

name

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

type

May be one of the following:

`wxBITMAP_TYPE_BMP` Load a Windows bitmap file.

`wxBITMAP_TYPE_BMP_RESOURCE` Load a Windows bitmap resource from the executable. Windows only.

`wxBITMAP_TYPE_PICT_RESOURCE` Load a PICT image resource from the executable. Mac OS only.

`wxBITMAP_TYPE_GIF` Load a GIF bitmap file.

`wxBITMAP_TYPE_XBM` Load an X bitmap file.

`wxBITMAP_TYPE_XPM` Load an XPM bitmap file.

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

In addition, `wxBitmap` can read all formats that `wxImage` (p. 790) can, which

currently include `wxBITMAP_TYPE_JPEG`, `wxBITMAP_TYPE_TIF`, `wxBITMAP_TYPE_PNG`, `wxBITMAP_TYPE_GIF`, `wxBITMAP_TYPE_PCX`, and `wxBITMAP_TYPE_PNM`. Of course, you must have wxImage handlers loaded.

img

Platform-independent wxImage object.

Remarks

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

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

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

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

The sixth form constructs a new bitmap.

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

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

```
#include "mybitmap.xpm"

...

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

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

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

See also

wxBitmap::LoadFile (p. 92)

wxPython note: Constructors supported by wxPython are:

wxBitmap(name, flag)	Loads a bitmap from a file
wxEmptyBitmap(width, height, depth = -1)	Creates an empty bitmap with the given specifications
wxBitmapFromXPMData(listOfStrings)	Create a bitmap from a Python list of strings whose contents are XPM

data.

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

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

wxPerl note: Constructors supported by wxPerl are:

- ::Bitmap->new(width, height, depth = -1)
- ::Bitmap->new(name, type)
- ::Bitmap->new(icon)
- ::Bitmap->newFromBits(bits, width, height, depth = 1)
- ::Bitmap->newFromXPM(data)

wxBitmap::~~wxBitmap

~wxBitmap()

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

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

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

wxBitmap::AddHandler

static void AddHandler(wxBitmapHandler* handler)

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

handler

A new bitmap format handler object. There is usually only one instance of a given handler class in an application session.

See also

wxBitmapHandler (p. 104)

wxBitmap::CleanUpHandlers

static void CleanUpHandlers()

Deletes all bitmap handlers.

This function is called by `wxWidgets` on exit.

`wxBitmap::ConvertToImage`

`wxImage ConvertToImage()`

Creates an image from a platform-dependent bitmap. This preserves mask information so that bitmaps and images can be converted back and forth without loss in that respect.

`wxBitmap::CopyFromIcon`

`bool CopyFromIcon(const wxIcon& icon)`

Creates the bitmap from an icon.

`wxBitmap::Create`

`virtual bool Create(int width, int height, int depth = -1)`

Creates a fresh bitmap. If the final argument is omitted, the display depth of the screen is used.

`virtual bool Create(void* data, int type, int width, int height, int depth = -1)`

Creates a bitmap from the given data, which can be of arbitrary type.

Parameters

width

The width of the bitmap in pixels.

height

The height of the bitmap in pixels.

depth

The depth of the bitmap in pixels. If this is -1, the screen depth is used.

data

Data whose type depends on the value of *type*.

type

A bitmap type identifier - see `wxBitmap::wxBitmap` (p. 84) 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. 84)

wxBitmap::FindHandler

static wxBitmapHandler* FindHandler(const wxString& name)

Finds the handler with the given name.

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

Finds the handler associated with the given extension and type.

static wxBitmapHandler* FindHandler(wxBitmapType bitmapType)

Finds the handler associated with the given bitmap type.

name

The handler name.

extension

The file extension, such as "bmp".

bitmapType

The bitmap type, such as wxBITMAP_TYPE_BMP.

Return value

A pointer to the handler if found, NULL otherwise.

See also

wxBitmapHandler (p. 104)

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. 104)

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. **Error! Bookmark not defined.**)

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. 95), *wxMask* (p. 920)

wxBitmap::GetWidth

int GetWidth() const

Gets the width of the bitmap in pixels.

See also

wxBitmap::GetHeight (p. 91)

wxBitmap::GetSubBitmap

wxBitmap GetSubBitmap(const wxRect&rect) const

Returns a sub bitmap of the current one as long as the rect belongs entirely to the bitmap. This function preserves bit depth and mask information.

wxBitmap::InitStandardHandlers

static void InitStandardHandlers()

Adds the standard bitmap format handlers, which, depending on wxWidgets configuration, can be handlers for Windows bitmap, Windows bitmap resource, and XPM.

This function is called by wxWidgets on startup.

See also

wxBitmapHandler (p. 104)

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. 104)

wxBitmap::LoadFile**bool LoadFile(const wxString& name, wxBitmapType type)**

Loads a bitmap from a file or resource.

Parameters

name

Either a filename or a Windows resource name. The meaning of *name* is determined by the *type* parameter.

type

One of the following values:

wxBITMAP_TYPE_BMP Load a Windows bitmap file.

wxBITMAP_TYPE_BMP_RESOURCE Load a Windows bitmap resource from the executable.

wxBITMAP_TYPE_PICT_RESOURCE Load a PICT image resource from the executable. Mac OS only.

wxBITMAP_TYPE_GIF Load a GIF bitmap file.

wxBITMAP_TYPE_XBM Load an X bitmap file.

wxBITMAP_TYPE_XPM Load an XPM bitmap file.

The validity of these flags depends on the platform and wxWidgets configuration.

In addition, wxBitmap can read all formats that *wxImage* (p. 790) 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. 91) member.

See also

wxBitmap::SaveFile (p. 93)

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. 104)

wxBitmap::SaveFile

bool SaveFile(const wxString& name, wxBitmapType type, wxPalette* palette = NULL)

Saves a bitmap in the named file.

Parameters

name

A filename. The meaning of *name* is determined by the *type* parameter.

type

One of the following values:

wxBITMAP_TYPE_BMP Save a Windows bitmap file.

wxBITMAP_TYPE_GIF Save a GIF bitmap file.

wxBITMAP_TYPE_XBM Save an X bitmap file.

wxBITMAP_TYPE_XPM Save an XPM bitmap file.

The validity of these flags depends on the platform and wxWidgets configuration.

In addition, wxBitmap can save all formats that *wxImage* (p. 790) can (wxBITMAP_TYPE_JPEG, wxBITMAP_TYPE_PNG). (Of course you must have wxImage handlers loaded.)

palette

An optional palette used for saving the bitmap.

Return value

true if the operation succeeded, false otherwise.

Remarks

Depending on how wxWidgets has been configured, not all formats may be available.

See also

wxBitmap::LoadFile (p. 92)

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. 91), *wxMask* (p. 920)

wxBitmap::SetPalette

void SetPalette(const wxPalette& *palette*)

Sets the associated palette. (Not implemented under GTK+).

Parameters

palette

The palette to set.

See also

wxPalette (p. **Error! Bookmark not defined.**)

wxBitmap::SetWidth

void SetWidth(int *width*)

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

Parameters

width

Bitmap width in pixels.

wxBitmap::operator =

wxBitmap& operator =(const wxBitmap& *bitmap*)

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

Parameters

bitmap

Bitmap to assign.

Return value

Returns 'this' object.

wxBitmap::operator ==**bool operator ==(const wxBitmap& *bitmap*)**

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

Parameters

bitmap

Bitmap to compare with 'this'

Return value

Returns true if the bitmaps were effectively equal, false otherwise.

wxBitmap::operator !=**bool operator !=(const wxBitmap& *bitmap*)**

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

Parameters

bitmap

Bitmap to compare with 'this'

Return value

Returns true if the bitmaps were unequal, false otherwise.

wxBitmapButton

A bitmap button is a control that contains a bitmap. It may be placed on a *dialog box* (p. 412) or *panel* (p. **Error! Bookmark not defined.**), or indeed almost any other window.

Derived from*wxButton* (p. 122)*wxControl* (p. 218)*wxWindow* (p. **Error! Bookmark not defined.**)*wxEvtHandler* (p. 490)*wxObject* (p. **Error! Bookmark not defined.**)**Include files**

<wx/bmpbuttn.h>

Remarks

A bitmap button can be supplied with a single bitmap, and wxWidgets will draw all button states using this bitmap. If the application needs more control, additional bitmaps for the selected state, unpressed focused state, and greyed-out state may be supplied.

Button states

This class supports bitmaps for several different states:

normal	This is the bitmap shown in the default state, it must be always valid while all the other bitmaps are optional and don't have to be set.
disabled	Bitmap shown when the button is disabled.
selected	Bitmap shown when the button is pushed (e.g. while the user keeps the mouse button pressed on it)
focus	Bitmap shown when the button has keyboard focus but is not pressed.
hover	Bitmap shown when the mouse is over the button (but it is not pressed). Notice that if hover bitmap is not specified but the current platform UI uses hover images for the buttons (such as Windows XP or GTK+), then the focus bitmap is used for hover state as well. This makes it possible to set focus bitmap only to get reasonably good behaviour on all platforms.

Window styles

wxBU_AUTODRAW	If this is specified, the button will be drawn automatically using the label bitmap only, providing a 3D-look border. If this style is not specified, the button will be drawn without borders and using all provided bitmaps. WIN32 only.
wxBU_LEFT	Left-justifies the bitmap label. WIN32 only.
wxBU_TOP	Aligns the bitmap label to the top of the button. WIN32 only.

wxBU_RIGHT	Right-justifies the bitmap label. WIN32 only.
wxBU_BOTTOM	Aligns the bitmap label to the bottom of the button. WIN32 only.

Note that **wxBU_EXACTFIT** supported by *wxButton* (p. 122) is *not* used by this class as bitmap buttons don't have any minimal standard size by default.

See also *window styles overview* (p. **Error! Bookmark not defined.**).

Event handling

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

See also

wxButton (p. 122)

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. 96).

validator

Window validator.

name

Window name.

Remarks

The *bitmap* parameter is normally the only bitmap you need to provide, and *wxWidgets* will draw the button correctly in its different states. If you want more control, call any of the functions *wxBitmapButton::SetBitmapSelected* (p. 102), *wxBitmapButton::SetBitmapFocus* (p. 101), *wxBitmapButton::SetBitmapDisabled* (p. 101).

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

See also

wxBitmapButton::Create (p. 99), *wxValidator* (p. **Error! Bookmark not defined.**)

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. 98).

wxBitmapButton::GetBitmapDisabled

const wxBitmap& GetBitmapDisabled() const**wxBitmap& GetBitmapDisabled()**

Returns the bitmap for the disabled state, may be invalid.

Return value

A reference to the disabled state bitmap.

See also

wxBitmapButton::SetBitmapDisabled (p. 101)

wxBitmapButton::GetBitmapFocus

const wxBitmap& GetBitmapFocus() const **wxBitmap& GetBitmapFocus()**

Returns the bitmap for the focused state, may be invalid.

Return value

A reference to the focused state bitmap.

See also

wxBitmapButton::SetBitmapFocus (p. 101)

wxBitmapButton::GetBitmapHover

const wxBitmap& GetBitmapHover() const **wxBitmap& GetBitmapHover()**

Returns the bitmap used when the mouse is over the button, may be invalid.

See also

wxBitmapButton::SetBitmapHover (p. 102)

wxBitmapButton::GetBitmapLabel

const wxBitmap& GetBitmapLabel() const **wxBitmap& GetBitmapLabel()**

Returns the label bitmap (the one passed to the constructor), always valid.

Return value

A reference to the button's label bitmap.

See also

wxBitmapButton::SetBitmapLabel (p. 102)

wxBitmapButton::GetBitmapSelected

wxBitmap& GetBitmapSelected() const **wxBitmap& GetBitmapSelected()**

Returns the bitmap for the pushed button state, may be invalid.

Return value

A reference to the selected state bitmap.

See also

wxBitmapButton::SetBitmapSelected (p. 102)

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. 99), *wxBitmapButton::SetBitmapLabel* (p. 102), *wxBitmapButton::SetBitmapSelected* (p. 102), *wxBitmapButton::SetBitmapFocus* (p. 101)

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. 100), *wxBitmapButton::SetBitmapLabel* (p. 102), *wxBitmapButton::SetBitmapSelected* (p. 102), *wxBitmapButton::SetBitmapDisabled* (p. 101)

wxBitmapButton::SetBitmapHover

void SetBitmapHover(const wxBitmap& *bitmap*)

Sets the bitmap to be shown when the mouse is over the button.

This function is new since wxWidgets version 2.7.0 and the hover bitmap is currently only supported in wxMSW.

See also

wxBitmapButton::GetBitmapHover (p. 100)

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. 100)

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. 101), *wxBitmapButton::SetBitmapLabel* (p. 102), *wxBitmapButton::SetBitmapFocus* (p. 101), *wxBitmapButton::SetBitmapDisabled* (p. 101)

wxBitmapDataObject

wxBitmapDataObject is a specialization of *wxDataObject* for bitmap data. It can be used without change to paste data into the *wxClipboard* (p. 154) or a *wxDropSource* (p. 472). 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. 103) may be overridden to increase efficiency.

Derived from

wxDataObjectSimple (p. 247)
wxDataObject (p. 242)

Include files

<wx/dataobj.h>

See also

Clipboard and drag and drop overview (p. **Error! Bookmark not defined.**),
wxDataObject (p. 242), *wxDataObjectSimple* (p. 247), *wxFileDataObject* (p. 514),
wxTextDataObject (p. **Error! Bookmark not defined.**), *wxDataObject* (p. 242)

wxBitmapDataObject(const wxBitmap& bitmap = wxNullBitmap)

Constructor, optionally passing a bitmap (otherwise use *SetBitmap* (p. 103) later).

wxBitmapDataObject::GetBitmap

virtual wxBitmap GetBitmap() const

Returns the bitmap associated with the data object. You may wish to override this method when offering data on-demand, but this is not required by wxWidgets' internals. Use this method to get data in bitmap form from the *wxClipboard* (p. 154).

wxBitmapDataObject::SetBitmap

virtual void SetBitmap(const wxBitmap& bitmap)

Sets the bitmap associated with the data object. This method is called when the data object receives data. Usually there will be no reason to override this function.

wxBitmapHandler

Overview (p. **Error! Bookmark not defined.**)

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. 88) in your application initialisation.

Derived from

wxObject (p. **Error! Bookmark not defined.**)

Include files

<wx/bitmap.h>

See also

wxBitmap (p. 84), *wxIcon* (p. 778), *wxCursor* (p. 230)

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. 84) 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. 84) for values this can take.

Return value

true if the operation succeeded, false otherwise.

See also

wxBitmap::LoadFile (p. 92)

wxBitmap::SaveFile (p. 93)

wxBitmapHandler::SaveFile (p. 106)

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. 84) 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. 92)

wxBitmap::SaveFile (p. 93)

wxBitmapHandler::LoadFile (p. 105)

wxBitmapHandler::SetName

void SetName(const wxString& *name*)

Sets the handler name.

Parameters

name

Handler name.

wxBitmapHandler::SetExtension

void SetExtension(const wxString& *extension*)

Sets the handler extension.

Parameters

extension

Handler extension.

wxBitmapHandler::SetType

void SetType(long type)

Sets the handler type.

Parameters

name

Handler type.

wxBoxSizer

The basic idea behind a box sizer is that windows will most often be laid out in rather simple basic geometry, typically in a row or a column or several hierarchies of either.

For more information, please see *Programming with wxBoxSizer* (p. **Error! Bookmark not defined.**).

Derived from

wxSizer (p. **Error! Bookmark not defined.**)

wxObject (p. **Error! Bookmark not defined.**)

Include files

<wx/sizer.h>

See also

wxSizer (p. **Error! Bookmark not defined.**), *Sizer overview* (p. **Error! Bookmark not defined.**)

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. **Error! Bookmark not defined.**) 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. 609)

wxObject (p. **Error! Bookmark not defined.**)

Include files

<wx/brush.h>

Predefined objects

Objects:

wxNullBrush

Pointers:

wxBLUE_BRUSH
wxGREEN_BRUSH
wxWHITE_BRUSH
wxBLACK_BRUSH
wxGREY_BRUSH
wxMEDIUM_GREY_BRUSH
wxLIGHT_GREY_BRUSH
wxTRANSPARENT_BRUSH
wxCYAN_BRUSH
wxRED_BRUSH

Remarks

On a monochrome display, wxWidgets shows all brushes as white unless the colour is really black.

Do not initialize objects on the stack before the program commences, since other required structures may not have been set up yet. Instead, define global pointers to objects and create them in *wxApp::OnInit* (p. 42) 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. 114), *wxDC* (p. 372), *wxDC::SetBrush* (p. 388)

wxBrush::wxBrush

wxBrush()

Default constructor. The brush will be uninitialised, and *wxBrush::Ok* (p. 112) will return false.

wxBrush(const wxColour& colour, int style = wxSOLID)

Constructs a brush from a colour object and style.

wxBrush(const wxString& colourName, int style)

Constructs a brush from a colour name and style.

wxBrush(const wxBitmap& stippleBitmap)

Constructs a stippled brush using a bitmap.

wxBrush(const wxBrush& brush)

Copy constructor. This uses reference counting so is a cheap operation.

Parameters

colour

Colour object.

colourName

Colour name. The name will be looked up in the colour database.

style

One of:

wxTRANSPARENT	Transparent (no fill).
wxSOLID	Solid.
wxSTIPPLE	Uses a bitmap as a stipple.
wxBDIAGONAL_HATCH	Backward diagonal hatch.
wxCROSSDIAG_HATCH	Cross-diagonal hatch.
wxFDIAGONAL_HATCH	Forward diagonal hatch.
wxCROSS_HATCH	Cross hatch.
wxHORIZONTAL_HATCH	Horizontal hatch.
wxVERTICAL_HATCH	Vertical hatch.

brush

Pointer or reference to a brush to copy.

stippleBitmap

A bitmap to use for stippling.

Remarks

If a stipple brush is created, the brush style will be set to `wxSTIPPLE`.

See also

wxBrushList (p. 114), *wxColour* (p. 168), *wxColourDatabase* (p. 173)

wxBrush::~wxBrush

~wxBrush()

Destructor.

Remarks

The destructor may not delete the underlying brush object of the native windowing system, since `wxBrush` uses a reference counting system for efficiency.

Although all remaining brushes are deleted when the application exits, the application should try to clean up all brushes itself. This is because `wxWidgets` cannot know if a pointer to the brush object is stored in an application data structure, and there is a risk of double deletion.

wxBrush::GetColour

wxColour& GetColour() const

Returns a reference to the brush colour.

See also

wxBrush::SetColour (p. 113)

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. 93) returns false).

See also

wxBrush::SetStipple (p. 113)

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. 113), *wxBrush::SetColour* (p. 113), *wxBrush::SetStipple* (p. 113)

wxBrush::IsHatch**bool IsHatch() const**

Returns true if the style of the brush is any of hatched fills.

See also

wxBrush::GetStyle (p. 112)

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(unsigned char red, unsigned char green, unsigned char blue)

Sets the brush colour using red, green and blue values.

See also

wxBrush::GetColour (p. 111)

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. 84)

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. 112)

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. 851)

wxObject (p. **Error! Bookmark not defined.**)

Include files

<wx/gdicmn.h>

Remarks

There is only one instance of this class: **wxTheBrushList**. Use this object to search for a previously created brush of the desired type and create it if not already found. In some windowing systems, the brush may be a scarce resource, so it can pay to reuse old resources if possible. When an application finishes, all brushes will be deleted and their resources freed, eliminating the possibility of 'memory leaks'. However, it is best not to rely on this automatic cleanup because it can lead to double deletion in some circumstances.

There are two mechanisms in recent versions of wxWidgets which make the brush list less useful than it once was. Under Windows, scarce resources are cleaned up internally if they are not being used. Also, a reference counting mechanism applied to all GDI objects means that some sharing of underlying resources is possible. You don't have to keep track of pointers, working out when it is safe delete a brush, because the reference counting does it for you. For example, you can set a brush in a device context, and then immediately delete the brush you passed, because the brush is 'copied'.

So you may find it easier to ignore the brush list, and instead create and copy brushes as you see fit. If your Windows resource meter suggests your application is using too many resources, you can resort to using GDI lists to share objects explicitly.

The only compelling use for the brush list is for wxWidgets to keep track of brushes in order to clean them up on exit. It is also kept for backward compatibility with earlier versions of wxWidgets.

See also

wxBrush (p. 108)

wxBrushList::wxBrushList**void wxBrushList()**

Constructor. The application should not construct its own brush list: use the object pointer **wxTheBrushList**.

wxBrushList::AddBrush**void AddBrush(wxBrush *brush)**

Used internally by wxWidgets to add a brush to the list.

wxBrushList::FindOrCreateBrush**wxBrush * FindOrCreateBrush(const wxColour& colour, int style = wxSOLID)**

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

Parameters

colour

Colour object.

style

Brush style. See *wxBrush::SetStyle* (p. 113) for a list of styles.

wxBrushList::RemoveBrush**void RemoveBrush(wxBrush *brush)**

Used by wxWidgets to remove a brush from the list.

wxBufferedDC

This simple class provides a simple way to avoid flicker: when drawing on it, everything is in fact first drawn on an in-memory buffer (a *wxBitmap* (p. 84)) and then copied to the screen only once, when this object is destroyed.

It can be used in the same way as any other device context. *wxBufferedDC* itself typically replaces *wxClientDC* (p. 151), if you want to use it in your *OnPaint()* handler, you should look at *wxBufferedPaintDC* (p. 118).

Derived from

wxMemoryDC (p. **Error! Bookmark not defined.**)

wxDC (p. 372)

wxObject (p. **Error! Bookmark not defined.**)

Include files

<wx/dcbuffer.h>

See also

wxDC (p. 372)

wxBufferedDC::wxBufferedDC**wxBufferedDC()**

wxBufferedDC(wxDC *dc, const wxSize& area, int style = wxBUFFER_CLIENT_AREA)

wxBufferedDC(wxDC *dc, const wxBitmap& buffer, int style = wxBUFFER_CLIENT_AREA)

If you use the first, default, constructor, you must call one of the *Init* (p. 117) methods later in order to use the object.

The other constructors initialize the object immediately and *Init()* must not be called after using them.

Parameters

dc

The underlying DC: everything drawn to this object will be flushed to this DC when this object is destroyed. You may pass NULL in order to just initialize the buffer, and not flush it.

area

The size of the bitmap to be used for buffering (this bitmap is created internally when it is not given explicitly).

buffer

Explicitly provided bitmap to be used for buffering: this is the most efficient solution as the bitmap doesn't have to be recreated each time but it also requires more memory as the bitmap is never freed. The bitmap should have appropriate size, anything drawn outside of its bounds is clipped.

style

wxBUFFER_CLIENT_AREA to indicate that just the client area of the window is buffered, or wxBUFFER_VIRTUAL_AREA to indicate that the buffer bitmap covers the virtual area (in which case PrepareDC is automatically called for the actual window device context).

wxBufferedDC::Init

```
void Init(wxDC *dc, const wxSize& area, int style = wxBUFFER_CLIENT_AREA)
```

```
void Init(wxDC *dc, const wxBitmap& buffer, int style = wxBUFFER_CLIENT_AREA)
```

These functions initialize the object created using the default constructor. Please see *constructors documentation* (p. 117) for details.

```
wxBufferedDC::~wxBufferedDC
```

Copies everything drawn on the DC so far to the underlying DC associated with this object, if any.

wxBufferedPaintDC

This is a subclass of *wxBufferedDC* (p. 116) which can be used inside of an `OnPaint()` event handler. Just create an object of this class instead of *wxPaintDC* (p. **Error! Bookmark not defined.**) and that's all you have to do to (mostly) avoid flicker. The only thing to watch out for is that if you are using this class together with *wxScrolledWindow* (p. **Error! Bookmark not defined.**), you probably do **not** want to call *PrepareDC* (p. **Error! Bookmark not defined.**) on it as it already does this internally for the real underlying *wxPaintDC*.

Derived from

wxMemoryDC (p. **Error! Bookmark not defined.**)
wxDC (p. 372)
wxObject (p. **Error! Bookmark not defined.**)

Include files

```
<wx/dcbuffer.h>
```

wxBufferedPaintDC::wxBufferedPaintDC

```
wxBufferedPaintDC(wxWindow *window, const wxBitmap& buffer, int style =  
wxBUFFER_CLIENT_AREA)
```

```
wxBufferedPaintDC(wxWindow *window, int style = wxBUFFER_CLIENT_AREA)
```

As with *wxBufferedDC* (p. 117), you may either provide the bitmap to be used for buffering or let this object create one internally (in the latter case, the size of the client part of the window is used).

Pass *wxBUFFER_CLIENT_AREA* for the *style* parameter to indicate that just the client area of the window is buffered, or *wxBUFFER_VIRTUAL_AREA* to indicate that the buffer bitmap covers the virtual area (in which case *PrepareDC* is automatically called for the actual window device context).

```
wxBufferedPaintDC::~wxBufferedPaintDC
```

Copies everything drawn on the DC so far to the window associated with this object, using a *wxPaintDC* (p. **Error! Bookmark not defined.**).

wxBufferedInputStream

This stream acts as a cache. It caches the bytes read from the specified input stream (See *wxFilterInputStream* (p. 551)). 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. 551)

Include files

<wx/stream.h>

See also

wxStreamBuffer (p. **Error! Bookmark not defined.**), *wxInputStream* (p. 826), *wxBufferedOutputStream* (p. 119)

wxBufferedOutputStream

This stream acts as a cache. It caches the bytes to be written to the specified output stream (See *wxFilterOutputStream* (p. 552)). 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. 552)

Include files

<wx/stream.h>

See also

wxStreamBuffer (p. **Error! Bookmark not defined.**), *wxOutputStream* (p. **Error! Bookmark not defined.**)

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.

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. **Error! Bookmark not defined.**) in the constructor, and *wxEndBusyCursor* (p. **Error! Bookmark not defined.**) in the destructor.

Derived from

None

Include files

<wx/utils.h>

See also

wxBeginBusyCursor (p. **Error! Bookmark not defined.**), *wxEndBusyCursor* (p. **Error! Bookmark not defined.**), *wxWindowDisabler* (p. **Error! Bookmark not defined.**)

wxBusyCursor::wxBusyCursor

wxBusyCursor(wxCursor* cursor = wxHOURLASS_CURSOR)

Constructs a busy cursor object, calling *wxBeginBusyCursor* (p. **Error! Bookmark not defined.**).

wxBusyCursor::~~wxBusyCursor

~wxBusyCursor()

Destroys the busy cursor object, calling *wxEndBusyCursor* (p. **Error! Bookmark not defined.**).

wxBusyInfo

This class makes it easy to tell your user that the program is temporarily busy. Just create a *wxBusyInfo* object on the stack, and within the current scope, a message window will be shown.

For example:

```
wxBusyInfo wait("Please wait, working...");  
  
for (int i = 0; i < 100000; i++)  
{  
    DoACalculation();  
}
```

It works by creating a window in the constructor, and deleting it in the destructor.

You may also want to call *wxTheApp->Yield()* to refresh the window periodically (in case it had been obscured by other windows, for example) like this:

```
wxWindowDisabler disableAll;  
  
wxBusyInfo wait("Please wait, working...");  
  
for (int i = 0; i < 100000; i++)  
{  
    DoACalculation();  
  
    if ( !(i % 1000) )  
        wxTheApp->Yield();  
}
```

but take care to not cause undesirable reentrancies when doing it (see *wxApp::Yield()* (p. 46) for more details). The simplest way to do it is to use *wxWindowDisabler* (p. **Error! Bookmark not defined.**) class as illustrated in the above example.

Derived from

None

Include files

<wx/busyinfo.h>

wxBusyInfo::wxBusyInfo

wxBusyInfo(const wxString& *msg*, wxWindow* *parent* = NULL)

Constructs a busy info window as child of *parent* and displays *msg* in it.

NB: If *parent* is not NULL you must ensure that it is not closed while the busy info is shown.

wxBusyInfo::~~wxBusyInfo

~wxBusyInfo()

Hides and closes the window containing the information text.

wxButton

A button is a control that contains a text string, and is one of the most common elements of a GUI. It may be placed on a *dialog box* (p. 412) or *panel* (p. **Error! Bookmark not defined.**), or indeed almost any other window.

Derived from

wxControl (p. 218)

wxWindow (p. **Error! Bookmark not defined.**)

wxEvtHandler (p. 490)

wxObject (p. **Error! Bookmark not defined.**)

Include files

<wx/button.h>

Window styles

wxBU_LEFT	Left-justifies the label. Windows and GTK+ only.
wxBU_TOP	Aligns the label to the top of the button. Windows and GTK+ only.
wxBU_RIGHT	Right-justifies the bitmap label. Windows and GTK+ only.
wxBU_BOTTOM	Aligns the label to the bottom of the button. Windows and GTK+ only.
wxBU_EXACTFIT	Creates the button as small as possible instead of making it of the standard size (which is the default behaviour).
wxNO_BORDER	Creates a flat button. Windows and GTK+ only.

See also *window styles overview* (p. **Error! Bookmark not defined.**).

Event handling

EVT_BUTTON(id, func)

Process a
wxEVT_COMMAND_BUTTON_CLICKED
event, when the button is clicked.

See also

wxBitmapButton (p. 96)

wxButton::wxButton

wxButton()

Default constructor.

wxButton(wxWindow* parent, wxWindowID id, const wxString& label = wxEmptyString, const wxPoint& pos = wxDefaultPosition, const wxSize& size = wxDefaultSize, long style = 0, const wxValidator& validator = wxDefaultValidator, const wxString& name = "button")

Constructor, creating and showing a button.

The preferred way to create standard buttons is to use default value of *label*. If no label is supplied and *id* is one of standard IDs from *this list* (p. **Error! Bookmark not defined.**), standard label will be used. In addition to that, the button will be decorated with stock icons under GTK+ 2.

Parameters

parent

Parent window. Must not be NULL.

id

Button identifier. A value of wxID_ANY indicates a default value.

label

Text to be displayed on the button.

pos

Button position.

size

Button size. If the default size is specified then the button is sized appropriately for the text.

style

Window style. See *wxButton* (p. 122).

validator

Window validator.

name

Window name.

See also

wxButton::Create (p. 124), *wxValidator* (p. **Error! Bookmark not defined.**)

wxButton::~~wxButton

~wxButton()

Destructor, destroying the button.

wxButton::Create

bool Create(wxWindow* parent, wxWindowID id, const wxString& label = wxEmptyString, const wxPoint& pos = wxDefaultPosition, const wxSize& size = wxDefaultSize, long style = 0, const wxValidator& validator, const wxString& name = "button")

Button creation function for two-step creation. For more details, see *wxButton::wxButton* (p. 123).

wxButton::GetLabel

wxString GetLabel() const

Returns the string label for the button.

Return value

The button's label.

See also

wxButton::SetLabel (p. 125)

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. **Error! Bookmark not defined.**) which sets the keyboard focus for windows and text panel items, and *wxPanel::SetDefaultItem* (p. **Error! Bookmark not defined.**).

Note that under Motif, calling this function immediately after creation of a button and before the creation of other buttons will cause misalignment of the row of buttons, since default buttons are larger. To get around this, call *SetDefault* after you have created a row of buttons: *wxWidgets* will then set the size of all buttons currently on the panel to the same size.

wxButton::SetLabel

void SetLabel(const wxString& label)

Sets the string label for the button.

Parameters

label

The label to set.

See also

wxButton::GetLabel (p. 124)

wxCalculateLayoutEvent

This event is sent by *wxLayoutAlgorithm* (p. 846) to calculate the amount of the remaining client area that the window should occupy.

Derived from

wxEvent (p. 487)

wxObject (p. **Error! Bookmark not defined.**)

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. **Error! Bookmark not defined.**), *wxSashLayoutWindow* (p. **Error! Bookmark not defined.**), *wxLayoutAlgorithm* (p. 846).

wxCalculateLayoutEvent::wxCalculateLayoutEvent

wxCalculateLayoutEvent(wxWindowID *id* = 0)

Constructor.

wxCalculateLayoutEvent::GetFlags

int GetFlags() const

Returns the flags associated with this event. Not currently used.

wxCalculateLayoutEvent::GetRect

wxRect GetRect() const

Before the event handler is entered, returns the remaining parent client area that the window could occupy. When the event handler returns, this should contain the remaining parent client rectangle, after the event handler has subtracted the area that its window occupies.

wxCalculateLayoutEvent::SetFlags

void SetFlags(int *flags*)

Sets the flags associated with this event. Not currently used.

wxCalculateLayoutEvent::SetRect

void SetRect(const wxRect& *rect*)

Call this to specify the new remaining parent client area, after the space occupied by the window has been subtracted.

wxCalendarCtrl

The calendar control allows the user to pick a date. For this, it displays a window containing several parts: a control at the top to pick the month and the year (either or both of them may be disabled), and a month area below them which shows all the days in the month. The user can move the current selection using the keyboard and select the

date (generating `EVT_CALENDAR` event) by pressing `<Return>` or double clicking it.

It has advanced possibilities for the customization of its display. All global settings (such as colours and fonts used) can, of course, be changed. But also, the display style for each day in the month can be set independently using `wxCalendarDateAttr` (p. 132) class.

An item without custom attributes is drawn with the default colours and font and without border, but setting custom attributes with `SetAttr` (p. 131) 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. **Error! Bookmark not defined.**) using `SetHoliday` (p. 133) 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. 218)
`wxWindow` (p. **Error! Bookmark not defined.**)
`wxEvtHandler` (p. 490)
`wxObject` (p. **Error! Bookmark not defined.**)

Include files

`<wx/calctrl.h>`

Window styles

wxCAL_SUNDAY_FIRST Show Sunday as the first day in the week

wxCAL_MONDAY_FIRST Show Monday as the first day in the week

wxCAL_SHOW_HOLIDAYS Highlight holidays in the calendar

wxCAL_NO_YEAR_CHANGE Disable the year changing

wxCAL_NO_MONTH_CHANGE Disable the month (and, implicitly, the year) changing

wxCAL_SHOW_SURROUNDING_WEEKS Show the neighbouring weeks in the previous and next months

wxCAL_SEQUENTIAL_MONTH_SELECTION Use alternative, more compact, style for the month and year selection controls.

The default calendar style is `wxCAL_SHOW_HOLIDAYS`.

Event table macros

To process input from a calendar control, use these event handler macros to direct input

to member functions that take a *wxCalendarEvent* (p. 135) 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. 132) 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. **Error! Bookmark not defined.**)

wxCalendarDateAttr (p. 132)

wxCalendarEvent (p. 135)

wxCalendarCtrl::wxCalendarCtrl

wxCalendarCtrl()

Default constructor, use *Create* (p. 129) after it.

```
wxCalendarCtrl(wxWindow* parent, wxWindowID id, const wxDateTime& date =
wxDefaultDateTime, const wxPoint& pos = wxDefaultPosition, const wxSize& size =
wxDefaultSize, long style = wxCAL_SHOW_HOLIDAYS, const wxString& name =
wxCalendarNameStr)
```

Does the same as *Create* (p. 129) 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. **Error! Bookmark not defined.**) 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. 130)

wxCalendarCtrl::GetHeaderColourBg**const wxColour& GetHeaderColourBg() const**

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

See also

SetHeaderColours (p. 130)

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. 130)

wxCalendarCtrl::GetHighlightColourBg**const wxColour& GetHighlightColourBg() const**

Gets the background highlight colour.

See also

SetHighlightColours (p. 130)

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. 131)

wxCalendarCtrl::GetHolidayColourBg

const wxColour& GetHolidayColourBg() const

Return the background colour currently used for holiday highlighting.

See also

SetHolidayColours (p. 131)

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. 127) 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. 127).

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. 127)

Include files

<wx/calctrl.h>

wxCalendarDateAttr::wxCalendarDateAttr

wxCalendarDateAttr()

wxCalendarDateAttr(const wxColour& colText, const wxColour& colBack = wxNullColour, const wxColour& colBorder = wxNullColour, const wxFont& font = wxNullFont, wxCalendarDateBorder border = wxCAL_BORDER_NONE)

wxCalendarDateAttr(wxCalendarDateBorder border, const wxColour& colBorder = wxNullColour)

The constructors.

wxCalendarDateAttr::SetTextColour

void SetTextColour(const wxColour& colText)

Sets the text (foreground) colour to use.

wxCalendarDateAttr::SetBackgroundColour

void SetBackgroundColour(const wxColour& colBack)

Sets the text background colour to use.

wxCalendarDateAttr::SetBorderColour

void SetBorderColour(const wxColour& col)

Sets the border colour to use.

wxCalendarDateAttr::SetFont

void SetFont(const wxFont& font)

Sets the font to use.

wxCalendarDateAttr::SetBorder

void SetBorder(wxCalendarDateBorder border)

Sets the *border kind* (p. 132)

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

wxCalendarEvent

The `wxCalendarEvent` class is used together with `wxCalendarCtrl` (p. 127).

Derived from

`wxDateEvent` (p. 251)
`wxCommandEvent` (p. 184)
`wxEvent` (p. 487)
`wxObject` (p. **Error! Bookmark not defined.**)

Include files

<wx/calctrl.h>

See also

`wxCalendarCtrl` (p. 127)

wxCalendarEvent::GetWeekDay

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.

wxCalendarEvent::SetWeekDay

void SetWeekDay(wxDateTime::WeekDay day)

Sets the week day carried by the event, normally only used by the library internally.

wxCaret

A caret is a blinking cursor showing the position where the typed text will appear. The text controls usually have a caret but `wxCaret` class also allows to use a caret in other windows.

Currently, the caret appears as a rectangle of the given size. In the future, it will be possible to specify a bitmap to be used for the caret shape.

A caret is always associated with a window and the current caret can be retrieved using `wxWindow::GetCaret` (p. **Error! Bookmark not defined.**). The same caret can't be reused in two different windows.

Derived from

No base class

Include files

<wx/caret.h>

Data structures

wxCaret::wxCaret

wxCaret()

Default constructor: you must use one of Create() functions later.

wxCaret(wxWindow* window, int width, int height)

wxCaret(wxWindowBase* window, const wxSize& size)

Create the caret of given (in pixels) width and height and associates it with the given window.

wxCaret::Create

bool Create(wxWindowBase* window, int width, int height)

bool Create(wxWindowBase* window, const wxSize& size)

Create the caret of given (in pixels) width and height and associates it with the given window (same as constructor).

wxCaret::GetBlinkTime

static int GetBlinkTime()

Returns the blink time which is measured in milliseconds and is the time elapsed between 2 inversions of the caret (blink time of the caret is the same for all carets, so this functions is static).

wxCaret::GetPosition

void GetPosition(int* x, int* y) const

wxPoint GetPosition() const

Get the caret position (in pixels).

wxPerl note: In wxPerl there are two methods instead of a single overloaded method:

GetPosition()

Returns a Wx::Point

GetPositionXY()

Returns a 2-element list (x, y)

wxCaret::GetSize

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

wxSize GetSize() const

Get the caret size.

wxPerl note: In wxPerl there are two methods instead of a single overloaded method:

GetSize()

Returns a `Wx::Size`

GetSizeWH()

Returns a 2-element list (`width`,
`height`)

wxCaret::GetWindow

wxWindow* GetWindow() const

Get the window the caret is associated with.

wxCaret::Hide

void Hide()

Same as `wxCaret::Show(false)` (p. 138).

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. 136)

wxCaret::SetSize

void SetSize(int *width*, int *height*)

void SetSize(const wxSize& *size*)

Changes the size of the caret.

wxCaret::Show

void Show(bool *show* = true)

Shows or hides the caret. Notice that if the caret was hidden N times, it must be shown N times as well to reappear on the screen.

wxCheckBox

A checkbox is a labelled box which by default is either on (checkmark is visible) or off (no checkmark). Optionally (when the wxCHK_3STATE style flag is set) it can have a third state, called the mixed or undetermined state. Often this is used as a "Does Not Apply" state.

Derived from

wxControl (p. 218)

wxWindow (p. **Error! Bookmark not defined.**)

wxEvtHandler (p. 490)

wxObject (p. **Error! Bookmark not defined.**)

Include files

<wx/checkbox.h>

Window styles

wxCHK_2STATE

Create a 2-state checkbox. This is the default.

wxCHK_3STATE

Create a 3-state checkbox. Not implemented in wxMGL, wxOS2 and wxGTK built against GTK+ 1.2.

wxCHK_ALLOW_3RD_STATE_FOR_USER

By default a user can't set a 3-state checkbox to the third state. It can only be done

from code. Using this flags allows the user to set the checkbox to the third state by clicking.

wxALIGN_RIGHT

Makes the text appear on the left of the checkbox.

See also *window styles overview* (p. **Error! Bookmark not defined.**).

Event handling**EVT_CHECKBOX(id, func)**

Process a `wxEVT_COMMAND_CHECKBOX_CLICKED` event, when the checkbox is clicked.

See also

`wxRadioButton` (p. **Error! Bookmark not defined.**), `wxCommandEvent` (p. 184)

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. 138).

validator

Window validator.

name

Window name.

See also

wxCheckBox::Create (p. 140), *wxValidator* (p. **Error! Bookmark not defined.**)

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. 139) for details.

wxCheckBox::GetValue

bool GetValue() const

Gets the state of a 2-state checkbox.

Return value

Returns `true` if it is checked, `false` otherwise.

wxCheckBox::Get3StateValue

wxCheckBoxState Get3StateValue() const

Gets the state of a 3-state checkbox.

Return value

Returns `wxCHK_UNCHECKED` when the checkbox is unchecked, `wxCHK_CHECKED` when it is checked and `wxCHK_UNDETERMINED` when it's in the undetermined state. Asserts when the function is used with a 2-state checkbox.

wxCheckBox::Is3rdStateAllowedForUser**bool Is3rdStateAllowedForUser() const**

Returns whether or not the user can set the checkbox to the third state.

Return value

Returns `true` if the user can set the third state of this checkbox, `false` if it can only be set programmatically or if it's a 2-state checkbox.

wxCheckBox::Is3State**bool Is3State() const**

Returns whether or not the checkbox is a 3-state checkbox.

Return value

Returns `true` if this checkbox is a 3-state checkbox, `false` if it's a 2-state checkbox.

wxCheckBox::IsChecked**bool IsChecked() const**

This is just a maybe more readable synonym for *GetValue* (p. 141): just as the latter, it returns `true` if the checkbox is checked and `false` otherwise.

wxCheckBox::SetValue**void SetValue(bool state)**

Sets the checkbox to the given state. This does not cause a `wxEVT_COMMAND_CHECKBOX_CLICKED` event to get emitted.

Parameters*state*

If `true`, the check is on, otherwise it is off.

wxCheckBox::Set3StateValue**void Set3StateValue(const wxCheckBoxState state)**

Sets the checkbox to the given state. This does not cause a `wxEVT_COMMAND_CHECKBOX_CLICKED` event to get emitted.

Parameters*state*

Can be one of: `wxCHK_UNCHECKED` (Check is off), `wxCHK_CHECKED` (Check

is on) or `wxCHK_UNDETERMINED` (Check is mixed). Asserts when the checkbox is a 2-state checkbox and setting the state to `wxCHK_UNDETERMINED`.

wxCheckListBox

A checklistbox is like a listbox, but allows items to be checked or unchecked.

When using this class under Windows `wxWidgets` must be compiled with `USE_OWNER_DRAWN` set to 1.

Only the new functions for this class are documented; see also *wxListBox* (p. 858).

Please note that `wxCheckListBox` uses client data in its implementation, and therefore this is not available to the application.

Derived from

wxListBox (p. 858)
wxControl (p. 218)
wxWindow (p. **Error! Bookmark not defined.**)
wxEvtHandler (p. 490)
wxObject (p. **Error! Bookmark not defined.**)

Include files

<wx/checklst.h>

Window styles

See *wxListBox* (p. 858).

Event handling

EVT_CHECKLISTBOX(id, func)	Process a <code>wxEVT_COMMAND_CHECKLISTBOX_TOGGLED</code> event, when an item in the check list box is checked or unchecked.
-----------------------------------	--

See also

wxListBox (p. 858), *wxChoice* (p. 145), *wxComboBox* (p. 176), *wxListCtrl* (p. 864), *wxCommandEvent* (p. 184)

wxCheckListBox::wxCheckListBox

wxCheckListBox()

Default constructor.

wxCheckListBox(wxWindow* parent, wxWindowID id, const wxPoint& pos = wxDefaultPosition, const wxSize& size = wxDefaultSize, int n, const wxString

```
choices[] = NULL, long style = 0, const wxValidator& validator = wxDefaultValidator,  
const wxString& name = "listBox")
```

```
wxCheckListBox(wxWindow* parent, wxWindowID id, const wxPoint& pos, const  
wxSize& size, const wxString& choices, long style = 0, const wxValidator&  
validator = wxDefaultValidator, const wxString& name = "listBox")
```

Constructor, creating and showing a list box.

Parameters

parent

Parent window. Must not be NULL.

id

Window identifier. A value of -1 indicates a default value.

pos

Window position.

size

Window size. If the default size (-1, -1) is specified then the window is sized appropriately.

n

Number of strings with which to initialise the control.

choices

An array of strings with which to initialise the control.

style

Window style. See *wxCheckListBox* (p. 142).

validator

Window validator.

name

Window name.

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

wxPerl note: In wxPerl there is just an array reference in place of *nand choices*.

wxCheckListBox::~~wxCheckListBox

void ~wxCheckListBox()

Destructor, destroying the list box.

wxCheckListBox::Check**void Check(int item, bool check = true)**

Checks the given item. Note that calling this method doesn't result in `wxEVT_COMMAND_CHECKLISTBOX_TOGGLED` being emitted.

Parameters

item

Index of item to check.

check

true if the item is to be checked, false otherwise.

wxCheckListBox::IsChecked**bool IsChecked(unsigned int item) const**

Returns true if the given item is checked, false otherwise.

Parameters

item

Index of item whose check status is to be returned.

wxChoice

A choice item is used to select one of a list of strings. Unlike a listbox, only the selection is visible until the user pulls down the menu of choices.

Derived from

wxControlWithItems (p. 219)

wxControl (p. 218)

wxWindow (p. **Error! Bookmark not defined.**)

wxEvtHandler (p. 490)

wxObject (p. **Error! Bookmark not defined.**)

Include files

<wx/choice.h>

Window styles

There are no special styles for `wxChoice`.

See also *window styles overview* (p. [Error! Bookmark not defined.](#)).

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. 858), *wxComboBox* (p. 176), *wxCommandEvent* (p. 184)

wxChoice::wxChoice

wxChoice()

Default constructor.

wxChoice(wxWindow *parent, wxWindowID id, const wxPoint& pos, const wxSize& size, int n, const wxString choices[], long style = 0, const wxValidator& validator = wxDefaultValidator, const wxString& name = "choice")

wxChoice(wxWindow *parent, wxWindowID id, const wxPoint& pos, const wxSize& size, const wxStringArray& 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. 145).

validator

Window validator.

name

Window name.

See also

wxChoice::Create (p. 147), *wxValidator* (p. **Error! Bookmark not defined.**)

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

wxPerl note: In wxPerl there is just an array reference in place of `nand choices`.

wxChoice::~~wxChoice

~wxChoice()

Destructor, destroying the choice item.

wxChoice::Create

bool Create(wxWindow *parent, wxWindowID id, const wxPoint& pos, const wxSize& size, int n, const wxString choices[], long style = 0, const wxValidator& validator = wxDefaultValidator, const wxString& name = "choice")

bool Create(wxWindow *parent, wxWindowID id, const wxPoint& pos, const wxSize& size, const wxString& choices, long style = 0, const wxValidator& validator = wxDefaultValidator, const wxString& name = "choice")

Creates the choice for two-step construction. See *wxChoice::wxChoice* (p. 145).

wxChoice::GetColumns

int GetColumns() const

Gets the number of columns in this choice item.

Remarks

This is implemented for Motif only and always returns 1 for the other platforms.

wxChoice::GetCurrentSelection

int GetCurrentSelection() const

Unlike *GetSelection* (p. 222) which only returns the accepted selection value, i.e. the selection in the control once the user closes the dropdown list, this function returns the current selection. That is, while the dropdown list is shown, it returns the currently selected item in it. When it is not shown, its result is the same as for the other function.

This function is new since wxWidgets version 2.6.2 (before this version *GetSelection* (p. 222) itself behaved like this).

wxChoice::SetColumns**void SetColumns(int *n* = 1)**

Sets the number of columns in this choice item.

Parameters

n

Number of columns.

Remarks

This is implemented for Motif only and doesn't do anything under other platforms.wxChoicebook

wxChoicebook is a class similar to *wxNotebook* (p. **Error! Bookmark not defined.**) but which uses a *wxChoice* (p. 145) to show the labels instead of the tabs.

There is no documentation for this class yet but its usage is identical to *wxNotebook* (except for the features clearly related to tabs only), so please refer to that class documentation for now. You can also use the *notebook sample* (p. **Error! Bookmark not defined.**) to see wxChoicebook in action.

Derived from

wxControl (p. 218)

wxWindow (p. **Error! Bookmark not defined.**)

wxEvtHandler (p. 490)

wxObject (p. **Error! Bookmark not defined.**)

Include files

<wx/choicebk.h>

Window styles**wxCHB_DEFAULT**

Choose the default location for the labels depending on the current platform (left everywhere except Mac where it is top).

wxCHB_TOP	Place labels above the page area.
wxCHB_LEFT	Place labels on the left side.
wxCHB_RIGHT	Place labels on the right side.
wxCHB_BOTTOM	Place labels below the page area.

See also

wxBookCtrl (p. **Error! Bookmark not defined.**), *wxNotebook* (p. **Error! Bookmark not defined.**), *notebook sample* (p. **Error! Bookmark not defined.**)

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. **Error! Bookmark not defined.**), *wxObject* (p. **Error! Bookmark not defined.**)

wxClassInfo::wxClassInfo

wxClassInfo(const wxChar * *className*, const wxClassInfo * *baseClass1*, const wxClassInfo * *baseClass2*, int *size*, wxObjectConstructorFn *fn*)

Constructs a wxClassInfo object. The supplied macros implicitly construct objects of this class, so there is no need to create such objects explicitly in an application.

wxClassInfo::CreateObject

wxObject* CreateObject()

Creates an object of the appropriate kind. Returns NULL if the class has not been declared dynamically creatable (typically, it is an abstract class).

wxClassInfo::FindClass

static wxClassInfo * FindClass(wxChar * *name*)

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

`wxClassInfo::GetBaseClassName1`

`wxChar * GetBaseClassName1() const`

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

`wxClassInfo::GetBaseClassName2`

`wxChar * GetBaseClassName2() const`

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

`wxClassInfo::GetClassName`

`wxChar * GetClassName() const`

Returns the string form of the class name.

`wxClassInfo::GetSize`

`int GetSize() const`

Returns the size of the class.

`wxClassInfo::InitializeClasses`

`static void InitializeClasses()`

Initializes pointers in the `wxClassInfo` objects for fast execution of `IsKindOf`. Called in base `wxWidgets` library initialization.

`wxClassInfo::IsKindOf`

`bool IsKindOf(wxClassInfo* info)`

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

`wxClient`

A `wxClient` object represents the client part of a client-server DDE-like (Dynamic Data Exchange) conversation. The actual DDE-based implementation using `wxDDEClient` is available on Windows only, but a platform-independent, socket-based version of this API is available using `wxTCPClient`, which has the same API.

To create a client which can communicate with a suitable server, you need to derive a class from `wxConnection` and another from `wxClient`. The custom `wxConnection` class will intercept communications in a 'conversation' with a server, and the custom `wxClient`

is required so that a user-overridden `wxClient::OnMakeConnection` (p. 151) member can return a `wxConnection` of the required class, when a connection is made. Look at the IPC sample and the *Interprocess communications overview* (p. **Error! Bookmark not defined.**) for an example of how to do this.

Derived from

`wxClientBase`
`wxObject` (p. **Error! Bookmark not defined.**)

Include files

<wx/ipc.h>

See also

`wxServer` (p. **Error! Bookmark not defined.**), `wxConnection` (p. 210), *Interprocess communications overview* (p. **Error! Bookmark not defined.**)

wxClient::wxClient

`wxClient()`

Constructs a client object.

wxClient::MakeConnection

wxConnectionBase * MakeConnection(const wxString& host, const wxString& service, const wxString& topic)

Tries to make a connection with a server by host (machine name under UNIX - use 'localhost' for same machine; ignored when using native DDE in Windows), service name and topic string. If the server allows a connection, a `wxConnection` object will be returned. The type of `wxConnection` returned can be altered by overriding the `wxClient::OnMakeConnection` (p. 151) member to return your own derived connection object.

Under Unix, the service name may be either an integer port identifier in which case an Internet domain socket will be used for the communications, or a valid file name (which shouldn't exist and will be deleted afterwards) in which case a Unix domain socket is created.

SECURITY NOTE: Using Internet domain sockets is extremely insecure for IPC as there is absolutely no access control for them, use Unix domain sockets whenever possible!

wxClient::OnMakeConnection

wxConnectionBase * OnMakeConnection()

Called by `wxClient::MakeConnection` (p. 151), by default this simply returns a new

`wxConnection` object. Override this method to return a `wxConnection` descendant customised for the application.

The advantage of deriving your own connection class is that it will enable you to intercept messages initiated by the server, such as `wxConnection::OnAdvise` (p. 212). You may also want to store application-specific data in instances of the new class.

wxClient::ValidHost

bool ValidHost(const wxString& host)

Returns true if this is a valid host name, false otherwise. This always returns true under MS Windows.

wxClientDC

A `wxClientDC` must be constructed if an application wishes to paint on the client area of a window from outside an **OnPaint** event. This should normally be constructed as a temporary stack object; don't store a `wxClientDC` object.

To draw on a window from within **OnPaint**, construct a `wxPaintDC` (p. **Error! Bookmark not defined.**) object.

To draw on the whole window including decorations, construct a `wxWindowDC` (p. **Error! Bookmark not defined.**) object (Windows only).

Derived from

`wxWindowDC` (p. **Error! Bookmark not defined.**)
`wxDC` (p. 372)

Include files

<wx/dcclient.h>

See also

`wxDC` (p. 372), `wxMemoryDC` (p. **Error! Bookmark not defined.**), `wxPaintDC` (p. **Error! Bookmark not defined.**), `wxWindowDC` (p. **Error! Bookmark not defined.**), `wxScreenDC` (p. **Error! Bookmark not defined.**)

wxClientDC::wxClientDC

wxClientDC(wxWindow* window)

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

wxClientData

All classes deriving from *wxEvtHandler* (p. 490) (such as all controls and *wxApp* (p. 36)) can hold arbitrary data which is here referred to as "client data". This is useful e.g. for scripting languages which need to handle shadow objects for most of *wxWidgets'* classes and which store a handle to such a shadow class as client data in that class. This data can either be of type *void* - in which case the *datacontainer* does not take care of freeing the data again or it is of type *wxClientData* or its derivatives. In that case the container (e.g. a control) will free the memory itself later. Note that you *must not* assign both *void* data and data derived from the *wxClientData* class to a container.

Some controls can hold various items and these controls can additionally hold client data for each item. This is the case for *wxChoice* (p. 145), *wxComboBox* (p. 176) and *wxListBox* (p. 858). *wxTreeCtrl* (p. **Error! Bookmark not defined.**) has a specialized class *wxTreeItemData* (p. **Error! Bookmark not defined.**) for each item in the tree.

If you want to add client data to your own classes, you may use the mix-in class *wxClientDataContainer* (p. 153).

Include files

<wx/clntdata.h>

See also

wxEvtHandler (p. 490), *wxTreeItemData* (p. **Error! Bookmark not defined.**), *wxStringClientData* (p. **Error! Bookmark not defined.**), *wxClientDataContainer* (p. 153)

wxClientData::wxClientData

wxClientData()

Constructor.

wxClientData::~~wxClientData

~wxClientData()

Virtual destructor.

wxClientDataContainer

This class is a mixin that provides storage and management of "client data." This data can either be of type *void* - in which case the *datacontainer* does not take care of freeing the data again or it is of type *wxClientData* or its derivatives. In that case the container will free the memory itself later. Note that you *must not* assign both *void* data and data derived from the *wxClientData* class to a container.

NOTE: This functionality is currently duplicated in *wxEvtHandler* in order to avoid having more than one vtable in that class hierarchy.

See also

wxEvtHandler (p. 490), *wxClientData* (p. 152)

Derived from

No base class

Include files

<wx/clntdata.h>

Data structures**wxClientDataContainer::wxClientDataContainer**

wxClientDataContainer()

wxClientDataContainer::~~wxClientDataContainer

~wxClientDataContainer()

wxClientDataContainer::GetClientData

void* GetClientData() const

Get the untyped client data.

wxClientDataContainer::GetClientObject

wxClientData* GetClientObject() const

Get a pointer to the client data object.

wxClientDataContainer::SetClientData

void SetClientData(void* data)

Set the untyped client data.

wxClientDataContainer::SetClientObject

void SetClientObject(wxClientData* data)

Set the client data object. Any previous object will be deleted.

wxClipboard

A class for manipulating the clipboard. Note that this is not compatible with the clipboard class from wxWidgets 1.xx, which has the same name but a different implementation.

To use the clipboard, you call member functions of the global **wxTheClipboard** object.

See also the *wxDataObject overview* (p. **Error! Bookmark not defined.**) for further information.

Call *wxClipboard::Open* (p. 157) to get ownership of the clipboard. If this operation returns true, you now own the clipboard. Call *wxClipboard::SetData* (p. 157) to put data on the clipboard, or *wxClipboard::GetData* (p. 156) to retrieve data from the clipboard. Call *wxClipboard::Close* (p. 156) 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. **Error! Bookmark not defined.**)

Include files

<wx/clipbrd.h>

See also

Drag and drop overview (p. **Error! Bookmark not defined.**), *wxDataObject* (p. 242)

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. 156).

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

See also

wxClipboard::SetData (p. 157)

wxClipboard::Clear**void Clear()**

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

wxClipboard::Close**void Close()**

Call this function to close the clipboard, having opened it with *wxClipboard::Open* (p. 157).

wxClipboard::Flush**bool Flush()**

Flushes the clipboard: this means that the data which is currently on clipboard will stay available even after the application exits (possibly eating memory), otherwise the clipboard will be emptied on exit. Returns false if the operation is unsuccessful for any reason.

wxClipboard::GetData**bool GetData(wxDataObject& data)**

Call this function to fill *data* with data on the clipboard, if available in the required format. Returns true on success.

wxClipboard::IsOpened

bool IsOpened() const

Returns true if the clipboard has been opened.

wxClipboard::IsSupported**bool IsSupported(const wxDataFormat& *format*)**

Returns true if there is data which matches the data format of the given data object currently **available** (IsSupported sounds like a misnomer, FIXME: better deprecate this name?) on the clipboard.

wxClipboard::Open**bool Open()**

Call this function to open the clipboard before calling *wxClipboard::SetData* (p. 157) and *wxClipboard::GetData* (p. 156).

Call *wxClipboard::Close* (p. 156) 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. 155)

wxClipboard::UsePrimarySelection**void UsePrimarySelection(bool *primary* = true)**

On platforms supporting it (currently only GTK), selects the so called PRIMARY SELECTION as the clipboard as opposed to the normal clipboard, if *primary* is true.

wxCloseEvent

This event class contains information about window and session close events.

The handler function for EVT_CLOSE is called when the user has tried to close a a frame or dialog box using the window manager (X) or system menu (Windows). It can

also be invoked by the application itself programmatically, for example by calling the `wxWindow::Close` (p. **Error! Bookmark not defined.**) function.

You should check whether the application is forcing the deletion of the window using `wxCloseEvent::CanVeto` (p. 158). If this is `false`, you *must* destroy the window using `wxWindow::Destroy` (p. **Error! Bookmark not defined.**). If the return value is `true`, it is up to you whether you respond by destroying the window.

If you don't destroy the window, you should call `wxCloseEvent::Veto` (p. 159) to let the calling code know that you did not destroy the window. This allows the `wxWindow::Close` (p. **Error! Bookmark not defined.**) function to return `true` or `false` depending on whether the close instruction was honoured or not.

Derived from

`wxEvent` (p. 487)

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::Close` (p. **Error! Bookmark not defined.**), *Window deletion overview* (p. **Error! Bookmark not defined.**)

`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 just logging off or false if the system is shutting down. This method can only be called for end session and query end session events, it doesn't make sense for close window event.

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. 158) returns true.

wxCmdLineParser

wxCmdLineParser is a class for parsing the command line.

It has the following features:

1. distinguishes options, switches and parameters; allows option grouping
2. allows both short and long options
3. automatically generates the usage message from the command line description

4. does type checks on the options values (number, date, ...).

To use it you should follow these steps:

1. *construct* (p. 162) 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. 166). 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. 167) 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. 167) allows one more parameter - this may be changed by giving non-default flags to it, i.e. use `wxCMD_LINE_OPTION_MANDATORY` to require that the option is given and `wxCMD_LINE_PARAM_OPTIONAL` to make a parameter optional. Also, `wxCMD_LINE_PARAM_MULTIPLE` may be specified if the programs accepts a variable number of parameters - but it only can be given for the last parameter in the command line description. If you use this flag, you will probably need to use *GetParamCount* (p. 168) to retrieve the number of parameters effectively specified after calling *Parse* (p. 167).

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. 37) and `wxApp::argv` (p. 37)
console sample

Construction

Before *Parse* (p. 167) 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. 167).

To specify the command line to parse you may use either one of constructors accepting it (*wxCmdLineParser(argc, argv)* (p. 163) or *wxCmdLineParser* (p. 164) usually) or, if you use *the default constructor* (p. 163), you can do it later by calling *SetCmdLine* (p. 164).

The same holds for command line description: it can be specified either in the constructor (*without command line* (p. 164) or *together with it* (p. 164)) or constructed later using either *SetDesc* (p. 166) or combination of *AddSwitch* (p. 167), *AddOption* (p. 167) and *AddParam* (p. 167) methods.

Using constructors or *SetDesc* (p. 166) 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. 167).

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. 166).

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. 165) method.

Finally, *SetLogo* (p. 166) can be used to show some application-specific text before the

explanation given by *Usage* (p. 167) function.

Parsing command line

After the command line description was constructed and the desired options were set, you can finally call *Parse* (p. 167) method. It returns 0 if the command line was correct and was parsed, -1 if the help option was specified (this is a separate case as, normally, the program will terminate after this) or a positive number if there was an error during the command line parsing.

In the latter case, the appropriate error message and usage information are logged by *wxCmdLineParser* itself using the standard *wxWidgets* logging functions.

Getting results

After calling *Parse* (p. 167) (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. 168) 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. 164) later.

wxCmdLineParser::wxCmdLineParser

wxCmdLineParser(int argc, char argv)**

wxCmdLineParser(int argc, wchar_t argv)**

Constructor specifies the command line to parse. This is the traditional (Unix) command line format. The parameters *argc* and *argv* have the same meaning as for *main*() function.

The second overloaded constructor is only available in Unicode build. The first one is available in both ANSI and Unicode modes because under some platforms the command line arguments are passed as ASCII strings even to Unicode programs.

wxCmdLineParser::wxCmdLineParser

wxCmdLineParser(const wxString& cmdline)

Constructor specifies the command line to parse in Windows format. The parameter *cmdline* has the same meaning as the corresponding parameter of *WinMain*().

wxCmdLineParser::wxCmdLineParser**wxCmdLineParser(const wxCmdLineEntryDesc* desc)**

Same as *wxCmdLineParser* (p. 163), but also specifies the *command line description* (p. 166).

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

Same as *wxCmdLineParser* (p. 163), but also specifies the *command line description* (p. 166).

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

Same as *wxCmdLineParser* (p. 164), but also specifies the *command line description* (p. 166).

wxCmdLineParser::ConvertStringToArgs**static wxArrayString ConvertStringToArgs(const wxChar *cmdline)**

Breaks down the string containing the full command line in words. The words are separated by whitespace. The quotes can be used in the input string to quote the white space and the back slashes can be used to quote the quotes.

wxCmdLineParser::SetCmdLine**void SetCmdLine(int argc, char** argv)****void SetCmdLine(int argc, wchar_t** argv)**

Set command line to parse after using one of the constructors which don't do it. The second overload of this function is only available in Unicode build.

See also*wxCmdLineParser* (p. 163)**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. 164)

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. 162) and *AreLongOptionsEnabled* (p. 166)

wxCmdLineParser::DisableLongOptions**void DisableLongOptions()**

Identical to *EnableLongOptions(false)* (p. 165).

wxCmdLineParser::AreLongOptionsEnabled**bool AreLongOptionsEnabled()**

Returns true if long options are enabled, otherwise false.

See also

EnableLongOptions (p. 165)

wxCmdLineParser::SetLogo**void SetLogo(const wxString& logo)**

logo is some extra text which will be shown by *Usage* (p. 167) method.

wxCmdLineParser::SetDesc

void SetDesc(const wxCmdLineEntryDesc* desc)

Construct the command line description

Take the command line description from the wxCMD_LINE_NONE terminated table.

Example of usage:

```
static const wxCmdLineEntryDesc cmdLineDesc[] =
{
    { wxCMD_LINE_SWITCH, "v", "verbose", "be verbose" },
    { wxCMD_LINE_SWITCH, "q", "quiet", "be quiet" },

    { wxCMD_LINE_OPTION, "o", "output", "output file" },
    { wxCMD_LINE_OPTION, "i", "input", "input dir" },
    { wxCMD_LINE_OPTION, "s", "size", "output block size",
wxCMD_LINE_VAL_NUMBER },
    { wxCMD_LINE_OPTION, "d", "date", "output file date",
wxCMD_LINE_VAL_DATE },

    { wxCMD_LINE_PARAM, NULL, NULL, "input file",
wxCMD_LINE_VAL_STRING, wxCMD_LINE_PARAM_MULTIPLE },

    { wxCMD_LINE_NONE }
};

wxCmdLineParser parser;

parser.SetDesc(cmdLineDesc);
```

wxCmdLineParser::AddSwitch

void AddSwitch(const wxString& name, const wxString& lng = wxEmptyString, const wxString& desc = wxEmptyString, int flags = 0)

Add a switch *name* with an optional long name *lng* (no long name if it is empty, which is default), description *desc* and flags *flags* to the command line description.

wxCmdLineParser::AddOption

void AddOption(const wxString& name, const wxString& lng = wxEmptyString, const wxString& desc = wxEmptyString, wxCmdLineParamType type = wxCMD_LINE_VAL_STRING, int flags = 0)

Add an option *name* with an optional long name *lng* (no long name if it is empty, which is default) taking a value of the given type (string by default) to the command line description.

wxCmdLineParser::AddParam

void AddParam(const wxString& desc = wxEmptyString, wxCmdLineParamType type = wxCMD_LINE_VAL_STRING, int flags = 0)

Add a parameter of the given *type* to the command line description.

wxCmdLineParser::Parse**int Parse**(bool *giveUsage* = *true*)

Parse the command line, return 0 if ok, -1 if "-h" or "--help" option was encountered and the help message was given or a positive value if a syntax error occurred.

Parameters*giveUsage*

If *true* (default), the usage message is given if a syntax error was encountered while parsing the command line or if help was requested. If *false*, only error messages about possible syntax errors are given, use *Usage* (p. 167) to show the usage message from the caller if needed.

wxCmdLineParser::Usage**void Usage**()

Give the standard usage message describing all program options. It will use the options and parameters descriptions specified earlier, so the resulting message will not be helpful to the user unless the descriptions were indeed specified.

See also*SetLogo* (p. 166)**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. 168)

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. 173) for how a pointer to a predefined, named colour may be returned instead of creating a new colour.

Valid RGB values are in the range 0 to 255.

You can retrieve the current system colour settings with *wxSystemSettings* (p. **Error! Bookmark not defined.**).

Derived from

wxObject (p. **Error! Bookmark not defined.**)

Include files

<wx/colour.h>

Predefined objects

Objects:

wxNullColour

Pointers:

wxBLACK

wxWHITE

wxRED

wxBLUE

wxGREEN
wxCYAN
wxLIGHT_GREY

See also

wxColourDatabase (p. 173), *wxPen* (p. **Error! Bookmark not defined.**), *wxBrush* (p. 108), *wxColourDialog* (p. 175), *wxSystemSettings* (p. **Error! Bookmark not defined.**)

wxColour::wxColour

wxColour()

Default constructor.

wxColour(unsigned char *red*, unsigned char *green*, 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. 173)

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(unsigned char *red*, unsigned char *green*, 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. 173)

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. **Error! Bookmark not defined.**)

Include files

<wx/cmndata.h>

See also

wxColour (p. 168), *wxColourDialog* (p. 175), *wxColourDialog overview* (p. **Error! Bookmark not defined.**)

wxColourData::wxColourData

wxColourData()

Constructor. Initializes the custom colours to `wxNullColour`, the *data colour* setting to black, and the *choose full* setting to true.

wxColourData::~~wxColourData

~wxColourData()

Destructor.

wxColourData::GetChooseFull

bool GetChooseFull() const

Under Windows, determines whether the Windows colour dialog will display the full dialog with custom colour selection controls. Under PalmOS, determines whether colour dialog will display full rgb colour picker or only available palette indexer. Has no meaning under other platforms.

The default value is true.

wxColourData::GetColour

wxColour& GetColour() const

Gets the current colour associated with the colour dialog.

The default colour is black.

wxColourData::GetCustomColour

wxColour& GetCustomColour(int i) const

Gets the *i*th custom colour associated with the colour dialog. *i* should be an integer between 0 and 15.

The default custom colours are invalid colours.

wxColourData::SetChooseFull

void SetChooseFull(const bool flag)

Under Windows, tells the Windows colour dialog to display the full dialog with custom colour selection controls. Under other platforms, has no effect.

The default value is true.

wxColourData::SetColour

void SetColour(const wxColour& colour)

Sets the default colour for the colour dialog.

The default colour is black.

wxColourData::SetCustomColour

void SetCustomColour(int i, const wxColour& colour)

Sets the *i*th custom colour for the colour dialog. *i* should be an integer between 0 and 15.

The default custom colours are invalid colours.

wxColourData::operator =

void operator =(const wxColourData& data)

Assignment operator for the colour data.

wxColourDatabase

wxWidgets maintains a database of standard RGB colours for a predefined set of named colours (such as "BLACK", "LIGHT GREY"). The application may add to this set if desired by using *AddColour* (p. 174) and may use it to look up colours by names using *Find* (p. 174) or find the names for the standard colour using *FindName* (p. 175).

There is one predefined instance of this class called **wxTheColourDatabase**.

Derived from

None

Include files

<wx/gdicmn.h>

Remarks

The standard database contains at least the following colours:

AQUAMARINE, BLACK, BLUE, BLUE VIOLET, BROWN, CADET BLUE, CORAL, CORNFLOWER BLUE, CYAN, DARK GREY, DARK GREEN, DARK OLIVE GREEN, DARK ORCHID, DARK SLATE BLUE, DARK SLATE GREY, DARK TURQUOISE, DIM GREY, FIREBRICK, FOREST GREEN, GOLD, GOLDENROD, GREY, GREEN, GREEN YELLOW, INDIAN RED, KHAKI, LIGHT BLUE, LIGHT GREY, LIGHT STEEL BLUE, LIME GREEN, MAGENTA, MAROON, MEDIUM AQUAMARINE, MEDIUM BLUE, MEDIUM FOREST GREEN, MEDIUM GOLDENROD, MEDIUM ORCHID, MEDIUM SEA GREEN, MEDIUM SLATE BLUE, MEDIUM SPRING GREEN, MEDIUM TURQUOISE, MEDIUM VIOLET RED, MIDNIGHT BLUE, NAVY, ORANGE, ORANGE RED, ORCHID, PALE GREEN, PINK, PLUM, PURPLE, RED, SALMON, SEA GREEN, SIENNA, SKY BLUE, SLATE BLUE, SPRING GREEN, STEEL BLUE, TAN, THISTLE, TURQUOISE, VIOLET, VIOLET RED, WHEAT, WHITE, YELLOW, YELLOW GREEN.

See also

wxColour (p. 168)

wxColourDatabase::wxColourDatabase

wxColourDatabase()

Constructs the colour database. It will be initialized at the first use.

wxColourDatabase::AddColour

void AddColour(const wxString& colourName, const wxColour& colour)

void AddColour(const wxString& colourName, wxColour* colour)

Adds a colour to the database. If a colour with the same name already exists, it is replaced.

Please note that the overload taking a pointer is deprecated and will be removed in the next wxWidgets version, please don't use it.

wxColourDatabase::Find

wxColour Find(const wxString& colourName)

Finds a colour given the name. Returns an invalid colour object (that is, such that its *Ok()* (p. 171) method returns *false*) if the colour wasn't found in the database.

wxColourDatabase::FindName

wxString FindName(const wxColour& colour) const

Finds a colour name given the colour. Returns an empty string if the colour is not found in the database.

wxColourDialog

This class represents the colour chooser dialog.

Derived from

wxDialog (p. 412)

wxWindow (p. **Error! Bookmark not defined.**)

wxEvtHandler (p. 490)

wxObject (p. **Error! Bookmark not defined.**)

Include files

<wx/colordlg.h>

See also

wxColourDialog Overview (p. **Error! Bookmark not defined.**),

wxColour (p. 168),

wxColourData (p. 172),

wxGetColourFromUser (p. **Error! Bookmark not defined.**)

wxColourDialog::wxColourDialog**wxColourDialog**(wxWindow* *parent*, wxColourData* *data* = NULL)

Constructor. Pass a parent window, and optionally a pointer to a block of colour data, which will be copied to the colour dialog's colour data. Custom colours from colour data object will be used in dialog's colour palette. Invalid entries in custom colours list will be ignored on some platforms (GTK) or replaced with white colour on platforms where custom colours palette has fixed size (MSW).

See also*wxColourData* (p. 172)**wxColourDialog::~wxColourDialog****~wxColourDialog**()

Destructor.

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

Same as *constructor* (p. 175).

wxColourDialog::GetColourData**wxColourData& GetColourData**()

Returns the *colour data* (p. 172) associated with the colour dialog.

wxColourDialog::ShowModal**int ShowModal**()

Shows the dialog, returning wxID_OK if the user pressed OK, and wxID_CANCEL otherwise.

wxComboBox

A combobox is like a combination of an edit control and a listbox. It can be displayed as static list with editable or read-only text field; or a drop-down list with text field; or a drop-down list without a text field.

A combobox permits a single selection only. Combobox items are numbered from zero.

Derived from

wxControlWithItems (p. 219)
wxControl (p. 218)
wxWindow (p. **Error! Bookmark not defined.**)
wxEvtHandler (p. 490)
wxObject (p. **Error! Bookmark not defined.**)

Include files

<wx/combobox.h>

Window styles

wxCB_SIMPLE	Creates a combobox with a permanently displayed list. Windows only.
wxCB_DROPDOWN	Creates a combobox with a drop-down list.
wxCB_READONLY	Same as wxCB_DROPDOWN but only the strings specified as the combobox choices can be selected, it is impossible to select (even from a program) a string which is not in the choices list.
wxCB_SORT	Sorts the entries in the list alphabetically.
wxPROCESS_ENTER	The control will generate the event wxEVT_COMMAND_TEXT_ENTER (otherwise pressing Enter key is either processed internally by the control or used for navigation between dialog controls). Windows only.

See also *window styles overview* (p. **Error! Bookmark not defined.**).

Event handling

EVT_COMBOBOX(id, func)	Process a wxEVT_COMMAND_COMBOBOX_SELECTED event, when an item on the list is selected. Note that calling <i>GetValue</i> (p. 180) returns the new value of selection.
EVT_TEXT(id, func)	Process a wxEVT_COMMAND_TEXT_UPDATED event, when the combobox text changes.
EVT_TEXT_ENTER(id, func)	Process a wxEVT_COMMAND_TEXT_ENTER event, when <RETURN> is pressed in the combobox.

See also

wxListBox (p. 858), *wxTextCtrl* (p. **Error! Bookmark not defined.**), *wxChoice* (p. 145), *wxCommandEvent* (p. 184)

wxComboBox::wxComboBox

wxComboBox()

Default constructor.

```
wxComboBox(wxWindow* parent, wxWindowID id, const wxString& value = "",  
const wxPoint& pos = wxDefaultPosition, const wxSize& size = wxDefaultSize, int n,  
const wxString choices[], long style = 0, const wxValidator& validator =  
wxDefaultValidator, const wxString& name = "comboBox")
```

```
wxComboBox(wxWindow* parent, wxWindowID id, const wxString& value, const  
wxPoint& pos, const wxSize& size, const wxStringArray& choices, long style = 0,  
const wxValidator& validator = wxDefaultValidator, const wxString& name =  
"comboBox")
```

Constructor, creating and showing a combobox.

Parameters

parent

Parent window. Must not be NULL.

id

Window identifier. A value of -1 indicates a default value.

value

Initial selection string. An empty string indicates no selection.

pos

Window position.

size

Window size. If the default size (-1, -1) is specified then the window is sized appropriately.

n

Number of strings with which to initialise the control.

choices

An array of strings with which to initialise the control.

style

Window style. See *wxComboBox* (p. 176).

validator

Window validator.

name

Window name.

See also

wxComboBox::Create (p. 179), *wxValidator* (p. **Error! Bookmark not defined.**)

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

wxPerl note: In wxPerl there is just an array reference in place of *nand choices*.

wxComboBox::~~wxComboBox

~wxComboBox()

Destructor, destroying the combobox.

wxComboBox::Create

bool Create(wxWindow* parent, wxWindowID id, const wxString& value = "", const wxPoint& pos = wxDefaultPosition, const wxSize& size = wxDefaultSize, int n, const wxString choices[], long style = 0, const wxValidator& validator = wxDefaultValidator, const wxString& name = "comboBox")

bool Create(wxWindow* parent, wxWindowID id, const wxString& value, const wxPoint& pos, const wxSize& size, const wxArrayString& choices, long style = 0, const wxValidator& validator = wxDefaultValidator, const wxString& name = "comboBox")

Creates the combobox for two-step construction. Derived classes should call or replace this function. See *wxComboBox::wxComboBox* (p. 177) for further details.

wxComboBox::CanCopy

bool CanCopy() const

Returns true if the combobox is editable and there is a text selection to copy to the clipboard. Only available on Windows.

wxComboBox::CanCut

bool CanCut() const

Returns true if the combobox is editable and there is a text selection to copy to the clipboard. Only available on Windows.

wxComboBox::CanPaste**bool CanPaste() const**

Returns true if the combobox is editable and there is text on the clipboard that can be pasted into the text field. Only available on Windows.

wxComboBox::CanRedo**bool CanRedo() const**

Returns true if the combobox is editable and the last undo can be redone. Only available on Windows.

wxComboBox::CanUndo**bool CanUndo() const**

Returns true if the combobox is editable and the last edit can be undone. Only available on Windows.

wxComboBox::Copy**void Copy()**

Copies the selected text to the clipboard.

wxComboBox::Cut**void Cut()**

Copies the selected text to the clipboard and removes the selection.

wxComboBox::GetInsertionPoint**long GetInsertionPoint() const**

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

Note: Under wxMSW, this function always returns 0 if the combobox doesn't have the focus.

wxComboBox::GetLastPosition**virtual wxTextPos GetLastPosition() const**

Returns the last position in the combobox text field.

wxComboBox::GetValue

wxString GetValue() const

Returns the current value in the combobox text field.

wxComboBox::Paste**void Paste()**

Pastes text from the clipboard to the text field.

wxComboBox::Redo**void Redo()**

Redoes the last undo in the text field. Windows only.

wxComboBox::Replace**void Replace(long *from*, long *to*, const wxString& *text*)**

Replaces the text between two positions with the given text, in the combobox text field.

Parameters

from

The first position.

to

The second position.

text

The text to insert.

wxComboBox::Remove**void Remove(long *from*, long *to*)**

Removes the text between the two positions in the combobox text field.

Parameters

from

The first position.

to

The last position.

wxComboBox::SetInsertionPoint

void SetInsertionPoint(long pos)

Sets the insertion point in the combobox text field.

Parameters

pos

The new insertion point.

wxComboBox::SetInsertionPointEnd

void SetInsertionPointEnd()

Sets the insertion point at the end of the combobox text field.

wxComboBox::SetSelection

void SetSelection(long from, long to)

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

Parameters

from

The first position.

to

The second position.

wxPython note: This method is called `SetMark` in wxPython, `SetSelectionname` is kept for `wxControlWithItems::SetSelection` (p. 225).

wxComboBox::SetValue

void SetValue(const wxString& text)

Sets the text for the combobox text field.

NB: For a combobox with `wxCB_READONLY` style the string must be in the combobox choices list, otherwise the call to `SetValue()` is ignored.

Parameters

text

The text to set.

wxComboBox::Undo

void Undo()

Undoes the last edit in the text field. Windows only.

wxCommand

`wxCommand` is a base class for modelling an application command, which is an action usually performed by selecting a menu item, pressing a toolbar button or any other means provided by the application to change the data or view.

Derived from

`wxObject` (p. **Error! Bookmark not defined.**)

Include files

<wx/cmdproc.h>

See also

Overview (p. **Error! Bookmark not defined.**)

wxCommand::wxCommand

wxCommand(*bool canUndo* = *false*, **const wxString&** *name* = *NULL*)

Constructor. `wxCommand` is an abstract class, so you will need to derive a new class and call this constructor from your own constructor.

canUndo tells the command processor whether this command is undo-able. You can achieve the same functionality by overriding the `CanUndo` member function (if for example the criteria for undoability is context-dependent).

name must be supplied for the command processor to display the command name in the application's edit menu.

wxCommand::~~wxCommand

~wxCommand()

Destructor.

wxCommand::CanUndo

bool `CanUndo`()

Returns true if the command can be undone, false otherwise.

wxCommand::Do

bool `Do`()

Override this member function to execute the appropriate action when called. Return true to indicate that the action has taken place, false otherwise. Returning false will indicate to the command processor that the action is not undoable and should not be added to the command history.

wxCommand::GetName

wxString GetName()

Returns the command name.

wxCommand::Undo

bool Undo()

Override this member function to un-execute a previous Do. Return true to indicate that the action has taken place, false otherwise. Returning false will indicate to the command processor that the action is not redoable and no change should be made to the command history.

How you implement this command is totally application dependent, but typical strategies include:

- Perform an inverse operation on the last modified piece of data in the document. When redone, a copy of data stored in command is pasted back or some operation reapplied. This relies on the fact that you know the ordering of Undos; the user can never Undo at an arbitrary position in the command history.
- Restore the entire document state (perhaps using document transactioning). Potentially very inefficient, but possibly easier to code if the user interface and data are complex, and an 'inverse execute' operation is hard to write.

The docview sample uses the first method, to remove or restore segments in the drawing.

wxCommandEvent

This event class contains information about command events, which originate from a variety of simple controls. More complex controls, such as *wxTreeCtrl* (p. **Error! Bookmark not defined.**), have separate command event classes.

Derived from

wxEvent (p. 487)

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 <code>wxEVT_COMMAND_BUTTON_CLICKED</code> command, which is generated by a <code>wxButton</code> control.
EVT_CHECKBOX(id, func)	Process a <code>wxEVT_COMMAND_CHECKBOX_CLICKED</code> command, which is generated by a <code>wxCheckBox</code> control.
EVT_CHOICE(id, func)	Process a <code>wxEVT_COMMAND_CHOICE_SELECTED</code> command, which is generated by a <code>wxChoice</code> control.
EVT_COMBOBOX(id, func)	Process a <code>wxEVT_COMMAND_COMBOBOX_SELECTED</code> command, which is generated by a <code>wxComboBox</code> control.
EVT_LISTBOX(id, func)	Process a <code>wxEVT_COMMAND_LISTBOX_SELECTED</code> command, which is generated by a <code>wxListBox</code> control.
EVT_LISTBOX_DCLICK(id, func)	Process a <code>wxEVT_COMMAND_LISTBOX_DOUBLECLICKED</code> command, which is generated by a <code>wxListBox</code> control.
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_CONTEXT_MENU(func)	Process the event generated when the user has requested a popup menu to appear by pressing a special keyboard key (under

Windows) or by right clicking the mouse.

EVT_RADIOBOX(id, func)	Process a <code>wxEVT_COMMAND_RADIOBOX_SELECTED</code> command, which is generated by a <code>wxRadioBox</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>wxScrollEvent</i> (p. Error! Bookmark not defined.)).
EVT_SLIDER(id, func)	Process a <code>wxEVT_COMMAND_SLIDER_UPDATED</code> command, which is generated by a <code>wxSlider</code> control.
EVT_TEXT(id, func)	Process a <code>wxEVT_COMMAND_TEXT_UPDATED</code> command, which is generated by a <code>wxTextCtrl</code> control.
EVT_TEXT_ENTER(id, func)	Process a <code>wxEVT_COMMAND_TEXT_ENTER</code> command, which is generated by a <code>wxTextCtrl</code> 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_TEXT_MAXLEN(id, func)	Process a <code>wxEVT_COMMAND_TEXT_MAXLEN</code> command, which is generated by a <code>wxTextCtrl</code> control when the user tries to enter more characters into it than the limit previously set with <i>SetMaxLength</i> (p. Error! Bookmark not defined.).
EVT_TOGGLEBUTTON(id, func)	Process a <code>wxEVT_COMMAND_TOGGLEBUTTON_CLICKED</code> event.
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 wxEVT_COMMAND_TOOL_CLICKED event for a range of identifiers. Pass the ids of the tools.
EVT_TOOL_RCLICKED(id, func)	Process a wxEVT_COMMAND_TOOL_RCLICKED event. Pass the id of the tool.
EVT_TOOL_RCLICKED_RANGE(id1, id2, func)	Process a wxEVT_COMMAND_TOOL_RCLICKED event for a range of ids. Pass the ids of the tools.
EVT_TOOL_ENTER(id, func)	Process a wxEVT_COMMAND_TOOL_ENTER event. Pass the id of the toolbar itself. The value of wxCommandEvent::GetSelection is the tool id, or -1 if the mouse cursor has moved off a tool.
EVT_COMMAND_LEFT_CLICK(id, func)	Process a wxEVT_COMMAND_LEFT_CLICK command, which is generated by a control (Windows 95 and NT only).
EVT_COMMAND_LEFT_DCLICK(id, func)	Process a wxEVT_COMMAND_LEFT_DCLICK command, which is generated by a control (Windows 95 and NT only).
EVT_COMMAND_RIGHT_CLICK(id, func)	Process a wxEVT_COMMAND_RIGHT_CLICK command, which is generated by a control (Windows 95 and NT only).
EVT_COMMAND_SET_FOCUS(id, func)	Process a wxEVT_COMMAND_SET_FOCUS command, which is generated by a control (Windows 95 and NT only).
EVT_COMMAND_KILL_FOCUS(id, func)	Process a wxEVT_COMMAND_KILL_FOCUS command, which is generated by a control (Windows 95 and NT only).
EVT_COMMAND_ENTER(id, func)	Process a wxEVT_COMMAND_ENTER command, which is generated by a control.

wxCommandEvent::wxCommandEvent

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

Constructor.

wxCommandEvent::Checked

bool Checked() const

Deprecated, use *IsChecked* (p. 188) instead.

wxCommandEvent::GetClientData

void* GetClientData()

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

wxCommandEvent::GetClientObject

wxClientData * GetClientObject()

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

wxCommandEvent::GetExtraLong

long GetExtraLong()

Returns extra information dependant on the event objects type. If the event comes from a listbox selection, it is a boolean determining whether the event was a selection (true) or a deselection (false). A listbox deselection only occurs for multiple-selection boxes, and in this case the index and string values are indeterminate and the listbox must be examined by the application.

wxCommandEvent::GetInt

int GetInt()

Returns the integer identifier corresponding to a listbox, choice or radiobox selection (only if the event was a selection, not a deselection), or a boolean value representing the value of a checkbox.

wxCommandEvent::GetSelection

int GetSelection()

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

wxCommandEvent::GetString

wxString GetString()

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

wxCommandEvent::IsChecked

bool IsChecked() const

This method can be used with checkbox and menu events: for the checkboxes, the method returns `true` for a selection event and `false` for a deselection one. For the menu events, this method indicates if the menu item just has become checked or unchecked (and thus only makes sense for checkable menu items).

wxCommandEvent::IsSelection

bool IsSelection()

For a listbox or similar event, returns true if it is a selection, false if it is a deselection.

wxCommandEvent::SetClientData

void SetClientData(void* clientData)

Sets the client data for this event.

wxCommandEvent::SetClientObject

void SetClientObject(wxClientData* clientObject)

Sets the client object for this event. The client object is *not* owned by the event object and the event object will not delete the client object in its destructor. The client object must be owned and deleted by another object (e.g. a control) that has longer life time than the event object.

wxCommandEvent::SetExtraLong

void SetExtraLong(int extraLong)

Sets the `m_extraLong` member.

wxCommandEvent::SetInt

void SetInt(int intCommand)

Sets the `m_commandInt` member.

wxCommandEvent::SetString

void SetString(const wxString& string)

Sets the `m_commandString` member.

wxCommandProcessor

wxCommandProcessor is a class that maintains a history of wxCommands, with undo/redo functionality built-in. Derive a new class from this if you want different behaviour.

Derived from

wxObject (p. **Error! Bookmark not defined.**)

Include files

<wx/cmdproc.h>

See also

wxCommandProcessor overview (p. **Error! Bookmark not defined.**), *wxCommand* (p. 182)

wxCommandProcessor::wxCommandProcessor

wxCommandProcessor(int *maxCommands* = -1)

Constructor.

maxCommands may be set to a positive integer to limit the number of commands stored to it, otherwise (and by default) the list of commands can grow arbitrarily.

wxCommandProcessor::~~wxCommandProcessor

~wxCommandProcessor()

Destructor.

wxCommandProcessor::CanUndo

virtual bool CanUndo()

Returns true if the currently-active command can be undone, false otherwise.

wxCommandProcessor::ClearCommands

virtual void ClearCommands()

Deletes all commands in the list and sets the current command pointer to `NULL`.

wxCommandProcessor::Redo

virtual bool Redo()

Executes (redoes) the current command (the command that has just been undone if any).

wxCommandProcessor::GetCommands**wxList& GetCommands() const**

Returns the list of commands.

wxCommandProcessor::GetMaxCommands**int GetMaxCommands() const**

Returns the maximum number of commands that the command processor stores.

wxCommandProcessor::GetEditMenu**wxMenu* GetEditMenu() const**

Returns the edit menu associated with the command processor.

wxCommandProcessor::GetRedoAccelerator**const wxString& GetRedoAccelerator() const**

Returns the string that will be appended to the Redo menu item.

wxCommandProcessor::GetRedoMenuLabel**wxString GetRedoMenuLabel() const**

Returns the string that will be shown for the redo menu item.

wxCommandProcessor::GetUndoAccelerator**const wxString& GetUndoAccelerator() const**

Returns the string that will be appended to the Undo menu item.

wxCommandProcessor::GetUndoMenuLabel**wxString GetUndoMenuLabel() const**

Returns the string that will be shown for the undo menu item.

wxCommandProcessor::Initialize**virtual void Initialize()**

Initializes the command processor, setting the current command to the last in the list (if

any), and updating the edit menu (if one has been specified).

wxCommandProcessor::IsDirty

virtual bool IsDirty()

Returns a boolean value that indicates if changes have been made since the last save operation. This only works if *wxCommandProcessor::MarkAsSaved* (p. 192) is called whenever the project is saved.

wxCommandProcessor::MarkAsSaved

virtual void MarkAsSaved()

You must call this method whenever the project is saved if you plan to use *wxCommandProcessor::IsDirty* (p. 191).

wxCommandProcessor::SetEditMenu

void SetEditMenu(wxMenu* menu)

Tells the command processor to update the Undo and Redo items on this menu as appropriate. Set this to NULL if the menu is about to be destroyed and command operations may still be performed, or the command processor may try to access an invalid pointer.

wxCommandProcessor::SetMenuStrings

void SetMenuStrings()

Sets the menu labels according to the currently set menu and the current command state.

wxCommandProcessor::SetRedoAccelerator

void SetRedoAccelerator(const wxString&accel)

Sets the string that will be appended to the Redo menu item.

wxCommandProcessor::SetUndoAccelerator

void SetUndoAccelerator(const wxString&accel)

Sets the string that will be appended to the Undo menu item.

wxCommandProcessor::Submit

virtual bool Submit(wxCommand *command, bool storeIt = true)

Submits a new command to the command processor. The command processor calls

`wxCommand::Do` to execute the command; if it succeeds, the command is stored in the history list, and the associated edit menu (if any) updated appropriately. If it fails, the command is deleted immediately. Once `Submit` has been called, the passed command should not be deleted directly by the application.

`storeIt` indicates whether the successful command should be stored in the history list.

wxCommandProcessor::Undo

virtual bool Undo()

Undoes the command just executed.

wxCondition

`wxCondition` variables correspond to `pthread` conditions or to Win32 event objects. They may be used in a multithreaded application to wait until the given condition becomes true which happens when the condition becomes signaled.

For example, if a worker thread is doing some long task and another thread has to wait until it is finished, the latter thread will wait on the condition object and the worker thread will signal it on exit (this example is not perfect because in this particular case it would be much better to just `Wait()` (p. **Error! Bookmark not defined.**) for the worker thread, but if there are several worker threads it already makes much more sense).

Note that a call to `Signal()` (p. 195) may happen before the other thread calls `Wait()` (p. 196) and, just as with the `pthread` conditions, the signal is then lost and so if you want to be sure that you don't miss it you must keep the mutex associated with the condition initially locked and lock it again before calling `Signal()` (p. 195). Of course, this means that this call is going to block until `Wait()` (p. 196) is called by another thread.

Example

This example shows how a main thread may launch a worker thread which starts running and then waits until the main thread signals it to continue:

```
class MySignallingThread : public wxThread
{
public:
    MySignallingThread(wxMutex *mutex, wxCondition *condition)
    {
        m_mutex = mutex;
        m_condition = condition;

        Create();
    }

    virtual ExitCode Entry()
    {
        ... do our job ...

        // tell the other(s) thread(s) that we're about to
        terminate: we must
        // lock the mutex first or we might signal the condition
        before the
```

```
        // waiting threads start waiting on it!
        wxMutexLocker lock(m_mutex);
        m_condition.Broadcast(); // same as Signal() here -- one
waiter only

        return 0;
    }

private:
    wxCondition *m_condition;
    wxMutex *m_mutex;
};

int main()
{
    wxMutex mutex;
    wxCondition condition(mutex);

    // the mutex should be initially locked
    mutex.Lock();

    // create and run the thread but notice that it won't be able
to    // exit (and signal its exit) before we unlock the mutex below
    MySignallingThread *thread = new MySignallingThread(&mutex,
&condition);

    thread->Run();

    // wait for the thread termination: Wait() atomically unlocks
the mutex    // which allows the thread to continue and starts waiting
    condition.Wait();

    // now we can exit
    return 0;
}
```

Of course, here it would be much better to simply use a joinable thread and call `wxThread::Wait` (p. **Error! Bookmark not defined.**) on it, but this example does illustrate the importance of properly locking the mutex when using `wxCondition`.

Constants

The following return codes are returned by `wxCondition` member functions:

```
enum wxCondError
{
    wxCOND_NO_ERROR = 0,        // successful completion
    wxCOND_INVALID,            // object hasn't been initialized
successfully
    wxCOND_TIMEOUT,            // WaitTimeout() has timed out
    wxCOND_MISC_ERROR          // some other error
};
```

Derived from

None.

Include files

<wx/thread.h>

See also

wxThread (p. **Error! Bookmark not defined.**), *wxMutex* (p. **Error! Bookmark not defined.**)

wxCondition::wxCondition

wxCondition(wxMutex& mutex)

Default and only constructor. The *mutex* must be locked by the caller before calling *Wait* (p. 196) function.

Use *IsOk* (p. 195) to check if the object was successfully initialized.

wxCondition::~~wxCondition

~wxCondition()

Destroys the *wxCondition* object. The destructor is not virtual so this class should not be used polymorphically.

wxCondition::Broadcast

void Broadcast()

Broadcasts to all waiting threads, waking all of them up. Note that this method may be called whether the mutex associated with this condition is locked or not.

See also

wxCondition::Signal (p. 195)

wxCondition::IsOk

bool IsOk() const

Returns *true* if the object had been initialized successfully, *false* if an error occurred.

wxCondition::Signal

void Signal()

Signals the object waking up at most one thread. If several threads are waiting on the same condition, the exact thread which is woken up is undefined. If no threads are waiting, the signal is lost and the condition would have to be signalled again to wake up any thread which may start waiting on it later.

Note that this method may be called whether the mutex associated with this condition is

locked or not.

See also

wxCondition::Broadcast (p. 195)

wxCondition::Wait**wxCondError Wait()**

Waits until the condition is signalled.

This method atomically releases the lock on the mutex associated with this condition (this is why it must be locked prior to calling *Wait*) and puts the thread to sleep until *Signal* (p. 195) or *Broadcast* (p. 195) is called.

Note that even if *Signal* (p. 195) had been called before *Wait* without waking up any thread, the thread would still wait for another one and so it is important to ensure that the condition will be signalled after *Wait* or the thread may sleep forever.

Return value

Returns `wxCOND_NO_ERROR` on success, another value if an error occurred.

See also

WaitTimeout (p. 196)

wxCondition::WaitTimeout**wxCondError WaitTimeout(unsigned long milliseconds)**

Waits until the condition is signalled or the timeout has elapsed.

This method is identical to *Wait* (p. 196) except that it returns, with the return code of `wxCOND_TIMEOUT` as soon as the given timeout expires.

Parameters

milliseconds

Timeout in milliseconds

Return value

Returns `wxCOND_NO_ERROR` if the condition was signalled, `wxCOND_TIMEOUT` if the timeout elapsed before this happened or another error code from `wxCondError` enum.

wxConfigBase

`wxConfigBase` class defines the basic interface of all config classes. It can not be used by itself (it is an abstract base class) and you will always use one of its derivations:

wxFileConfig (p. 513), *wxRegConfig* or any other.

However, usually you don't even need to know the precise nature of the class you're working with but you would just use the *wxConfigBase* methods. This allows you to write the same code regardless of whether you're working with the registry under Win32 or text-based config files under Unix (or even Windows 3.1 .INI files if you're really unlucky). To make writing the portable code even easier, *wxWidgets* provides a typedef *wxConfig* which is mapped onto the native *wxConfigBase* implementation on the given platform: i.e. *wxRegConfig* under Win32 and *wxFileConfig* otherwise.

See *config overview* (p. **Error! Bookmark not defined.**) 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. 198)

Derived from

No base class

Include files

<wx/config.h> (to let *wxWidgets* choose a *wxConfig* class for your platform)
<wx/confbase.h> (base config class)
<wx/fileconf.h> (*wxFileConfig* class)
<wx/msw/regconf.h> (*wxRegConfig* class)

Example

Here is how you would typically use this class:

```
// using wxConfig instead of writing wxFileConfig or wxRegConfig
enhances
// portability of the code
wxConfig *config = new wxConfig("MyAppName");

wxString str;
if ( config->Read("LastPrompt", &str) ) {
    // last prompt was found in the config file/registry and its
    value is now
    // in str
    ...
}
else {
    // no last prompt...
}

// another example: using default values and the full path
instead of just
// key name: if the key is not found , the value 17 is returned
long value = config->Read("/LastRun/CalculatedValues/MaxValue",
17);
...
...
...
// at the end of the program we would save everything back
config->Write("LastPrompt", str);
config->Write("/LastRun/CalculatedValues/MaxValue", value);
```

```
// the changes will be written back automatically  
delete config;
```

This basic example, of course, doesn't show all `wxConfig` features, such as enumerating, testing for existence and deleting the entries and groups of entries in the config file, its abilities to automatically store the default values or expand the environment variables on the fly. However, the main idea is that using this class is easy and that it should normally do what you expect it to.

NB: in the documentation of this class, the words "config file" also mean "registry hive" for `wxRegConfig` and, generally speaking, might mean any physical storage where a `wxConfigBase`-derived class stores its data.

Static functions

These functions deal with the "default" config object. Although its usage is not at all mandatory it may be convenient to use a global config object instead of creating and deleting the local config objects each time you need one (especially because creating a `wxFileConfig` object might be a time consuming operation). In this case, you may create this global config object in the very start of the program and `Set()` it as the default. Then, from anywhere in your program, you may access it using the `Get()` function. Note that you must delete this object (usually in `wxApp::OnExit` (p. 42)) in order to avoid memory leaks, `wxWidgets` won't do it automatically.

As it happens, you may even further simplify the procedure described above: you may forget about calling `Set()`. When `Get()` is called and there is no current object, it will create one using `Create()` function. To disable this behaviour `DontCreateOnDemand()` is provided.

Note: You should use either `Set()` or `Get()` because `wxWidgets` library itself would take advantage of it and could save various information in it. For example `wxFontMapper` (p. 578) or Unix version of `wxFileDialog` (p. 515) have the ability to use `wxConfig` class.

`Set` (p. 209)

`Get` (p. 205)

`Create` (p. 204)

`DontCreateOnDemand` (p. 204)

Constructor and destructor

`wxConfigBase` (p. 202)

`~wxConfigBase` (p. 203)

Path management

As explained in *config overview* (p. **Error! Bookmark not defined.**), 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. 209)

GetPath (p. 206)

Enumeration

The functions in this section allow to enumerate all entries and groups in the config file. All functions here return `false` when there are no more items.

You must pass the same index to `GetNext` and `GetFirst` (don't modify it). Please note that it is **not** the index of the current item (you will have some great surprises with `wxRegConfig` if you assume this) and you shouldn't even look at it: it is just a "cookie" which stores the state of the enumeration. It can't be stored inside the class because it would prevent you from running several enumerations simultaneously, that's why you must pass it explicitly.

Having said all this, enumerating the config entries/groups is very simple:

```
wxConfigBase *config = ...;
wxArrayString aNames;

// enumeration variables
wxString str;
long dummy;

// first enum all entries
bool bCont = config->GetFirstEntry(str, dummy);
while ( bCont ) {
    aNames.Add(str);

    bCont = GetConfig()->GetNextEntry(str, dummy);
}

... we have all entry names in aNames...

// now all groups...
bCont = GetConfig()->GetFirstGroup(str, dummy);
while ( bCont ) {
    aNames.Add(str);

    bCont = GetConfig()->GetNextGroup(str, dummy);
}

... we have all group (and entry) names in aNames...
```

There are also functions to get the number of entries/subgroups without actually enumerating them, but you will probably never need them.

GetFirstGroup (p. 205)
GetNextGroup (p. 206)
GetFirstEntry (p. 205)
GetNextEntry (p. 206)
GetNumberOfEntries (p. 206)
GetNumberOfGroups (p. 206)

Tests of existence

HasGroup (p. 207)
HasEntry (p. 206)
Exists (p. 204)
GetEntryType (p. 205)

Miscellaneous functions

GetAppName (p. 205)
GetVendorName (p. 206)
SetUmask (p. 513)

Key access

These functions are the core of `wxConfigBase` class: they allow you to read and write config file data. All *Read* functions 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. 207)
Write (p. 209)
Flush (p. 204)

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. 208)
RenameGroup (p. 209)

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. 204)
DeleteGroup (p. 204)
DeleteAll (p. 204)

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. 207)

SetExpandEnvVars (p. 209)

SetRecordDefaults (p. 209)

IsRecordingDefaults (p. 207)

`wxConfigBase::wxConfigBase`

`wxConfigBase(const wxString& appName = wxEmptyString, const wxString& vendorName = wxEmptyString, const wxString& localFilename = wxEmptyString, const wxString& globalFilename = wxEmptyString, long style = 0, wxMBConv& conv = wxConvUTF8)`

This is the default and only constructor of the `wxConfigBase` class, and derived classes.

Parameters

appName

The application name. If this is empty, the class will normally use `wxApp::GetAppName` (p. 38) to set it. The application name is used in the registry key on Windows, and can be used to deduce the local filename parameter if that is missing.

vendorName

The vendor name. If this is empty, it is assumed that no vendor name is wanted, if this is optional for the current config class. The vendor name is appended to the application name for `wxRegConfig`.

localFilename

Some config classes require a local filename. If this is not present, but required, the application name will be used instead.

globalFilename

Some config classes require a global filename. If this is not present, but required, the application name will be used instead.

style

Can be one of `wxCONFIG_USE_LOCAL_FILE` and `wxCONFIG_USE_GLOBAL_FILE`. The style interpretation depends on the config class and is ignored by some. For `wxFileConfig`, these styles determine whether a local or global config file is created or used. If the flag is present but the parameter is empty, the parameter will be set to a default. If the parameter is present but the style flag not, the relevant flag will be added to the style. For `wxFileConfig` you can also add `wxCONFIG_USE_RELATIVE_PATH` by logically or'ing it to either of the `_FILE` options to tell `wxFileConfig` to use relative instead of absolute paths. For `wxFileConfig`, you can also add `wxCONFIG_USE_NO_ESCAPE_CHARACTERS` which will turn off character escaping for the values of entries stored in the config file: for example a `foo` key with some backslash characters will be stored as `foo=C:\mydir` instead of the usual storage of `foo=C:\\mydir`. For `wxRegConfig`, this flag refers to HKLM, and provides read-only access.

The `wxCONFIG_USE_NO_ESCAPE_CHARACTERS` style can be helpful if your config file must be read or written to by a non-`wxWidgets` program (which might not understand the escape characters). Note, however, that if `wxCONFIG_USE_NO_ESCAPE_CHARACTERS` style is used, it is now your application's responsibility to ensure that there is no newline or other illegal characters in a value, before writing that value to the file.

conv

This parameter is only used by `wxFileConfig` when compiled in Unicode mode. It specifies the encoding in which the configuration file is written.

Remarks

By default, environment variable expansion is on and recording defaults is off.

wxConfigBase::~wxConfigBase**~wxConfigBase()**

Empty but ensures that dtor of all derived classes is virtual.

wxConfigBase::Create**static wxConfigBase * Create()**

Create a new config object: this function will create the "best" implementation of

`wxConfig` available for the current platform, see comments near the definition of `wxCONFIG_WIN32_NATIVE` for details. It returns the created object and also sets it as the current one.

`wxConfigBase::DontCreateOnDemand`

`void DontCreateOnDemand()`

Calling this function will prevent `Get()` from automatically creating a new config object if the current one is `NULL`. It might be useful to call it near the program end to prevent "accidental" creation of a new config object.

`wxConfigBase::DeleteAll`

`bool DeleteAll()`

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

`wxConfigBase::DeleteEntry`

`bool DeleteEntry(const wxString& key, bool bDeleteGroupIfEmpty = true)`

Deletes the specified entry and the group it belongs to if it was the last key in it and the second parameter is true.

`wxConfigBase::DeleteGroup`

`bool DeleteGroup(const wxString& key)`

Delete the group (with all subgroups)

`wxConfigBase::Exists`

`bool Exists(wxString& strName) const`

returns `true` if either a group or an entry with a given name exists

`wxConfigBase::Flush`

`bool Flush(bool bCurrentOnly = false)`

permanently writes all changes (otherwise, they're only written from object's destructor)

`wxConfigBase::Get`

`static wxConfigBase * Get(bool CreateOnDemand = true)`

Get the current config object. If there is no current object and `CreateOnDemand` is true, creates one (using `Create`) unless `DontCreateOnDemand` was called previously.

wxConfigBase::GetAppName**wxString GetAppName() const**

Returns the application name.

wxConfigBase::GetEntryType**enum wxConfigBase::EntryType GetEntryType(const wxString& name) const**

Returns the type of the given entry or *Unknown* if the entry doesn't exist. This function should be used to decide which version of `Read()` should be used because some of `wxConfig` implementations will complain about type mismatch otherwise: e.g., an attempt to read a string value from an integer key with `wxRegConfig` will fail.

The result is an element of enum `EntryType`:

```
enum EntryType
{
    Type_Unknown,
    Type_String,
    Type_Boolean,
    Type_Integer,
    Type_Float
};
```

wxConfigBase::GetFirstGroup**bool GetFirstGroup(wxString& str, long& index) const**

Gets the first group.

wxPython note: The `wxPython` version of this method returns a 3-tuple consisting of the continue flag, the value string, and the index for the next call.

wxPerl note: In `wxPerl` this method takes no arguments and returns a 3-element list (`continue, str, index`).

wxConfigBase::GetFirstEntry**bool GetFirstEntry(wxString& str, long& index) const**

Gets the first entry.

wxPython note: The `wxPython` version of this method returns a 3-tuple consisting of the continue flag, the value string, and the index for the next call.

wxPerl note: In `wxPerl` this method takes no arguments and returns a 3-element list (`continue, str, index`).

wxConfigBase::GetNextGroup**bool GetNextGroup(wxString& str, long& index) const**

Gets the next group.

wxPython note: The wxPython version of this method returns a 3-tuple consisting of the continue flag, the value string, and the index for the next call.

wxPerl note: In wxPerl this method only takes the `index` parameter and returns a 3-element list (`continue`, `str`, `index`).

wxConfigBase::GetNextEntry

bool GetNextEntry(wxString& str, long& index) const

Gets the next entry.

wxPython note: The wxPython version of this method returns a 3-tuple consisting of the continue flag, the value string, and the index for the next call.

wxPerl note: In wxPerl this method only takes the `index` parameter and returns a 3-element list (`continue`, `str`, `index`).

wxConfigBase::GetNumberOfEntries

uint GetNumberOfEntries(bool bRecursive = false) const

wxConfigBase::GetNumberOfGroups

uint GetNumberOfGroups(bool bRecursive = false) const

Get number of entries/subgroups in the current group, with or without its subgroups.

wxConfigBase::GetPath

const wxString& GetPath() const

Retrieve the current path (always as absolute path).

wxConfigBase::GetVendorName

wxString GetVendorName() const

Returns the vendor name.

wxConfigBase::HasEntry

bool HasEntry(wxString& strName) const

returns `true` if the entry by this name exists

wxConfigBase::HasGroup

bool HasGroup(const wxString& strName) const

returns `true` if the group by this name exists

wxConfigBase::IsExpandingEnvVars

bool IsExpandingEnvVars() const

Returns `true` if we are expanding environment variables in key values.

wxConfigBase::IsRecordingDefaults

bool IsRecordingDefaults() const

Returns `true` if we are writing defaults back to the config file.

wxConfigBase::Read

bool Read(const wxString& key, wxString* str) const

Read a string from the key, returning `true` if the value was read. If the key was not found, *str* is not changed.

bool Read(const wxString& key, wxString* str, const wxString& defaultVal) const

Read a string from the key. The default value is returned if the key was not found.

Returns `true` if value was really read, `false` if the default was used.

wxString Read(const wxString& key, const wxString& defaultVal) const

Another version of *Read()*, returning the string value directly.

bool Read(const wxString& key, long* l) const

Reads a long value, returning `true` if the value was found. If the value was not found, *l* is not changed.

bool Read(const wxString& key, long* l, long defaultVal) const

Reads a long value, returning `true` if the value was found. If the value was not found, *defaultVal* is used instead.

long Read(const wxString& key, long defaultVal) const

Reads a long value from the key and returns it. *defaultVal* is returned if the key is not found.

NB: writing

```
conf->Read("key", 0);
```

won't work because the call is ambiguous: compiler can not choose between two *Read* functions. Instead, write:

```
conf->Read("key", 01);
```

bool Read(const wxString& key, double* d) const

Reads a double value, returning `true` if the value was found. If the value was not found, *d* is not changed.

bool Read(const wxString& key, double* d, double defaultVal) const

Reads a double value, returning `true` if the value was found. If the value was not found, *defaultVal* is used instead.

bool Read(const wxString& key, bool* b) const

Reads a bool value, returning `true` if the value was found. If the value was not found, *b* is not changed.

bool Read(const wxString& key, bool* d, bool defaultVal) const

Reads a bool value, returning `true` if the value was found. If the value was not found, *defaultVal* is used instead.

wxPython note: In place of a single overloaded method name, wxPython implements the following methods:

Read(key, default="")	Returns a string.
ReadInt(key, default=0)	Returns an int.
ReadFloat(key, default=0.0)	Returns a floating point number.

wxPerl note: In place of a single overloaded method, wxPerl uses:

Read(key, default="")	Returns a string
ReadInt(key, default=0)	Returns an integer
ReadFloat(key, default=0.0)	Returns a floating point number
ReadBool(key, default=0)	Returns a boolean

wxConfigBase::RenameEntry

bool RenameEntry(const wxString& oldName, const wxString& newName)

Renames an entry in the current group. The entries names (both the old and the new one) shouldn't contain backslashes, i.e. only simple names and not arbitrary paths are accepted by this function.

Returns `false` if *oldName* doesn't exist or if *newName* already exists.

wxConfigBase::RenameGroup

bool RenameGroup(const wxString& oldName, const wxString& newName)

Renames a subgroup of the current group. The subgroup names (both the old and the new one) shouldn't contain backslashes, i.e. only simple names and not arbitrary paths are accepted by this function.

Returns `false` if *oldName* doesn't exist or if *newName* already exists.

wxConfigBase::Set

static wxConfigBase * Set(wxConfigBase *pConfig)

Sets the config object as the current one, returns the pointer to the previous current object (both the parameter and returned value may be NULL)

wxConfigBase::SetExpandEnvVars

void SetExpandEnvVars (bool bDolt = true)

Determine whether we wish to expand environment variables in key values.

wxConfigBase::SetPath

void SetPath(const wxString& strPath)

Set current path: if the first character is '/', it is the absolute path, otherwise it is a relative path. '.' is supported. If strPath doesn't exist it is created.

wxConfigBase::SetRecordDefaults

void SetRecordDefaults(bool bDolt = true)

Sets whether defaults are recorded to the config file whenever an attempt to read the value which is not present in it is done.

If on (default is off) all default values for the settings used by the program are written back to the config file. This allows the user to see what config options may be changed and is probably useful only for wxFileConfig.

wxConfigBase::Write

bool Write(const wxString& key, const wxString& value)

bool Write(const wxString& key, long value)

bool Write(const wxString& key, double value)

bool Write(const wxString& key, bool value)

These functions write the specified value to the config file and return `true` on success.

wxPython note: In place of a single overloaded method name, wxPython implements the following methods:

Write(key, value)	Writes a string.
WriteInt(key, value)	Writes an int.
WriteFloat(key, value)	Writes a floating point number.

wxPerl note: In place of a single overloaded method, wxPerl uses:

Write(key, value)	Writes a string
WriteInt(key, value)	Writes an integer
WriteFloat(key, value)	Writes a floating point number
WriteBool(key, value)	Writes a boolean

wxConnection

A `wxConnection` object represents the connection between a client and a server. It is created by making a connection using a `wxClient` (p. 150) object, or by the acceptance of a connection by a `wxServer` (p. **Error! Bookmark not defined.**) object. The bulk of a DDE-like (Dynamic Data Exchange) conversation is controlled by calling members in a **wxConnection** object or by overriding its members. The actual DDE-based implementation using `wxDDEConnection` is available on Windows only, but a platform-independent, socket-based version of this API is available using `wxTCPConnection`, which has the same API.

An application should normally derive a new connection class from `wxConnection`, in order to override the communication event handlers to do something interesting.

Derived from

`wxConnectionBase`
`wxObject` (p. **Error! Bookmark not defined.**)

Include files

`<wx/ipc.h>`

Types

`wxIPCFormat` is defined as follows:

```
enum wxIPCFormat
{
```



```

wxIPC_INVALID = 0,
wxIPC_TEXT = 1, /* CF_TEXT */
wxIPC_BITMAP = 2, /* CF_BITMAP */
wxIPC_METAFILE = 3, /* CF_METAFILEPICT */
wxIPC_SYLK = 4,
wxIPC_DIF = 5,
wxIPC_TIFF = 6,
wxIPC_OEMTEXT = 7, /* CF_OEMTEXT */
wxIPC_DIB = 8, /* CF_DIB */
wxIPC_PALETTE = 9,
wxIPC_PENDATA = 10,
wxIPC_RIFF = 11,
wxIPC_WAVE = 12,
wxIPC_UNICODETEXT = 13,
wxIPC_ENHMETAFILE = 14,
wxIPC_FILENAME = 15, /* CF_HDROP */
wxIPC_LOCALE = 16,
wxIPC_PRIVATE = 20
};

```

See also

wxClient (p. 150), *wxServer* (p. **Error! Bookmark not defined.**), *Interprocess communications overview* (p. **Error! Bookmark not defined.**)

wxConnection::wxConnection

wxConnection()

wxConnection(char* buffer, int size)

Constructs a connection object. If no user-defined connection object is to be derived from `wxConnection`, then the constructor should not be called directly, since the default connection object will be provided on requesting (or accepting) a connection. However, if the user defines his or her own derived connection object, the *wxServer::OnAcceptConnection* (p. **Error! Bookmark not defined.**) and/or *wxClient::OnMakeConnection* (p. 151) members should be replaced by functions which construct the new connection object.

If the arguments of the `wxConnection` constructor are void then the `wxConnection` object manages its own connection buffer, allocating memory as needed. A programmer-supplied buffer cannot be increased if necessary, and the program will assert if it is not large enough. The programmer-supplied buffer is included mainly for backwards compatibility.

wxConnection::Advise

bool Advise(const wxString& item, char* data, int size = -1, wxIPCFormat format = wxCF_TEXT)

Called by the server application to advise the client of a change in the data associated with the given item. Causes the client connection's *wxConnection::OnAdvise* (p. 212) member to be called. Returns true if successful.

wxConnection::Execute**bool Execute(char* data, int size = -1, wxIPCFormat format = wxCF_TEXT)**

Called by the client application to execute a command on the server. Can also be used to transfer arbitrary data to the server (similar to *wxConnection::Poke* (p. 213) in that respect). Causes the server connection's *wxConnection::OnExecute* (p. 212) member to be called. Returns true if successful.

wxConnection::Disconnect**bool Disconnect()**

Called by the client or server application to disconnect from the other program; it causes the *wxConnection::OnDisconnect* (p. 212) message to be sent to the corresponding connection object in the other program. Returns true if successful or already disconnected. The application that calls **Disconnect** must explicitly delete its side of the connection.

wxConnection::OnAdvise**virtual bool OnAdvise(const wxString& topic, const wxString& item, char* data, int size, wxIPCFormat format)**

Message sent to the client application when the server notifies it of a change in the data associated with the given item, using *Advise* (p. 211).

wxConnection::OnDisconnect**virtual bool OnDisconnect()**

Message sent to the client or server application when the other application notifies it to end the connection. The default behaviour is to delete the connection object and return true, so applications should generally override **OnDisconnect** (finally calling the inherited method as well) so that they know the connection object is no longer available.

wxConnection::OnExecute**virtual bool OnExecute(const wxString& topic, char* data, int size, wxIPCFormat format)**

Message sent to the server application when the client notifies it to execute the given data, using *Execute* (p. 212). Note that there is no item associated with this message.

wxConnection::OnPoke**virtual bool OnPoke(const wxString& topic, const wxString& item, char* data, int size, wxIPCFormat format)**

Message sent to the server application when the client notifies it to accept the given

data.

wxConnection::OnRequest

virtual char* OnRequest(const wxString& topic, const wxString& item, int *size, wxIPCFormat format)

Message sent to the server application when the client calls *wxConnection::Request* (p. 214). The server's *OnRequest* (p. 213) method should respond by returning a character string, or NULL to indicate no data, and setting *size. The character string must of course persist after the call returns.

wxConnection::OnStartAdvise

virtual bool OnStartAdvise(const wxString& topic, const wxString& item)

Message sent to the server application by the client, when the client wishes to start an 'advise loop' for the given topic and item. The server can refuse to participate by returning false.

wxConnection::OnStopAdvise

virtual bool OnStopAdvise(const wxString& topic, const wxString& item)

Message sent to the server application by the client, when the client wishes to stop an 'advise loop' for the given topic and item. The server can refuse to stop the advise loop by returning false, although this doesn't have much meaning in practice.

wxConnection::Poke

bool Poke(const wxString& item, char* data, int size = -1, wxIPCFormat format = wxCF_TEXT)

Called by the client application to poke data into the server. Can be used to transfer arbitrary data to the server. Causes the server connection's *wxConnection::OnPoke* (p. 213) member to be called. If size is -1 the size is computed from the string length of data.

Returns true if successful.

wxConnection::Request

char* Request(const wxString& item, int *size, wxIPCFormat format = wxIPC_TEXT)

Called by the client application to request data from the server. Causes the server connection's *wxConnection::OnRequest* (p. 213) member to be called. Size may be NULL or a pointer to a variable to receive the size of the requested item.

Returns a character string (actually a pointer to the connection's buffer) if successful, NULL otherwise. This buffer does not need to be deleted.

wxConnection::StartAdvise**bool StartAdvise(const wxString& item)**

Called by the client application to ask if an advise loop can be started with the server. Causes the server connection's *wxConnection::OnStartAdvise* (p. 213) member to be called. Returns true if the server okays it, false otherwise.

wxConnection::StopAdvise**bool StopAdvise(const wxString& item)**

Called by the client application to ask if an advise loop can be stopped. Causes the server connection's *wxConnection::OnStopAdvise* (p. 213) member to be called. Returns true if the server okays it, false otherwise.

wxContextMenuEvent

This class is used for context menu events, sent to give the application a chance to show a context (popup) menu.

Derived from*wxCommandEvent* (p. 184)*wxEvent* (p. 487)*wxObject* (p. **Error! Bookmark not defined.**)**Include files**

<wx/event.h>

Event table macros

To process a menu event, use these event handler macros to direct input to member functions that take a *wxContextMenuEvent* argument.

EVT_CONTEXT_MENU(func) A right click (or other context menu command depending on platform) has been detected.

See also*Command events* (p. 184),*Event handling overview* (p. **Error! Bookmark not defined.**)**wxContextMenuEvent::wxContextMenuEvent****wxContextMenuEvent(WXTYPE id = 0, int id = 0, const wxPoint& pos=wxDefaultPosition)**

Constructor.

wxContextMenuEvent::GetPosition**wxPoint GetPosition() const**

Returns the position at which the menu should be shown.

wxContextMenuEvent::SetPosition**void SetPosition(const wxPoint& point)**

Sets the position at which the menu should be shown.

wxContextHelp

This class changes the cursor to a query and puts the application into a 'context-sensitive help mode'. When the user left-clicks on a window within the specified window, a `wxEVT_HELP` event is sent to that control, and the application may respond to it by popping up some help.

For example:

```
wxContextHelp contextHelp(myWindow);
```

There are a couple of ways to invoke this behaviour implicitly:

- Use the `wxDIALOG_EX_CONTEXTHELP` style for a dialog (Windows only). This will put a question mark in the titlebar, and Windows will put the application into context-sensitive help mode automatically, with further programming.
- Create a `wxContextHelpButton` (p. 216), whose predefined behaviour is to create a context help object. Normally you will write your application so that this button is only added to a dialog for non-Windows platforms (use `wxDIALOG_EX_CONTEXTHELP` on Windows).

Note that on Mac OS X, the cursor does not change when in context-sensitive help mode.

Derived from

`wxObject` (p. **Error! Bookmark not defined.**)

Include files

<wx/cshelp.h>

See also

`wxHelpEvent` (p. 701), `wxHelpController` (p. 694), `wxContextHelpButton` (p. 216)

wxContextHelp::wxContextHelp

wxContextHelp(wxWindow* *window* = NULL, bool *doNow* = true)

Constructs a context help object, calling *BeginContextHelp* (p. 216) if *doNow* is true (the default).

If *window* is NULL, the top window is used.

wxContextHelp::~~wxContextHelp

~wxContextHelp()

Destroys the context help object.

wxContextHelp::BeginContextHelp

bool BeginContextHelp(wxWindow* *window* = NULL)

Puts the application into context-sensitive help mode. *window* is the window which will be used to catch events; if NULL, the top window will be used.

Returns true if the application was successfully put into context-sensitive help mode. This function only returns when the event loop has finished.

wxContextHelp::EndContextHelp

bool EndContextHelp()

Ends context-sensitive help mode. Not normally called by the application.

wxContextHelpButton

Instances of this class may be used to add a question mark button that when pressed, puts the application into context-help mode. It does this by creating a *wxContextHelp* (p. 215) object which itself generates a wxEVT_HELP event when the user clicks on a window.

On Windows, you may add a question-mark icon to a dialog by use of the wxDIALOG_EX_CONTEXTHELP extra style, but on other platforms you will have to add a button explicitly, usually next to OK, Cancel or similar buttons.

Derived from

wxBitmapButton (p. 96)

wxButton (p. 122)

wxControl (p. 218)

wxWindow (p. **Error! Bookmark not defined.**)

wxEvtHandler (p. 490)

wxObject (p. **Error! Bookmark not defined.**)

Include files

<wx/cshelp.h>

See also

wxBitmapButton (p. 96), *wxContextHelp* (p. 215)

wxContextHelpButton::wxContextHelpButton

wxContextHelpButton()

Default constructor.

wxContextHelpButton(*wxWindow** parent, **wxWindowID** id = *wxID_CONTEXT_HELP*, **const wxPoint&** pos = *wxDefaultPosition*, **const wxSize&** size = *wxDefaultSize*, **long** style = *wxBU_AUTODRAW*)

Constructor, creating and showing a context help button.

Parameters

parent

Parent window. Must not be NULL.

id

Button identifier. Defaults to *wxID_CONTEXT_HELP*.

pos

Button position.

size

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

style

Window style.

Remarks

Normally you need pass only the parent window to the constructor, and use the defaults for the remaining parameters.

wxControl

This is the base class for a control or "widget".

A control is generally a small window which processes user input and/or displays one or more item of data.

Derived from

wxWindow (p. **Error! Bookmark not defined.**)

wxEvtHandler (p. 490)

wxObject (p. **Error! Bookmark not defined.**)

Include files

<wx/control.h>

See also

wxValidator (p. **Error! Bookmark not defined.**)

wxControl::Command

void Command(wxCommandEvent& event)

Simulates the effect of the user issuing a command to the item. See *wxCommandEvent* (p. 184).

wxControl::GetLabel

wxString& GetLabel()

Returns the control's text.

Note that the returned string contains the mnemonics (& characters) if any.

wxControl::SetLabel

void SetLabel(const wxString& label)

Sets the item's text.

The & characters in the *label* are special and indicate that the following character is a mnemonic for this control and can be used to activate it from the keyboard (typically by using *Alt* key in combination with it). To insert a literal ampersand character, you need to double it, i.e. use "&&".

wxControlWithItems

This class is an abstract base class for some *wxWidgets* controls which contain several items, such as *wxListBox* (p. 858) and *wxCheckListBox* (p. 142) derived from it, *wxChoice* (p. 145) and *wxComboBox* (p. 176).

It defines the methods for accessing the controls items and although each of the derived classes implements them differently, they still all conform to the same interface.

The items in a *wxControlWithItems* have (non empty) string labels and, optionally, client

data associated with them. Client data may be of two different kinds: either simple untyped (`void *`) pointers which are simply stored by the control but not used in any way by it, or typed pointers (`wxClientData *`) which are owned by the control meaning that the typed client data (and only it) will be deleted when an item is *deleted* (p. 220) or the entire control is *cleared* (p. 220) (which also happens when it is destroyed). Finally note that in the same control all items must have client data of the same type (typed or untyped), if any. This type is determined by the first call to *Append* (p. 219) (the version with client data pointer) or *SetClientData* (p. 224).

Derived from

wxControl (p. 218)
wxWindow (p. **Error! Bookmark not defined.**)
wxEvtHandler (p. 490)
wxObject (p. **Error! Bookmark not defined.**)

Include files

<wx/ctrlsub.h> but usually never included directly

wxControlWithItems::Append

int Append(const wxString& item)

Adds the item to the end of the list box.

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

int Append(const wxString& item, wxClientData *clientData)

Adds the item to the end of the list box, associating the given, typed or untyped, client data pointer with the item.

void Append(const wxArrayString& strings)

Appends several items at once to the control. Notice that calling this method may be much faster than appending the items one by one if you need to add a lot of items.

Parameters

item

String to add.

clientData

Client data to associate with the item.

Return value

When appending a single item, the return value is the index of the newly added item which may be different from the last one if the control is sorted (e.g. has `wxLB_SORT` or

`wxCB_SORT` style).

`wxControlWithItems::Clear`

`void Clear()`

Removes all items from the control.

Clear() also deletes the client data of the existing items if it is owned by the control.

`wxControlWithItems::Delete`

`void Delete(unsigned int n)`

Deletes an item from the control. The client data associated with the item will be also deleted if it is owned by the control.

Note that it is an error (signalled by an assert failure in debug builds) to remove an item with the index negative or greater or equal than the number of items in the control.

Parameters

n

The zero-based item index.

See also

Clear (p. 220)

`wxControlWithItems::FindString`

`int FindString(const wxString& string, bool caseSensitive = false)`

Finds an item whose label matches the given string.

Parameters

string

String to find.

caseSensitive

Whether search is case sensitive (default is not).

Return value

The zero-based position of the item, or `wxNOT_FOUND` if the string was not found.

`wxControlWithItems::GetClientData`

`void * GetClientData(unsigned int n) const`

Returns a pointer to the client data associated with the given item (if any). It is an error to call this function for a control which doesn't have untyped client data at all although it is ok to call it even if the given item doesn't have any client data associated with it (but other items do).

Parameters

n

The zero-based position of the item.

Return value

A pointer to the client data, or `NULL` if not present.

`wxControlWithItems::GetClientObject`

`wxClientData * GetClientObject(unsigned int n) const`

Returns a pointer to the client data associated with the given item (if any). It is an error to call this function for a control which doesn't have typed client data at all although it is ok to call it even if the given item doesn't have any client data associated with it (but other items do).

Parameters

n

The zero-based position of the item.

Return value

A pointer to the client data, or `NULL` if not present.

`wxControlWithItems::GetCount`

`unsigned int GetCount() const`

Returns the number of items in the control.

See also

IsEmpty (p. 223)

`wxControlWithItems::GetSelection`

`int GetSelection() const`

Returns the index of the selected item or `wxNOT_FOUND` if no item is selected.

Return value

The position of the current selection.

Remarks

This method can be used with single selection list boxes only, you should use `wxListBox::GetSelections` (p. 861) for the list boxes with `wxLB_MULTIPLE` style.

See also

`SetSelection` (p. 225), `GetStringSelection` (p. 223)

wxControlWithItems::GetString

wxString GetString(unsigned int *n*) const

Returns the label of the item with the given index.

Parameters

n

The zero-based index.

Return value

The label of the item or an empty string if the position was invalid.

wxControlWithItems::GetStringSelection

wxString GetStringSelection() const

Returns the label of the selected item or an empty string if no item is selected.

See also

`GetSelection` (p. 222)

wxControlWithItems::Insert

int Insert(const wxString& *item*, unsigned int *pos*)

Inserts the item into the list before *pos*. Not valid for `wxLB_SORT` or `wxCB_SORT` styles, use `Append` instead.

int Insert(const wxString& *item*, unsigned int *pos*, void **clientData*)

int Insert(const wxString& *item*, unsigned int *pos*, wxClientData **clientData*)

Inserts the item into the list before *pos*, associating the given, typed or untyped, client data pointer with the item. Not valid for `wxLB_SORT` or `wxCB_SORT` styles, use `Append` instead.

Parameters

item

String to add.

pos

Position to insert item before, zero based.

clientData

Client data to associate with the item.

Return value

The return value is the index of the newly inserted item. If the insertion failed for some reason, -1 is returned.

wxControlWithItems::IsEmpty

bool IsEmpty() const

Returns `true` if the control is empty or `false` if it has some items.

See also

GetCount (p. 222)

wxControlWithItems::Number

int Number() const

Obsolescence note: This method is obsolete and was replaced with *GetCount* (p. 222), please use the new method in the new code. This method is only available if `wxWidgets` was compiled with `WXWIN_COMPATIBILITY_2_2` defined and will disappear completely in future versions.

wxControlWithItems::Select

void Select(int *n*)

This is the same as *SetSelection* (p. 225) and exists only because it is slightly more natural for controls which support multiple selection.

wxControlWithItems::SetClientData

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

Associates the given untyped client data pointer with the given item. Note that it is an error to call this function if any typed client data pointers had been associated with the control items before.

Parameters

n

The zero-based item index.

data

The client data to associate with the item.

wxControlWithItems::SetClientObject

void SetClientObject(unsigned int *n*, wxClientData **data*)

Associates the given typed client data pointer with the given item: the *data* object will be deleted when the item is deleted (either explicitly by using *Deletes* (p. 220) or implicitly when the control itself is destroyed).

Note that it is an error to call this function if any untyped client data pointers had been associated with the control items before.

Parameters

n

The zero-based item index.

data

The client data to associate with the item.

wxControlWithItems::SetSelection

void SetSelection(int *n*)

Sets the selection to the given item *n* or removes the selection entirely if *n* == wxNOT_FOUND.

Note that this does not cause any command events to be emitted nor does it deselect any other items in the controls which support multiple selections.

Parameters

n

The string position to select, starting from zero.

See also

SetString (p. 225), *SetStringSelection* (p. 225)

wxControlWithItems::SetString

void SetString(unsigned int *n*, const wxString& *string*)

Sets the label for the given item.

Parameters

n

The zero-based item index.

string

The label to set.

wxControlWithItems::SetStringSelection

bool SetStringSelection(const wxString& *string*)

Selects the item with the specified string in the control. This doesn't cause any command events being emitted.

Parameters

string

The string to select.

Return value

`true` if the specified string has been selected, `false` if it wasn't found in the control.

See also

***SetSelection* (p. 225) 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. **Error! Bookmark not defined.**) `wxStreamBase` (p. **Error! Bookmark not defined.**)

Include files

<wx/stream.h>

wxCountingOutputStream::wxCountingOutputStream

wxCountingOutputStream()

Creates a `wxCountingOutputStream` object.

wxCountingOutputStream::~~wxCountingOutputStream

~wxCountingOutputStream()

Destructor.

wxCountingOutputStream::GetSize

size_t GetSize() const

Returns the current size of the stream.

wxCriticalSection

A critical section object is used for exactly the same purpose as *mutexes* (p. **Error! Bookmark not defined.**). 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. 228) class whenever possible instead of directly using *wxCriticalSection* for the same reasons *wxMutexLocker* (p. **Error! Bookmark not defined.**) is preferable to *wxMutex* (p. **Error! Bookmark not defined.**) - please see *wxMutex* for an example.

Derived from

None.

Include files

<wx/thread.h>

See also

wxThread (p. **Error! Bookmark not defined.**), *wxCondition* (p. 193),
wxCriticalSectionLocker (p. 228)

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. 227) objects. A *wxCriticalSectionLocker* enters the critical section in the constructor and leaves it in the destructor making it much more difficult to forget to leave a critical section (which, in general, will lead to serious and difficult to debug problems).

Example of using it:

```
void Set Foo()
{
    // gs_critSect is some (global) critical section guarding
    // access to the
    // object "foo"
    wxCriticalSectionLocker locker(gs_critSect);

    if ( ... )
    {
        // do something
        ...

        return;
    }

    // do something else
    ...

    return;
}
```

Without *wxCriticalSectionLocker*, you would need to remember to manually leave the critical section before each `return`.

Derived from

None.

Include files

<wx/thread.h>

See also

wxCriticalSection (p. 227), *wxMutexLocker* (p. **Error! Bookmark not defined.**)

wxCriticalSectionLocker::wxCriticalSectionLocker

wxCriticalSectionLocker(**wxCriticalSection&** *criticalsection*)

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

wxCriticalSectionLocker::~~wxCriticalSectionLocker

~wxCriticalSectionLocker()

Destructor leaves the critical section.

wxCSCnv

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

Derived from

wxMBConv (p. 923)

Include files

<wx/strconv.h>

See also

wxMBConv (p. 923), *wxEncodingConverter* (p. 482), *wxMBConv classes overview* (p. **Error! Bookmark not defined.**)

wxCSCnv::wxCSCnv

wxCSCnv(const **wxChar*** *charset*)

wxCSCnv(**wxFontEncoding** *encoding*)

Constructor. You may specify either the name of the character set you want to convert from/to or an encoding constant. If the character set name is not recognized, ISO 8859-1 is used as fall back.

wxCSSConv::~~wxCSSConv**~wxCSSConv()**

Destructor frees any resources needed to perform the conversion.

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

Converts from the selected character set to Unicode. Returns length of string written to destination buffer.

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

Converts from Unicode to the selected character set. Returns length of string written to destination buffer.

wxCursor

A cursor is a small bitmap usually used for denoting where the mouse pointer is, with a picture that might indicate the interpretation of a mouse click. As with icons, cursors in X and MS Windows are created in a different manner. Therefore, separate cursors will be created for the different environments. Platform-specific methods for creating a **wxCursor** object are catered for, and this is an occasion where conditional compilation will probably be required (see *wxIcon* (p. 778) for an example).

A single cursor object may be used in many windows (any subwindow type). The wxWidgets convention is to set the cursor for a window, as in X, rather than to set it globally as in MS Windows, although a global `::wxSetCursor` (p. **Error! Bookmark not defined.**) is also available for MS Windows use.

Derived from*wxBitmap* (p. 84)*wxGDIObject* (p. 609)*wxObject* (p. **Error! Bookmark not defined.**)**Include files**

<wx/cursor.h>

Predefined objects

Objects:

wxNullCursor

Pointers:

wxSTANDARD_CURSOR
wxHOURLASS_CURSOR
wxCROSS_CURSOR

See also

wxBitmap (p. 84), *wxIcon* (p. 778), *wxWindow::SetCursor* (p. **Error! Bookmark not defined.**), *::wxSetCursor* (p. **Error! Bookmark not defined.**)

wxCursor::wxCursor

wxCursor()

Default constructor.

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

Constructs a cursor by passing an array of bits (Motif and GTK+ only). *maskBits* is used only under Motif and GTK+. The parameters *fg* and *bg* are only present on GTK+, and force the cursor to use particular background and foreground colours.

If either *hotSpotX* or *hotSpotY* is -1, the hotspot will be the centre of the cursor image (Motif only).

wxCursor(const wxString& cursorName, long type, int hotSpotX=0, int hotSpotY=0)

Constructs a cursor by passing a string resource name or filename.

On MacOS when specifying a string resource name, first the color cursors 'crsr' and then the black/white cursors 'CURS' in the resource chain are scanned through.

hotSpotX and *hotSpotY* are currently only used under Windows when loading from an icon file, to specify the cursor hotspot relative to the top left of the image.

wxCursor(int cursorId)

Constructs a cursor using a cursor identifier.

wxCursor(const wxImage& image)

Constructs a cursor from a *wxImage*. The cursor is monochrome, colors with the RGB elements all greater than 127 will be foreground, colors less than this background. The mask (if any) will be used as transparent.

In MSW the foreground will be white and the background black. If the cursor is larger than 32x32 it is resized. In GTK, the two most frequent colors will be used for foreground and background. The cursor will be displayed at the size of the image. On MacOS if the cursor is larger than 16x16 it is resized and currently only shown as black/white (mask respected).

wxCursor(const wxCursor& cursor)

Copy constructor. This uses reference counting so is a cheap operation.

Parameters

bits

An array of bits.

maskBits

Bits for a mask bitmap.

width

Cursor width.

height

Cursor height.

hotSpotX

Hotspot x coordinate.

hotSpotY

Hotspot y coordinate.

type

Icon type to load. Under Motif, *type* defaults to **wxBITMAP_TYPE_XBM**. Under Windows, it defaults to **wxBITMAP_TYPE_CUR_RESOURCE**. Under MacOS, it defaults to **wxBITMAP_TYPE_MACCURSOR_RESOURCE**.

Under X, the permitted cursor types are:

wxBITMAP_TYPE_XBM Load an X bitmap file.

Under Windows, the permitted types are:

wxBITMAP_TYPE_CUR Load a cursor from a .cur cursor file (only if **USE_RESOURCE_LOADING_IN_MSW** is enabled in **setup.h**).

wxBITMAP_TYPE_CUR_RESOURCE Load a Windows resource (as specified in the .rc file).

wxBITMAP_TYPE_ICO Load a cursor from a .ico icon file (only if **USE_RESOURCE_LOADING_IN_MSW** is enabled in **setup.h**). Specify *hotSpotX* and *hotSpotY*.

cursorId

A stock cursor identifier. May be one of:

wxCURSOR_ARROW	A standard arrow cursor.
wxCURSOR_RIGHT_ARROW	A standard arrow cursor pointing to the right.
wxCURSOR_BLANK	Transparent cursor.
wxCURSOR_BULLSEYE	Bullseye cursor.
wxCURSOR_CHAR	Rectangular character cursor.
wxCURSOR_CROSS	A cross cursor.
wxCURSOR_HAND	A hand cursor.
wxCURSOR_IBEAM	An I-beam cursor (vertical line).
wxCURSOR_LEFT_BUTTON	Represents a mouse with the left button depressed.
wxCURSOR_MAGNIFIER	A magnifier icon.
wxCURSOR_MIDDLE_BUTTON	Represents a mouse with the middle button depressed.
wxCURSOR_NO_ENTRY	A no-entry sign cursor.
wxCURSOR_PAINT_BRUSH	A paintbrush cursor.
wxCURSOR_PENCIL	A pencil cursor.
wxCURSOR_POINT_LEFT	A cursor that points left.
wxCURSOR_POINT_RIGHT	A cursor that points right.
wxCURSOR_QUESTION_ARROW	An arrow and question mark.
wxCURSOR_RIGHT_BUTTON	Represents a mouse with the right button depressed.
wxCURSOR_SIZENESW	A sizing cursor pointing NE-SW.
wxCURSOR_SIZENS	A sizing cursor pointing N-S.
wxCURSOR_SIZENWSE	A sizing cursor pointing NW-SE.
wxCURSOR_SIZEWE	A sizing cursor pointing W-E.
wxCURSOR_SIZING	A general sizing cursor.
wxCURSOR_SPRAYCAN	A spraycan cursor.
wxCURSOR_WAIT	A wait cursor.
wxCURSOR_WATCH	A watch cursor.
wxCURSOR_ARROWWAIT	A cursor with both an arrow and an hourglass,

(windows.)

Note that not all cursors are available on all platforms.

cursor

Pointer or reference to a cursor to copy.

wxPython note: Constructors supported by wxPython are:

wxCursor(name, flags, hotSpotX=0, hotSpotY=0) Constructs a cursor from a filename

wxStockCursor(id) Constructs a stock cursor

wxPerl note: Constructors supported by wxPerl are:

- ::Cursor->new(name, type, hotSpotX = 0, hotSpotY = 0)
- ::Cursor->new(id)
- ::Cursor->new(image)
- ::Cursor->newData(bits, width, height, hotSpotX = -1, hotSpotY = -1, maskBits = 0)

Example

The following is an example of creating a cursor from 32x32 bitmap data (`down_bits`) and a mask (`down_mask`) where 1 is black and 0 is white for the bits, and 1 is opaque and 0 is transparent for the mask. It works on Windows and GTK+.

```
static char down_bits[] = { 255, 255, 255, 255, 31,
    255, 255, 255, 31, 255, 255, 255, 31, 255, 255, 255,
    31, 255, 255, 255, 31, 255, 255, 255, 31, 255, 255,
    255, 31, 255, 255, 255, 31, 255, 255, 255, 25, 243,
    255, 255, 19, 249, 255, 255, 7, 252, 255, 255, 15, 254,
    255, 255, 31, 255, 255, 255, 191, 255, 255, 255, 255,
    255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
    255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
    255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
    255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
    255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
    255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
    255 };

static char down_mask[] = { 240, 1, 0, 0, 240, 1,
    0, 0, 240, 1, 0, 0, 240, 1, 0, 0, 240, 1, 0, 0, 240, 1,
    0, 0, 240, 1, 0, 0, 240, 1, 0, 0, 255, 31, 0, 0, 255,
    31, 0, 0, 254, 15, 0, 0, 252, 7, 0, 0, 248, 3, 0, 0,
    240, 1, 0, 0, 224, 0, 0, 0, 64, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0 };

#ifdef __WXMSW__
wxBitmap down_bitmap(down_bits, 32, 32);
```

```
wxBitmap down_mask_bitmap(down_mask, 32, 32);

down_bitmap.SetMask(new wxMask(down_mask_bitmap));
wxImage down_image = down_bitmap.ConvertToImage();
down_image.SetOption(wxIMAGE_OPTION_CUR_HOTSPOT_X, 6);
down_image.SetOption(wxIMAGE_OPTION_CUR_HOTSPOT_Y, 14);
wxCursor down_cursor = wxCursor(down_image);
#else
wxCursor down_cursor = wxCursor(down_bits, 32, 32,
    6, 14, down_mask, wxWHITE, wxBLACK);
#endif
```

wxCursor::~~wxCursor

~wxCursor()

Destroys the cursor. A cursor can be reused for more than one window, and does not get destroyed when the window is destroyed. `wxWidgets` destroys all cursors on application exit, although it is best to clean them up explicitly.

wxCursor::Ok

bool Ok() const

Returns true if cursor data is present.

wxCursor::operator =

wxCursor& operator =(const wxCursor& cursor)

Assignment operator, using reference counting. Returns a reference to 'this'.

wxCursor::operator ==

bool operator ==(const wxCursor& cursor)

Equality operator. Two cursors are equal if they contain pointers to the same underlying cursor data. It does not compare each attribute, so two independently-created cursors using the same parameters will fail the test.

wxCursor::operator !=

bool operator !=(const wxCursor& cursor)

Inequality operator. Two cursors are not equal if they contain pointers to different underlying cursor data. It does not compare each attribute.

wxCustomDataObject

`wxCustomDataObject` is a specialization of `wxDataObjectSimple` (p. 247) 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. 237) or *TakeData* (p. 237) 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. 237), *GetData* (p. 237) and *SetData* (p. 237) (or may be only the first two or only the last one if you only allow reading/writing the data)

Derived from

wxDataObjectSimple (p. 247)
wxDataObject (p. 242)

Include files

<wx/dataobj.h>

See also

wxDataObject (p. 242)

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. 248) 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. 237) function, the base class version will always be called (C++ doesn't allow calling virtual functions from constructors or destructors), so if you override *Free()*, you should override the destructor in your class as well (which would probably just call the derived class' version of *Free()*).

wxCustomDataObject::Alloc

virtual void * Alloc(size_t size)

This function is called to allocate *size* bytes of memory from `SetData()`. The default version just uses the operator `new`.

wxCustomDataObject::Free

virtual void Free()

This function is called when the data is freed, you may override it to anything you want (or may be nothing at all). The default version calls `operator delete[]` on the data.

wxCustomDataObject::GetSize

virtual size_t GetSize() const

Returns the data size in bytes.

wxCustomDataObject::GetData

virtual void * GetData() const

Returns a pointer to the data.

wxCustomDataObject::SetData

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

Set the data. The data object will make an internal copy.

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

wxCustomDataObject::TakeData

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

Like *SetData* (p. 237), but doesn't copy the data - instead the object takes ownership of the pointer.

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

wxDataFormat

A `wxDataFormat` is an encapsulation of a platform-specific format handle which is used by the system for the clipboard and drag and drop operations. The applications are usually only interested in, for example, pasting data from the clipboard only if the data is in a format the program understands and a data format is something which uniquely identifies this format.

On the system level, a data format is usually just a number (`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:

<code>wxDF_INVALID</code>	An invalid format - used as default argument for functions taking a <code>wxDataFormat</code> argument sometimes
<code>wxDF_TEXT</code>	Text format (<code>wxString</code>)
<code>wxDF_BITMAP</code>	A bitmap (<code>wxBitmap</code>)
<code>wxDF_METAFILE</code>	A metafile (<code>wxMetafile</code> , Windows only)
<code>wxDF_FILENAME</code>	A list of filenames
<code>wxDF_HTML</code>	An HTML string. This is only valid when passed to <code>wxSetClipboardData</code> when compiled with Visual C++ in non-Unicode mode

As mentioned above, these standard formats may be passed to any function taking `wxDataFormat` argument because `wxDataFormat` has an implicit conversion from them (or, to be precise from the type `wxDataFormat::NativeFormat` which is the type used by the underlying platform for data formats).

Aside the standard formats, the application may also use custom formats which are identified by their names (strings) and not numeric identifiers. Although internally custom format must be created (or *registered*) first, you shouldn't care about it because it is done automatically the first time the `wxDataFormat` object corresponding to a given format name is created. The only implication of this is that you should avoid having global `wxDataFormat` objects with non-default constructor because their constructors are executed before the program has time to perform all necessary initialisations and so an attempt to do clipboard format registration at this time will usually lead to a crash!

Virtual functions to override

None

Derived from

None

See also

Clipboard and drag and drop overview (p. **Error! Bookmark not defined.**), *DnD sample* (p. **Error! Bookmark not defined.**), *wxDataObject* (p. 242)

Include files

<wx/dataobj.h>

wxDataFormat::wxDataFormat

wxDataFormat(NativeFormat *format* = wxDF_INVALID)

Constructs a data format object for one of the standard data formats or an empty data object (use *SetType* (p. 239) or *SetId* (p. 239) later in this case)

wxPerl note: In wxPerl this function is named `newNative`.

wxDataFormat::wxDataFormat

wxDataFormat(const wxChar **format*)

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

wxPerl note: In wxPerl this function is named `newUser`.

wxDataFormat::operator ==

bool operator ==(const wxDataFormat& *format*) const

Returns true if the formats are equal.

wxDataFormat::operator !=

bool operator !=(const wxDataFormat& *format*) const

Returns true if the formats are different.

wxDataFormat::GetId

wxString GetId() const

Returns the name of a custom format (this function will fail for a standard format).

wxDataFormat::GetType

NativeFormat GetType() const

Returns the platform-specific number identifying the format.

wxDataFormat::SetId

void SetId(const wxChar **format*)

Sets the format to be the custom format identified by the given name.

wxDataFormat::SetType

void SetType(NativeFormat *format*)

Sets the format to the given value, which should be one of wxDF_XXX constants.

wxDataInputStream

This class provides functions that read binary data types in a portable way. Data can be read in either big-endian or little-endian format, little-endian being the default on all architectures.

If you want to read data from text files (or streams) use *wxTextInputStream* (p. **Error! Bookmark not defined.**) 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. 248).

Derived from

None

Include files

<wx/datstrm.h>

wxDataInputStream::wxDataInputStream

wxDataInputStream(wxInputStream& *stream*)

wxDataInputStream(wxInputStream& *stream*, wxMBConv& *conv* = wxMBConvUTF8)

Constructs a datastream object from an input stream. Only read methods will be available. The second form is only available in Unicode build of wxWidgets.

Parameters

stream

The input stream.

conv

Charset conversion object object used to decode strings in Unicode mode (see *wxDataInputStream::ReadString* (p. 242) documentation for detailed description). Note that you must not destroy *conv* before you destroy this *wxDataInputStream* instance!

wxDataInputStream::~~wxDataInputStream

~wxDataInputStream()

Destroys the *wxDataInputStream* object.

wxDataInputStream::BigEndianOrdered

void BigEndianOrdered(bool *be_order*)

If *be_order* is true, all data will be read in big-endian order, such as written by programs on a big endian architecture (e.g. Sparc) or written by Java-Streams (which always use big-endian order). *wxDataInputStream::Read8*

wxUInt8 Read8()

Reads a single byte from the stream.

void Read8(wxUInt8 **buffer*, size_t *size*)

Reads bytes from the stream in a specified buffer. The amount of bytes to read is specified by the *size* variable.

wxDataInputStream::Read16

wxUInt16 Read16()

Reads a 16 bit unsigned integer from the stream.

void Read16(wxUInt16 **buffer*, size_t *size*)

Reads 16 bit unsigned integers from the stream in a specified buffer. the amount of 16 bit unsigned integer to read is specified by the *size* variable.

wxDataInputStream::Read32

wxUInt32 Read32()

Reads a 32 bit unsigned integer from the stream.

void Read32(wxUInt32 **buffer*, size_t *size*)

Reads 32 bit unsigned integers from the stream in a specified buffer. the amount of 32 bit unsigned integer to read is specified by the *size* variable.

wxDataInputStream::Read64

wxUint64 Read64()

Reads a 64 bit unsigned integer from the stream.

void Read64(wxUint64 *buffer, size_t size)

Reads 64 bit unsigned integers from the stream in a specified buffer. the amount of 64 bit unsigned integer to read is specified by the *size* variable.

wxDataInputStream::ReadDouble**double ReadDouble()**

Reads a double (IEEE encoded) from the stream.

void ReadDouble(double *buffer, size_t size)

Reads double data (IEEE encoded) from the stream in a specified buffer. the amount of double to read is specified by the *size* variable.

wxDataInputStream::ReadString**wxString ReadString()**

Reads a string from a stream. Actually, this function first reads a long integer specifying the length of the string (without the last null character) and then reads the string.

In Unicode build of wxWidgets, the function first reads multibyte (char*) string from the stream and then converts it to Unicode using the *convobject* passed to constructor and returns the result as wxString. You are responsible for using the same convertor as when writing the stream.

See also *wxDataOutputStream::WriteString* (p. 251).

wxDataObject

A wxDataObject represents data that can be copied to or from the clipboard, or dragged and dropped. The important thing about wxDataObject is that this is a 'smart' piece of data unlike 'dumb' data containers such as memory buffers or files. Being 'smart' here means that the data object itself should know what data formats it supports and how to render itself in each of its supported formats.

A supported format, incidentally, is exactly the format in which the data can be requested from a data object or from which the data object may be set. In the general case, an object may support different formats on 'input' and 'output', i.e. it may be able to render itself in a given format but not be created from data on this format or vice versa.

wxDataObject defines an enumeration type

```
enum Direction
{
    Get    = 0x01,    // format is supported by GetDataHere()
    Set    = 0x02    // format is supported by SetData()
```

```
};
```

which distinguishes between them. See *wxDataFormat* (p. 237) documentation for more about formats.

Not surprisingly, being 'smart' comes at a price of added complexity. This is reasonable for the situations when you really need to support multiple formats, but may be annoying if you only want to do something simple like cut and paste text.

To provide a solution for both cases, wxWidgets has two predefined classes which derive from *wxDataObject*: *wxDataObjectSimple* (p. 247) and *wxDataObjectComposite* (p. 246). *wxDataObjectSimple* (p. 247) is the simplest *wxDataObject* possible and only holds data in a single format (such as HTML or text) and *wxDataObjectComposite* (p. 246) is the simplest way to implement a *wxDataObject* that does support multiple formats because it achieves this by simply holding several *wxDataObjectSimple* objects.

So, you have several solutions when you need a *wxDataObject* class (and you need one as soon as you want to transfer data via the clipboard or drag and drop):

- 1. Use one of the built-in classes** You may use *wxTextDataObject*, *wxBitmapDataObject* or *wxFileDataObject* in the simplest cases when you only need to support one format and your data is either text, bitmap or list of files.
- 2. Use *wxDataObjectSimple*** Deriving from *wxDataObjectSimple* is the simplest solution for custom data - you will only support one format and so probably won't be able to communicate with other programs, but data transfer will work in your program (or between different copies of it).
- 3. Use *wxDataObjectComposite*** This is a simple but powerful solution which allows you to support any number of formats (either standard or custom if you combine it with the previous solution).
- 4. Use *wxDataObject* directly** This is the solution for maximal flexibility and efficiency, but it is also the most difficult to implement.

Please note that the easiest way to use drag and drop and the clipboard with multiple formats is by using *wxDataObjectComposite*, but it is not the most efficient one as each *wxDataObjectSimple* would contain the whole data in its respective formats. Now imagine that you want to paste 200 pages of text in your proprietary format, as well as Word, RTF, HTML, Unicode and plain text to the clipboard and even today's computers are in trouble. For this case, you will have to derive from *wxDataObject* directly and make it enumerate its formats and provide the data in the requested format on demand.

Note that neither the GTK+ data transfer mechanisms for clipboard and drag and drop, nor OLE data transfer, copy any data until another application actually requests the data. This is in contrast to the 'feel' offered to the user of a program who would normally think that the data resides in the clipboard after having pressed 'Copy' - in reality it is only declared to be available.

There are several predefined data object classes derived from *wxDataObjectSimple*: *wxFileDataObject* (p. 514), *wxTextDataObject* (p. **Error! Bookmark not defined.**) and

wxBitmapDataObject (p. 103) which can be used without change.

You may also derive your own data object classes from *wxCustomDataObject* (p. 235) for user-defined types. The format of user-defined data is given as a mime-type string literal, such as "application/word" or "image/png". These strings are used as they are under Unix (so far only GTK+) to identify a format and are translated into their Windows equivalent under Win32 (using the OLE *IDataObject* for data exchange to and from the clipboard and for drag and drop). Note that the format string translation under Windows is not yet finished.

wxPython note: At this time this class is not directly usable from wxPython. Derive a class from *wxPyDataObjectSimple* (p. 247) instead.

wxPerl note: This class is not currently usable from wxPerl; you may use *Wx::PIDataObjectSimple* (p. 247) 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. 245).

Derived from

None

Include files

<wx/dataobj.h>

See also

Clipboard and drag and drop overview (p. **Error! Bookmark not defined.**), *DnD sample* (p. **Error! Bookmark not defined.**), *wxFileDataObject* (p. 514), *wxTextDataObject* (p. **Error! Bookmark not defined.**), *wxBitmapDataObject* (p. 103), *wxCustomDataObject* (p. 235), *wxDropTarget* (p. 475), *wxDropSource* (p. 472), *wxTextDropTarget* (p. **Error! Bookmark not defined.**), *wxFileDropTarget* (p. 519)

wxDataObject::wxDataObject

wxDataObject()

Constructor.

wxDataObject::~~wxDataObject

~wxDataObject()

Destructor.

wxDataObject::GetAllFormats**virtual void GetAllFormats(wxDataFormat *formats, Direction dir = Get) const**

Copy all supported formats in the given direction to the array pointed to by *formats*. There is enough space for GetFormatCount(dir) formats in it.

wxPerl note: In wxPerl this method only takes the `dir` parameter. In scalar context it returns the first format, in list context it returns a list containing all the supported formats.

wxDataObject::GetDataHere**virtual bool GetDataHere(const wxDataFormat& format, void *buf) const**

The method will write the data of the format *format* in the buffer *buf* and return true on success, false on failure.

wxDataObject::GetDataSize**virtual size_t GetDataSize(const wxDataFormat& format) const**

Returns the data size of the given format *format*.

wxDataObject::GetFormatCount**virtual size_t GetFormatCount(Direction dir = Get) const**

Returns the number of available formats for rendering or setting the data.

wxDataObject::GetPreferredFormat**virtual wxDataFormat GetPreferredFormat(Direction dir = Get) const**

Returns the preferred format for either rendering the data (if *dir* is `Get`, its default value) or for setting it. Usually this will be the native format of the `wxDataObject`.

wxDataObject::SetData**virtual bool SetData(const wxDataFormat& format, size_t len, const void *buf)**

Set the data in the format *format* of the length *len* provided in the buffer *buf*.

Returns true on success, false on failure.

wxDataObjectComposite

`wxDataObjectComposite` is the simplest `wxDataObject` (p. 242) derivation which may be used to support multiple formats. It contains several `wxDataObjectSimple` (p. 247) 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. 247)) and its format determines the preferred format of the composite data object as well.

See *wxDataObject* (p. 242) 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. 242)

Include files

<wx/dataobj.h>

See also

Clipboard and drag and drop overview (p. **Error! Bookmark not defined.**),
wxDataObject (p. 242), *wxDataObjectSimple* (p. 247), *wxFileDataObject* (p. 514),
wxTextDataObject (p. **Error! Bookmark not defined.**), *wxBitmapDataObject* (p. 103)

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. 242) class. The data object of (a class derived from) this class only supports one format, so the number of virtual functions to be implemented is reduced.

Notice that this is still an abstract base class and cannot be used but should be derived from.

wxPython note: If you wish to create a derived *wxDataObjectSimple* class in wxPython you should derive the class from *wxPyDataObjectSimple* in order to get Python-aware capabilities for the various virtual methods.

wxPerl note: In wxPerl, you need to derive your data object class from `Wx::PIDataObjectSimple`.

Virtual functions to override

The objects supporting rendering the data must override *GetDataSize* (p. 248) and *GetDataHere* (p. 248) while the objects which may be set must override *SetData* (p. 248). Of course, the objects supporting both operations must override all three methods.

Derived from

wxDataObject (p. 242)

Include files

<wx/dataobj.h>

See also

Clipboard and drag and drop overview (p. **Error! Bookmark not defined.**), *DnD sample* (p. **Error! Bookmark not defined.**), *wxFileDataObject* (p. 514), *wxTextDataObject* (p. **Error! Bookmark not defined.**), *wxBitmapDataObject* (p. 103)

wxDataObjectSimple::wxDataObjectSimple

wxDataObjectSimple(const wxDataFormat& format = wxFormatInvalid)

Constructor accepts the supported format (none by default) which may also be set later with *SetFormat* (p. 248).

wxDataObjectSimple::GetFormat

const wxDataFormat& GetFormat() const

Returns the (one and only one) format supported by this object. It is supposed that the format is supported in both directions.

wxDataObjectSimple::SetFormat

void SetFormat(const wxDataFormat& format)

Sets the supported format.

wxDataObjectSimple::GetDataSize

virtual size_t GetDataSize() const

Gets the size of our data. Must be implemented in the derived class if the object supports rendering its data.

wxDataObjectSimple::GetDataHere**virtual bool GetDataHere(void *buf) const**

Copy the data to the buffer, return true on success. Must be implemented in the derived class if the object supports rendering its data.

wxPython note: When implementing this method in wxPython, no additional parameters are required and the data should be returned from the method as a string.

wxDataObjectSimple::SetData**virtual bool SetData(size_t len, const void *buf)**

Copy the data from the buffer, return true on success. Must be implemented in the derived class if the object supports setting its data.

wxPython note: When implementing this method in wxPython, the data comes as a single string parameter rather than the two shown here.

wxDataOutputStream

This class provides functions that write binary data types in a portable way. Data can be written in either big-endian or little-endian format, little-endian being the default on all architectures.

If you want to write data to text files (or streams) use *wxTextOutputStream* (p. **Error! Bookmark not defined.**) instead.

The << operator is overloaded and you can use this class like a standard C++ iostream. See *wxDataInputStream* (p. 239) for its usage and caveats.

See also *wxDataInputStream* (p. 239).

Derived from

None

Include files

<wx/datstrm.h>

wxDataOutputStream::wxDataOutputStream**wxDataOutputStream(wxOutputStream& stream)****wxDataOutputStream(wxOutputStream& stream, wxMBConv& conv = wxMBConvUTF8)**

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

available. The second form is only available in Unicode build of wxWidgets.

Parameters

stream

The output stream.

conv

Charset conversion object object used to encoding Unicode strings before writing them to the stream in Unicode mode (see `wxDataOutputStream::WriteString` (p. 251) documentation for detailed description). Note that you must not destroy *conv* before you destroy this `wxDataOutputStream` instance! It is recommended to use default value (UTF-8).

wxDataOutputStream::~~wxDataOutputStream

~wxDataOutputStream()

Destroys the `wxDataOutputStream` object.

wxDataOutputStream::BigEndianOrdered

void BigEndianOrdered(bool *be_order*)

If *be_order* is true, all data will be written in big-endian order, e.g. for reading on a Sparc or from Java-Streams (which always use big-endian order), otherwise data will be written in little-endian order.

wxDataOutputStream::Write8

void Write8(wxUInt8 *i8*)

Writes the single byte *i8* to the stream.

void Write8(const wxUInt8 **buffer*, size_t *size*)

Writes an array of bytes to the stream. The amount of bytes to write is specified with the *size* variable.

wxDataOutputStream::Write16

void Write16(wxUInt16 *i16*)

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

void Write16(const wxUInt16 **buffer*, size_t *size*)

Writes an array of 16 bit unsigned integer to the stream. The amount of 16 bit unsigned integer to write is specified with the *size* variable.

wxDataOutputStream::Write32**void Write32(wxUint32 *i32*)**

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

void Write32(const wxUint32 **buffer*, size_t *size*)

Writes an array of 32 bit unsigned integer to the stream. The amount of 32 bit unsigned integer to write is specified with the *size* variable.

wxDataOutputStream::Write64**void Write64(wxUint64 *i64*)**

Writes the 64 bit unsigned integer *i64* to the stream.

void Write64(const wxUint64 **buffer*, size_t *size*)

Writes an array of 64 bit unsigned integer to the stream. The amount of 64 bit unsigned integer to write is specified with the *size* variable.

wxDataOutputStream::WriteDouble**void WriteDouble(double *f*)**

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

void WriteDouble(const double **buffer*, size_t *size*)

Writes an array of double to the stream. The amount of double to write is specified with the *size* variable.

wxDataOutputStream::WriteString**void WriteString(const wxString& *string*)**

Writes *string* to the stream. Actually, this method writes the size of the string before writing *string* itself.

In ANSI build of wxWidgets, the string is written to the stream in exactly same way it is represented in memory. In Unicode build, however, the string is first converted to multibyte representation with *conv* object passed to stream's constructor (consequently, ANSI application can read data written by Unicode application, as long as they agree on encoding) and this representation is written to the stream. UTF-8 is used by default.

wxDateEvent

This event class holds information about a date change and is used together with *wxDatePickerCtrl* (p. 251). It also serves as a base class for *wxCalendarEvent* (p. 135).

Derived from

wxCommandEvent (p. 184)

wxEvent (p. 487)

wxObject (p. **Error! Bookmark not defined.**)

Include files

<wx/dateevt.h>

wxDateEvent::GetDate

const wxDateTime& GetDate() const

Returns the date.

wxDateEvent::SetDate

void SetDate(const wxDateTime& date)

Sets the date carried by the event, normally only used by the library internally.

wxDatePickerCtrl

This control allows the user to select a date. Unlike *wxCalendarCtrl* (p. 127), which is a relatively big control, *wxDatePickerCtrl* is implemented as a small window showing the currently selected date. The control can be edited using the keyboard, and can also display a popup window for more user-friendly date selection, depending on the styles used and the platform, except PalmOS where date is selected using native dialog.

It is only available if `wxUSE_DATEPICKCTRL` is set to 1.

Derived from

wxControl (p. 218)

wxWindow (p. **Error! Bookmark not defined.**)

wxEvtHandler (p. 490)

wxObject (p. **Error! Bookmark not defined.**)

Include files

<wx/datectrl.h>

(only available if `wxUSE_DATEPICKCTRL` is set to 1)

Window styles**wxDP_SPIN**

Creates a control without a month calendar drop down but with spin-control-like arrows to change individual date components. This style is not supported by the generic

version.

wxDP_DROPDOWN	Creates a control with a month calendar drop-down part from which the user can select a date.
wxDP_DEFAULT	Creates a control with the style that is best supported for the current platform (currently wxDP_SPIN under Windows and wxDP_DROPDOWN elsewhere).
wxDP_ALLOWNONE	With this style, the control allows the user to not enter any valid date at all. Without it - the default - the control always has some valid date.
wxDP_SHOWCENTURY	Forces display of the century in the default date format. Without this style the century could be displayed, or not, depending on the default date representation in the system.

Event handling

EVT_DATE_CHANGED(id, func)	This event fires when the user changes the current selection in the control.
-----------------------------------	--

See also

wxCalendarCtrl (p. 127),
wxDateEvent (p. 251)

wxDatePickerCtrl::wxDatePickerCtrl

wxDatePickerCtrl(*wxWindow *parent*, **wxWindowID** *id*, **const wxDateTime&** *dt* = *wxDefaultDateTime*, **const wxPoint&** *pos* = *wxDefaultPosition*, **const wxSize&** *size* = *wxDefaultSize*, **long** *style* = *wxDP_DEFAULT* | *wxDP_SHOWCENTURY*, **const wxValidator&** *validator* = *wxDefaultValidator*, **const wxString&** *name* = "datectrl")

Initializes the object and calls *Create* (p. 253) with all the parameters.

wxDatePickerCtrl::Create

bool *Create*(*wxWindow *parent*, **wxWindowID** *id*, **const wxDateTime&** *dt* = *wxDefaultDateTime*, **const wxPoint&** *pos* = *wxDefaultPosition*, **const wxSize&** *size* = *wxDefaultSize*, **long** *style* = *wxDP_DEFAULT* | *wxDP_SHOWCENTURY*, **const wxValidator&** *validator* = *wxDefaultValidator*, **const wxString&** *name* = "datectrl")

Parameters

parent

Parent window, must not be non-NULL.

id

The identifier for the control.

dt

The initial value of the control, if an invalid date (such as the default value) is used, the control is set to today.

pos

Initial position.

size

Initial size. If left at default value, the control chooses its own best size by using the height approximately equal to a text control and width large enough to show the date string fully.

style

The window style, should be left at 0 as there are no special styles for this control in this version.

validator

Validator which can be used for additional date checks.

name

Control name.

Return value

`true` if the control was successfully created or `false` if creation failed.

wxDatePickerCtrl::GetRange

bool GetRange(wxDateTime *dt1, wxDateTime *dt2) const

If the control had been previously limited to a range of dates using *SetRange()* (p. 254), returns the lower and upper bounds of this range. If no range is set (or only one of the bounds is set), *dt1* and/or *dt2* are set to be invalid.

Parameters

dt1

Pointer to the object which receives the lower range limit or becomes invalid if it is not set. May be `NULL` if the caller is not interested in lower limit

dt2

Same as above but for the upper limit

Return value

`false` if no range limits are currently set, `true` if at least one bound is set.

wxDatePickerCtrl::GetValue

wxDateTime GetValue() const

Returns the currently selected. If there is no selection or the selection is outside of the current range, an invalid object is returned.

wxDatePickerCtrl::SetFormat

void SetFormat(const wxChar* format)

Sets the display format for the date in the control. See `wxDateTime` for the meaning of format strings.

Remarks

If the format parameter is invalid, the behaviour is undefined.

wxDatePickerCtrl::SetRange

void SetRange(const wxDateTime& dt1, const wxDateTime& dt2)

Sets the valid range for the date selection. If *dt1* is valid, it becomes the earliest date (inclusive) accepted by the control. If *dt2* is valid, it becomes the latest possible date.

Remarks

If the current value of the control is outside of the newly set range bounds, the behaviour is undefined.

wxDatePickerCtrl::SetValue

void SetValue(const wxDateTime& dt)

Changes the current value of the control. The date should be valid and included in the currently selected range, if any.

Calling this method does not result in a date change event.

wxDateSpan

This class is a "logical time span" and is useful for implementing program logic for such things as "add one month to the date" which, in general, doesn't mean to add $60 \times 60 \times 24 \times 31$ seconds to it, but to take the same date the next month (to understand that this is indeed different consider adding one month to Feb, 15 -- we want to get Mar, 15, of course).

When adding a month to the date, all lesser components (days, hours, ...) won't be changed unless the resulting date would be invalid: for example, Jan 31 + 1 month will

be Feb 28, not (non existing) Feb 31.

Because of this feature, adding and subtracting back again the same `wxDateSpan` will **not**, in general give back the original date: Feb 28 - 1 month will be Jan 28, not Jan 31!

`wxDateSpan` objects can be either positive or negative. They may be multiplied by scalars which multiply all deltas by the scalar: i.e. `2*(1 month and 1 day)` is 2 months and 2 days. They can be added together and with `wxDateTime` (p. 260) or `wxTimeSpan` (p. **Error! Bookmark not defined.**), but the type of result is different for each case.

Beware about weeks: if you specify both weeks and days, the total number of days added will be `7*weeks + days`! See also `GetTotalDays()` function.

Equality operators are defined for `wxDateSpans`. Two datespans are equal if and only if they both give the same target date when added to **every** source date. Thus `wxDateSpan::Months(1)` is not equal to `wxDateSpan::Days(30)`, because they don't give the same date when added to 1 Feb. But `wxDateSpan::Days(14)` is equal to `wxDateSpan::Weeks(2)`

Finally, notice that for adding hours, minutes and so on you don't need this class at all: `wxTimeSpan` (p. **Error! Bookmark not defined.**) will do the job because there are no subtleties associated with those (we don't support leap seconds).

Derived from

No base class

Include files

`<wx/datetime.h>`

See also

Date classes overview (p. **Error! Bookmark not defined.**), `wxDateTime` (p. 260)

`wxDateSpan::wxDateSpan`

`wxDateSpan(int years = 0, int months = 0, int weeks = 0, int days = 0)`

Constructs the date span object for the given number of years, months, weeks and days. Note that the weeks and days add together if both are given.

`wxDateSpan::Add`

`wxDateSpan Add(const wxDateSpan& other) const`

`wxDateSpan& Add(const wxDateSpan& other)`

`wxDateSpan& operator+=(const wxDateSpan& other)`

Returns the sum of two date spans. The first version returns a new object, the second

and third ones modify this object in place.

wxDateSpan::Day

static wxDateSpan Day()

Returns a date span object corresponding to one day.

See also

Days (p. 256)

wxDateSpan::Days

static wxDateSpan Days(int days)

Returns a date span object corresponding to the given number of days.

See also

Day (p. 256)

wxDateSpan::GetDays

int GetDays() const

Returns the number of days (only, that it not counting the weeks component!) in this date span.

See also

GetTotalDays (p. 257)

wxDateSpan::GetMonths

int GetMonths() const

Returns the number of the months (not counting the years) in this date span.

wxDateSpan::GetTotalDays

int GetTotalDays() const

Returns the combined number of days in this date span, counting both weeks and days. It still doesn't take neither months nor years into the account.

See also

GetWeeks (p. 257), *GetDays* (p. 256)

wxDateSpan::GetWeeks

int GetWeeks() const

Returns the number of weeks in this date span.

See also

GetTotalDays (p. 257)

wxDateSpan::GetYears**int GetYears() const**

Returns the number of years in this date span.

wxDateSpan::Month**static wxDateSpan Month()**

Returns a date span object corresponding to one month.

See also

Months (p. 258)

wxDateSpan::Months**static wxDateSpan Months(int mon)**

Returns a date span object corresponding to the given number of months.

See also

Month (p. 257)

wxDateSpan::Multiply**wxDateSpan Multiply(int factor) const****wxDateSpan& Multiply(int factor)****wxDateSpan& operator*=(int factor)**

Returns the product of the date span by the specified *factor*. The product is computed by multiplying each of the components by the factor.

The first version returns a new object, the second and third ones modify this object in place.

wxDateSpan::Negate**wxDateSpan Negate() const**

Returns the date span with the opposite sign.

See also

Neg (p. 258)

wxDateSpan::Neg

wxDateSpan& Neg()

wxDateSpan& operator-()

Changes the sign of this date span.

See also

Negate (p. 258)

wxDateSpan::SetDays

wxDateSpan& SetDays(int *n*)

Sets the number of days (without modifying any other components) in this date span.

wxDateSpan::SetYears

wxDateSpan& SetYears(int *n*)

Sets the number of years (without modifying any other components) in this date span.

wxDateSpan::SetMonths

wxDateSpan& SetMonths(int *n*)

Sets the number of months (without modifying any other components) in this date span.

wxDateSpan::SetWeeks

wxDateSpan& SetWeeks(int *n*)

Sets the number of weeks (without modifying any other components) in this date span.

wxDateSpan::Subtract

wxDateSpan Subtract(const wxDateSpan& *other*) const

wxDateSpan& Subtract(const wxDateSpan& *other*)

wxDateSpan& operator+=(const wxDateSpan& *other*)

Returns the difference of two date spans. The first version returns a new object, the second and third ones modify this object in place.

wxDateSpan::Week**static wxDateSpan Week()**

Returns a date span object corresponding to one week.

See also

Weeks (p. 259)

wxDateSpan::Weeks**static wxDateSpan Weeks(int weeks)**

Returns a date span object corresponding to the given number of weeks.

See also

Week (p. 259)

wxDateSpan::Year**static wxDateSpan Year()**

Returns a date span object corresponding to one year.

See also

Years (p. 260)

wxDateSpan::Years**static wxDateSpan Years(int years)**

Returns a date span object corresponding to the given number of years.

See also

Year (p. 259)

wxDateSpan::operator==**bool operator==(wxDateSpan& other) const**

Returns `true` if this date span is equal to the other one. Two date spans are considered equal if and only if they have the same number of years and months and the same total number of days (counting both days and weeks).

wxDateSpan::operator!=**bool operator!=(wxDateSpan& other) const**

Returns `true` if this date span is different from the other one.

See also

`operator==` (p. 260)

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 to be
    // consequent numbers, so writing something like `GMT0 +
    // offset' is
    // safe if abs(offset) <= 12

    // underscore stands for minus
    GMT_12, GMT_11, GMT_10, GMT_9, GMT_8, GMT_7,
    GMT_6, GMT_5, GMT_4, GMT_3, GMT_2, GMT_1,
    GMT0,
    GMT1, GMT2, GMT3, GMT4, GMT5, GMT6,
    GMT7, GMT8, GMT9, GMT10, GMT11, GMT12,
    // Note that GMT12 and GMT_12 are not the same: there is a
    // difference
    // of exactly one day between them

    // some symbolic names for TZ

    // Europe
    WET = GMT0, // Western Europe Time
    WEST = GMT1, // Western Europe

    Summer Time
    CET = GMT1, // Central Europe Time
    CEST = GMT2, // Central Europe

    Summer Time
    EET = GMT2, // Eastern Europe Time
    EEST = GMT3, // Eastern Europe

    Summer Time
    MSK = GMT3, // Moscow Time
```

```

        MSD = GMT4,                                // Moscow Summer Time
        // US and Canada
        AST = GMT_4,                                // Atlantic Standard
Time    ADT = GMT_3,                                // Atlantic Daylight
Time    EST = GMT_5,                                // Eastern Standard
Time    EDT = GMT_4,                                // Eastern Daylight
Saving Time CST = GMT_6,                            // Central Standard
Time    CDT = GMT_5,                                // Central Daylight
Saving Time MST = GMT_7,                            // Mountain Standard
Time    MDT = GMT_6,                                // Mountain Daylight
Saving Time PST = GMT_8,                            // Pacific Standard
Time    PDT = GMT_7,                                // Pacific Daylight
Saving Time HST = GMT_10,                           // Hawaiian Standard
Time    AKST = GMT_9,                                // Alaska Standard
Time    AKDT = GMT_8,                                // Alaska Daylight
Saving Time
        // Australia
        A_WST = GMT8,                                // Western Standard
Time    A_CST = GMT12 + 1,                           // Central Standard
Time (+9.5) A_EST = GMT10,                            // Eastern Standard
Time    A_ESST = GMT11,                              // Eastern Summer Time

        // Universal Coordinated Time = the new and politically
correct name
        // for GMT
        UTC = GMT0
    };

```

Month names: Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec and Inv_Month for an invalid.month value are the values of wxDateTime::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. 269) and GetWeekDayName (p. 270) 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 parameters is one of the following values:

```
enum Calendar
{
    Gregorian, // calendar currently in use in Western
countries
    Julian      // calendar in use since -45 until the 1582
(or later)
};
```

Date calculations often depend on the country and `wxDateTime` allows to set the country whose conventions should be used using `SetCountry` (p. 271). 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. 277) and `GetWeekOfMonth` (p. 277)).

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. **Error! Bookmark not defined.**), *wxTimeSpan* (p. **Error! Bookmark not defined.**), *wxDateSpan* (p. 255), *wxCalendarCtrl* (p. 127)

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. 271)
GetCountry (p. 269)
IsWestEuropeanCountry (p. 271)
GetCurrentYear (p. 269)
ConvertYearToBC (p. 268)
GetCurrentMonth (p. 269)
IsLeapYear (p. 271)
GetCentury (p. 269)
GetNumberOfDays (p. 270)
GetNumberOfDays (p. 270)
GetMonthName (p. 269)
GetWeekDayName (p. 270)
GetAmPmStrings (p. 268)
IsDSTApplicable (p. 271)
GetBeginDST (p. 268)
GetEndDST (p. 269)
Now (p. 271)
UNow (p. 272)
Today (p. 272)

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

wxDateTime() (p. 272)
wxDateTime(time_t) (p. 272)
wxDateTime(struct tm) (p. 272)

wxDateTime(double jdn) (p. 272)
wxDateTime(h, m, s, ms) (p. 273)
wxDateTime(day, mon, year, h, m, s, ms) (p. 273)
SetToCurrent (p. 273)
Set(time_t) (p. 273)
Set(struct tm) (p. 273)
Set(double jdn) (p. 273)
Set(h, m, s, ms) (p. 274)
Set(day, mon, year, h, m, s, ms) (p. 274)
SetFromDOS(unsigned long ddt) (p. 278)
ResetTime (p. 274)
SetYear (p. 274)
SetMonth (p. 274)
SetDay (p. 274)
SetHour (p. 275)
SetMinute (p. 275)
SetSecond (p. 275)
SetMillisecond (p. 275)
operator=(time_t) (p. 275)
operator=(struct tm) (p. 275)

Accessors

Here are the trivial accessors. Other functions, which might have to perform some more complicated calculations to find the answer are under the *Calendar calculations* (p. 267) section.

IsValid (p. 275)
GetTicks (p. 275)
GetYear (p. 276)
GetMonth (p. 276)
GetDay (p. 276)
GetWeekDay (p. 276)
GetHour (p. 276)
GetMinute (p. 276)
GetSecond (p. 276)
GetMillisecond (p. 276)
GetDayOfYear (p. 277)
GetWeekOfYear (p. 277)
GetWeekOfMonth (p. 277)
GetYearDay (p. 285)
IsWorkDay (p. 277)
IsGregorianDate (p. 277)
GetAsDOS (p. 278)

Date comparison

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

IsEqualTo (p. 278)

IsEarlierThan (p. 278)
IsLaterThan (p. 278)
IsStrictlyBetween (p. 278)
IsBetween (p. 278)
IsSameDate (p. 279)
IsSameTime (p. 279)
IsEqualUpTo (p. 279)

Date arithmetics

These functions carry out *arithmetics* (p. **Error! Bookmark not defined.**) 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. 279)
Add(wxDateSpan) (p. 279)
Subtract(wxTimeSpan) (p. 279)
Subtract(wxDateSpan) (p. 280)
Subtract(wxDateTime) (p. 280)
operator+=(wxTimeSpan) (p. 279)
operator+=(wxDateSpan) (p. 279)
operator-=(wxTimeSpan) (p. 279)
operator-=(wxDateSpan) (p. 280)

Parsing and formatting dates

These functions convert `wxDateTime` objects to and from text. The conversions to text are mostly trivial: you can either do it using the default date and time representations for the current locale (*FormatDate* (p. 282) and *FormatTime* (p. 282)), using the international standard representation defined by ISO 8601 (*FormatISODate* (p. 282) and *FormatISOTime* (p. 282)) or by specifying any format at all and using *Format* (p. 282) directly.

The conversions from text are more interesting, as there are much more possibilities to care about. The simplest cases can be taken care of with *ParseFormat* (p. 280) which can parse any date in the given (rigid) format. *ParseRfc822Date* (p. 280) 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. 282), hence the need for a separate function.

But the most interesting functions are *ParseTime* (p. 281), *ParseDate* (p. 281) and *ParseDateTime* (p. 281). 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. 281) can parse the strings such as "tomorrow", "March first" and even "next Sunday".

ParseRfc822Date (p. 280)

ParseFormat (p. 280)

ParseDateTime (p. 281)

ParseDate (p. 281)

ParseTime (p. 281)

Format (p. 282)

FormatDate (p. 282)

FormatTime (p. 282)

FormatISODate (p. 282)

FormatISOTime (p. 282)

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. 282)

GetWeekDayInSameWeek (p. 283)

SetToNextWeekDay (p. 283)

GetNextWeekDay (p. 283)

SetToPrevWeekDay (p. 283)

GetPrevWeekDay (p. 283)

SetToWeekDay (p. 283)

GetWeekDay (p. 284)

SetToLastWeekDay (p. 284)

GetLastWeekDay (p. 284)

SetToWeekOfYear (p. 284)

SetToLastMonthDay (p. 284)

GetLastMonthDay (p. 285)

SetToYearDay (p. 285)

GetYearDay (p. 285)

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 *aJDN* (p. 273) and you may also get its *JDN*, *MJD* (p. 285) or *Rata Die number* (p. 286) from it.

wxDateTime(double jdn) (p. 272)

Set(double jdn) (p. 273)

GetJulianDayNumber (p. 285)

GetJDN (p. 285)

GetModifiedJulianDayNumber (p. 285)

GetMJD (p. 286)

GetRataDie (p. 286)

Time zone and DST support

Please see the *time zone overview* (p. **Error! Bookmark not defined.**) for more information about time zones. Normally, these functions should be rarely used.

FromTimezone (p. 286)

ToTimezone (p. 286)

MakeTimezone (p. 286)

MakeFromTimezone (p. 286)

ToUTC (p. 287)

MakeUTC (p. 287)

GetBeginDST (p. 268)

GetEndDST (p. 269)

IsDST (p. 287)

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. **Error! Bookmark not defined.**).

See also

GetEndDST (p. 269)

wxDatetime::GetCountry

static Country GetCountry()

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

See also

SetCountry (p. 271)

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. 268)

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. 270)

wxDatetime::GetNumberOfDays

static wxDateTime_t GetNumberOfDays(int *year*, **Calendar** *cal* = *Gregorian*)

static wxDateTime_t GetNumberOfDays(Month *month*, int *year* = *Inv_Year*, **Calendar** *cal* = *Gregorian*)

Returns the number of days in the given year or in the given month of the year.

The only supported value for *cal* parameter is currently *Gregorian*.

wxPython note: These two methods are named *GetNumberOfDaysInYear* and *GetNumberOfDaysInMonth* in wxPython.

wxDatetime::GetTimeNow

static time_t GetTimeNow()

Returns the current time.

wxDatetime::GetTmNow

static struct tm * GetTmNow(struct tm **tm*)

Returns the current time broken down, uses the buffer whose address is passed to the function via *tm* to store the result.

wxDatetime::GetTmNow

static struct tm * GetTmNow()

Returns the current time broken down. Note that this function returns a pointer to a static buffer that's reused by calls to this function and certain C library functions (e.g. *localtime*). If there is any chance your code might be used in a multi-threaded application, you really should use the flavour of function *wxDatetime::GetTmNow* (p. 270) taking a parameter.

wxDatetime::GetWeekDayName

static wxString GetWeekDayName(WeekDay *weekday*, **NameFlags** *flags* = *Name_Full*)

Gets the full (default) or abbreviated (specify *Name_Abbbr*) name of the given week day.

See also

GetMonthName (p. 269)

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

See also

Today (p. 272)

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. 260).

See also

GetCountry (p. 269)

wxDateTime::Today

static wxDateTime Today()

Returns the object corresponding to the midnight of the current day (i.e. the same as *Now()* (p. 271), but the time part is set to 0).

See also

Now (p. 271)

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. 271)

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. 272).

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

wxDateTime::wxDateTime

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

Same as *Set* (p. 272)

wxPython note: Unsupported.

wxDateTime::wxDateTime

wxDateTime& wxDateTime(double *jdn*)

Same as *Set* (p. 272)

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. 273)

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

wxDateTime::wxDateTime

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

Same as *Set* (p. 274)

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. 271) 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. 272) and the time from supplied parameters.

wxPython note: This method is named `SetHMS` in wxPython.

wxDateTime::Set

wxDateTime& Set(wxDateTime_t day, Month month = Inv_Month, int year = Inv_Year, wxDateTime_t hour = 0, wxDateTime_t minute = 0, wxDateTime_t second = 0, wxDateTime_t millisec = 0)

Sets the date and time from the parameters.

wxDateTime::ResetTime

wxDateTime& ResetTime()

Reset time to midnight (00:00:00) without changing the date.

wxDateTime::SetYear

wxDateTime& SetYear(int year)

Sets the year without changing other date components.

wxDateTime::SetMonth

wxDateTime& SetMonth(Month month)

Sets the month without changing other date components.

wxDateTime::SetDay

wxDateTime& SetDay(wxDateTime_t day)

Sets the day without changing other date components.

wxDateTime::SetHour

wxDateTime& SetHour(wxDateTime_t hour)

Sets the hour without changing other date components.

wxDateTime::SetMinute

wxDateTime& SetMinute(wxDateTime_t minute)

Sets the minute without changing other date components.

wxDateTime::SetSecond

wxDateTime& SetSecond(wxDateTime_t second)

Sets the second without changing other date components.

wxDateTime::SetMillisecond

wxDateTime& SetMillisecond(wxDateTime_t millisecond)

Sets the millisecond without changing other date components.

wxDateTime::operator=

wxDateTime& operator(time_t timet)

Same as *Set* (p. 273).

wxDateTime::operator=

wxDateTime& operator(const struct tm& tm)

Same as *Set* (p. 273).

wxDateTime::IsValid

bool IsValid() const

Returns `true` if the object represents a valid time moment.

wxDateTime::GetTm

Tm GetTm(const TimeZone& tz = Local) const

Returns broken down representation of the date and time.

wxDateTime::GetTicks

time_t GetTicks() const

Returns the number of seconds since Jan 1, 1970. An assert failure will occur if the date is not in the range covered by `time_t` type.

wxDateTime::GetYear**int GetYear(const TimeZone& tz = Local) const**

Returns the year in the given timezone (local one by default).

wxDateTime::GetMonth**Month GetMonth(const TimeZone& tz = Local) const**

Returns the month in the given timezone (local one by default).

wxDateTime::GetDay**wxDateTime_t GetDay(const TimeZone& tz = Local) const**

Returns the day in the given timezone (local one by default).

wxDateTime::GetWeekDay**WeekDay GetWeekDay(const TimeZone& tz = Local) const**

Returns the week day in the given timezone (local one by default).

wxDateTime::GetHour**wxDateTime_t GetHour(const TimeZone& tz = Local) const**

Returns the hour in the given timezone (local one by default).

wxDateTime::GetMinute**wxDateTime_t GetMinute(const TimeZone& tz = Local) const**

Returns the minute in the given timezone (local one by default).

wxDateTime::GetSecond**wxDateTime_t GetSecond(const TimeZone& tz = Local) const**

Returns the seconds in the given timezone (local one by default).

wxDateTime::GetMillisecond**wxDateTime_t GetMillisecond(const TimeZone& tz = Local) const**

Returns the milliseconds in the given timezone (local one by default).

wxDateTime::GetDayOfYear

wxDateTime_t GetDayOfYear(const TimeZone& tz = Local) const

Returns the day of the year (in 1...366 range) in the given timezone (local one by default).

wxDateTime::GetWeekOfYear**wxDateTime_t GetWeekOfYear(WeekFlags flags = Monday_First, const TimeZone& tz = Local) const**

Returns the number of the week of the year this date is in. The first week of the year is, according to international standards, the one containing Jan 4 or, equivalently, the first week which has Thursday in this year. Both of these definitions are the same as saying that the first week of the year must contain more than half of its days in this year. Accordingly, the week number will always be in 1...53 range (52 for non leap years).

The function depends on the *week start* (p. 260) convention specified by the *flags* argument but its results for *Sunday_First* are not well-defined as the ISO definition quoted above applies to the weeks starting on Monday only.

wxDateTime::GetWeekOfMonth**wxDateTime_t GetWeekOfMonth(WeekFlags flags = Monday_First, const TimeZone& tz = Local) const**

Returns the ordinal number of the week in the month (in 1...5 range).

As *GetWeekOfYear* (p. 277), this function supports both conventions for the week start. See the description of these *week start* (p. 260) 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::SetFromDOS**wxDateTime& Set(unsigned long ddt)**

Sets the date from the date and time in DOS

(http://developer.novell.com/ndk/doc/smscomp/index.html?page=/ndk/doc/smscomp/sms_docs/data/hc2vlu5i.html)format.

wxDateTime::GetAsDOS**unsigned long GetAsDOS() const**

Returns the date and time inDOS

(http://developer.novell.com/ndk/doc/smscomp/index.html?page=/ndk/doc/smscomp/sms_docs/data/hc2vlu5i.html)format.

wxDateTime::IsEqualTo**bool IsEqualTo(const wxDateTime& *datetime*) const**Returns `true` if the two dates are strictly identical.**wxDateTime::IsEarlierThan****bool IsEarlierThan(const wxDateTime& *datetime*) const**Returns `true` if this date precedes the given one.**wxDateTime::IsLaterThan****bool IsLaterThan(const wxDateTime& *datetime*) const**Returns `true` if this date is later than the given one.**wxDateTime::IsStrictlyBetween****bool IsStrictlyBetween(const wxDateTime& *t1*, const wxDateTime& *t2*) const**Returns `true` if this date lies strictly between the two others,**See also***IsBetween* (p. 278)**wxDateTime::IsBetween****bool IsBetween(const wxDateTime& *t1*, const wxDateTime& *t2*) const**Returns `true` if *IsStrictlyBetween* (p. 278) is `true` or if the date is equal to one of the limit values.**See also***IsStrictlyBetween* (p. 278)**wxDateTime::IsSameDate****bool IsSameDate(const wxDateTime& *dt*) const**

Returns `true` if the date is the same without comparing the time parts.

wxDatetime::IsSameTime

bool IsSameTime(const wxDateTime& dt) const

Returns `true` if the time is the same (although dates may differ).

wxDatetime::IsEqualUpTo

bool IsEqualUpTo(const wxDateTime& dt, const wxTimeSpan& ts) const

Returns `true` if the date is equal to another one up to the given time interval, i.e. if the absolute difference between the two dates is less than this interval.

wxDatetime::Add

wxDatetime Add(const wxTimeSpan& diff) const

wxDatetime& Add(const wxTimeSpan& diff)

wxDatetime& operator+=(const wxTimeSpan& diff)

Adds the given time span to this object.

wxPython note: This method is named `AddTS` in wxPython.

wxDatetime::Add

wxDatetime Add(const wxDateSpan& diff) const

wxDatetime& Add(const wxDateSpan& diff)

wxDatetime& operator+=(const wxDateSpan& diff)

Adds the given date span to this object.

wxPython note: This method is named `AddDS` in wxPython.

wxDatetime::Subtract

wxDatetime Subtract(const wxTimeSpan& diff) const

wxDatetime& Subtract(const wxTimeSpan& diff)

wxDatetime& operator-=(const wxTimeSpan& diff)

Subtracts the given time span from this object.

wxPython note: This method is named `SubtractTS` in wxPython.

wxDatetime::Subtract

wxDateTime Subtract(const wxDateSpan& diff) const

wxDateTime& Subtract(const wxDateSpan& diff)

wxDateTime& operator-=(const wxDateSpan& diff)

Subtracts the given date span from this object.

wxPython note: This method is named `SubtractDS` in wxPython.

wxDateTime::Subtract

wxTimeSpan Subtract(const wxDateTime& dt) const

Subtracts another date from this one and returns the difference between them as `wxTimeSpan`.

wxDateTime::ParseRfc822Date

const wxChar * ParseRfc822Date(const wxChar* date)

Parses the string *date* looking for a date formatted according to the RFC 822 in it. The exact description of this format may, of course, be found in the RFC (section 5), but, briefly, this is the format used in the headers of Internet email messages and one of the most common strings expressing date in this format may be something like "Sat, 18 Dec 1999 00:48:30 +0100".

Returns `NULL` if the conversion failed, otherwise return the pointer to the character immediately following the part of the string which could be parsed. If the entire string contains only the date in RFC 822 format, the returned pointer will be pointing to a `NUL` character.

This function is intentionally strict, it will return an error for any string which is not RFC 822 compliant. If you need to parse date formatted in more free ways, you should use *ParseDateTime* (p. 281) or *ParseDate* (p. 281) instead.

wxDateTime::ParseFormat

const wxChar * ParseFormat(const wxChar *date, const wxChar *format = wxDefaultDateTimeFormat, const wxDateTime& dateDef = wxDefaultDateTime)

This function parses the string *date* according to the given *format*. The system `strptime(3)` function is used whenever available, but even if it is not, this function is still implemented, although support for locale-dependent format specifiers such as "%c", "%x" or "%X" may not be perfect and GNU extensions such as "%z" and "%Z" are not implemented. This function does handle the month and weekday names in the current locale on all platforms, however.

Please see the description of the ANSI C function `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. 272) 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. 280), 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. 281), but it only allows the date to be specified. It is thus less flexible than *ParseDateTime* (p. 281), 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 function is like *ParseDateTime* (p. 281), but only allows the time to be specified in the input string.

Returns `NULL` if the conversion failed, otherwise return the pointer to the character which stopped the scan.

wxDateTime::Format

wxString Format(const wxChar *format = wxDefaultDateTimeFormat, const TimeZone& tz = Local) const

This function does the same as the standard ANSI C `strftime(3)` function. Please see its description for the meaning of *format* parameter.

It also accepts a few wxWidgets-specific extensions: you can optionally specify the width of the field to follow using `printf(3)`-like syntax and the format specification `%l` can be used to get the number of milliseconds.

See also

ParseFormat (p. 280)

wxDatetime::FormatDate

wxString FormatDate() const

Identical to calling *Format()* (p. 282) with "%x" argument (which means 'preferred date representation for the current locale').

wxDatetime::FormatTime

wxString FormatTime() const

Identical to calling *Format()* (p. 282) with "%X" argument (which means 'preferred time representation for the current locale').

wxDatetime::FormatISODate

wxString FormatISODate() const

This function returns the date representation in the ISO 8601 format (YYYY-MM-DD).

wxDatetime::FormatISOTime

wxString FormatISOTime() const

This function returns the time representation in the ISO 8601 format (HH:MM:SS).

wxDatetime::SetToWeekDayInSameWeek

wxDatetime& SetToWeekDayInSameWeek(WeekDay weekday, WeekFlags flags = Monday_First)

Adjusts the date so that it will still lie in the same week as before, but its week day will be the given one.

Returns the reference to the modified object itself.

wxDatetime::GetWeekDayInSameWeek

wxDatetime GetWeekDayInSameWeek(WeekDay weekday, WeekFlags flags = Monday_First) const

Returns the copy of this object to which *SetToWeekDayInSameWeek* (p. 282) 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. 283) 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. 283) 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. 283) 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. 284) was applied.

wxDatetime::SetToWeekOfYear

static wxDateTime SetToWeekOfYear(int year, wxDateTime_t numWeek, WeekDay weekday = Mon)

Set the date to the given *weekday* in the week number *numWeek* of the given *year*. The number should be in range 1...53.

Note that the returned date may be in a different year than the one passed to this function because both the week 1 and week 52 or 53 (for leap years) contain days from different years. See *GetWeekOfYear* (p. 277) for the explanation of how the year weeks are counted.

wxDatetime::SetToLastMonthDay

wxDatetime& SetToLastMonthDay(Month month = Inv_Month, int year = Inv_Year)

Sets the date to the last day in the specified month (the current one by default).

Returns the reference to the modified object itself.

wxDatetime::GetLastMonthDay

wxDatetime GetLastMonthDay(Month month = Inv_Month, int year = Inv_Year)
const

Returns the copy of this object to which *SetToLastMonthDay* (p. 284) 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. 285) was applied.

wxDateTime::GetJulianDayNumber

double GetJulianDayNumber() const

Returns the *JDN* (p. 273) corresponding to this date. Beware of rounding errors!

See also

GetModifiedJulianDayNumber (p. 285)

wxDateTime::GetJDN

double GetJDN() const

Synonym for *GetJulianDayNumber* (p. 285).

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 the noons like JDN. The MJD 0 is Nov 17, 1858.

wxDateTime::GetMJD

double GetMJD() const

Synonym for *GetModifiedJulianDayNumber* (p. 285).

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::FromTimezone

wxDateTime FromTimezone(const TimeZone& tz, bool noDST = false) const

Transform the date from the given time zone to the local one. If *noDST* is `true`, no DST adjustments will be made.

Returns the date in the local time zone.

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::MakeFromTimezone

wxDateTime& MakeFromTimezone(const TimeZone& tz, bool noDST = false)

Same as *FromTimezone* (p. 286) but modifies the object in place.

wxDateTime::ToUTC

wxDateTime ToUTC(bool noDST = false) const

This is the same as calling *ToTimezone* (p. 286) with the argument `GMT0`.

wxDateTime::MakeUTC

wxDateTime& MakeUTC(bool noDST = false)

This is the same as calling *MakeTimezone* (p. 286) 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. 268) and *GetEndDST* (p. 269)

wxDateTimeHolidayAuthority

TODO

wxDateTimeWorkDays

TODO

wxDB

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

See the *database classes overview* (p. **Error! Bookmark not defined.**) for an introduction to using the ODBC classes.

Include files

<wx/db.h>

Helper classes and data structures

The following classes and structs are defined in db.cpp/.h for use with the wxDb class.

- *wxDbColFor* (p. 321)
- *wxDbColInf* (p. 322)
- *wxDbTableInf* (p. 367)
- *wxDbInf* (p. 329)

Constants

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

wxDB_PATH_MAX passed to data	Maximum path length allowed to be the ODBC driver to indicate where the file(s) are located.
DB_MAX_COLUMN_NAME_LEN of a	Maximum supported length for the name column
DB_MAX_ERROR_HISTORY retained in new	Maximum number of error messages the queue before being overwritten by errors.

DB_MAX_ERROR_MSG_LEN message	Maximum supported length of an error returned by the ODBC classes
DB_MAX_STATEMENT_LEN complete SQL driver	Maximum supported length for a statement to be passed to the ODBC
DB_MAX_TABLE_NAME_LEN of a	Maximum supported length for the name table
DB_MAX_WHERE_CLAUSE_LEN that	Maximum supported WHERE clause length can be passed to the ODBC driver
DB_TYPE_NAME_LEN column's	Maximum length of the name of a data type

Enumerated types

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. 299) will return one of these enumerated values listed below.

- DB2
- DBase (IV, V)**
- Firebird
- INFORMIX
- Interbase
- MS SQL Server (v7 - minimal testing)
- MS Access (97, 2000, 2002, and 2003)
- MySQL (2.x and 3.5 - use the 2.5x drivers though)
- Oracle (v7, v8, v8i)
- Pervasive SQL
- PostgreSQL
- Sybase (ASA and ASE)

- XBase Sequiter
- VIRTUOSO

See the remarks in *wxD::Dbms* (p. 299) 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 *wxD::GetNextError* (p. 308). Contains the count of bytes in the *wxD::errorMsg* string.

int wxDb::DB_STATUS

The last ODBC error/status that occurred on this data connection. Possible codes are:

DB_ERR_GENERAL_WARNING	// SqlState =
'01000'	
DB_ERR_DISCONNECT_ERROR	// SqlState =
'01002'	
DB_ERR_DATA_TRUNCATED	// SqlState =
'01004'	
DB_ERR_PRIV_NOT_REVOKED	// SqlState =
'01006'	
DB_ERR_INVALID_CONN_STR_ATTR	// SqlState =
'01S00'	
DB_ERR_ERROR_IN_ROW	// SqlState =
'01S01'	
DB_ERR_OPTION_VALUE_CHANGED	// SqlState =
'01S02'	
DB_ERR_NO_ROWS_UPD_OR_DEL	// SqlState =
'01S03'	
DB_ERR_MULTI_ROWS_UPD_OR_DEL	// SqlState =
'01S04'	
DB_ERR_WRONG_NO_OF_PARAMS	// SqlState =
'07001'	
DB_ERR_DATA_TYPE_ATTR_VIOL	// SqlState =
'07006'	
DB_ERR_UNABLE_TO_CONNECT	// SqlState =
'08001'	
DB_ERR_CONNECTION_IN_USE	// SqlState =
'08002'	
DB_ERR_CONNECTION_NOT_OPEN	// SqlState =
'08003'	
DB_ERR_REJECTED_CONNECTION	// SqlState =
'08004'	
DB_ERR_CONN_FAIL_IN_TRANS	// SqlState =
'08007'	
DB_ERR_COMM_LINK_FAILURE	// SqlState =
'08S01'	
DB_ERR_INSERT_VALUE_LIST_MISMATCH	// SqlState =
'21S01'	
DB_ERR_DERIVED_TABLE_MISMATCH	// SqlState =
'21S02'	
DB_ERR_STRING_RIGHT_TRUNC	// SqlState =
'22001'	
DB_ERR_NUMERIC_VALUE_OUT_OF_RNG	// SqlState =
'22003'	

```

    DB_ERR_ERROR_IN_ASSIGNMENT                // SqlState =
'22005'
    DB_ERR_DATETIME_FLD_OVERFLOW              // SqlState =
'22008'
    DB_ERR_DIVIDE_BY_ZERO                     // SqlState =
'22012'
    DB_ERR_STR_DATA_LENGTH_MISMATCH           // SqlState =
'22026'
    DB_ERR_INTEGRITY_CONSTRAINT_VIOL          // SqlState =
'23000'
    DB_ERR_INVALID_CURSOR_STATE               // SqlState =
'24000'
    DB_ERR_INVALID_TRANS_STATE               // SqlState =
'25000'
    DB_ERR_INVALID_AUTH_SPEC                 // SqlState =
'28000'
    DB_ERR_INVALID_CURSOR_NAME               // SqlState =
'34000'
    DB_ERR_SYNTAX_ERROR_OR_ACCESS_VIOL        // SqlState =
'37000'
    DB_ERR_DUPLICATE_CURSOR_NAME             // SqlState =
'3C000'
    DB_ERR_SERIALIZATION_FAILURE              // SqlState =
'40001'
    DB_ERR_SYNTAX_ERROR_OR_ACCESS_VIOL2      // SqlState =
'42000'
    DB_ERR_OPERATION_ABORTED                 // SqlState =
'70100'
    DB_ERR_UNSUPPORTED_FUNCTION               // SqlState =
'IM001'
    DB_ERR_NO_DATA_SOURCE                    // SqlState =
'IM002'
    DB_ERR_DRIVER_LOAD_ERROR                 // SqlState =
'IM003'
    DB_ERR_SQLALLOCENV_FAILED                // SqlState =
'IM004'
    DB_ERR_SQLALLOCCONNECT_FAILED            // SqlState =
'IM005'
    DB_ERR_SQLSETCONNECTOPTION_FAILED         // SqlState =
'IM006'
    DB_ERR_NO_DATA_SOURCE_DLG_PROHIB         // SqlState =
'IM007'
    DB_ERR_DIALOG_FAILED                     // SqlState =
'IM008'
    DB_ERR_UNABLE_TO_LOAD_TRANSLATION_DLL     // SqlState =
'IM009'
    DB_ERR_DATA_SOURCE_NAME_TOO_LONG         // SqlState =
'IM010'
    DB_ERR_DRIVER_NAME_TOO_LONG              // SqlState =
'IM011'
    DB_ERR_DRIVER_KEYWORD_SYNTAX_ERROR        // SqlState =
'IM012'
    DB_ERR_TRACE_FILE_ERROR                  // SqlState =
'IM013'
    DB_ERR_TABLE_OR_VIEW_ALREADY_EXISTS       // SqlState =
'S0001'
    DB_ERR_TABLE_NOT_FOUND                   // SqlState =
'S0002'
    DB_ERR_INDEX_ALREADY_EXISTS               // SqlState =
'S0011'
    DB_ERR_INDEX_NOT_FOUND                   // SqlState =
'S0012'
    DB_ERR_COLUMN_ALREADY_EXISTS              // SqlState =
'S0021'

```

```

        DB_ERR_COLUMN_NOT_FOUND                // SqlState =
'S0022'
        DB_ERR_NO_DEFAULT_FOR_COLUMN           // SqlState =
'S0023'
        DB_ERR_GENERAL_ERROR                   // SqlState =
'S1000'
        DB_ERR_MEMORY_ALLOCATION_FAILURE        // SqlState =
'S1001'
        DB_ERR_INVALID_COLUMN_NUMBER           // SqlState =
'S1002'
        DB_ERR_PROGRAM_TYPE_OUT_OF_RANGE       // SqlState =
'S1003'
        DB_ERR_SQL_DATA_TYPE_OUT_OF_RANGE      // SqlState =
'S1004'
        DB_ERR_OPERATION_CANCELLED              // SqlState =
'S1008'
        DB_ERR_INVALID_ARGUMENT_VALUE          // SqlState =
'S1009'
        DB_ERR_FUNCTION_SEQUENCE_ERROR         // SqlState =
'S1010'
        DB_ERR_OPERATION_INVALID_AT_THIS_TIME  // SqlState =
'S1011'
        DB_ERR_INVALID_TRANS_OPERATION_CODE     // SqlState =
'S1012'
        DB_ERR_NO_CURSOR_NAME_AVAIL            // SqlState =
'S1015'
        DB_ERR_INVALID_STR_OR_BUF_LEN          // SqlState =
'S1090'
        DB_ERR_DESCRIPTOR_TYPE_OUT_OF_RANGE     // SqlState =
'S1091'
        DB_ERR_OPTION_TYPE_OUT_OF_RANGE         // SqlState =
'S1092'
        DB_ERR_INVALID_PARAM_NO                // SqlState =
'S1093'
        DB_ERR_INVALID_SCALE_VALUE             // SqlState =
'S1094'
        DB_ERR_FUNCTION_TYPE_OUT_OF_RANGE       // SqlState =
'S1095'
        DB_ERR_INF_TYPE_OUT_OF_RANGE            // SqlState =
'S1096'
        DB_ERR_COLUMN_TYPE_OUT_OF_RANGE         // SqlState =
'S1097'
        DB_ERR_SCOPE_TYPE_OUT_OF_RANGE          // SqlState =
'S1098'
        DB_ERR_NULLABLE_TYPE_OUT_OF_RANGE       // SqlState =
'S1099'
        DB_ERR_UNIQUENESS_OPTION_TYPE_OUT_OF_RANGE // SqlState =
'S1100'
        DB_ERR_ACCURACY_OPTION_TYPE_OUT_OF_RANGE // SqlState =
'S1101'
        DB_ERR_DIRECTION_OPTION_OUT_OF_RANGE    // SqlState =
'S1103'
        DB_ERR_INVALID_PRECISION_VALUE          // SqlState =
'S1104'
        DB_ERR_INVALID_PARAM_TYPE              // SqlState =
'S1105'
        DB_ERR_FETCH_TYPE_OUT_OF_RANGE          // SqlState =
'S1106'
        DB_ERR_ROW_VALUE_OUT_OF_RANGE           // SqlState =
'S1107'
        DB_ERR_CONCURRENCY_OPTION_OUT_OF_RANGE  // SqlState =
'S1108'
        DB_ERR_INVALID_CURSOR_POSITION          // SqlState =
'S1109'

```

```

        DB_ERR_INVALID_DRIVER_COMPLETION           // SqlState =
'S1110'
        DB_ERR_INVALID_BOOKMARK_VALUE             // SqlState =
'S1111'
        DB_ERR_DRIVER_NOT_CAPABLE                 // SqlState =
'S1C00'
        DB_ERR_TIMEOUT_EXPIRED                   // SqlState =
'S1T00'

```

struct **wxDb::dbInf**

This structure is internal to the wxDb class and contains details of the ODBC datasource that the current instance of the wxDb is connected to in its members. When the datasource is opened, all of the information contained in the dbInf structure is queried from the datasource. This information is used almost exclusively within the ODBC class library. Where there may be a need for particular portions of this information outside of the class library, member functions (e.g. *wxDbTable::IsCursorClosedOnCommit* (p. 350)) have been added for ease of use.

```

        wxChar dbmsName[40]           - Name of the dbms product
        wxChar dbmsVer[64]            - Version # of the dbms product
        wxChar driverName[40]         - Driver name
        wxChar odbcVer[60]            - ODBC version of the driver
        wxChar drvMgrOdbcVer[60]      - ODBC version of the driver manager
        wxChar driverVer[60]          - Driver version
        wxChar serverName[80]         - Server Name, typically a connect
string
        wxChar databaseName[128]      - Database filename
        wxChar outerJoins[2]          - Does datasource support outer
joins
        wxChar procedureSupport[2]    - Does datasource support stored
procedures
        UWORD maxConnections          - Maximum # of connections
datasource
                                supports
        UWORD maxStmts                - Maximum # of HSTMTs per HDBC
        UWORD apiConfLvl              - ODBC API conformance level
        UWORD cliConfLvl              - Is datasource SAG compliant
        UWORD sqlConfLvl              - SQL conformance level
        UWORD cursorCommitBehavior    - How cursors are affected on db
commit
        UWORD cursorRollbackBehavior - How cursors are affected on db
rollback
        UWORD supportNotNullClause    - Does datasource support NOT
NULL
                                clause
        wxChar supportIEF[2]          - Integrity Enhancement Facility
(Ref.
                                Integrity)
        UDWORD txnIsolation           - Transaction isolation level
supported by
                                driver
        UDWORD txnIsolationOptions    - Transaction isolation level
options
                                available
        UDWORD fetchDirections        - Fetch directions supported
        UDWORD lockTypes              - Lock types supported in SQLSetPos
        UDWORD posOperations          - Position operations supported in
SQLSetPos
        UDWORD posStmts               - Position statements supported
        UDWORD scrollConcurrency       - Scrollable cursor concurrency

```


options		
UDWORD scrollOptions	supported	- Scrollable cursor options
UDWORD staticSensitivity		- Can additions/deletions/updates be detected
UWORD txnCapable		- Indicates if datasource supports transactions
UDWORD loginTimeout		- Number seconds to wait for a login request

wxChar wxDb::errorList[DB_MAX_ERROR_HISTORY][DB_MAX_ERROR_MSG_LEN]

The last n ODBC errors that have occurred on this database connection.

wxChar wxDb::errorMsg[SQL_MAX_MESSAGE_LENGTH]

This member variable is populated as a result of calling *wxDb::GetNextError* (p. 308). It contains the ODBC error message text.

SDWORD wxDb::nativeError

Set by *wxDb::DispAllErrors*, *wxDb::GetNextError*, and *wxDb::DispNextError*. It contains the datasource-specific error code returned by the datasource to the ODBC driver. Used for reporting ODBC errors.

wxChar wxDb::sqlState[20]

Set by *wxDb::TranslateSqlState()*. Indicates the error state after a failed ODBC operation. Used for reporting ODBC errors.

Remarks

Default cursor scrolling is defined by *wxODBC_FWD_ONLY_CURSORS* in *setup.h* when the *wxWidgets* library is built. This behavior can be overridden when an instance of a *wxDb* is created (see *wxDb constructor* (p. 296)). Default setting of this value true, as not all databases/drivers support both types of cursors.

See also

wxDbColFor (p. 321), *wxDbColInf* (p. 322), *wxDbTable* (p. 329), *wxDbTableInf* (p. 367), *wxDbInf* (p. 329)

Associated non-class functions

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

void wxDbCloseConnections()

Remarks

Closes all cached connections that have been made through use of the *wxDbGetConnection* (p. 294) function.

NOTE: These connections are closed regardless of whether they are in use or not. This function should only be called after the program has finished using the connections and

all *wxDbTable* instances that use any of the connections have been closed.

This function performs a *wxDb::CommitTrans* (p. 298) on the connection before closing it to commit any changes that are still pending, as well as to avoid any function sequence errors upon closing each connection.

int wxDbConnectionsInUse()

Remarks

Returns a count of how many database connections are currently free (not being used) that have been cached through use of the *wxDbGetConnection* (p. 294) function.

bool wxDbFreeConnection(wxDb *pDb)

Remarks

Searches the list of cached database connections connection for one matching the passed in *wxDb* instance. If found, that cached connection is freed.

Freeing a connection means that it is marked as available (free) in the cache of connections, so that a call to *wxDbGetConnection* (p. 294) is able to return a pointer to the *wxDb* instance for use. Freeing a connection does NOT close the connection, it only makes the connection available again.

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

Remarks

This function is used to request a "new" *wxDb* instance for use by the program. The *wxDb* instance returned is also opened (see *wxDb::Open* (p. 312)).

This function (along with *wxDbFreeConnection()* and *wxDbCloseConnection()*) maintain a cache of *wxDb* instances for user/re-use by a program. When a program needs a *wxDb* instance, it may call this function to obtain a *wxDb* instance. If there is a *wxDb* instance in the cache that is currently unused that matches the connection requirements specified in '*pDbConfig*' then that cached connection is marked as no longer being free, and a pointer to the *wxDb* instance is returned.

If there are no connections available in the cache that meet the requirements given in '*pDbConfig*', then a new *wxDb* instance is created to connect to the datasource specified in '*pDbConfig*' using the userID and password given in '*pDbConfig*'.

NOTE: The caching routine also uses the *wxDb::Open* (p. 312) connection datatype copying code. If the call to *wxDbGetConnection()* requests a connection to a datasource, and there is not one available in the cache, a new connection is created. But when the connection is opened, instead of polling the datasource over again for its datatypes, if a connection to the same datasource (using the same userID/password) has already been done previously, the new connection skips querying the datasource for its datatypes, and uses the same datatypes determined previously by the other connection(s) for that same datasource. This cuts down greatly on network traffic, database load, and connection creation time.

When the program is done using a connection created through a call to `wxDdbGetConnection()`, the program should call `wxDdbFreeConnection()` to release the `wxDdb` instance back to the cache. DO NOT DELETE THE `wxDdb` INSTANCE! Deleting the `wxDdb` instance returned can cause a crash/memory corruption later in the program when the cache is cleaned up.

When exiting the program, call `wxDdbCloseConnections()` to close all the cached connections created by calls to `wxDdbGetConnection()`.

const wxChar * wxDbLogExtendedErrorMsg(const wxChar *userText, wxDb *pDb, wxChar *ErrFile, int ErrLine)

Writes a message to the `wxLog` window (stdout usually) when an internal error situation occurs.

bool wxDbSqlLog(wxDdbSqlLogState state, const wxString &filename = SQL_LOG_FILENAME)

Remarks

This function sets the sql log state for all open `wxDdb` objects

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

Remarks

This routine queries the ODBC driver manager for a list of available datasources. Repeatedly call this function to obtain all the datasources available through the ODBC driver manager on the current workstation.

```
wxArrayString strArray;

while (wxDdbGetDataSource(DbConnectInf.GetHenv(), Dsn,
SQL_MAX_DSN_LENGTH+1, DsDesc, 255))
    strArray.Add(Dsn);
```

wxDdb::wxDdb

wxDdb()

Default constructor.

wxDdb(const HENV &aHenv, bool FwdOnlyCursors=(bool)wxODBC_FWD_ONLY_CURSORS)

Constructor, used to create an ODBC connection to a datasource.

Parameters

aHenv

Environment handle used for this connection. See `wxDConnectInf::AllocHenv` (p.

325)

FwdOnlyCursors

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

Remarks

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

Example

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

wxDb sampleDB(ConnectInf.GetHenv());
if (!sampleDB.Open(ConnectInf.GetDsn(), ConnectInf.GetUserID(),
                  ConnectInf.GetPassword()))
{
    // Error opening datasource
}
```

See also

wxDbGetConnection (p. 294)

wxDb::Catalog

```
bool Catalog(wxChar * userID, const wxString &fileName =
SQL_CATALOG_FILENAME)
```

Allows a data "dictionary" of the datasource to be created, dumping pertinent information about all data tables to which the user specified in *userID* has access.

Parameters

userID

Database user name to use in accessing the database. All tables to which this user has rights will be evaluated in the catalog.

fileName

OPTIONAL. Name of the text file to create and write the DB catalog to. Default is SQL_CATALOG_FILENAME.

Return value

Returns true if the catalog request was successful, or false if there was some reason that the catalog could not be generated.

Example

```
=====
```

TABLE NAME	COLUMN NAME	DATA TYPE	PRECISION	LENGTH
=====	=====	=====	=====	=====
EMPLOYEE	RECID	(0008)NUMBER	15	8
EMPLOYEE	USER_ID	(0012)VARCHAR2	13	13
EMPLOYEE	FULL_NAME	(0012)VARCHAR2	26	26
EMPLOYEE	PASSWORD	(0012)VARCHAR2	26	26
EMPLOYEE	START_DATE	(0011)DATE	19	16

wxDb::Close

void Close()

Closes the database connection.

Remarks

At the end of your program, when you have finished all of your database work, you must close the ODBC connection to the datasource. There are actually four steps involved in doing this as illustrated in the example.

Any wxDbTable instances which use this connection must be deleted before closing the database connection.

Example

```
// Commit any open transactions on the datasource
sampleDB.CommitTrans();

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

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

wxDb::CommitTrans

bool CommitTrans()

Permanently "commits" changes (insertions/deletions/updates) to the database.

Return value

Returns true if the commit was successful, or false if the commit failed.

Remarks

Transactions begin implicitly as soon as you make a change to the database with an insert/update/delete, or any other direct SQL command that performs one of these operations against the datasource. At any time thereafter, to save the changes to disk permanently, "commit" them by calling this function.

Calling this member function commits ALL open transactions on this ODBC connection. For example, if three different wxDbTable instances used the same connection to the datasource, committing changes made on one of those wxDbTable instances commits any pending transactions on all three wxDbTable instances.

Until a call to `wxDdb::CommitTrans()` is made, no other user or cursor is able to see any changes made to the row(s) that have been inserted/modified/deleted.

Special Note : *Cursors*

It is important to understand that different database/ODBC driver combinations handle transactions differently. One thing in particular that you must pay attention to is cursors, in regard to transactions. Cursors are what allow you to scroll through records forward and backward and to manipulate records as you scroll through them. When you issue a query, a cursor is created behind the scenes. The cursor keeps track of the query and keeps track of the current record pointer. After you commit or rollback a transaction, the cursor may be closed automatically. This is database dependent, and with some databases this behavior can be controlled through management functions. This means you would need to requery the datasource before you can perform any additional work using this cursor. This is only necessary however if the datasource closes the cursor after a commit or rollback. Use the `wxDdbTable::IsCursorClosedOnCommit` (p. 350) member function to determine the datasource's transaction behavior. Note, in many situations it is very inefficient to assume the cursor is closed and always requery. This could put a significant, unnecessary load on datasources that leave the cursors open after a transaction.

wxDdb::CreateView

bool CreateView(const wxString & viewName, const wxString & colList, const wxString & pSqlStmt)

Creates a SQL VIEW of one or more tables in a single datasource. Note that this function will only work against databases which support views (currently only Oracle as of November 21 2000).

Parameters

viewName

The name of the view. e.g. PARTS_V

colList

OPTIONAL Pass in a comma delimited list of column names if you wish to explicitly name each column in the result set. If not desired, pass in an empty string and the column names from the associated table(s) will be used.

pSqlStmt

Pointer to the select statement portion of the CREATE VIEW statement. Must be a complete, valid SQL SELECT statement.

Remarks

A 'view' is a logical table that derives columns from one or more other tables or views. Once the view is created, it can be queried exactly like any other table in the database.

NOTE: Views are not available with all datasources. Oracle is one example of a

datasource which does support views.

Example

```
// Incomplete code sample
db.CreateView("PARTS_SD1", "PN, PD, QTY",
              "SELECT PART_NUM, PART_DESC, QTY_ON_HAND * 1.1
FROM PARTS \
              WHERE STORAGE_DEVICE = 1");

// PARTS_SD1 can now be queried just as if it were a data
table.
// e.g. SELECT PN, PD, QTY FROM PARTS_SD1
```

wxDb::Dbms

wxDBMS Dbms()

Remarks

The return value will be of the enumerated type wxDBMS. This enumerated type contains a list of all the currently tested and supported databases.

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

Possible values returned by this function can be viewed in the *Enumerated types* (p. 288) section of wxDb.

There are known issues with conformance to the ODBC standards with several datasources supported by the wxWidgets ODBC classes. Please see the overview for specific details on which datasources have which issues.

Return value

The return value will indicate which of the supported datasources is currently connected to by this connection. In the event that the datasource is not recognized, a value of 'dbmsUNIDENTIFIED' is returned.

wxDb::DispAllErrors

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

Used to log all database errors that occurred as a result of an executed database command. This logging is automatic and also includes debug logging when compiled in debug mode via *wxLogDebug* (p. **Error! Bookmark not defined.**). If logging is turned on via *wxDb::SetSqlLogging* (p. 315), then an entry is also logged to the defined log file.

Parameters

aHenv

Handle to the ODBC environment.

aHdbc

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

aHstmt

Handle to the ODBC statement being executed against. Pass this in if the ODBC function call that failed required a hstmt argument.

Remarks

This member function will log all of the ODBC error messages for the last ODBC function call that was made. This function is normally used internally within the ODBC class library, but can be used programmatically after calling ODBC functions directly (i.e. `SQLFreeEnv()`).

Return value

The function always returns false, so a call to this function can be made in the return statement of a code block in the event of a failure to perform an action (see the example below).

See also

`wxDb::SetSqlLogging` (p. 315), `wxDbSqlLog`

Example

```
if (SQLExecDirect(hstmt, (UCHAR FAR *) pSqlStmt, SQL_NTS) !=
    SQL_SUCCESS)
    // Display all ODBC errors for this stmt
    return(db.DispAllErrors(db.henv, db.hdbc, hstmt));
```

`wxDb::DispNextError`**`void DispNextError()`****Remarks**

This function is normally used internally within the ODBC class library. It could be used programmatically after calling ODBC functions directly. This function works in conjunction with `wxDb::GetNextError` (p. 308) when errors (or sometimes informational messages) returned from ODBC need to be analyzed rather than simply displaying them as an error. `GetNextError()` retrieves the next ODBC error from the ODBC error queue. The `wxDb` member variables "sqlState", "nativeError" and "errorMsg" could then be evaluated. To display the error retrieved, `DispNextError()` could then be called. The combination of `GetNextError()` and `DispNextError()` can be used to iteratively step through the errors returned from ODBC evaluating each one in context and displaying the ones you choose.

Example

```
// Drop the table before attempting to create it
sprintf(sqlStmt, "DROP TABLE %s", tableName);
```



```
// Execute the drop table statement
if (SQLExecDirect(hstmt, (UCHAR FAR *)sqlStmt, SQL_NTS) !=
SQL_SUCCESS)
{
    // Check for sqlState = S0002, "Table or view not found".
    // Ignore this error, bomb out on any other error.
    pDb->GetNextError(henv, hdbc, hstmt);
    if (wxStrcmp(pDb->sqlState, "S0002"))
    {
        pDb->DispNextError(); // Displayed error retrieved
        pDb->DispAllErrors(henv, hdbc, hstmt); // Display all
other errors, if any
        pDb->RollbackTrans(); // Rollback the transaction
        CloseCursor();        // Close the cursor
        return(false);        // Return Failure
    }
}
```

wxDp::DropView

bool DropView(const wxString &viewName)

Drops the data table view named in 'viewName'.

Parameters

viewName

Name of the view to be dropped.

Remarks

If the view does not exist, this function will return true. Note that views are not supported with all datasources.

wxDp::ExecSql

bool ExecSql(const wxString &pSqlStmt)

bool ExecSql(const wxString &pSqlStmt, wxDbCollnf **columns, short &numcols)

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.

To get column amount and column names or other information about returned columns, pass '*columns*' and '*numcols*' parameters to the function also.

Parameters

pSqlStmt

Pointer to the SQL statement to be executed.

columns

On success, this function will set this pointer to point to array of *wxDboColInf* (p. 322) objects, holding information about columns returned by the query. You need to call *delete[]* for the pointer you pass here after you don't use it anymore to prevent memory leak.

numcols

Reference to variable where amount of objects in '*columns*'-parameter will be set.

Remarks

This member extends the *wxDbo* class and allows you to build and execute ANY VALID SQL statement against the datasource. This allows you to extend the class library by being able to issue any SQL statement that the datasource is capable of processing.

See also

wxDbo::GetData (p. 305), *wxDbo::GetNext* (p. 308)

wxDbo::FwdOnlyCursors

bool IsFwdOnlyCursors()

Older form (pre-2.3/2.4 of *wxWidgets*) of the *wxDbo::IsFwdOnlyCursors* (p. 310). This method is provided for backward compatibility only. The method *wxDbo::IsFwdOnlyCursors* (p. 310) should be used in place of this method.

wxDboInf * GetCatalog(const wxChar *userID)

wxDbo::GetCatalog

wxDboInf * GetCatalog(const wxChar *userID)

Returns a *wxDboInf* (p. 329) pointer that points to the catalog (datasource) name, schema, number of tables accessible to the current user, and a *wxDboTableInf* pointer to all data pertaining to all tables in the users catalog.

Parameters

userID

Owner/Schema of the table. Specify a *userID* when the datasource you are connected to allows multiple unique tables with the same name to be owned by different users. *userID* is evaluated as follows:

```
userID == NULL    ... UserID is ignored (DEFAULT)
userID == ""      ... UserID set equal to 'this->uid'
userID != ""      ... UserID set equal to 'userID'
```

Remarks

The returned catalog will only contain catalog entries for tables to which the user

specified in 'userID' has sufficient privileges. If no user is specified (NULL passed in), a catalog pertaining to all tables in the datasource accessible to the connected user (permissions apply) via this connection will be returned.

wxDb::GetColumnCount

int GetColumnCount(const wxString &tableName, const wxChar *userID)

Parameters

tableName

The table name you wish to obtain column information about.

userID

Name of the user that owns the table(s) (also referred to as schema). Required for some datasources for situations where there may be multiple tables with the same name in the datasource, but owned by different users. *userID* is evaluated in the following manner:

```
userID == NULL    ... UserID is ignored (DEFAULT)
userID == ""      ... UserID set equal to 'this->uid'
userID != ""      ... UserID set equal to 'userID'
```

Return value

Returns a count of how many columns are in the specified table. If an error occurs retrieving the number of columns, this function will return a -1.

wxDb::GetColumns

wxDbColInf * GetColumns(const wxString &tableName, UWORD *numCols, const wxChar *userID=NULL)

wxDbColInf * GetColumns(wxChar *tableName[], const wxChar *userID)

Parameters

tableName

The table name you wish to obtain column information about.

numCols

Pointer to a UWORD which will hold a count of the number of columns returned by this function

tableName[]

An array of pointers to table names you wish to obtain column information about. The last element of this array must be a NULL string.

userID

Name of the user that owns the table(s) (also referred to as schema). Required for some datasources for situations where there may be multiple tables with the same name in the datasource, but owned by different users. *userID* is evaluated in the following manner:

```
userID == NULL    ... UserID is ignored (DEFAULT)
userID == ""      ... UserID set equal to 'this->uid'
userID != ""      ... UserID set equal to 'userID'
```

Return value

This function returns a pointer to an array of *wxDdbColInf* (p. 322) structures, allowing you to obtain information regarding the columns of the named table(s). If no columns were found, or an error occurred, this pointer will be NULL.

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

ALL column bindings associated with this *wxDdb* instance are unbound by this function, including those used by any *wxDdbTable* instances that use this *wxDdb* instance. This function should use its own *wxDdb* instance to avoid undesired unbinding of columns.

See also

wxDdbColInf (p. 322)

Example

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

wxDdb::GetData

bool **GetData**(**UWORD** *colNumber*, **SWORD** *cType*, **PTR** *pData*, **SDWORD** *maxLen*, **SDWORD FAR** * *cbReturned*)

Used to retrieve result set data without binding column values to memory variables (i.e. not using a *wxDdbTable* instance to access table data).

Parameters

colNumber

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

cType

The C data type that is to be returned. See a partial list in *wxDdbTable::SetColDefs*

(p. 358)

pData

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

maxLen

Maximum size of the buffer *pData* in characters. NOTE: Not UNICODE safe. If this is a numeric field, a value of 0 may be passed for this parameter, as the API knows the size of the expected return value.

cbReturned

Pointer to the buffer containing the length of the actual data returned. If this value comes back as SQL_NULL_DATA, then the *wxDdb::GetData* (p. 305) call has failed.

See also

wxDdb::GetNext (p. 308), *wxDdb::ExecSql* (p. 302)

Example

```
SDWORD cb;
ULONG reqQty;
wxString sqlStmt;
sqlStmt = "SELECT SUM(REQUIRED_QTY - PICKED_QTY) FROM
ORDER_TABLE WHERE \
          PART_RECID = 1450 AND REQUIRED_QTY > PICKED_QTY";

// Perform the query
if (!pDb->ExecSql(sqlStmt.c_str()))
{
    // ERROR
    return(0);
}

// Request the first row of the result set
if (!pDb->GetNext())
{
    // ERROR
    return(0);
}

// Read column #1 of the row returned by the call to
::GetNext()
// and return the value in 'reqQty'
if (!pDb->GetData(1, SQL_C_ULONG, &reqQty, 0, &cb))
{
    // ERROR
    return(0);
}

// Check for a NULL result
if (cb == SQL_NULL_DATA)
    return(0);
```

Remarks

When requesting multiple columns to be returned from the result set (for example, the

SQL query requested 3 columns be returned), the calls to this function must request the columns in ordinal sequence (1,2,3 or 1,3 or 2,3).

wxDdb::GetDatabaseName

const wxChar * GetDatabaseName()

Returns the name of the database engine.

wxDdb::GetDatasourceName

const wxString & GetDatasourceName()

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(const wxString &tableName, wxDbCollnf *collnf, UWORD numColumns)

Used to determine which columns are members of primary or non-primary indexes on the specified table. If a column is a member of a foreign key for some other table, that information is detected also.

This function is primarily for use by the *wxDdb::GetColumns* (p. 304) 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. 304)). This function populates the *PkCol*, *PkTableName*, and *FkTableName* members of the *collnf* structure.

numColumns

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

Return value

Currently always returns true.

See also

wxDdbCollnf (p. 322), *wxDdb::GetColumns* (p. 304)

wxDdb::GetNext**bool GetNext()**

Called after executing a query, this function requests the next row in the result set after the current position of the cursor.

See also

wxDdb::ExecSql (p. 302), *wxDdb::GetData* (p. 305)

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

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

aHstmt

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

Example

```
if (SQLExecDirect(hstmt, (UCHAR FAR *) pSqlStmt, SQL_NTS) !=
    SQL_SUCCESS)
{
```

```
        return(db.GetNextError(db.henv, db.hdbc, hstmt));  
    }
```

See also

wxDdb::DispNextError (p. 301), *wxDdb::DispAllErrors* (p. 300)

wxDdb::GetPassword

const wxString & GetPassword()

Returns the password used to establish this connection to the datasource.

wxDdb::GetTableCount

int GetTableCount()

Returns the number of *wxDdbTable*() instances currently using this datasource connection.

wxDdb::GetUsername

const wxString & GetUsername()

Returns the user name (uid) used to establish this connection to the datasource.

wxDdb::Grant

bool Grant(int privileges, const wxString &tableName, const wxString &userList = "PUBLIC")

Use this member function to GRANT privileges to users for accessing tables in the datasource.

Parameters

privileges

Use this argument to select which privileges you want to grant. Pass `DB_GRANT_ALL` to grant all privileges. To grant individual privileges pass one or more of the following OR'd together:

<code>DB_GRANT_SELECT</code>	<code>= 1</code>	
<code>DB_GRANT_INSERT</code>	<code>= 2</code>	
<code>DB_GRANT_UPDATE</code>	<code>= 4</code>	
<code>DB_GRANT_DELETE</code>	<code>= 8</code>	
<code>DB_GRANT_ALL</code>	<code>= DB_GRANT_SELECT</code>	<code> DB_GRANT_INSERT</code>
	<code>DB_GRANT_UPDATE</code>	<code> DB_GRANT_DELETE</code>

tableName

The name of the table you wish to grant privileges on.

userList

OPTIONAL. A comma delimited list of users to grant the privileges to. If this argument is not passed in, the privileges will be given to the general PUBLIC.

Remarks

Some databases require user names to be specified in all capital letters (i.e. Oracle). This function does not automatically capitalize the user names passed in the comma-separated list. This is the responsibility of the calling routine.

The currently logged in user must have sufficient grantor privileges for this function to be able to successfully grant the indicated privileges.

Example

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

wxDb::IsFwdOnlyCursors

bool IsFwdOnlyCursors()

This setting indicates whether this database connection was created as being capable of using only forward scrolling cursors.

This function does NOT indicate if the ODBC driver or datasource supports backward scrolling cursors. There is no standard way of detecting if the driver or datasource can support backward scrolling cursors.

If a wxDb instance was created as being capable of only forward scrolling cursors, then even if the datasource and ODBC driver support backward scrolling cursors, tables using this database connection would only be able to use forward scrolling cursors.

The default setting of whether a wxDb connection to a database allows forward-only or also backward scrolling cursors is defined in setup.h by the value of wxODBC_FWD_ONLY_CURSORS. This default setting can be overridden when the wxDb connection is initially created (see *wxDb constructor* (p. 296) and *wxDbGetConnection* (p. 294)).

Return value

Returns true if this datasource connection is defined as using only forward scrolling cursors, or false if the connection is defined as being allowed to use backward scrolling cursors and their associated functions (see note above).

Remarks

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

This function is not available in versions prior to v2.4. You should use *wxDb::FwdOnlyCursors* (p. 303) for wxWidgets versions prior to 2.4.

See also

wxDdb constructor (p. 296), *wxDdbGetConnection* (p. 294)

wxDdb::IsOpen**bool IsOpen()**

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

Remarks

This function may indicate that the database connection is open, even if the call to *wxDdb::Open* (p. 312) may have failed to fully initialize the connection correctly. The connection to the database *is* open and can be used via the direct SQL commands, if this function returns true. Other functions which depend on the *wxDdb::Open* (p. 312) to have completed correctly may not function as expected. The return result from *wxDdb::Open* (p. 312) is the only way to know if complete initialization of this *wxDdb* connection was successful or not. See *wxDdb::Open* (p. 312) for more details on partial failures to open a connection instance.

wxDdb::LogError

void LogError(const wxString &errMsg const wxString &SQLState= "")

errMsg

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

SQLState

OPTIONAL. Native SQL state error. Default is 0.

Remarks

Calling this function will enter a log message in the error list maintained for the database connection. This log message is free form and can be anything the programmer wants to enter in the error list.

If SQL logging is turned on, the call to this function will also log the text into the SQL log file.

See also

wxDdb::WriteSqlLog (p. 319)

wxDdb::ModifyColumn

void ModifyColumn(const wxString &tableName const wxString &ColumnName int dataType ULONG columnLength=0 const wxString &optionalParam= "")

Used to change certain properties of a column such as the length, or whether a column allows NULLs or not.

tableName

Name of the table that the column to be modified is in.

columnName

Name of the column to be modified. NOTE: Name of column cannot be changed with this function.

dataType

Any one of DB_DATA_TYPE_VARCHAR, DB_DATA_TYPE_INTEGER, DB_DATA_TYPE_FLOAT, DB_DATA_TYPE_DATE.

columnLength

New size of the column. Valid only for DB_DATA_TYPE_VARCHAR dataType fields. Default is 0.

optionalParam

Default is "".

Remarks

Cannot be used to modify the precision of a numeric column, therefore 'columnLength' is ignored unless the dataType is DB_DATA_TYPE_VARCHAR.

Some datasources do not allow certain properties of a column to be changed if any rows currently have data stored in that column. Those datasources that do allow columns to be changed with data in the rows many handle truncation and/or expansion in different ways. Please refer to the reference material for the datasource being used for behavioral descriptions.

Example

```
ok = pDb->ModifyColumn( "CONTACTS", "ADDRESS2",
                        DB_, colDefs[j].SzDataObj,
                        wxT( "NOT NULL" ) );
```

wxDb::Open

bool Open(const wxString &Dsn, const wxString &Uid, const wxString &AuthStr, bool failOnDataTypeUnsupported)

bool Open(const wxString &inConnectStr, bool failOnDataTypeUnsupported)

bool Open(wxDbConnectInf *dbConnectInf, bool failOnDataTypeUnsupported)

bool Open(wxDb *copyDb)

Opens a connection to the datasource, sets certain behaviors of the datasource to confirm to the accepted behaviors (e.g. cursor position maintained on commits), and queries the datasource for its representations of the basic datatypes to determine the form in which the data going to/from columns in the data tables are to be handled.

The second form of this function, which accepts a "wxDb *" as a parameter, can be used to avoid the overhead (execution time, database load, network traffic) which are needed to determine the data types and representations of data that are necessary for cross-datasource support by these classes.

Normally the first form of the `wxDB::Open()` function will open the connection and then send a series of queries to the datasource asking it for its representation of data types, and all the features it supports. If one connection to the datasource has already been made previously, the information gathered when that connection was created can just be copied to any new connections to the same datasource by passing a pointer to the first connection in as a parameter to the `wxDB::Open()` function. Note that this new connection created from the first connections information will use the same `Dsn/Uid/AuthStr` as the first connection used.

Parameters

Dsn

datasource name. The name of the ODBC datasource as assigned when the datasource is initially set up through the ODBC data source manager.

Uid

User ID. The name (ID) of the user you wish to connect as to the datasource. The user name (ID) determines what objects you have access to in the datasource and what datasource privileges you have. Privileges include being able to create new objects, update objects, delete objects and so on. Users and privileges are normally administered by the database administrator.

AuthStr

The password associated with the `Uid`.

failOnDataTypeUnsupporte

As part of connecting to a database, the `wxDB::Open()` function will query the database to find out the native types that it supports. With some databases, some data types may not be supported, or not sufficiently supported, for use with the `wxODBC` classes. Normally a program should fail in this case, so as not to try to use a data type that is not supported. This parameter allows the programmer to override the failure if they wish and continue on using the connection.

dbConnectInf

Contains a `DSN`, `Uid`, `Password`, or a connection string to be used in opening a new connection to the database. If a connection string is present, then the connection string will be used. If there is no connection string present, then the `DSN`, `Uid`, and `Password` are used.

inConnectStr

A valid ODBC connection string used to connect to a database

copyDb

Already completely configured and opened datasource connection from which all Dsn, Uid, AuthStr, connection string, and data typing information is to be copied from for use by this datasource connection. If 'copyDb' used a connection string to create its connection originally, then the connection being made by this call to wxDb::Open() will use that same connection string.

Remarks

After a wxDb instance is created, it must then be opened. When opening a datasource, there must be three pieces of information passed. The data source name, user name (ID) and the password for the user. No database activity on the datasource can be performed until the connection is opened. This is normally done at program startup and the datasource remains open for the duration of the program/module run.

It is possible to have connections to multiple datasources open at the same time to support distributed database connections by having separate instances of wxDb objects that use either the same or different Dsn/Uid/AuthStr settings.

If this function returns a value of false, it does not necessarily mean that the connection to the datasource was not opened. It may mean that some portion of the initialization of the connection failed (such as a datatype not being able to be determined how the datasource represents it). To determine if the connection to the database failed, use the wxDb::IsOpen (p. 311) function after receiving a false result back from this function to determine if the connection was opened or not. If this function returns false, but wxDb::IsOpen (p. 311) returns true, then direct SQL commands may be passed to the database connection and can be successfully executed, but use of the datatypes (such as by a wxDbTable instance) that are normally determined during open will not be possible.

The *Dsn*, *Uid*, and *AuthStr* string pointers that are passed in are copied. NOT the strings themselves, only the pointers. The calling routine must maintain the memory for these three strings for the life of the wxDb instance.

Example

```
wxD b sampleDB(DbConnectInf.GetHenv());
if (!sampleDB.Open("Oracle 7.1 HP/UX", "gtasker",
"myPassword"))
{
    if (sampleDb.IsOpen())
    {
        // Connection is open, but the initialization of
        // datatypes and parameter settings failed
    }
    else
    {
        // Error opening datasource
    }
}
```

wxD b::RollbackTrans

bool RollbackTrans()

Function to "undo" changes made to the database. After an insert/update/delete, the

operation may be "undone" by issuing this command any time before a `wxDb::CommitTrans` (p. 298) is called on the database connection.

Remarks

Transactions begin implicitly as soon as you make a change to the database. The transaction continues until either a commit or rollback is executed. Calling `wxDb::RollbackTrans()` will result in ALL changes done using this database connection that have not already been committed to be "undone" back to the last commit/rollback that was successfully executed.

Calling this member function rolls back ALL open (uncommitted) transactions on this ODBC connection, including all `wxDbTable` instances that use this connection.

See also

`wxDb::CommitTrans` (p. 298) for a special note on cursors

wxDb::SetDebugErrorMessages

void SetDebugErrorMessages(bool state)

state

Either true (debug messages are logged) or false (debug messages are not logged).

Remarks

Turns on/off debug error messages from the ODBC class library. When this function is passed true, errors are reported to the user/logged automatically in a text or pop-up dialog when an ODBC error occurs. When passed false, errors are silently handled.

When compiled in release mode (FINAL=1), this setting has no affect.

See also

`wxDb` constructor (p. 296)

wxDb::SetSqlLogging

bool SetSqlLogging(wxDbSqlLogState state, const wxString &filename = SQL_LOG_FILENAME, bool append = false)

Parameters

state

Either `sqlLogOFF` or `sqlLogON` (see *enum wxDbSqlLogState* (p. 321)). Turns logging of SQL commands sent to the datasource OFF or ON.

filename

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

SQL_LOG_FILENAME.

append

OPTIONAL. Whether the file is appended to or overwritten. Default is false.

Remarks

When called with *sqlLogON*, all commands sent to the datasource engine are logged to the file specified by *filename*. Logging is done by embedded *wxD::WriteSqlLog* (p. 319) calls in the database member functions, or may be manually logged by adding calls to *wxD::WriteSqlLog* (p. 319) in your own source code.

When called with *sqlLogOFF*, the logging file is closed, and any calls to *wxD::WriteSqlLog* (p. 319) are ignored.

wxD::SQLColumnName

const wxString SQLColumnName(const char * colName)

Returns the column name in a form ready for use in SQL statements. In most cases, the column name is returned verbatim. But some databases (e.g. MS Access, SQL Server, MSDE) allow for spaces in column names, which must be specially quoted. For example, if the datasource allows spaces in the column name, the returned string will have the correct enclosing marks around the name to allow it to be properly included in a SQL statement for the DBMS that is currently connected to with this connection.

Parameters

colName

Native name of the column in the table that is to be evaluated to determine if any special quoting marks needed to be added to it before including the column name in a SQL statement

See also

wxD::SQLTableName (p. 316)

wxD::SQLTableName

const wxString SQLTableName(const char * tableName)

Returns the table name in a form ready for use in SQL statements. In most cases, the table name is returned verbatim. But some databases (e.g. MS Access, SQL Server, MSDE) allow for spaces in table names, which must be specially quoted. For example, if the datasource allows spaces in the table name, the returned string will have the correct enclosing marks around the name to allow it to be properly included in a SQL statement for the data source that is currently connected to with this connection.

Parameters

tableName

Native name of the table that is to be evaluated to determine if any special quoting marks needed to be added to it before including the table name in a SQL statement

See also

wxDdb::SQLColumnName (p. 316)

wxDdb::TableExists

bool TableExists(const wxString &tableName, const wxChar *userID=NULL, const wxString &path="")

Checks the ODBC datasource for the existence of a table. If a *userID* is specified, then the table must be accessible by that user (user must have at least minimal privileges to the table).

Parameters

tableName

Name of the table to check for the existence of.

userID

Owner of the table (also referred to as schema). Specify a *userID* when the datasource you are connected to allows multiple unique tables with the same name to be owned by different users. *userID* is evaluated as follows:

```
userID == NULL    ... UserID is ignored (DEFAULT)
userID == ""      ... UserID set equal to 'this->uid'
userID != ""      ... UserID set equal to 'userID'
```

Remarks

tableName may refer to a table, view, alias or synonym.

This function does not indicate whether or not the user has privileges to query or perform other functions on the table. Use the *wxDdb::TablePrivileges* (p. 318) to determine if the user has sufficient privileges or not.

See also

wxDdb::TablePrivileges (p. 318)

wxDdb::TablePrivileges

bool TablePrivileges(const wxString &tableName, const wxString &priv, const wxChar *userID=NULL, const wxChar *schema=NULL, const wxString &path="")

Checks the ODBC datasource for the existence of a table. If a *userID* is specified, then the table must be accessible by that user (user must have at least minimal privileges to the table).

Parameters

tableName

Name of the table on which to check privileges. *tableName* may refer to a table, view, alias or synonym.

priv

The table privilege being evaluated. May be one of the following (or a datasource specific privilege):

SELECT for	: The connected user is permitted to retrieve data one or more columns of the table.
INSERT rows	: The connected user is permitted to insert new containing data for one or more columns into the table.
UPDATE data in	: The connected user is permitted to update the one or more columns of the table.
DELETE of	: The connected user is permitted to delete rows data from the table.
REFERENCES or (for	: Is the connected user permitted to refer to one more columns of the table within a constraint example, a unique, referential, or table check constraint).

userID

OPTIONAL. User for which to determine if the privilege specified to be checked is granted or not. Default is "". *userID* is evaluated as follows:

```
userID == NULL    ... NOT ALLOWED!
userID == ""      ... UserID set equal to 'this->uid'
userID != ""      ... UserID set equal to 'userID'
```

schema

OPTIONAL. Owner of the table. Specify a *userID* when the datasource you are connected to allows multiple unique tables with the same name to be owned by different users. Specifying the table owner makes determination of the users privileges MUCH faster. Default is NULL. *userID* is evaluated as follows:

```
schema == NULL    ... Any owner (DEFAULT)
schema == ""      ... Owned by 'this->uid'
schema != ""      ... Owned by userID specified in 'schema'
```

path

OPTIONAL. Path to the table. Default is "". Currently unused.

Remarks

The scope of privilege allowed to the connected user by a given table privilege is datasource dependent.

For example, the privilege UPDATE might allow the connected user to update all columns in a table on one datasource, but only those columns for which the grantor (the user that granted the connected user) has the UPDATE privilege on another datasource.

Looking up a user's privileges to a table can be time consuming depending on the datasource and ODBC driver. This time can be minimized by passing a *schema* as a parameter. With some datasources/drivers, the difference can be several seconds of time difference.

wxDb::TranslateSqlState

int TranslateSqlState(const wxString &SQLState)

Converts an ODBC sqlstate to an internal error code.

Parameters

SQLState

State to be converted.

Return value

Returns the internal class DB_ERR code. See *wxDb::DB_STATUS* (p. 287) definition.

wxDb::WriteSqlLog

bool WriteSqlLog(const wxString &logMsg)

Parameters

logMsg

Free form string to be written to the log file.

Remarks

Very useful debugging tool that may be turned on/off during run time (see (see *wxDb::SetSqlLogging* (p. 315) for details on turning logging on/off). The passed in string *logMsg* will be written to a log file if SQL logging is turned on.

Return value

If SQL logging is off when a call to WriteSqlLog() is made, or there is a failure to write the log message to the log file, the function returns false without performing the requested log, otherwise true is returned.

See also

wxDdb::SetSqlLogging (p. 315)

wxDdbColDataPtr

Pointer to dynamic column definitions for use with a *wxDdbTable* instance. Currently there are no member functions for this class.

See the *database classes overview* (p. **Error! Bookmark not defined.**) for an introduction to using the ODBC classes.

```
void    *PtrDataObj;
int      SzDataObj;
SWORD    SqlCtype;
```

wxDdbColDef

This class is used to hold information about the columns bound to an instance of a *wxDdbTable* object.

Each instance of this class describes one column in the *wxDdbTable* object. When calling the *wxDdb constructor* (p. 296), a parameter passed in indicates the number of columns that will be defined for the *wxDdbTable* object. The constructor uses this information to allocate adequate memory for all of the column descriptions in your *wxDdbTable* object. Private member *wxDdbTable::colDefs* is a pointer to this chunk of memory maintained by the *wxDdbTable* class (and can be retrieved using the *wxDdbTable::GetColDefs* (p. 344) function). To access the *n*th column definition of your *wxDdbTable* object, just reference *wxDdbColDefs* element [*n* - 1].

Typically, *wxDdbTable::SetColDefs* (p. 358) is used to populate an array of these data structures for the *wxDdbTable* instance.

Currently there are no accessor functions for this class, so all members are public.

```
wxChar    ColName[DB_MAX_COLUMN_NAME_LEN+1];  // Column Name
int        DbDataType;    - Logical Data Type;
                        e.g. DB_DATA_TYPE_INTEGER
SWORD      SqlCtype;      - C data type; e.g. SQL_C_LONG
void       *PtrDataObj;   - Address of the data object
int        SzDataObj;     - Size, in bytes, of the data object
bool       KeyField;      - Is column part of the PRIMARY KEY for
the
                                table? -- Date fields should NOT be
                                KeyFields
bool       Updateable;    - Column is updateable?
bool       InsertAllowed; - Column included in INSERT statements?
bool       DerivedCol;    - Column is a derived value?
SDWORD     CbValue;       - !!!Internal use only!!!
bool       Null;          - NOT FULLY IMPLEMENTED
                                Allows NULL values in Inserts and
Updates
```

See also

database classes overview (p. **Error! Bookmark not defined.**), *wxDdbTable::GetColDefs*

(p. 344), *wxDdb constructor* (p. 296)

Include files

<wx/db.h>

wxDdbColDef::Initialize

Simply initializes all member variables to a cleared state. Called by the constructor automatically.

wxDdbColFor

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

```

        wxString      s_Field;           // Formatted String for Output
        wxString      s_Format[7];      // Formatted Objects - TIMESTAMP
has
        wxString      s_Amount[7];      // Formatted Objects - amount of
                                         the biggest (7)
                                         things that can be formatted
        int           i_Amount[7];      // Formatted Objects -
                                         TT MM YYYY HH MM SS m
        int           i_Nation;         // 0 = timestamp
                                         1 = EU
                                         2 = UK
                                         3 = International
                                         4 = US
        int           i_dbDataType;     // conversion of the
'sqlDataType'
by
        SWORD         i_sqlDataType;    to the generic data type used
                                         these classes

```

The constructor for this class initializes all the values to zero or NULL.

The destructor does nothing at this time.

Only one function is provided with this class currently.

See the *database classes overview* (p. **Error! Bookmark not defined.**) for an introduction to using the ODBC classes.

Include files

<wx/db.h>

wxDdbColFor::Format

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

decimalDigits)

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

wxDbColFor::Initialize

Simply initializes all member variables to a cleared state. Called by the constructor automatically.

wxDbColInf

Used with the *wxDb::GetColumns* (p. 304) functions for obtaining all retrievable information about a column's definition.

```

wxChar      catalog[128+1];
wxChar      schema[128+1];
wxChar      tableName[DB_MAX_TABLE_NAME_LEN+1];
wxChar      colName[DB_MAX_COLUMN_NAME_LEN+1];
SWORD      sqlDataType;
wxChar      typeName[128+1];
SWORD      columnSize;
SWORD      bufferLength;
short      decimalDigits;
short      numPrecRadix;
short      nullable;
wxChar      remarks[254+1];
int         dbDataType; // conversion of the 'sqlDataType'
                        // to the generic data type used by
                        // these classes
int         PkCol;      // Primary key column
                        0 = No
                        1 = First Key
                        2 = Second Key, etc...
wxChar      PkTableName[DB_MAX_TABLE_NAME_LEN+1];
                        // Tables that use this PKey as a
FKey
int         FkCol;      // Foreign key column
                        0 = No
                        1 = First Key
                        2 = Second Key, etc...
wxChar      FkTableName[DB_MAX_TABLE_NAME_LEN+1];
                        // Foreign key table name
wxDbColFor *pColFor;    // How should this column be
formatted

```

The constructor for this class initializes all the values to zero, "", or NULL.

The destructor for this class takes care of deleting the pColFor member if it is non-NULL.

See the *database classes overview* (p. **Error! Bookmark not defined.**) for an introduction to using the ODBC classes.

Include files

<wx/db.h>

wxDbCollnf::Initialize

Simply initializes all member variables to a cleared state. Called by the constructor automatically.

wxDbConnectInf

This class is used for holding the data necessary for connecting to the ODBC datasource. That information includes: SQL environment handle, datasource name, user ID, password and default directory path (used with dBase). Other optional fields held in this class are and file type, both for future functions planned to be added for creating/manipulating datasource definitions.

wxDbConnectInf::wxDbConnectInf

wxDbConnectInf()

Default constructor.

wxDbConnectInf(HENV henv, const wxString &dsn, const wxString &userID="", const wxString &password, const wxString &defaultDir="", const wxString &description="", const wxString &fileType="")

Constructor which allows initial settings of all the classes member variables.

See the special note below on the henv parameter for forcing this constructor to create a SQL environment handle automatically, rather than needing to pass one in to the function.

Parameters

henv

Environment handle used for this connection. See *wxDConnectInf::AllocHenv* (p. 325) for how to create an SQL environment handle. NOTE: Passing in a NULL for this parameter will inform the constructor that it should create its own SQL environment handle. If NULL is passed for this parameter, the constructor will call *wxDConnectInf::AllocHenv* (p. 325) internally. A flag is set internally also to indicate that the HENV was created by the constructor so that when the default class destructor is called, the destructor will call *wxDConnectInf::FreeHenv* (p. 325) to free the environment handle automatically.

dsn

Name of the datasource to be used in creating wxDb instances for creating connection(s) to a datasource.

userID

OPTIONAL Many datasources allow (or even require) use of a username to determine privileges that connecting user is allowed to have when accessing the datasource or the data tables. Default is "".

password

OPTIONAL Password to be associated with the user ID specified in 'userID'.
Default is "".

defaultDir

OPTIONAL Used for datasources which require the path to where the data file is stored to be specified. dBase is one example of the type of datasource which requires this information. Default is "".

description

OPTIONAL FUTURE USE Default is "".

fileType

OPTIONAL FUTURE USE Default is "".

Remarks

It is strongly recommended that programs use the longer form of the constructor and allow the constructor to create the SQL environment handle automatically, and manage the destruction of the handle.

Example

```
wxDbConnectInf *DbConnectInf;  
  
    DbConnectInf = new wxDbConnectInf(0, "MY_DSN", "MY_USER",  
    "MY_PASSWORD");  
  
    ....the rest of the program  
  
    delete DbConnectInf;
```

See also

wxDConnectInf::AllocHenv (p. 325), *wxDConnectInf::FreeHenv* (p. 325)

wxDbConnectInf::~~wxDbConnectInf

~wxDbConnectInf()

Handles the default destruction of the instance of the class. If the long form of the *wxDConnectInf* (p. 323) was used, then this destructor also takes care of calling *wxDConnectInf::FreeHenv* (p. 325) to free the SQL environment handle.

wxDbConnectInf::AllocHenv

bool AllocHenv()

Allocates a SQL environment handle that will be used to interface with an ODBC datasource.

Remarks

This function can be automatically called by the long form of the *wxDbConnectInf* (p. 323) constructor.

wxDbConnectInf::FreeHenv**void FreeHenv()**

Frees the SQL environment handle being managed by the instance of this class.

Remarks

If the SQL environment handle was created using the long form of the *wxDbConnectInf* (p. 323) constructor, then the flag indicating that the HENV should be destroyed when the classes destructor is called is reset to be false, so that any future handles created using the *wxDbConnectInf::AllocHenv* (p. 325) function must be manually released with a call to this function.

wxDbConnectInf::Initialize

Simply initializes all member variables to a cleared state. Called by the constructor automatically.

wxDbConnectInf::GetAuthStr**const wxChar * GetAuthStr()**

Accessor function to return the password assigned for this class instance that will be used with the user ID.

Synonymous with *wxDbConnectInf::GetPassword* (p. 326)

wxDbConnectInf::GetDefaultDir**const wxChar * GetDefaultDir()**

Accessor function to return the default directory in which the datasource's data table is stored. This directory is only used for file based datasources like dBase. MS-Access does not require this to be set, as the path is set in the ODBC Administrator for MS-Access.

wxDbConnectInf::GetDescription**const wxChar * GetDescription()**

Accessor function to return the description assigned for this class instance.

NOTE: Description is a FUTURE USE item and is unused currently.

wxDbConnectInf::GetDsn

const wxChar * GetDsn()

Accessor function to return the datasource name assigned for this class instance.

wxDbConnectInf::GetFileType

const wxChar * GetFileType()

Accessor function to return the filetype of the ODBC datasource assigned for this class instance.

NOTE: FileType is a FUTURE USE item and is unused currently.

wxDbConnectInf::GetHenv

const HENV GetHenv()

Accessor function to return the SQL environment handle being managed by this class instance.

wxDbConnectInf::GetPassword

const wxChar * GetPassword()

Accessor function to return the password assigned for this class instance that will be used with the user ID.

Synonymous with *wxDbConnectInf::GetAuthStr* (p. 325)

wxDbConnectInf::GetUid

const wxChar * GetUid()

Accessor function to return the user ID assigned for this class instance.

wxDbConnectInf::GetUserID

const wxChar * GetUserID()

Accessor function to return the user ID assigned for this class instance.

wxDbConnectInf::SetAuthStr

SetAuthStr(const wxString &authstr)

Accessor function to assign the password for this class instance that will be used with the user ID.

Synonymous with *wxDbConnectInf::SetPassword* (p. 328)

wxDbConnectInf::SetDefaultDir

SetDefaultDir(const wxString &defDir)

Accessor function to assign the default directory in which the datasource's data table is stored. This directory is only used for file based datasources like dBase. MS-Access does not require this to be set, as the path is set in the ODBC Administrator for MS-Access.

wxDbConnectInf::SetDescription**SetDescription**(const wxString &desc)

Accessor function to assign the description assigned for this class instance.

NOTE: Description is a FUTURE USE item and is unused currently.

wxDbConnectInf::SetDsn**SetDsn**(const wxString &dsn)

Accessor function to assign the datasource name for this class instance.

wxDbConnectInf::SetFileType**SetFileType**(const wxString &)

Accessor function to return the filetype of the ODBC datasource assigned for this class instance.

NOTE: FileType is a FUTURE USE item and is unused currently.

wxDbConnectInf::SetHenv**void SetHenv**(const HENV henv)

Accessor function to set the SQL environment handle for this class instance.

wxDbConnectInf::SetPassword**SetPassword**(const wxString &password)

Accessor function to assign the password for this class instance that will be used with the user ID.

Synonymous with *wxDbConnectInf::SetAuthStr* (p. 327)

wxDbConnectInf::SetUid**SetUid**(const wxString &uid)

Accessor function to set the user ID for this class instance.

wxDbConnectInf::SetUserID

SetUserID(const wxString &userID)

Accessor function to assign the user ID for this class instance.

wxDbIdxDef

Used in creation of non-primary indexes. Currently there are no member functions for this class.

```
wxChar  ColName[DB_MAX_COLUMN_NAME_LEN+1]
                                     // Name of column
bool    Ascending                  // Is index maintained in
                                     ASCENDING sequence?
```

There are no constructors/destructors as of this time, and no member functions.

See the *database classes overview* (p. **Error! Bookmark not defined.**) for an introduction to using the ODBC classes.

Include files

<wx/db.h>

wxDbInf

Contains information regarding the database connection (datasource name, number of tables, etc). A pointer to a wxDbTableInf is included in this class so a program can create a wxDbTableInf array instance to maintain all information about all tables in the datasource to have all the datasource's information in one memory structure.

Primarily, this class is used internally by the wxWidgets ODBC classes.

```
wxChar      catalog[128+1];
wxChar      schema[128+1]; // typically means owner of
table(s)
int          numTables;    // How many tables does this
                           datasource have
wxDbTableInf *pTableInf;   // Equals a new
                           wxDbTableInf[numTables];
```

The constructor for this class initializes all the values to zero, "", or NULL.

The destructor for this class takes care of deleting the pTableInf member if it is non-NULL.

See the *database classes overview* (p. **Error! Bookmark not defined.**) for an introduction to using the ODBC classes.

Include files

<wx/db.h>

wxDblnf::Initialize

Simply initializes all member variables to a cleared state. Called by the constructor automatically.

wxDbTable

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

See the *database classes overview* (p. **Error! Bookmark not defined.**) for an introduction to using the ODBC classes.

Include files

<wx/dbtable.h>
<wx/db.h>

Helper classes and data structures

The following classes and structs are defined in dbtable.cpp/.h for use with the wxDbTable class.

- *wxDbColDef* (p. 320)
- *wxDbColDataPtr* (p. 320)
- *wxDblIdxDef* (p. 328)

Constants

wxDB_DEFAULT_CURSOR based	Primary cursor normally used for cursor operations.
wxDB_QUERY_ONLY opened insert/update/deletes overhead query faster	Used to indicate whether a table that is is for query only, or if will be performed on the table. Less (cursors and memory) are allocated for only tables, plus read access times are with some datasources.
wxDB_ROWID_LEN CanUpdateByRowID() faster column	[Oracle only] - Used when is true. Optimizes updates so they are by updating on the Oracle-specific ROWID rather than some other index.

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

wxDBTable::wxDBTable

wxDBTable(wxDB *pwxDb, const wxString &tblName, const UWORD numColumns, const wxString &qryTblName = "", bool qryOnly = !wxDB_QUERY_ONLY, const wxString &tblPath = "")

Default constructor.

Parameters

pwxDb

Pointer to the wxDb instance to be used by this wxDbTable instance.

tblName

The name of the table in the RDBMS.

numColumns

The number of columns in the table. (Do NOT include the ROWID column in the count if using Oracle).

qryTblName

OPTIONAL. The name of the table or view to base your queries on. This argument allows you to specify a table/view other than the base table for this object to base your queries on. This allows you to query on a view for example, but all of the INSERT, UPDATE and DELETES will still be performed on the base table for this wxDbTable object. Basing your queries on a view can provide a substantial performance increase in cases where your queries involve many tables with multiple joins. Default is "".

qryOnly

OPTIONAL. Indicates whether the table will be accessible for query purposes only, or should the table create the necessary cursors to be able to insert, update, and delete data from the table. Default is !wxDB_QUERY_ONLY.

tblPath

OPTIONAL. Some datasources (such as dBase) require a path to where the table is stored on the system. Default is "".

wxDbTable::wxDbTable**virtual ~wxDbTable()**

Virtual default destructor.

wxDbTable::BuildDeleteStmt

void BuildDeleteStmt(wxString &pSqlStmt, int typeOfDel, const wxString &pWhereClause= "")

Constructs the full SQL statement that can be used to delete all rows matching the criteria in the pWhereClause.

Parameters

pSqlStmt

Pointer to buffer for the SQL statement retrieved. To be sure you have adequate space allocated for the SQL statement, allocate DB_MAX_STATEMENT_LEN bytes.

typeOfDel

The type of delete statement being performed. Can be one of three values: DB_DEL_KEYFIELDS, DB_DEL_WHERE or DB_DEL_MATCHING

pWhereClause

OPTIONAL. If the typeOfDel is DB_DEL_WHERE, then you must also pass in a SQL WHERE clause in this argument. Default is "".

Remarks

This member function constructs a SQL DELETE statement. This can be used for debugging purposes if you are having problems executing your SQL statement.

WHERE and FROM clauses specified using *wxDbTable::SetWhereClause* (p. 364) and *wxDbTable::SetFromClause* (p. 361) are ignored by this function.

wxDbTable::BuildSelectStmt

void BuildSelectStmt(wxString &pSqlStmt, int typeOfSelect, bool distinct)

Constructs the full SQL statement that can be used to select all rows matching the criteria in the pWhereClause. This function is called internally in the wxDbTable class whenever the function *wxDbTable::Query* (p. 353) is called.

NOTE: Only the columns specified in *wxDbTable::SetColDefs* (p. 358) statements are included in the list of columns returned by the SQL statement created by a call to this function.

Parameters

pSqlStmt

Pointer to storage for the SQL statement retrieved. To be sure you have adequate space allocated for the SQL statement, allocate DB_MAX_STATEMENT_LEN bytes.

typeOfSelect

The type of select statement being performed. Can be one of four values: DB_SELECT_KEYFIELDS, DB_SELECT_WHERE, DB_SELECT_MATCHING or DB_SELECT_STATEMENT.

distinct

Whether to select distinct records only.

Remarks

This member function constructs a SQL SELECT statement. This can be used for debugging purposes if you are having problems executing your SQL statement.

WHERE and FROM clauses specified using *wxDbTable::SetWhereClause* (p. 364) and *wxDbTable::SetFromClause* (p. 361) are ignored by this function.

wxDbTable::BuildUpdateStmt

void BuildUpdateStmt(wxString &pSqlStmt, int typeOfUpd, const wxString &pWhereClause= "")

Constructs the full SQL statement that can be used to update all rows matching the criteria in the pWhereClause.

If typeOfUpdate is DB_UPD_KEYFIELDS, then the current values in the bound columns are used to determine which row(s) in the table are to be updated. The exception to this is when a datasource supports ROW IDs (Oracle). The ROW ID column is used for efficiency purposes when available.

NOTE: Only the columns specified in *wxDbTable::SetColDefs* (p. 358) statements are included in the list of columns updated by the SQL statement created by a call to this function. Any column definitions that were defined as being non-updateable will be excluded from the SQL UPDATE statement created by this function.

Parameters

pSqlStmt

Pointer to storage for the SQL statement retrieved. To be sure you have adequate space allocated for the SQL statement, allocate DB_MAX_STATEMENT_LEN bytes.

typeOfUpdate

The type of update statement being performed. Can be one of two values: DB_UPD_KEYFIELDS or DB_UPD_WHERE.

pWhereClause

OPTIONAL. If the `typeOfUpdate` is `DB_UPD_WHERE`, then you must also pass in a SQL WHERE clause in this argument. Default is "".

Remarks

This member function allows you to see what the SQL UPDATE statement looks like that the ODBC class library builds. This can be used for debugging purposes if you are having problems executing your SQL statement.

WHERE and FROM clauses specified using `wxDbTable::SetWhereClause` (p. 364) and `wxDbTable::SetFromClause` (p. 361) are ignored by this function.

wxDbTable::BuildWhereClause

void BuildWhereClause(wxString &pWhereClause, int typeOfWhere, const wxString &qualTableName="", bool useLikeComparison=false)

Constructs the portion of a SQL statement which would follow the word 'WHERE' in a SQL statement to be passed to the datasource. The returned string does NOT include the word 'WHERE'.

Parameters

pWhereClause

Pointer to storage for the SQL statement retrieved. To be sure you have adequate space allocated for the SQL statement, allocate `DB_MAX_STATEMENT_LEN` bytes.

typeOfWhere

The type of where clause to generate. Can be one of two values: `DB_WHERE_KEYFIELDS` or `DB_WHERE_MATCHING`.

qualTableName

OPTIONAL. Prepended to all base table column names. For use when a FROM clause has been specified with the `wxDbTable::SetFromClause` (p. 361), to clarify which table a column name reference belongs to. Default is "".

useLikeComparison

OPTIONAL. Should the constructed WHERE clause utilize the LIKE comparison operator. If false, then the '=' operator is used. Default is false.

Remarks

This member function allows you to see what the SQL WHERE clause looks like that the ODBC class library builds. This can be used for debugging purposes if you are having problems executing your own SQL statements.

If using 'typeOfWhere' set to `DB_WHERE_MATCHING`, any bound columns currently

containing a NULL value are not included in the WHERE clause's list of columns to use in the comparison.

wxDbTable::CanSelectForUpdate

bool CanSelectForUpdate()

Use this function to determine if the datasource supports SELECT ... FOR UPDATE. When the keywords "FOR UPDATE" are included as part of your SQL SELECT statement, all records *retrieved* (not just queried, but actually retrieved using `wxDbTable::GetNext` (p. 347), etc) from the result set are locked.

Remarks

Not all datasources support the "FOR UPDATE" clause, so you must use this member function to determine if the datasource currently connected to supports this behavior or not before trying to select using "FOR UPDATE".

If the `wxDbTable` instance was created with the parameter `wxDB_QUERY_ONLY`, then this function will return false. For all known databases which do not support the FOR UPDATE clause, this function will return false also.

wxDbTable::CanUpdateByROWID

bool CanUpdateByROWID()

CURRENTLY ONLY POSSIBLE IF USING ORACLE.

--- CURRENTLY DISABLED FOR *ALL* DATASOURCES --- NOV 1 2000 - gt

Every Oracle table has a hidden column named ROWID. This is a pointer to the physical location of the record in the datasource and allows for very fast updates and deletes. The key is to retrieve this ROWID during your query so it is available during an update or delete operation.

Use of the ROWID feature is always handled by the class library except in the case of `wxDbTable::QueryBySqlStmt` (p. 354). Since you are passing in the SQL SELECT statement, it is up to you to include the ROWID column in your query. If you do not, the application will still work, but may not be as optimized. The ROWID is always the last column in the column list in your SQL SELECT statement. The ROWID is not a column in the normal sense and should not be considered part of the column definitions for the `wxDbTable` object.

Remarks

The decision to include the ROWID in your SQL SELECT statement must be deferred until runtime since it depends on whether you are connected to an Oracle datasource or not.

Example

```
// Incomplete code sample
wxDbTable parts;
```

```
.....
if (parts.CanUpdateByROWID())
{
    // Note that the ROWID column must always be the last
column selected
    sqlStmt = "SELECT PART_NUM, PART_DESC, ROWID" FROM PARTS";
}
else
    sqlStmt = "SELECT PART_NUM, PART_DESC FROM PARTS";
```

wxDbTable::ClearMemberVar

void ClearMemberVar(UWORD colNumber, bool setToNull=false)

Same as *wxDbTable::ClearMemberVars* (p. 336) except that this function clears only the specified column of its values, and optionally sets the column to be a NULL column.

colNumber

Column number that is to be cleared. This number (between 0 and (numColumns-1)) is the index of the column definition created using the *wxDbTable::SetColDefs* (p. 358) function.

setToNull

OPTIONAL. Indicates whether the column should be flagged as being a NULL value stored in the bound memory variable. If true, then any value stored in the bound member variable is cleared. Default is false.

wxDbTable::ClearMemberVars

void ClearMemberVars(bool setToNull=false)

Initializes all bound columns of the *wxDbTable* instance to zero. In the case of a string, zero is copied to the first byte of the string.

setToNull

OPTIONAL. Indicates whether all columns should be flagged as having a NULL value stored in the bound memory variable. If true, then any value stored in the bound member variable is cleared. Default is false.

Remarks

This is useful before calling functions such as *wxDbTable::QueryMatching* (p. 356) or *wxDbTable::DeleteMatching* (p. 341) since these functions build their WHERE clauses from non-zero columns. To call either *wxDbTable::QueryMatching* (p. 356) or *wxDbTable::DeleteMatching* (p. 341) use this sequence:

- 1) *ClearMemberVars()*
- 2) Assign columns values you wish to match on
- 3) Call *wxDbTable::QueryMatching()* or *wxDbTable::DeleteMatching()*

wxDbTable::CloseCursor

bool CloseCursor(HSTMT *cursor*)

Closes the specified cursor associated with the wxDbTable object.

Parameters

cursor

The cursor to be closed.

Remarks

Typically handled internally by the ODBC class library, but may be used by the programmer if desired.

DO NOT CLOSE THE wxDB_DEFAULT_CURSOR!

wxDbTable::Count**ULONG Count(const wxString &*args*="*")**

Returns the number of records which would be in the result set using the current query parameters specified in the WHERE and FROM clauses.

Parameters

args

OPTIONAL. This argument allows the use of the DISTINCT keyword against a column name to cause the returned count to only indicate the number of rows in the result set that have a unique value in the specified column. An example is shown below. Default is "*", meaning a count of the total number of rows matching is returned, regardless of uniqueness.

Remarks

This function can be called before or after an actual query to obtain the count of records in the result set. Count() uses its own cursor, so result set cursor positioning is not affected by calls to Count().

WHERE and FROM clauses specified using *wxDbTable::SetWhereClause* (p. 364) and *wxDbTable::SetFromClause* (p. 361) ARE used by this function.

Example

```

USERS TABLE

FIRST_NAME      LAST_NAME
-----
John            Doe
Richard         Smith
Michael         Jones
John            Carpenter

// Incomplete code sample
wxDbTable users;
```

```
.....
users.SetWhereClause("");

// This Count() will return 4, as there are four users listed
above
// that match the query parameters
totalNumberOfUsers = users.Count();

// This Count() will return 3, as there are only 3 unique
first names
// in the table above - John, Richard, Michael.
totalNumberOfUniqueFirstNames = users.Count("DISTINCT
FIRST_NAME");
```

wxDbTable::CreateIndex

**bool CreateIndex(const wxString &IndexName, bool unique, UWORD
numIndexColumns, wxDbIdxDef *pIndexDefs, bool attemptDrop=true)**

This member function allows you to create secondary (non primary) indexes on your tables. You first create your table, normally specifying a primary index, and then create any secondary indexes on the table. Indexes in relational model are not required. You do not need indexes to look up records in a table or to join two tables together. In the relational model, indexes, if available, provide a quicker means to look up data in a table. To enjoy the performance benefits of indexes, the indexes must be defined on the appropriate columns and your SQL code must be written in such a way as to take advantage of those indexes.

Parameters

IndexName

Name of the Index. Name must be unique within the table space of the datasource.

unique

Indicates if this index is unique.

numIndexColumns

Number of columns in the index.

pIndexDefs

A pointer to an array *wxDblIdxDef* (p. 328) structures.

attemptDrop

OPTIONAL. Indicates if the function should try to execute a *wxDbTable::DropIndex* (p. 343) on the index name provided before trying to create the index name. Default is true.

Remarks

The first parameter, index name, must be unique and should be given a meaningful

name. Common practice is to include the table name as a prefix in the index name (e.g. For table PARTS, you might want to call your index PARTS_Index1). This will allow you to easily view all of the indexes defined for a given table grouped together alphabetically.

The second parameter indicates if the index is unique or not. Uniqueness is enforced at the RDBMS level preventing rows which would have duplicate indexes from being inserted into the table when violating a unique index's uniqueness.

In the third parameter, specify how many columns are in your index. This number must match the number of columns defined in the 'pIndexDefs' parameter.

The fourth parameter specifies which columns make up the index using the *wxDblIdxDef* (p. 328) structure. For each column in the index, you must specify two things, the column name and the sort order (ascending / descending). See the example below to see how to build and pass in the *wxDblIdxDef* (p. 328) structure.

The fifth parameter is provided to handle the differences in datasources as to whether they will automatically overwrite existing indexes with the same name or not. Some datasources require that the existing index must be dropped first, so this is the default behavior.

Some datasources (MySQL, and possibly others) require columns which are to be part of an index to be defined as NOT NULL. When this function is called, if a column is not defined to be NOT NULL, a call to this function will modify the column definition to change any columns included in the index to be NOT NULL. In this situation, if a NULL value already exists in one of the columns that is being modified, creation of the index will fail.

PostGres is unable to handle index definitions which specify whether the index is ascending or descending, and defaults to the system default when the index is created.

It is not necessary to call *wxDb::CommitTrans* (p. 298) after executing this function.

Example

```
// Create a secondary index on the PARTS table
wxDblIdxDef IndexDef[2]; // 2 columns make up the index

wxStrcpy(IndexDef[0].ColName, "PART_DESC"); // Column 1
IndexDef[0].Ascending = true;

wxStrcpy(IndexDef[1].ColName, "SERIAL_NO"); // Column 2
IndexDef[1].Ascending = false;

// Create a name for the index based on the table's name
wxString indexName;
indexName.Printf("%s_Index1", parts->GetTableName());
parts->CreateIndex(indexName, true, 2, IndexDef);
```

wxDbTable::CreateTable

bool CreateTable(bool attemptDrop=true)

Creates a table based on the definitions previously defined for this wxDbTable instance.

Parameters

attemptDrop

OPTIONAL. Indicates whether the driver should attempt to drop the table before trying to create it. Some datasources will not allow creation of a table if the table already exists in the table space being used. Default is true.

Remarks

This function creates the table and primary index (if any) in the table space associated with the connected datasource. The owner of these objects will be the user id that was given when *wxDdb::Open* (p. 312) was called. The objects will be created in the default schema/table space for that user.

In your derived *wxDdbTable* object constructor, the columns and primary index of the table are described through the *wxDdbColDef* (p. 320) structure. *wxDdbTable::CreateTable* (p. 339) uses this information to create the table and to add the primary index. See *wxDdbTable* (p. 329) ctor and *wxDdbColDef* description for additional information on describing the columns of the table.

It is not necessary to call *wxDdb::CommitTrans* (p. 298) after executing this function.

wxDdbTable::DB_STATUS

bool DB_STATUS()

Accessor function that returns the *wxDdb* private member variable *DB_STATUS* for the database connection used by this instance of *wxDdbTable*.

wxDdbTable::Delete

bool Delete()

Deletes the row from the table indicated by the current cursor.

Remarks

Use *wxDdbTable::GetFirst* (p. 345), *wxDdbTable::GetLast* (p. 346), *wxDdbTable::GetNext* (p. 347) or *wxDdbTable::GetPrev* (p. 347) to position the cursor to a valid record. Once positioned on a record, call this function to delete the row from the table.

A *wxDdb::CommitTrans* (p. 298) or *wxDdb::RollbackTrans* (p. 315) must be called after use of this function to commit or rollback the deletion.

NOTE: Most datasources have a limited size "rollback" segment. This means that it is only possible to insert/update/delete a finite number of rows without performing a *wxDdb::CommitTrans* (p. 298) or *wxDdb::RollbackTrans* (p. 315). Size of the rollback segment varies from database to database, and is user configurable in most databases. Therefore it is usually best to try to perform a commit or rollback at relatively small intervals when processing a larger number of actions that insert/update/delete rows in a table.

wxDbTable::DeleteCursor**bool DeleteCursor**(HSTMT *hstmtDel)

Allows a program to delete a cursor.

Parameters*hstmtDel*

Handle of the cursor to delete.

Remarks

For default cursors associated with the instance of wxDbTable, it is not necessary to specifically delete the cursors. This is automatically done in the wxDbTable destructor.

NOTE: If the cursor could not be deleted for some reason, an error is logged indicating the reason. Even if the cursor could not be deleted, the HSTMT that is passed in is deleted, and the pointer is set to NULL.

DO NOT DELETE THE wxDB_DEFAULT_CURSOR!

wxDbTable::DeleteMatching**bool DeleteMatching**()

This member function allows you to delete records from your wxDbTable object by specifying the data in the columns to match on.

Remarks

To delete all users with a first name of "JOHN", do the following:

1. Clear all "columns" using wxDbTable::ClearMemberVars().
2. Set the FIRST_NAME column equal to "JOHN".
3. Call wxDbTable::DeleteMatching().

The WHERE clause is built by the ODBC class library based on all non-NULL columns. This allows deletion of records by matching on any column(s) in your wxDbTable instance, without having to write the SQL WHERE clause.

A *wxDb::CommitTrans* (p. 298) or *wxDb::RollbackTrans* (p. 315) must be called after use of this function to commit or rollback the deletion.

NOTE: Row(s) should be locked before deleting them to make sure they are not already in use. This can be achieved by calling *wxDbTable::QueryMatching* (p. 356), and then retrieving the records, locking each as you go (assuming FOR UPDATE is allowed on the datasource). After the row(s) have been successfully locked, call this function.

NOTE: Most datasources have a limited "rollback" segment. This means that it is only possible to insert/update/delete a finite number of rows without performing a

`wxDb::CommitTrans` (p. 298) or `wxDb::RollbackTrans` (p. 315). Size of the rollback segment varies from database to database, and is user configurable in most databases. Therefore it is usually best to try to perform a commit or rollback at relatively small intervals when processing a larger number of actions that insert/update/delete rows in a table.

Example

```
// Incomplete code sample to delete all users with a first
name
// of "JOHN"
users.ClearMemberVars();
wxStrcpy(users.FirstName, "JOHN");
users.DeleteMatching();
```

`wxDbTable::DeleteWhere`

bool DeleteWhere(const wxString &pWhereClause)

Deletes all rows from the table which match the criteria specified in the WHERE clause that is passed in.

Parameters

pWhereClause

SQL WHERE clause. This WHERE clause determines which records will be deleted from the table interfaced through the `wxDbTable` instance. The WHERE clause passed in must be compliant with the SQL 92 grammar. Do not include the keyword 'WHERE'

Remarks

This is the most powerful form of the `wxDbTable` delete functions. This function gives access to the full power of SQL. This function can be used to delete records by passing a valid SQL WHERE clause. Sophisticated deletions can be performed based on multiple criteria using the full functionality of the SQL language.

A `wxDb::CommitTrans` (p. 298) must be called after use of this function to commit the deletions.

Note: This function is limited to deleting records from the table associated with this `wxDbTable` object only. Deletions on joined tables is not possible.

NOTE: Most datasources have a limited size "rollback" segment. This means that it is only possible to insert/update/delete a finite number of rows without performing a `wxDb::CommitTrans` (p. 298) or `wxDb::RollbackTrans` (p. 315). Size of the rollback segment varies from database to database, and is user configurable in most databases. Therefore it is usually best to try to perform a commit or rollback at relatively small intervals when processing a larger number of actions that insert/update/delete rows in a table.

WHERE and FROM clauses specified using `wxDbTable::SetWhereClause` (p. 364) and `wxDbTable::SetFromClause` (p. 361) are ignored by this function.

Example

```
// Delete parts 1 thru 10 from containers 'X', 'Y' and 'Z'
that
// are magenta in color
parts.DeleteWhere("(PART_NUMBER BETWEEN 1 AND 10) AND \
CONTAINER IN ('X', 'Y', 'Z') AND \
UPPER(COLOR) = 'MAGENTA'");
```

wxDATABASE::DropIndex**bool DropIndex(const wxString &IndexName)**

Allows an index on the associated table to be dropped (deleted) if the user login has sufficient privileges to do so.

Parameters*IndexName*

Name of the index to be dropped.

Remarks

If the index specified in the 'IndexName' parameter does not exist, an error will be logged, and the function will return a result of false.

It is not necessary to call *wxDATABASE::CommitTrans* (p. 298) after executing this function.

wxDATABASE::DropTable**bool DropTable()**

Deletes the associated table if the user has sufficient privileges to do so.

Remarks

This function returns true if the table does not exist, but only for supported databases (see *wxDATABASE::Dbms* (p. 299)). If a datasource is not specifically supported, and this function is called, the function will return false.

Most datasources/ODBC drivers will delete any indexes associated with the table automatically, and others may not. Check the documentation for your database to determine the behavior.

It is not necessary to call *wxDATABASE::CommitTrans* (p. 298) after executing this function.

wxDATABASE::From**const wxString & From()****void From(const wxString &From)**

Accessor function for the private class member *wxDATABASE::from*. Can be used as a

synonym for *wxDbTable::GetFromClause* (p. 346)(the first form of this function) or *wxDbTable::SetFromClause* (p. 361) (the second form of this function).

Parameters

From

A comma separated list of table names that are to be inner joined with the base table's columns so that the joined table's columns may be returned in the result set or used as a portion of a comparison with the base table's columns. NOTE that the base tables name must NOT be included in the FROM clause, as it is automatically included by the *wxDbTable* class in constructing query statements.

Return value

The first form of this function returns the current value of the *wxDbTable* member variable *::from*.

The second form of the function has no return value, as it will always set the from clause successfully.

See also

wxDbTable::GetFromClause (p. 346), *wxDbTable::SetFromClause* (p. 361)

wxDbTable::GetColDefs

wxDbColDef * GetColDefs()

Accessor function that returns a pointer to the array of column definitions that are bound to the columns that this *wxDbTable* instance is associated with.

To determine the number of elements pointed to by the returned *wxDbColDef* (p. 320) pointer, use the *wxDbTable::GetNumberOfColumns* (p. 347) function.

Remarks

These column definitions must not be manually redefined after they have been set.

wxDbTable::GetCursor

HSTMT GetCursor()

Returns the HSTMT value of the current cursor for this *wxDbTable* object.

Remarks

This function is typically used just before changing to use a different cursor so that after the program is finished using the other cursor, the current cursor can be set back to being the cursor in use.

See also

wxDbTable::SetCursor (p. 360), *wxDbTable::GetNewCursor* (p. 346)

wxDbTable::GetDb**wxDb * GetDb()**

Accessor function for the private member variable `pDb` which is a pointer to the datasource connection that this `wxDbTable` instance uses.

wxDbTable::GetFirst**bool GetFirst()**

Retrieves the FIRST row in the record set as defined by the current query. Before retrieving records, a query must be performed using `wxDbTable::Query` (p. 353), `wxDbTable::QueryOnKeyFields` (p. 357), `wxDbTable::QueryMatching` (p. 356) or `wxDbTable::QueryBySqlStmt` (p. 354).

Remarks

This function can only be used if the datasource connection used by the `wxDbTable` instance was created with `FwdOnlyCursors` set to false. If the connection does not allow backward scrolling cursors, this function will return false, and the data contained in the bound columns will be undefined.

See also

`wxDb::IsFwdOnlyCursors` (p. 310)

wxDbTable::GetFromClause**const wxString & GetFromClause()**

Accessor function that returns the current FROM setting assigned with `wxDbTable::SetFromClause` (p. 361).

See also

`wxDbTable::From` (p. 344)

wxDbTable::GetLast**bool GetLast()**

Retrieves the LAST row in the record set as defined by the current query. Before retrieving records, a query must be performed using `wxDbTable::Query` (p. 353), `wxDbTable::QueryOnKeyFields` (p. 357), `wxDbTable::QueryMatching` (p. 356) or `wxDbTable::QueryBySqlStmt` (p. 354).

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

wxDdb::IsFwdOnlyCursors (p. 310)

wxDdbTable::GetNewCursor

HSTMT * GetNewCursor(**bool** *setCursor=false*,**bool** *bindColumns=true*)

This function will create a new cursor that can be used to access the table being referenced by this *wxDdbTable* instance, or to execute direct SQL commands on without affecting the cursors that are already defined and possibly positioned.

Parameters

setCursor

OPTIONAL. Should this new cursor be set to be the current cursor after successfully creating the new cursor. Default is false.

bindColumns

OPTIONAL. Should this new cursor be bound to all the memory variables that the default cursor is bound to. Default is true.

Remarks

This new cursor must be closed using *wxDdbTable::DeleteCursor* (p. 341) by the calling program before the *wxDdbTable* instance is deleted, or both memory and resource leaks will occur.

wxDdbTable::GetNext

bool GetNext()

Retrieves the NEXT row in the record set after the current cursor position as defined by the current query. Before retrieving records, a query must be performed using *wxDdbTable::Query* (p. 353), *wxDdbTable::QueryOnKeyFields* (p. 357), *wxDdbTable::QueryMatching* (p. 356) or *wxDdbTable::QueryBySqlStmt* (p. 354).

Return value

This function returns false when the current cursor has reached the end of the result set. When false is returned, data in the bound columns is undefined.

Remarks

This function works with both forward and backward scrolling cursors.

See also *wxDdbTable::++* (p. 366)

wxDdbTable::GetNumberOfColumns

UWORD GetNumberOfColumns()

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

wxDbTable::GetOrderByClause**const wxString & GetOrderByClause()**

Accessor function that returns the current ORDER BY setting assigned with the `wxDbTable::SetOrderByClause` (p. 363).

See also

`wxDbTable::OrderBy` (p. 352)

wxDbTable::GetPrev**bool GetPrev()**

Retrieves the PREVIOUS row in the record set before the current cursor position as defined by the current query. Before retrieving records, a query must be performed using `wxDbTable::Query` (p. 353), `wxDbTable::QueryOnKeyFields` (p. 357), `wxDbTable::QueryMatching` (p. 356) or `wxDbTable::QueryBySqlStmt` (p. 354).

Return value

This function returns false when the current cursor has reached the beginning of the result set and there are now other rows prior to the cursors current position. When false is returned, data in the bound columns is undefined.

Remarks

This function can only be used if the datasource connection used by the `wxDbTable` instance was created with `FwdOnlyCursors` set to false. If the connection does not allow backward scrolling cursors, this function will return false, and the data contained in the bound columns will be undefined.

See also

`wxDb::IsFwdOnlyCursors` (p. 310), `wxDbTable::--` (p. 366)

wxDbTable::GetQueryTableName**const wxString & GetQueryTableName()**

Accessor function that returns the name of the table/view that was indicated as being the table/view to query against when this `wxDbTable` instance was created.

See also

`wxDbTable` constructor (p. 330)

wxDbTable::GetRowNum**UWORD GetRowNum()**

Returns the ODBC row number for performing positioned updates and deletes.

Remarks

This function is not being used within the ODBC class library and may be a candidate for removal if no use is found for it.

Row number with some datasources/ODBC drivers is the position in the result set, while in others it may be a physical position in the database. Check your database documentation to find out which behavior is supported.

wxDbTable::GetTableName**const wxString & GetTableName()**

Accessor function that returns the name of the table that was indicated as being the table that this wxDbTable instance was associated with.

wxDbTable::GetTablePath**const wxString & GetTablePath()**

Accessor function that returns the path to the data table that was indicated during creation of this wxDbTable instance.

Remarks

Currently only applicable to dBase and MS-Access datasources.

wxDbTable::GetWhereClause**const wxString & GetWhereClause()**

Accessor function that returns the current WHERE setting assigned with *wxDbTable::SetWhereClause* (p. 364)

See also

wxDbTable::Where (p. 366)

wxDbTable::Insert**int Insert()**

Inserts a new record into the table being referenced by this wxDbTable instance. The values in the member variables of the wxDbTable instance are inserted into the columns of the new row in the database.

Return value

1)	DB_SUCCESS	Record inserted successfully (value =
	DB_FAILURE	Insert failed (value = 0)
	DB_ERR_INTEGRITY_CONSTRAINT_VIOL	The insert failed due to an integrity
unique		constraint violation (duplicate non-
		index entry) is attempted.

Remarks

A *wxD::CommitTrans* (p. 298) or *wxD::RollbackTrans* (p. 315) must be called after use of this function to commit or rollback the insertion.

Example

```
// Incomplete code snippet
wxStrcpy(parts->PartName, "10");
wxStrcpy(parts->PartDesc, "Part #10");
parts->Qty = 1000;
RETCODE retcode = parts->Insert();
switch(retcode)
{
    case DB_SUCCESS:
        parts->GetDb()->CommitTrans();
        return(true);
    case DB_ERR_INTEGRITY_CONSTRAINT_VIOL:
        // Current data would result in a duplicate key
        // on one or more indexes that do not allow duplicates
        parts->GetDb()->RollbackTrans();
        return(false);
    default:
        // Insert failed for some unexpected reason
        parts->GetDb()->RollbackTrans();
        return(false);
}
```

wxD::IsColNull

bool IsColNull(UWORD colNumber) const

Used primarily in the ODBC class library to determine if a column value is set to "NULL". Works for all data types supported by the ODBC class library.

Parameters

colNumber

The column number of the bound column as defined by the *wxD::SetColDefs* (p. 358) calls which defined the columns accessible to this *wxD* instance.

Remarks

NULL column support is currently not fully implemented as of wxWidgets 2.4.

wxDbTable::IsCursorClosedOnCommit**bool IsCursorClosedOnCommit()**

Accessor function to return information collected during the opening of the datasource connection that is used by this wxDbTable instance. The result returned by this function indicates whether an implicit closing of the cursor is done after a commit on the database connection.

Return value

Returns true if the cursor associated with this wxDbTable object is closed after a commit or rollback operation. Returns false otherwise.

Remarks

If more than one wxDbTable instance used the same database connection, all cursors which use the database connection are closed on the commit if this function indicates true.

wxDbTable::IsQueryOnly**bool IsQueryOnly()**

Accessor function that returns a value indicating if this wxDbTable instance was created to allow only queries to be performed on the bound columns. If this function returns true, then no actions may be performed using this wxDbTable instance that would modify (insert/delete/update) the table's data.

wxDbTable::Open**bool Open(bool checkPrivileges=false, bool checkTableExists=true)**

Every wxDbTable instance must be opened before it can be used. This function checks for the existence of the requested table, binds columns, creates required cursors, (insert/select and update if connection is not wxDB_QUERY_ONLY) and constructs the insert statement that is to be used for inserting data as a new row in the datasource.

NOTE: To retrieve data into an opened table, the of the table must be bound to the variables in the program via call(s) to *wxDbTable::SetColDefs* (p. 358) before calling *Open()*.

See the *database classes overview* (p. **Error! Bookmark not defined.**) for an introduction to using the ODBC classes.

Parameters*checkPrivileges*

Indicates whether the *Open()* function should check whether the current connected user has at least SELECT privileges to access the table to which they are trying to open. Default is false.

checkTableExists

Indicates whether the `Open()` function should check whether the table exists in the database or not before opening it. Default is true.

Remarks

If the function returns a false value due to the table not existing, a log entry is recorded for the datasource connection indicating the problem that was detected when checking for table existence. Note that it is usually best for the calling routine to check for the existence of the table and for sufficient user privileges to access the table in the mode (`wxDB_QUERY_ONLY` or `!wxDB_QUERY_ONLY`) before trying to open the table for the best possible explanation as to why a table cannot be opened.

Checking the user's privileges on a table can be quite time consuming during the open phase. With most applications, the programmer already knows that the user has sufficient privileges to access the table, so this check is normally not required.

For best performance, open the table, and then use the `wxDdb::TablePrivileges` (p. 318) function to check the users privileges. Passing a schema to the `TablePrivileges()` function can significantly speed up the privileges checks.

See also

`wxDdb::TableExists` (p. 317), `wxDdb::TablePrivileges` (p. 318) `wxDdbTable::SetColDefs` (p. 358)

wxDdbTable::OrderBy

const wxString & OrderBy()

void OrderBy(const wxString & OrderBy)

Accessor function for the private class member `wxDdbTable::orderBy`. Can be used as a synonym for `wxDdbTable::GetOrderByClause` (p. 347)(the first form of this function) or `wxDdbTable::SetOrderByClause` (p. 363)(the second form of this function).

Parameters

OrderBy

A comma separated list of column names that indicate the alphabetized/numeric sorting sequence that the result set is to be returned in. If a FROM clause has also been specified, each column name specified in the ORDER BY clause should be prefaced with the table name to which the column belongs using DOT notation (TABLE_NAME.COLUMN_NAME).

Return value

The first form of this function returns the current value of the `wxDdbTable` member variable `::orderBy`.

The second form of the function has no return value.

See also

wxDbTable::GetOrderByClause (p. 347), *wxDbTable::SetFromClause* (p. 361)

wxDbTable::Query

virtual bool Query(**bool** *forUpdate=false*, **bool** *distinct=false*)

Parameters

forUpdate

OPTIONAL. Gives you the option of locking records as they are retrieved. If the RDBMS is not capable of the FOR UPDATE clause, this argument is ignored. See *wxDbTable::CanSelectForUpdate* (p. 334) for additional information regarding this argument. Default is false.

distinct

OPTIONAL. Allows selection of only distinct values from the query (SELECT DISTINCT ... FROM ...). The notion of DISTINCT applies to all columns returned in the result set, not individual columns. Default is false.

Remarks

This function queries records from the datasource based on the three *wxDbTable* members: "where", "orderBy", and "from". Use *wxDbTable::SetWhereClause* (p. 364) to filter on records to be retrieved (e.g. All users with a first name of "JOHN"). Use *wxDbTable::SetOrderByClause* (p. 363) to change the sequence in which records are returned in the result set from the datasource (e.g. Ordered by LAST_NAME). Use *wxDbTable::SetFromClause* (p. 361) to allow inner joining of the base table (the one being associated with this instance of *wxDbTable*) with other tables which share a related field.

After each of these clauses are set/cleared, call *wxDbTable::Query()* to fetch the result set from the datasource.

This scheme has an advantage if you have to requery your record set frequently in that you only have to set your WHERE, ORDER BY, and FROM clauses once. Then to refresh the record set, simply call *wxDbTable::Query()* as frequently as needed.

Note that repeated calls to *wxDbTable::Query()* may tax the database server and make your application sluggish if done too frequently or unnecessarily.

The base table name is automatically prepended to the base column names in the event that the FROM clause has been set (is non-null) using *wxDbTable::SetFromClause* (p. 361).

The cursor for the result set is positioned *before* the first record in the result set after the query. To retrieve the first record, call either *wxDbTable::GetFirst* (p. 345) (only if backward scrolling cursors are available) or *wxDbTable::GetNext* (p. 347). Typically, no data from the result set is returned to the client driver until a request such as *wxDbTable::GetNext* (p. 347) is performed, so network traffic and database load are

not overwhelmed transmitting data until the data is actually requested by the client. This behavior is solely dependent on the ODBC driver though, so refer to the ODBC driver's reference material for information on its behaviors.

Values in the bound columns' memory variables are undefined after executing a call to this function and remain that way until a row in the result set is requested to be returned.

The `wxDBTable::Query()` function is defined as "virtual" so that it may be overridden for application specific purposes.

Be sure to set the `wxDBTable`'s "where", "orderBy", and "from" member variables to "" if they are not to be used in the query. Otherwise, the results returned may have unexpected results (or no results) due to improper or incorrect query parameters constructed from the uninitialized clauses.

Example

```
// Incomplete code sample
parts->SetWhereClause("DESCRIPTION = 'FOOD'");
parts->SetOrderByClause("EXPIRATION_DATE");
parts->SetFromClause("");
// Query the records based on the where, orderBy and from
clauses
// specified above
parts->Query();
// Display all records queried
while(parts->GetNext())
    dispPart(parts); // user defined function
```

wxDBTable::QueryBySqlStmt

bool QueryBySqlStmt(const wxString &pSqlStmt)

Performs a query against the datasource by accepting and passing verbatim the SQL SELECT statement passed to the function.

Parameters

pSqlStmt

Pointer to the SQL SELECT statement to be executed.

Remarks

This is the most powerful form of the query functions available. This member function allows a programmer to write their own custom SQL SELECT statement for requesting data from the datasource. This gives the programmer access to the full power of SQL for performing operations such as scalar functions, aggregate functions, table joins, and sub-queries, as well as datasource specific function calls.

The requirements of the SELECT statement are the following:

1. Must return the correct number of columns. In the derived `wxDBTable` constructor, it is specified how many columns are in the `wxDBTable` object. The SELECT statement must return exactly that many columns.

2. The columns must be returned in the same sequence as specified when defining the bounds columns `wxDbTable::SetColDefs` (p. 358), and the columns returned must be of the proper data type. For example, if column 3 is defined in the `wxDbTable` bound column definitions to be a float, the `SELECT` statement must return a float for column 3 (e.g. `PRICE * 1.10` to increase the price by 10).
3. The `ROWID` can be included in your `SELECT` statement as the **last** column selected, if the datasource supports it. Use `wxDbTable::CanUpdateByROWID()` to determine if the `ROWID` can be selected from the datasource. If it can, much better performance can be achieved on updates and deletes by including the `ROWID` in the `SELECT` statement.

Even though data can be selected from multiple tables (joins) in your select statement, only the base table associated with this `wxDbTable` object is automatically updated through the ODBC class library. Data from multiple tables can be selected for display purposes however. Include columns in the `wxDbTable` object and mark them as non-updateable (See `wxDbColDef` (p. 320) for details). This way columns can be selected and displayed from other tables, but only the base table will be updated automatically when performed through the `wxDbTable::Update` (p. 365) function after using this type of query. To update tables other than the base table, use the `wxDbTable::Update` (p. 365) function passing a SQL statement.

After this function has been called, the cursor is positioned before the first record in the record set. To retrieve the first record, call either `wxDbTable::GetFirst` (p. 345) or `wxDbTable::GetNext` (p. 347).

Example

```
// Incomplete code samples
wxString sqlStmt;
sqlStmt = "SELECT * FROM PARTS WHERE STORAGE_DEVICE = 'SD98' \
          AND CONTAINER = 12";
// Query the records using the SQL SELECT statement above
parts->QueryBySqlStmt(sqlStmt);
// Display all records queried
while(parts->GetNext())
    dispPart(&parts);

Example SQL statements
-----

// Table Join returning 3 columns
SELECT PART_NUM, part_desc, sd_name
from parts, storage_devices
where parts.storage_device_id =
       storage_devices.storage_device_id

// Aggregate function returning total number of
// parts in container 99
SELECT count(*) from PARTS where container = 99

// Order by clause; ROWID, scalar function
SELECT PART_NUM, substring(part_desc, 1, 10), qty_on_hand + 1,
ROWID
from parts
where warehouse = 10
order by PART_NUM desc           // descending order
```

```
// Subquery
SELECT * from parts
    where container in (select container
                        from storage_devices
                        where device_id = 12)
```

wxDbTable::QueryMatching

virtual bool QueryMatching(bool forUpdate=false, bool distinct=false)

QueryMatching allows querying of records from the table associated with the wxDbTable object by matching "columns" to values.

For example: To query the datasource for the row with a PART_NUMBER column value of "32", clear all column variables of the wxDbTable object, set the PartNumber variable that is bound to the PART_NUMBER column in the wxDbTable object to "32", and then call wxDbTable::QueryMatching().

Parameters

forUpdate

OPTIONAL. Gives you the option of locking records as they are queried (SELECT ... FOR UPDATE). If the RDBMS is not capable of the FOR UPDATE clause, this argument is ignored. See *wxDbTable::CanSelectForUpdate* (p. 334) for additional information regarding this argument. Default is false.

distinct

OPTIONAL. Allows selection of only distinct values from the query (SELECT DISTINCT ... FROM ...). The notion of DISTINCT applies to all columns returned in the result set, not individual columns. Default is false.

Remarks

The SQL WHERE clause is built by the ODBC class library based on all non-zero/non-NULL columns in your wxDbTable object. Matches can be on one, many or all of the wxDbTable's columns. The base table name is prepended to the column names in the event that the wxDbTable's FROM clause is non-null.

This function cannot be used to perform queries which will check for columns that are 0 or NULL, as the automatically constructed WHERE clause only will contain comparisons on column member variables that are non-zero/non-NULL.

The primary difference between this function and *wxDbTable::QueryOnKeyFields* (p. 357) is that this function can query on any column(s) in the wxDbTable object. Note however that this may not always be very efficient. Searching on non-indexed columns will always require a full table scan.

The cursor is positioned before the first record in the record set after the query is performed. To retrieve the first record, the program must call either *wxDbTable::GetFirst* (p. 345) or *wxDbTable::GetNext* (p. 347).

WHERE and FROM clauses specified using *wxDbTable::SetWhereClause* (p. 364) and

`wxDbTable::SetFromClause` (p. 361) are ignored by this function.

Example

```
// Incomplete code sample
parts->ClearMemberVars();           // Set all columns to zero
wxStrcpy(parts->PartNumber, "32");  // Set columns to query on
parts->OnHold = true;
parts->QueryMatching();             // Query
// Display all records queried
while(parts->GetNext())
    dispPart(parts); // Some application defined function
```

wxDbTable::QueryOnKeyFields

bool QueryOnKeyFields(bool forUpdate=false, bool distinct=false)

`QueryOnKeyFields` provides an easy mechanism to query records in the table associated with the `wxDbTable` object by the primary index column(s). Simply assign the primary index column(s) values and then call this member function to retrieve the record.

Note that since primary indexes are always unique, this function implicitly always returns a single record from the database. The base table name is prepended to the column names in the event that the `wxDbTable`'s `FROM` clause is non-null.

Parameters

forUpdate

OPTIONAL. Gives you the option of locking records as they are queried (`SELECT ... FOR UPDATE`). If the RDBMS is not capable of the `FOR UPDATE` clause, this argument is ignored. See `wxDbTable::CanSelectForUpdate` (p. 334) for additional information regarding this argument. Default is false.

distinct

OPTIONAL. Allows selection of only distinct values from the query (`SELECT DISTINCT ... FROM ...`). The notion of `DISTINCT` applies to all columns returned in the result set, not individual columns. Default is false.

Remarks

The cursor is positioned before the first record in the record set after the query is performed. To retrieve the first record, the program must call either `wxDbTable::GetFirst` (p. 345) or `wxDbTable::GetNext` (p. 347).

`WHERE` and `FROM` clauses specified using `wxDbTable::SetWhereClause` (p. 364) and `wxDbTable::SetFromClause` (p. 361) are ignored by this function.

Example

```
// Incomplete code sample
wxStrcpy(parts->PartNumber, "32");
parts->QueryOnKeyFields();
// Display all records queried
```

```
while(parts->GetNext())
    dispPart(parts); // Some application defined function
```

wxDbTable::Refresh

bool Refresh()

This function re-reads the bound columns into the memory variables, setting them to the current values stored on the disk.

The cursor position and result set are unaffected by calls to this function. (The one exception is in the case where the record to be refreshed has been deleted by some other user or transaction since it was originally retrieved as part of the result set. For most datasources, the default behavior in this situation is to return the value that was originally queried for the result set, even though it has been deleted from the database. But this is datasource dependent, and should be tested before relying on this behavior.)

Remarks

This routine is only guaranteed to work if the table has a unique primary index defined for it. Otherwise, more than one record may be fetched and there is no guarantee that the correct record will be refreshed. The table's columns are refreshed to reflect the current data in the database.

wxDbTable::SetColDefs

bool SetColDefs(UWORD index, const wxString &fieldName, int dataType, void *pData, SWORD cType, int size, bool keyField = false, bool updateable = true, bool insertAllowed = true, bool derivedColumn = false)

wxDbColDataPtr * SetColDefs(wxDbColInf *colInfs, UWORD numCols)

Parameters

index

Column number (0 to n-1, where n is the number of columns specified as being defined for this wxDbTable instance when the wxDbTable constructor was called.

fieldName

Column name from the associated data table.

dataType

Logical data type. Valid logical types include:

DB_DATA_TYPE_VARCHAR	: strings
DB_DATA_TYPE_INTEGER	: non-floating point numbers
DB_DATA_TYPE_FLOAT	: floating point numbers
DB_DATA_TYPE_DATE	: dates
DB_DATA_TYPE_BLOB	: binary large objects
DB_DATA_TYPE_MEMO	: large strings

pData

Pointer to the data object that will hold the column's value when a row of data is returned from the datasource.

cType

SQL C Type. This defines the data type that the SQL representation of the data is converted to to be stored in *pData*. Other valid types are available also, but these are the most common ones:

```
SQL_C_CHAR        // string - deprecated: use SQL_C_WXCHAR
SQL_C_WXCHAR      // string - Used transparently in unicode or
non-unicode builds
SQL_C_LONG
SQL_C_ULONG
SQL_C_SHORT
SQL_C_USHORT
SQL_C_FLOAT
SQL_C_DOUBLE
SQL_C_NUMERIC
SQL_C_TIMESTAMP

SQL_C_BOOLEAN     // defined in db.h
SQL_C_ENUM        // defined in db.h
```

size

Maximum size in bytes of the *pData* object.

keyField

OPTIONAL. Indicates if this column is part of the primary index. Default is false.

updateable

OPTIONAL. Are updates allowed on this column? Default is true.

insertAllowed

OPTIONAL. Inserts allowed on this column? Default is true.

derivedColumn

OPTIONAL. Is this a derived column (non-base table column for query only)? Default is false.

collInfs

Pointer to an array of *wxDboCollInf* instances which contains all the information necessary to create *numCols* column definitions.

numCols

Number of elements of *wxDboCollInf* type that are pointed to by *collInfs*, which are to have column definitions created from them.

Remarks

If *pData* is to hold a string of characters, be sure to include enough space for the NULL terminator in *pData* and in the byte count of *size*.

Using the first form of this function, if the column definition is not able to be created, a value of false is returned. If the specified index of the column exceeds the number of columns defined in the *wxDbTable* instance, an assert is thrown and logged (in debug builds) and a false is returned.

A failure to create the column definition in the second form results in a value of NULL being returned.

Both forms of this function provide a shortcut for defining the columns in your *wxDbTable* object. Use this function in any derived *wxDbTable* constructor when describing the column/columns in the *wxDbTable* object.

The second form of this function is primarily used when the *wxDb::GetColumns* (p. 304) function was used to query the datasource for the column definitions, so that the column definitions are already stored in *wxDbColInf* form. One example use of using *wxDb::GetColumns* (p. 304) then using this function is if a data table existed in one datasource, and the table's column definitions were to be copied over to another datasource or table.

Example

```
// Long way not using this function
wxStrcpy(colDefs[0].ColName, "PART_NUM");
colDefs[0].DbDataType = DB_DATA_TYPE_VARCHAR;
colDefs[0].PtrDataObj = PartNumber;
colDefs[0].SqlCtype = SQL_C_WXCHAR;
colDefs[0].SzDataObj = PART_NUMBER_LEN;
colDefs[0].KeyField = true;
colDefs[0].Updateable = false;
colDefs[0].InsertAllowed = true;
colDefs[0].DerivedCol = false;

// Shortcut using this function
SetColDefs(0, "PART_NUM", DB_DATA_TYPE_VARCHAR, PartNumber,
           SQL_C_WXCHAR, PART_NUMBER_LEN, true, false, true,
false);
```

wxDbTable::SetCursor

void SetCursor(HSTMT *hstmtActivate = (void **) wxDB_DEFAULT_CURSOR)

Parameters

hstmtActivate

OPTIONAL. Pointer to the cursor that is to become the current cursor. Passing no cursor handle will reset the cursor back to the *wxDbTable*'s default (original) cursor that was created when the *wxDbTable* instance was first created. Default is *wxDB_DEFAULT_CURSOR*.

Remarks

When swapping between cursors, the member variables of the `wxDbTable` object are automatically refreshed with the column values of the row that the current cursor is positioned at (if any). If the cursor is not positioned, then the data in member variables is undefined.

The only way to return back to the cursor that was in use before this function was called is to programmatically determine the current cursor's `HSTMTBEFORE` calling this function using `wxDbTable::GetCursor` (p. 345) and saving a pointer to that cursor.

See also

`wxDbTable::GetNewCursor` (p. 346), `wxDbTable::GetCursor` (p. 345), `wxDbTable::SetCursor` (p. 360)

`wxDbTable::SetFromClause`

`void SetFromClause(const wxString &From)`

Accessor function for setting the private class member `wxDbTable::from` that indicates what other tables should be inner joined with the `wxDbTable`'s base table for access to the columns in those other tables.

Synonym to this function is one form of `wxDbTable::From` (p. 344)

Parameters

From

A comma separated list of table names that are to be inner joined with the base table's columns so that the joined table's columns may be returned in the result set or used as a portion of a comparison with the base table's columns. NOTE that the base table's name must NOT be included in the FROM clause, as it is automatically included by the `wxDbTable` class in constructing query statements.

Remarks

Used by the `wxDbTable::Query` (p. 353) and `wxDbTable::Count` (p. 337) member functions to allow inner joining of records from multiple tables.

Do **not** include the keyword "FROM" when setting the FROM clause.

If using the FROM clause when performing a query, be certain to include in the corresponding WHERE clause a comparison of a column from either the base table or one of the other joined tables to each other joined table to ensure the datasource knows on which column values the tables should be joined on.

Example

```
...
// Base table is the "LOCATION" table, and it is being
// inner joined to the "PART" table via the field
"PART_NUMBER"
// that can be related between the two tables.
location->SetWhereClause("LOCATION.PART_NUMBER =
PART.PART_NUMBER")
```

```
location->SetFromClause( "PART" );  
...
```

See also

wxDbTable::From (p. 344), *wxDbTable::GetFromClause* (p. 346)

wxDbTable::SetColNull

bool SetColNull(UWORD *colNumber*, bool *set=true*)

bool SetColNull(const wxString &*colName*, bool *set=true*)

Both forms of this function allow a member variable representing a column in the table associated with this wxDbTable object to be set to NULL.

The first form allows the column to be set by the index into the column definitions used to create the wxDbTable instance, while the second allows the actual column name to be specified.

Parameters

colNumber

Index into the column definitions used when first defining this wxDbTable object.

colName

Actual data table column name that is to be set to NULL.

set

Whether the column is set to NULL or not. Passing true sets the column to NULL, passing false sets the column to be non-NULL. Default is true.

Remarks

No database updates are done by this function. It only operates on the member variables in memory. Use and insert or update function to store this value to disk.

wxDbTable::SetOrderByClause

void SetOrderByClause(const wxString &*OrderBy*)

Accessor function for setting the private class member wxDbTable::orderBy which determines sequence/ordering of the rows returned in the result set of a query.

A synonym to this function is one form of the function *wxDbTable::OrderBy* (p. 352)

Parameters

OrderBy

A comma separated list of column names that indicate the alphabetized sorting

sequence that the result set is to be returned in. If a FROM clause has also been specified, each column name specified in the ORDER BY clause should be prefaced with the table name to which the column belongs using DOT notation (TABLE_NAME.COLUMN_NAME).

Remarks

Do **not** include the keywords "ORDER BY" when setting the ORDER BY clause.

Example

```
...
parts->SetOrderByClause( "PART_DESCRIP, QUANTITY" );
...

...
location->SetOrderByClause( "LOCATION.POSITION,
PART.PART_NUMBER" );
...
```

See also

wxDbTable::OrderBy (p. 352), *wxDbTable::GetOrderByClause* (p. 347)

wxDbTable::SetQueryTimeout

bool SetQueryTimeout(UDWORD *nSeconds*)

Allows a time period to be set as the timeout period for queries.

Parameters

nSeconds

The number of seconds to wait for the query to complete before timing out.

Remarks

Neither Oracle or Access support this function as of yet. Other databases should be evaluated for support before depending on this function working correctly.

wxDbTable::SetWhereClause

void SetWhereClause(const wxString &*Where*)

Accessor function for setting the private class member wxDbTable::where that determines which rows are returned in the result set by the datasource.

A synonym to this function is one form of the function *wxDbTable::Where* (p. 366)

Parameters

Where

SQL "where" clause. This clause can contain any SQL language that is legal in

standard where clauses. If a FROM clause has also been specified, each column name specified in the ORDER BY clause should be prefaced with the table name to which the column belongs using DOT notation (TABLE_NAME.COLUMN_NAME).

Remarks

Do **not** include the keywords "WHERE" when setting the WHERE clause.

Example

```
...
// Simple where clause
parts->SetWhereClause("PART_NUMBER = '32'");
...
// Any comparison operators
parts->SetWhereClause("PART_DESCRIP LIKE 'HAMMER%'");
...
// Multiple comparisons, including a function call
parts->Where("QTY > 0 AND {fn UCASE(PART_DESCRIP)} LIKE
'%DRILL%'");
...
// Using parameters and multiple logical combinations
parts->Where("((QTY > 10) OR (ON_ORDER > 0)) AND ON_HOLD =
0");
...
// This query uses an inner join (requiring a FROM clause
also)
// that joins the PART and LOCATION table on the common field
// PART_NUMBER.
parts->Where("PART.ON_HOLD = 0 AND \
PART.PART_NUMBER = LOCATION.PART_NUMBER AND \
LOCATION.PART_NUMBER > 0");
```

See also

wxDbTable::Where (p. 366), *wxDbTable::GetWhereClause* (p. 349)

wxDbTable::Update

bool Update()

bool Update(const wxString &pSqlStmt)

The first form of this function will update the row that the current cursor is currently positioned at with the values in the memory variables that are bound to the columns. The actual SQL statement to perform the update is automatically created by the ODBC class, and then executed.

The second form of the function allows full access through SQL statements for updating records in the database. Write any valid SQL UPDATE statement and submit it to this function for execution. Sophisticated updates can be performed using the full power of the SQL dialect. The full SQL statement must have the exact syntax required by the driver/datasource for performing the update. This usually is in the form of:

```
UPDATE tablename SET col1=X, col2=Y, ... where ...
```

Parameters*pSqlStmt*

Pointer to SQL UPDATE statement to be executed.

Remarks

A *wxDdb::CommitTrans* (p. 298) or *wxDdb::RollbackTrans* (p. 315) must be called after use of this function to commit or rollback the update.

Example

```
wxString sqlStmt;  
sqlStmt = "update PART set QTY = 0 where PART_NUMBER = '32'";
```

wxDdbTable::UpdateWhere**bool UpdateWhere(const wxString &pWhereClause)**

Performs updates to the base table of the *wxDdbTable* object, updating only the rows which match the criteria specified in the *pWhereClause*.

All columns that are bound to member variables for this *wxDdbTable* instance that were defined with the "updateable" parameter set to true will be updated with the information currently held in the memory variable.

Parameters*pWhereClause*

Pointer to a valid SQL WHERE clause. Do not include the keyword 'WHERE'.

Remarks

Care should be used when updating columns that are part of indexes with this function so as not to violate an unique key constraints.

A *wxDdb::CommitTrans* (p. 298) or *wxDdb::RollbackTrans* (p. 315) must be called after use of this function to commit or rollback the update(s).

wxDdbTable::Where**const wxString & Where()****void Where(const wxString& Where)**

Accessor function for the private class member *wxDdbTable::where*. Can be used as a synonym for *wxDdbTable::GetWhereClause* (p. 349) (the first form of this function) to return the current where clause or *wxDdbTable::SetWhereClause* (p. 364) (the second form of this function) to set the where clause for this table instance.

Parameters

Where

A valid SQL WHERE clause. Do not include the keyword 'WHERE'.

Return value

The first form of this function returns the current value of the `wxDbTable` member variable `::where`.

The second form of the function has no return value, as it will always set the where clause successfully.

See also

`wxDbTable::GetWhereClause` (p. 349), `wxDbTable::SetWhereClause` (p. 364)

wxDbTable::operator ++

bool operator ++()

Synonym for `wxDbTable::GetNext` (p. 347)

See also

`wxDbTable::GetNext` (p. 347)

wxDbTable::operator --

bool operator --()

Synonym for `wxDbTable::GetPrev` (p. 347)

See also

`wxDbTable::GetPrev` (p. 347)

wxDbTableInf

```
tableName[0]    = 0;
tableType[0]    = 0;
tableRemarks[0] = 0;
numCols        = 0;
pColInf        = NULL;
```

Currently only used by `wxDb::GetCatalog` (p. 303) internally and `wxDbInf` (p. 329) 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 `awxDbCollnf` (p. 322) array instance (using `wxDb::GetColumns` (p. 304)) to maintain all information about the columns of a table in one memory structure.

Eventually, accessor functions will be added for this class

See the *database classes overview* (p. **Error! Bookmark not defined.**) for an

introduction to using the ODBC classes.

Include files

<wx/db.h>

wxDbTableInf::Initialize

Simply initializes all member variables to a cleared state. Called by the constructor automatically.

wxDbGridCollInfo

This class is used to define columns to be shown, names of the columns, order and type of data, when using *wxdbGridTableBase* (p. 369) to display a Table or query in a *wxGrid* (p. 621)

See the database grid example in *wxdbGridTableBase* (p. 369) for an introduction to using the *wxDbGrid* classes.

Include files

<wx/dbgrid.h>

wxDbGridCollInfo::wxDbGridCollInfo

wxDbGridCollInfo(int *colNumber*, wxString *type*, wxString *title*, wxDbGridCollInfo **next*)

Default constructor. See the database grid example in *wxdbGridTableBase* (p. 369) to see two different ways for adding columns.

Parameters

colNumber

Column number in the *wxDBTable* (p. 329) instance to be used (first column is 0).

type

Column type ,wxString specifying the grid name for the datatype in this column, or use wxGRID_VALUE_DBAUTO to determine the type automatically from the *wxDBColDef* (p. 320) definition

title

The column label to be used in the grid display

next

A pointer to the next `wxDbGridCollInfo` structure if using one-step construction, `NULL` terminates the list. Use `Null` also if using two step construction.

See the database grid example in `wxDbGridTableBase` (p. 369) to see two different ways for adding columns.

wxDbGridCollInfo::~~wxDbGridCollInfo

~wxDbGridCollInfo()

Destructor.

wxDbGridCollInfo::AddCollInfo

void AddCollInfo(int colNumber, wxString type, wxString title)

Use this member function for adding columns. See the database grid example in `wxDbGridTableBase` (p. 369) to see two different ways for adding columns.

It is important to note that this class is merely a specifier to the `wxDbGridTableBase` (p. 369) constructor. Changes made to this datatype after the `wxDbGridTableBase` (p. 369) is called will not have any effect.

Parameters*colNumber*

Column number in the `wxDbTable` (p. 329) instance to be used (first column is 0).

type

Column type ,`wxString` specifying the grid name for the datatype in this column, or use `wxGRID_VALUE_DBAUTO` to determine the type automatically from the `wxDbColDef` (p. 320) definition

title

The column label to be used in the grid display

Remarks

As `wxDbTable` must be defined with to have columns which match those to by a `wxDbGridCollInfo` info structure as this is the structure which informs the grid of how you want to display your `wxDbTable` (p. 329). If no datatype conversion or the referenced column number does not exist the the behavior is undefined.

See the example at `wxDbGridCollInfo::wxDbGridCollInfo` (p. 368).

wxDbGridTableBase

You can view a database table in a grid using this class.

If you are deriving your own `wxDbTable` subclass for your table, then you may consider overriding `GetCol()` and `SetCol()` to provide calculated fields. This does work but care should be taken when using `wxDbGridTableBase` in this way.

The constructor and `AssignDbTable()` call allows you to specify the ownership if the `wxDbTable` object pointer. If you tell `wxGridTableBase` to take ownership, it will delete the passed `wxDbTable` when a new one is assigned or `wxGridTableBase`'s destructor is called. However no checks for aliasing are done so `Assign(table,...,true);` `Assign(table,...,true);` is an error. If you need to request a table object the preferred way is that the client keeps ownership.

Derived From

`wxGridTableBase` (p. 678)

Include files

`<wx/dbgrid.h>`

Example

```
// First step, let's define wxDbTable
int numColumns = 2;
wxDbTable *table = new wxDbTable (db, tblName, numColumns);
int int_var;
wxChar string_name[255];
table->SetColDef (0, "column 0", DB_DATA_TYPE_INTEGER,
&int_var,
                SQL_C_LONG, sizeof(int_var), true);
table->SetColDef (1, "column 1", DB_DATA_TYPE_VARCHAR,
&string_name,
                SQL_C_LONG, sizeof(string_name), false);

// now let's define columns in the grid

// first way to do it
wxDbGridColInfo *columns;
columns = new wxDbGridColInfo(0, wxGRID_VALUE_LONG, "first
column",
                new wxDbGridColInfo(1, wxGRID_VALUE_STRING, "second
column",
                NULL));

// second way to do it
wxDbGridColInfo *columns;
// first column is special
columns = new wxDbGridColInfo(0, wxGRID_VALUE_LONG, "first
column", NULL);
// all the rest
columns->AddColInfo (1, wxGRID_VALUE_STRING, "second column");

// second way may be better when columns are not known at
compile time

// now, let's open the table and make a Query()
table->Open();
// this step is very important
table->SetRowMode (wxDbTable::WX_ROW_MODE_QUERY);
// in the grid we will see only the rows of the result query
m_dbTable->Query();
```

```
        wxDbGridTableBase *dbgrid = new wxDbGridTableBase(table,
columns, wxUSE_QUERY, true);
        delete columns; // not needed anymore
        wxGrid *grid = new wxGrid ( ... );
        grid->SetTable(dbgrid, true);
        grid->Fit();
```

Include files

<wx/dbgrid.h>

Helper classes and data structures

wxDbGridTableBase::wxDbGridTableBase

wxDbGridTableBase(wxDbTable *tab, wxDbGridCollInfo *CollInfo, int count = wxUSE_QUERY, bool takeOwnership = true)

Constructor.

Parameters

tab

The database table you want to display. Must be opened and queried before display the grid. See the example *above* (p. 369).

CollInfo

Columns titles, and other values. See *wxDbGridCollInfo* (p. 367).

count

You can use a query result set (wxUSE_QUERY, to use wxDbTable::Count(wxDbTable::Count() or you can fix the total number of rows (count >= 0) to display, or specify it if you already know the size in avoid calling

takeOwnership

If true, this class deletes wxDbTable when it stops referring to it, if false application must take care of deleting it.

wxDbGridTableBase::ValidateRow

void ValidateRow(int row)

It ensures that the row data is fetched from the database, and if the wxDbTable local buffer, the row number passed should be the grid row.

Parameters

row

Row where validation must be done.

wxDcGridTableBase::UpdateRow

bool UpdateRow(int row)

If row has changed it forces that row to be written back to the database, however support for detecting whether insert/update is required is currently not in wxDbTable, so this function is currently unsupported.

Parameters

row

Row you want to update.

wxDcGridTableBase::AssignDbTable

bool AssignDbTable(wxDcTable *tab, int count = wxUSE_QUERY, bool takeOwnership = true)

Resets the grid for using with a new database table, but using the same columns definition. This can be useful when re-querying the database and want to see the changes.

Parameters

tab

Database table you want to assign to the grid.

count

Number of rows you want to show or wxUSE_QUERY for using a query.

takeOwnership

If false, user must take care of deleting tab after deleting the wxDbGridTableBase. If true, deletion is made by destructor class.

wxDC

A wxDC is a *device context* onto which graphics and text can be drawn. It is intended to represent a number of output devices in a generic way, so a window can have a device context associated with it, and a printer also has a device context. In this way, the same piece of code may write to a number of different devices, if the device context is used as a parameter.

Notice that wxDC is an abstract base class and can't be created directly, please use *wxPaintDC* (p. **Error! Bookmark not defined.**), *wxClientDC* (p. 151), *wxWindowDC* (p. **Error! Bookmark not defined.**), *wxScreenDC* (p. **Error! Bookmark not defined.**), *wxMemoryDC* (p. **Error! Bookmark not defined.**) or *wxPrinterDC* (p. **Error! Bookmark not defined.**).

Please note that in addition to the versions of the methods documented here, there are also versions which accept single `wxPoint` parameter instead of two `wxCoord` ones or `wxPoint` and `wxSize` instead of four of them.

Derived from

wxObject (p. **Error! Bookmark not defined.**)

Include files

<wx/dc.h>

See also

Overview (p. **Error! Bookmark not defined.**)

wxDC::Blit

bool **Blit**(**wxCoord** *xdest*, **wxCoord** *ydest*, **wxCoord** *width*, **wxCoord** *height*, **wxDC*** *source*, **wxCoord** *xsrc*, **wxCoord** *ysrc*, **int** *logicalFunc* = `wxCOPY`, **bool** *useMask* = `false`, **wxCoord** *xsrcMask* = -1, **wxCoord** *ysrcMask* = -1)

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

Parameters

xdest

Destination device context x position.

ydest

Destination device context y position.

width

Width of source area to be copied.

height

Height of source area to be copied.

source

Source device context.

xsrc

Source device context x position.

ysrc

Source device context y position.

logicalFunc

Logical function to use: see *wxDC::SetLogicalFunction* (p. 389).

useMask

If true, Blit does a transparent blit using the mask that is associated with the bitmap selected into the source device context. The Windows implementation does the following if MaskBlt cannot be used:

1. Creates a temporary bitmap and copies the destination area into it.
2. Copies the source area into the temporary bitmap using the specified logical function.
3. Sets the masked area in the temporary bitmap to BLACK by ANDing the mask bitmap with the temp bitmap with the foreground colour set to WHITE and the bg colour set to BLACK.
4. Sets the unmasked area in the destination area to BLACK by ANDing the mask bitmap with the destination area with the foreground colour set to BLACK and the background colour set to WHITE.
5. ORs the temporary bitmap with the destination area.
6. Deletes the temporary bitmap.

This sequence of operations ensures that the source's transparent area need not be black, and logical functions are supported.

Note: on Windows, blitting with masks can be speeded up considerably by compiling *wxWidgets* with the *wxUSE_DC_CACHE* option enabled. You can also influence whether MaskBlt or the explicit mask blitting code above is used, by using *wxSystemOptions* (p. **Error! Bookmark not defined.**) and setting the **no-maskblt** option to 1.

xsrcMask

Source x position on the mask. If both *xsrcMask* and *ysrcMask* are -1, *xsrc* and *ysrc* will be assumed for the mask source position. Currently only implemented on Windows.

ysrcMask

Source y position on the mask. If both *xsrcMask* and *ysrcMask* are -1, *xsrc* and *ysrc* will be assumed for the mask source position. Currently only implemented on Windows.

Remarks

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

See *wxMemoryDC* (p. **Error! Bookmark not defined.**) for typical usage.

See also

wxMemoryDC (p. **Error! Bookmark not defined.**), *wxBitmap* (p. 84), *wxMask* (p. 920)

wxDC::CalcBoundingBox

void CalcBoundingBox(wxCoord x, wxCoord y)

Adds the specified point to the bounding box which can be retrieved with *MinX* (p. 387), *MaxX* (p. 386) and *MinY* (p. 387), *MaxY* (p. 386) functions.

See also

ResetBoundingBox (p. 387)

wxDC::Clear

void Clear()

Clears the device context using the current background brush.

wxDC::ComputeScaleAndOrigin

virtual void ComputeScaleAndOrigin()

Performs all necessary computations for given platform and context type after each change of scale and origin parameters. Usually called automatically internally after such changes.

wxDC::CrossHair

void CrossHair(wxCoord x, wxCoord y)

Displays a cross hair using the current pen. This is a vertical and horizontal line the height and width of the window, centred on the given point.

wxDC::DestroyClippingRegion

void DestroyClippingRegion()

Destroys the current clipping region so that none of the DC is clipped. See also *wxDC::SetClippingRegion* (p. 388).

wxDC::DeviceToLogicalX

wxCoord DeviceToLogicalX(wxCoord x)

Convert device X coordinate to logical coordinate, using the current mapping mode.

wxDC::DeviceToLogicalXRel**wxCoord DeviceToLogicalXRel(wxCoord x)**

Convert device X coordinate to relative logical coordinate, using the current mapping mode but ignoring the x axis orientation. Use this function for converting a width, for example.

wxDC::DeviceToLogicalY**wxCoord DeviceToLogicalY(wxCoord y)**

Converts device Y coordinate to logical coordinate, using the current mapping mode.

wxDC::DeviceToLogicalYRel**wxCoord DeviceToLogicalYRel(wxCoord y)**

Convert device Y coordinate to relative logical coordinate, using the current mapping mode but ignoring the y axis orientation. Use this function for converting a height, for example.

wxDC::DrawArc**void DrawArc(wxCoord x1, wxCoord y1, wxCoord x2, wxCoord y2, wxCoord xc, wxCoord yc)**

Draws an arc of a circle, centred on (xc, yc), with starting point (x1, y1) and ending at (x2, y2). The current pen is used for the outline and the current brush for filling the shape.

The arc is drawn in an anticlockwise direction from the start point to the end point.

wxDC::DrawBitmap**void DrawBitmap(const wxBitmap& bitmap, wxCoord x, wxCoord y, bool transparent)**

Draw a bitmap on the device context at the specified point. If *transparent* is true and the bitmap has a transparency mask, the bitmap will be drawn transparently.

When drawing a mono-bitmap, the current text foreground colour will be used to draw the foreground of the bitmap (all bits set to 1), and the current text background colour to draw the background (all bits set to 0). See also *SetTextForeground* (p. 391), *SetTextBackground* (p. 390) and *wxMemoryDC* (p. **Error! Bookmark not defined.**).

wxDC::DrawCheckMark**void DrawCheckMark(wxCoord x, wxCoord y, wxCoord width, wxCoord height)****void DrawCheckMark(const wxRect &rect)**

Draws a check mark inside the given rectangle.

wxDC::DrawCircle

void DrawCircle(wxCoord x, wxCoord y, wxCoord radius)

void DrawCircle(const wxPoint& pt, wxCoord radius)

Draws a circle with the given centre and radius.

See also

DrawEllipse (p. 377)

wxDC::DrawEllipse

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

void DrawEllipse(const wxPoint& pt, const wxSize& size)

void DrawEllipse(const wxRect& rect)

Draws an ellipse contained in the rectangle specified either with the given top left corner and the given size or directly. The current pen is used for the outline and the current brush for filling the shape.

See also

DrawCircle (p. 376)

wxDC::DrawEllipticArc

**void DrawEllipticArc(wxCoord x, wxCoord y, wxCoord width, wxCoord height,
double start, double end)**

Draws an arc of an ellipse. The current pen is used for drawing the arc and the current brush is used for drawing the pie.

x and *y* specify the *x* and *y* coordinates of the upper-left corner of the rectangle that contains the ellipse.

width and *height* specify the width and height of the rectangle that contains the ellipse.

start and *end* specify the start and end of the arc relative to the three-o'clock position from the center of the rectangle. Angles are specified in degrees (360 is a complete circle). Positive values mean counter-clockwise motion. If *start* is equal to *end*, a complete ellipse will be drawn.

wxDC::DrawIcon

void DrawIcon(const wxIcon& icon, wxCoord x, wxCoord y)

Draw an icon on the display (does nothing if the device context is PostScript). This can be the simplest way of drawing bitmaps on a window.

wxDC::DrawLabel

```
virtual void DrawLabel(const wxString& text,                const wxBitmap&  
image,                const wxRect& rect,                int alignment =  
wxALIGN_LEFT | wxALIGN_TOP,                int indexAccel = -1,  
wxRect *rectBounding = NULL)
```

```
void DrawLabel(const wxString& text, const wxRect& rect,                int  
alignment = wxALIGN_LEFT | wxALIGN_TOP,                int indexAccel = -1)
```

Draw optional bitmap and the text into the given rectangle and aligns it as specified by alignment parameter; it also will emphasize the character with the given index if it is != -1 and return the bounding rectangle if required.

wxDC::DrawLine

```
void DrawLine(wxCoord x1, wxCoord y1, wxCoord x2, wxCoord y2)
```

Draws a line from the first point to the second. The current pen is used for drawing the line. Note that the point (x2, y2) is *not* part of the line and is not drawn by this function (this is consistent with the behaviour of many other toolkits).

wxDC::DrawLines

```
void DrawLines(int n, wxPoint points[], wxCoord xoffset = 0, wxCoord yoffset = 0)
```

```
void DrawLines(wxList *points, wxCoord xoffset = 0, wxCoord yoffset = 0)
```

Draws lines using an array of *points* of size *n*, or list of pointers to points, adding the optional offset coordinate. The current pen is used for drawing the lines. The programmer is responsible for deleting the list of points.

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

wxPerl note: The wxPerl version of this method accepts as its first parameter a reference to an array of wxPoint objects.

wxDC::DrawPolygon

```
void DrawPolygon(int n, wxPoint points[], wxCoord xoffset = 0, wxCoord yoffset = 0,  
int fill_style = wxODDEVEN_RULE)
```

```
void DrawPolygon(wxList *points, wxCoord xoffset = 0, wxCoord yoffset = 0,  
int fill_style = wxODDEVEN_RULE)
```

Draws a filled polygon using an array of *points* of size *n*, or list of pointers to points, adding the optional offset coordinate.

The last argument specifies the fill rule: **wxODDEVEN_RULE** (the default) or **wxWINDING_RULE**.

The current pen is used for drawing the outline, and the current brush for filling the shape. Using a transparent brush suppresses filling. The programmer is responsible for deleting the list of points.

Note that wxWidgets automatically closes the first and last points.

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

wxPerl note: The wxPerl version of this method accepts `$points` as its first parameter a reference to an array of wxPoint objects.

wxDC::DrawPolyPolygon

```
void DrawPolyPolygon(int n, int count[], wxPoint points[], wxCoord xoffset = 0,  
wxCoord yoffset = 0,  
int fill_style = wxODDEVEN_RULE)
```

Draws two or more filled polygons using an array of *points*, adding the optional offset coordinates.

Notice that for the platforms providing a native implementation of this function (Windows and PostScript-based wxDC currently), this is more efficient than using *DrawPolygon* (p. 378) in a loop.

n specifies the number of polygons to draw, the array *count* of size *n* specifies the number of points in each of the polygons in the *points* array.

The last argument specifies the fill rule: **wxODDEVEN_RULE** (the default) or **wxWINDING_RULE**.

The current pen is used for drawing the outline, and the current brush for filling the shape. Using a transparent brush suppresses filling.

The polygons maybe disjoint or overlapping. Each polygon specified in a call to **DrawPolyPolygon** must be closed. Unlike polygons created by the *DrawPolygon* (p. 378) member function, the polygons created by **DrawPolyPolygon** are not closed automatically.

wxPython note: Not implemented yet

wxPerl note: Not implemented yet

wxDC::DrawPoint

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

Draws a point using the color of the current pen. Note that the other properties of the pen are not used, such as width etc..

wxDC::DrawRectangle**void DrawRectangle(wxCoord x, wxCoord y, wxCoord width, wxCoord height)**

Draws a rectangle with the given top left corner, and with the given size. The current pen is used for the outline and the current brush for filling the shape.

wxDC::DrawRotatedText**void DrawRotatedText(const wxString& text, wxCoord x, wxCoord y, double angle)**

Draws the text rotated by *angle* degrees.

NB: Under Win9x only TrueType fonts can be drawn by this function. In particular, a font different from `wxNORMAL_FONT` should be used as the latter is not a TrueType font. `wxSWISS_FONT` is an example of a font which is.

See also

DrawText (p. 380)

wxDC::DrawRoundedRectangle**void DrawRoundedRectangle(wxCoord x, wxCoord y, wxCoord width, wxCoord height, double radius)**

Draws a rectangle with the given top left corner, and with the given size. The corners are quarter-circles using the given radius. The current pen is used for the outline and the current brush for filling the shape.

If *radius* is positive, the value is assumed to be the radius of the rounded corner. If *radius* is negative, the absolute value is assumed to be the *proportion* of the smallest dimension of the rectangle. This means that the corner can be a sensible size relative to the size of the rectangle, and also avoids the strange effects X produces when the corners are too big for the rectangle.

wxDC::DrawSpline**void DrawSpline(int n, wxPoint points[])**

Draws a spline between all given control points, using the current pen.

void DrawSpline(wxList *points)

Draws a spline between all given control points, using the current pen. Doesn't delete the `wxList` and contents.

void DrawSpline(wxCoord x1, wxCoord y1, wxCoord x2, wxCoord y2, wxCoord x3, wxCoord y3)

Draws a three-point spline using the current pen.

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

wxPerl note: The wxPerl version of this method accepts a reference to an array of wxPoint objects.

wxDC::DrawText

void DrawText(const wxString& text, wxCoord x, wxCoord y)

Draws a text string at the specified point, using the current text font, and the current text foreground and background colours.

The coordinates refer to the top-left corner of the rectangle bounding the string. See *wxDC::GetTextExtent* (p. 384) 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. 382) 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::EndPage

void EndPage()

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

wxDC::FloodFill

bool FloodFill(wxCoord x, wxCoord y, const wxColour& colour, int style=wxFLOOD_SURFACE)

Flood fills the device context starting from the given point, using the *current brush colour*, and using a style:

- **wxFLOOD_SURFACE:** the flooding occurs until a colour other than the given colour is encountered.
- **wxFLOOD_BORDER:** the area to be flooded is bounded by the given colour.

Returns false if the operation failed.

Note: The present implementation for non-Windows platforms may fail to find colour borders if the pixels do not match the colour exactly. However the function will still return true.

wxDC::GetBackground**const wxBrush& GetBackground() const**

Gets the brush used for painting the background (see *wxDC::SetBackground* (p. 388)).

wxDC::GetBackgroundMode**int GetBackgroundMode() const**

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

See also

SetBackgroundMode (p. 388)

wxDC::GetBrush**const wxBrush& GetBrush() const**

Gets the current brush (see *wxDC::SetBrush* (p. 388)).

wxDC::GetCharHeight**wxCoord GetCharHeight()**

Gets the character height of the currently set font.

wxDC::GetCharWidth**wxCoord GetCharWidth()**

Gets the average character width of the currently set font.

wxDC::GetClippingBox**void GetClippingBox(wxCoord *x, wxCoord *y, wxCoord *width, wxCoord *height)**

Gets the rectangle surrounding the current clipping region.

wxPython note: No arguments are required and the four values defining the rectangle are returned as a tuple.

wxPerl note: This method takes no arguments and returns a four element list(`x`, `y`, `width`, `height`)

wxDC::GetFont**const wxFont& GetFont() const**

Gets the current font. Notice that even although each device context object has some

default font after creation, this method would return a `wxNullFont` initially and only after calling `wxDC::SetFont` (p. 389) a valid font is returned.

wxDC::GetLogicalFunction

int GetLogicalFunction()

Gets the current logical function (see `wxDC::SetLogicalFunction` (p. 389)).

wxDC::GetMapMode

int GetMapMode()

Gets the *mapping mode* for the device context (see `wxDC::SetMapMode` (p. 389)).

wxDC::GetPartialTextExtents

bool GetPartialTextExtents(const wxString& text, wxArrayInt& widths) const

Fills the *widths* array with the widths from the beginning of *text* to the corresponding character of *text*. The generic version simply builds a running total of the widths of each character using `GetTextExtent` (p. 384), however if the various platforms have a native API function that is faster or more accurate than the generic implementation then it should be used instead.

wxPython note: This method only takes the *text* parameter and returns a Python list of integers.

wxDC::GetPen

const wxPen& GetPen() const

Gets the current pen (see `wxDC::SetPen` (p. 390)).

wxDC::GetPixel

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

Gets in *colour* the colour at the specified location. Not available for `wxPostScriptDC` or `wxMetafileDC`.

Note that setting a pixel can be done using `DrawPoint` (p. 379).

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

wxPerl note: This method only takes the parameters *x* and *y* and returns a `Wx::Colour` value

wxDC::GetPPI

wxSize GetPPI() const

Returns the resolution of the device in pixels per inch.

wxDC::GetSize

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

wxSize GetSize() const

This gets the horizontal and vertical resolution in device units. It can be used to scale graphics to fit the page. For example, if *maxX* and *maxY* represent the maximum horizontal and vertical 'pixel' values used in your application, the following code will scale the graphic to fit on the printer page:

```
wxCoord w, h;
dc.GetSize(&w, &h);
double scaleX=(double)(maxX/w);
double scaleY=(double)(maxY/h);
dc.SetUserScale(min(scaleX,scaleY),min(scaleX,scaleY));
```

wxPython note: In place of a single overloaded method name, wxPython implements the following methods:

GetSize()	Returns a wxSize
GetSizeTuple()	Returns a 2-tuple (width, height)

wxPerl note: In place of a single overloaded method, wxPerl uses:

GetSize()	Returns a Wx::Size
GetSizeWH()	Returns a 2-element list (width, height)

wxDC::GetSizeMM

void GetSizeMM(wxCoord *width, wxCoord *height) const

wxSize GetSizeMM() const

Returns the horizontal and vertical resolution in millimetres.

wxDC::GetTextBackground

const wxColour& GetTextBackground() const

Gets the current text background colour (see *wxDC::SetTextBackground* (p. 390)).

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).

If the optional parameter *font* is specified and valid, then it is used for the text extent calculation. Otherwise the currently selected font is.

See also *wxFont* (p. 561), *wxDC::SetFont* (p. 389).

wxPython note: The following methods are implemented in wxPython:

GetTextExtent(string)	Returns a 2-tuple, (width, height)
GetFullTextExtent(string, font=NULL)	Returns a 4-tuple, (width, height, descent, externalLeading)

wxPerl note: In wxPerl this method is implemented as **GetTextExtent(string, font = undef)** returning a four element array (width, height, descent, externalLeading)

wxDC::GetTextForeground

```
const wxColour& GetTextForeground() const
```

Gets the current text foreground colour (see *wxDC::SetTextForeground* (p. 391)).

wxDC::GetUserScale

```
void GetUserScale(double *x, double *y)
```

Gets the current user scale factor (set by *SetUserScale* (p. 391)).

wxPerl note: In wxPerl this method takes no arguments and return a two element array (x, y)

wxDC::GradientFillConcentric

```
void GradientFillConcentric(const wxRect& rect, const wxColour& initialColour,  
    const wxColour& destColour)
```

```
void GradientFillConcentric(const wxRect& rect, const wxColour& initialColour,  
    const wxColour& destColour, const wxPoint& circleCenter)
```

Fill the area specified by *rect* with a radial gradient, starting from *initialColour* at the centre of the circle and fading to *destColour* on the circle outside.

circleCenter are the relative coordinates of centre of the circle in the specified *rect*. If not specified, the circle is placed at the centre of *rect*.

Note: Currently this function is very slow, don't use it for real-time drawing.

wxDC::GradientFillLinear

void GradientFillLinear(const wxRect& *rect*, const wxColour& *initialColour*, const wxColour& *destColour*, wxDirection *nDirection* = wxEAST)

Fill the area specified by *rect* with a linear gradient, starting from *initialColour* and eventually fading to *destColour*. The *nDirection* specifies the direction of the colour change, default is to use *initialColour* on the left part of the rectangle and *destColour* on the right one.

wxDC::LogicalToDeviceX

wxCoord LogicalToDeviceX(wxCoord *x*)

Converts logical X coordinate to device coordinate, using the current mapping mode.

wxDC::LogicalToDeviceXRel

wxCoord LogicalToDeviceXRel(wxCoord *x*)

Converts logical X coordinate to relative device coordinate, using the current mapping mode but ignoring the x axis orientation. Use this for converting a width, for example.

wxDC::LogicalToDeviceY

wxCoord LogicalToDeviceY(wxCoord *y*)

Converts logical Y coordinate to device coordinate, using the current mapping mode.

wxDC::LogicalToDeviceYRel

wxCoord LogicalToDeviceYRel(wxCoord *y*)

Converts logical Y coordinate to relative device coordinate, using the current mapping mode but ignoring the y axis orientation. Use this for converting a height, for example.

wxDC::MaxX

wxCoord MaxX()

Gets the maximum horizontal extent used in drawing commands so far.

wxDC::MaxY

wxCoord MaxY()

Gets the maximum vertical extent used in drawing commands so far.

wxDC::MinX**wxCoord MinX()**

Gets the minimum horizontal extent used in drawing commands so far.

wxDC::MinY**wxCoord MinY()**

Gets the minimum vertical extent used in drawing commands so far.

wxDC::Ok**bool Ok()**

Returns true if the DC is ok to use.

wxDC::ResetBoundingBox**void ResetBoundingBox()**

Resets the bounding box: after a call to this function, the bounding box doesn't contain anything.

See also

CalcBoundingBox (p. 374)

wxDC::SetAxisOrientation**void SetAxisOrientation(bool *xLeftRight*, bool *yBottomUp*)**

Sets the x and y axis orientation (i.e., the direction from lowest to highest values on the axis). The default orientation is x axis from left to right and y axis from top down.

Parameters*xLeftRight*

True to set the x axis orientation to the natural left to right orientation, false to invert it.

yBottomUp

True to set the y axis orientation to the natural bottom up orientation, false to invert it.

wxDC::SetBackground

void SetBackground(const wxBrush& brush)

Sets the current background brush for the DC.

wxDC::SetBackgroundMode

void SetBackgroundMode(int mode)

mode may be one of `wxSOLID` and `wxTRANSPARENT`. This setting determines whether text will be drawn with a background colour or not.

wxDC::SetBrush

void SetBrush(const wxBrush& brush)

Sets the current brush for the DC.

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

See also *wxBrush* (p. 108).

See also *wxMemoryDC* (p. **Error! Bookmark not defined.**) for the interpretation of colours when drawing into a monochrome bitmap.

wxDC::SetClippingRegion

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

void SetClippingRegion(const wxPoint& pt, const wxSize& sz)

void SetClippingRegion(const wxRect& rect)

void SetClippingRegion(const wxRegion& region)

Sets the clipping region for this device context to the intersection of the given region described by the parameters of this method and the previously set clipping region. You should call *DestroyClippingRegion* (p. 375) if you want to set the clipping region exactly to the region specified.

The clipping region is an area to which drawing is restricted. Possible uses for the clipping region are for clipping text or for speeding up window redraws when only a known area of the screen is damaged.

See also

wxDC::DestroyClippingRegion (p. 375), *wxRegion* (p. **Error! Bookmark not defined.**)

wxDC::SetDeviceOrigin

void SetDeviceOrigin(wxCoord x, wxCoord y)

Sets the device origin (i.e., the origin in pixels after scaling has been applied).

This function may be useful in Windows printing operations for placing a graphic on a page.

wxDC::SetFont

void SetFont(const wxFont& font)

Sets the current font for the DC. It must be a valid font, in particular you should not pass `wxNullFont` to this method.

See also *wxFont* (p. 561).

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. 373)) 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>
<code>wxNOR</code>	<code>(NOT src) AND (NOT dst)</code>
<code>wxNO_OP</code>	<code>dst</code>
<code>wxOR</code>	<code>src OR dst</code>
<code>wxOR_INVERT</code>	<code>(NOT src) OR dst</code>
<code>wxOR_REVERSE</code>	<code>src OR (NOT dst)</code>
<code>wxSET</code>	<code>1</code>
<code>wxSRC_INVERT</code>	<code>NOT src</code>
<code>wxXOR</code>	<code>src XOR dst</code>

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. 391)) scales the text appropriately.

In Windows, scalable TrueType fonts are always used; in X, results depend on availability of fonts, but usually a reasonable match is found.

The coordinate origin is always at the top left of the screen/printer.

Drawing to a Windows printer device context uses the current mapping mode, but mapping mode is currently ignored for PostScript output.

The mapping mode can be one of the following:

<code>wxMM_TWIPS</code>	Each logical unit is 1/20 of a point, or 1/1440 of an inch.
<code>wxMM_POINTS</code>	Each logical unit is a point, or 1/72 of an inch.
<code>wxMM_METRIC</code>	Each logical unit is 1 mm.
<code>wxMM_LOMETRIC</code>	Each logical unit is 1/10 of a mm.
<code>wxMM_TEXT</code>	Each logical unit is 1 pixel.

wxDC::SetPalette

void SetPalette(const wxPalette& palette)

If this is a window DC or memory DC, assigns the given palette to the window or bitmap associated with the DC. If the argument is `wxNullPalette`, the current palette is selected out of the device context, and the original palette restored.

See *wxPalette* (p. **Error! Bookmark not defined.**) for further details.

wxDC::SetPen

void SetPen(const wxPen& pen)

Sets the current pen for the DC.

If the argument is `wxNullPen`, the current pen is selected out of the device context, and the original pen restored.

See also *wxMemoryDC* (p. **Error! Bookmark not defined.**) 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. **Error! Bookmark not defined.**) for the interpretation of colours when drawing into a monochrome bitmap.

wxDC::SetUserScale

void SetUserScale(double xScale, double yScale)

Sets the user scaling factor, useful for applications which require 'zooming'.

wxDC::StartDoc

bool StartDoc(const wxString& message)

Starts a document (only relevant when outputting to a printer). Message is a message to show while printing.

wxDC::StartPage

bool StartPage()

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

wxDCClipper

This is a small helper class which sets the specified DC to its constructor clipping region and then automatically destroys it in its destructor. Using it ensures that an unwanted clipping region is not left set on the DC.

Derived from

No base class

Include files

<wx/dc.h>

See also

wxDC (p. 372)

wxDCClipper::wxDCClipper

wxDCClipper(wxDC& dc, wxCoord x, wxCoord y, wxCoord w, wxCoord h,)

wxDCClipper(wxDC& dc, const wxRect& rect)

Constructor: sets the clipping region for the given device context to the specified

rectangle.

wxDCClipper::~wxDCClipper

~wxDCClipper()

Destructor: destroys the clipping region set in the constructor.

wxDDEClient

A wxDDEClient object represents the client part of a client-server DDE (Dynamic Data Exchange) conversation.

To create a client which can communicate with a suitable server, you need to derive a class from wxDDEConnection and another from wxDDEClient. The custom wxDDEConnection class will intercept communications in a 'conversation' with a server, and the custom wxDDEServer is required so that a user-overridden *wxDDEClient::OnMakeConnection* (p. 393) 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. **Error! Bookmark not defined.**).

Derived from

wxClientBase
wxObject (p. **Error! Bookmark not defined.**)

Include files

<wx/dde.h>

See also

wxDDEServer (p. 397), *wxDDEConnection* (p. 393), *Interprocess communications overview* (p. **Error! Bookmark not defined.**)

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

wxDDEClient::OnMakeConnection

wxConnectionBase * OnMakeConnection()

The type of `wxDDEConnection` (p. 393) returned from a `wxDDEClient::MakeConnection` (p. 393) 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. 212). 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. 392) object, or by the acceptance of a connection by a `wxDDEServer` (p. 397) 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. **Error! Bookmark not defined.**).

Derived from

`wxConnectionBase`
`wxObject` (p. **Error! Bookmark not defined.**)

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. 392), *wxDDEServer* (p. 397), *Interprocess communications overview* (p. [Error! Bookmark not defined.](#))

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. 398) and/or *wxDDEClient::OnMakeConnection* (p. 393) 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. 395) 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. 396) in that respect). Causes the server connection's *wxDDEConnection::OnExecute* (p. 395) 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. 395) 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. 397). 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. 396) 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. 396) 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. 396) 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. 396) 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. **Error! Bookmark not defined.**).

Derived from

wxServerBase

Include files

<wx/dde.h>

See also

wxDDEClient (p. 392), *wxDDEConnection* (p. 393), *IPC overview* (p. **Error! Bookmark not defined.**)

wxDDEServer::wxDDEServer

wxDDEServer()

Constructs a server object.

wxDDEServer::Create

bool Create(const wxString& service)

Registers the server using the given service name. Under UNIX, the string must contain an integer id which is used as an Internet port number. false is returned if the call failed (for example, the port number is already in use).

wxDDEServer::OnAcceptConnection

virtual wxConnectionBase * OnAcceptConnection(const wxString& topic)

When a client calls **MakeConnection**, the server receives the message and this

member is called. The application should derive a member to intercept this message and return a connection object of either the standard `wxDDEConnection` type, or of a user-derived type. If the topic is "STDIO", the application may wish to refuse the connection. Under UNIX, when a server is created the `OnAcceptConnection` message is always sent for standard input and output, but in the context of DDE messages it doesn't make a lot of sense.

wxDebugContext

A class for performing various debugging and memory tracing operations. Full functionality (such as printing out objects currently allocated) is only present in a debugging build of `wxWidgets`, i.e. if the `__WXDEBUG__` symbol is defined. `wxDebugContext` and related functions and macros can be compiled out by setting `wxUSE_DEBUG_CONTEXT` to 0 in `setup.h`.

Derived from

No parent class.

Include files

`<wx/memory.h>`

See also

Overview (p. **Error! Bookmark not defined.**)

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. 401)

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. 402)

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. 903) functionality.

See also

wxDebugContext::SetLevel (p. 402)

wxDebugContext::GetStream**ostream& GetStream()**

Returns the output stream associated with the debug context.

This is obsolete, replaced by *wxLog* (p. 903) functionality.

See also

wxDebugContext::SetStream (p. 403)

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. 903) 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. 903) functionality.

See also

wxDebugContext::SetStream (p. 403), *wxDebugContext::GetStream* (p. 400)

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. 401)

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. 401)

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. 399)

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. 399)

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. 903) functionality.

See also

wxDebugContext::GetLevel (p. 399)

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. 903) 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. 903) 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. 400), *wxDebugContext::HasStream* (p. 400)

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. 903) functionality.

Derived from

streambuf

Include files

<wx/memory.h>

Example

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

See also

Overview (p. **Error! Bookmark not defined.**)

wxDebugReport

wxDebugReport is used to generate a debug report, containing information about the program current state. It is usually used from *wxApp::OnFatalException()* (p. 42) as shown in the *sample* (p. **Error! Bookmark not defined.**).

A wxDebugReport object contains one or more files. A few of them can be created by the class itself but more can be created from the outside and then added to the report. Also note that several virtual functions may be overridden to further customize the class behaviour.

Once a report is fully assembled, it can simply be left in the temporary directory so that the user can email it to the developers (in which case you should still use *wxDebugReportCompress* (p. 408) to compress it in a single file) or uploaded to a Web server using *wxDebugReportUpload* (p. 410) (setting up the Web server to accept uploads is your responsibility, of course). Other handlers, for example for automatically emailing the report, can be defined as well but are not currently included in wxWidgets.

Example of use

```
wxDebugReport report;  
wxDebugReportPreviewStd preview;  
  
report.AddCurrentContext(); // could also use AddAll()  
report.AddCurrentDump();   // to do both at once  
  
if ( preview.Show(report) )  
    report.Process();
```

Derived from

No base class

Include files

<wx/debugrpt.h>

Data structures

This enum is used for functions that report either the current state or the state during the last (fatal) exception:

```
enum wxDebugReport::Context
{
    Context_Current,
    Context_Exception
};
```

wxDebugReport::wxDebugReport

wxDebugReport()

The constructor creates a temporary directory where the files that will be included in the report are created. Use *IsOk()* (p. 407) to check for errors.

wxDebugReport::~~wxDebugReport

~wxDebugReport()

The destructor normally destroys the temporary directory created in the constructor with all the files it contains. Call *Reset()* (p. 408) to prevent this from happening.

wxDebugReport::AddAll

void AddAll(Context context = Context_Exception)

Adds all available information to the report. Currently this includes a text (XML) file describing the process context and, under Win32, a minidump file.

wxDebugReport::AddContext

bool AddContext(Context ctx)

Add an XML file containing the current or exception context and the stack trace.

wxDebugReport::AddCurrentContext

bool AddCurrentContext()

The same as *AddContext(Context_Current)* (p. 405).

wxDebugReport::AddCurrentDump

bool AddCurrentDump()

The same as *AddDump(Context_Current)* (p. 405).

wxDebugReport::AddDump

bool AddDump(Context ctx)

Adds the minidump file to the debug report.

Minidumps are only available under recent Win32 versions (`dbghlp32.dll` can be installed under older systems to make minidumps available).

wxDebugReport::AddExceptionContext

bool AddExceptionContext()

The same as *AddContext(Context_Exception)* (p. 405).

wxDebugReport::AddExceptionDump

bool AddExceptionDump()

The same as *AddDump(Context_Exception)* (p. 405).

wxDebugReport::AddFile

void AddFile(const wxString& filename, const wxString& description)

Add another file to the report. If *filename* is an absolute path, it is copied to a file in the debug report directory with the same name. Otherwise the file should already exist in this directory

description only exists to be displayed to the user in the report summary shown by *wxDebugReportPreview* (p. 409).

See also

GetDirectory() (p. 407),
AddText() (p. 406)

wxDebugReport::AddText

bool AddText(const wxString& filename, const wxString& text, const wxString& description)

This is a convenient wrapper around *AddFile* (p. 406). It creates the file with the given *name* and writes *text* to it, then adds the file to the report. The *filename* shouldn't contain the path.

Returns `true` if file could be added successfully, `false` if an IO error occurred.

wxDebugReport::DoAddCustomContext

void DoAddCustomContext(wxXmlNode * nodeRoot)

This function may be overridden to add arbitrary custom context to the XML context file created by *AddContext* (p. 405). By default, it does nothing.

wxDebugReport::DoAddExceptionInfo**bool DoAddExceptionInfo(wxXmlNode* nodeContext)**

This function may be overridden to modify the contents of the exception tag in the XML context file.

wxDebugReport::DoAddLoadedModules**bool DoAddLoadedModules(wxXmlNode* nodeModules)**

This function may be overridden to modify the contents of the modules tag in the XML context file.

wxDebugReport::DoAddSystemInfo**bool DoAddSystemInfo(wxXmlNode* nodeSystemInfo)**

This function may be overridden to modify the contents of the system tag in the XML context file.

wxDebugReport::GetDirectory**const wxString& GetDirectory() const**

Returns the name of the temporary directory used for the files in this report.

This method should be used to construct the full name of the files which you wish to add to the report using *AddFile* (p. 406).

wxDebugReport::GetFile**bool GetFile(size_t n, wxString* name, wxString* desc) const**

Retrieves the name (relative to *GetDirectory()* (p. 407)) and the description of the file with the given index. If *n* is greater than or equal to the number of files, *false* is returned.

wxDebugReport::GetFilesCount**size_t GetFilesCount() const**

Gets the current number files in this report.

wxDebugReport::GetReportName**wxString GetReportName() const**

Gets the name used as a base name for various files, by default *wxApp::GetAppName()* (p. 38) is used.

wxDebugReport::IsOk**bool IsOk() const**

Returns `true` if the object was successfully initialized. If this method returns `false` the report can't be used.

wxDebugReport::Process**bool Process()**

Processes this report: the base class simply notifies the user that the report has been generated. This is usually not enough -- instead you should override this method to do something more useful to you.

wxDebugReport::RemoveFile**void RemoveFile(const wxString& name)**

Removes the file from report: this is used by *wxDebugReportPreview* (p. 409) to allow the user to remove files potentially containing private information from the report.

wxDebugReport::Reset**void Reset()**

Resets the directory name we use. The object can't be used any more after this as it becomes uninitialized and invalid.

wxDebugReportCompress

wxDebugReportCompress is a *wxDebugReport* (p. 404) which compresses all the files in this debug report into a single .ZIP file in its *Process()* function.

Derived from

wxDebugReport (p. 404)

Include files

<wx/debugrpt.h>

wxDebugReportCompress::wxDebugReportCompress**wxDebugReportCompress()**

Default constructor does nothing special.

wxDebugReportCompress::GetCompressedFileName**const wxString& GetCompressedFileName() const**

Returns the full path of the compressed file (empty if creation failed).

wxDebugReportPreview

This class presents the debug report to the user and allows him to veto report entirely or remove some parts of it. Although not mandatory, using this class is strongly recommended as data included in the debug report might contain sensitive private information and the user should be notified about it as well as having a possibility to examine the data which had been gathered to check whether this is effectively the case and discard the debug report if it is.

wxDebugReportPreview is an abstract base class, currently the only concrete class deriving from it is *wxDebugReportPreviewStd* (p. 409).

Derived from

No base class

Include files

<wx/debugrpt.h>

wxDebugReportPreview::wxDebugReportPreview**wxDebugReportPreview()**

Trivial default constructor.

wxDebugReportPreview::~~wxDebugReportPreview**~wxDebugReportPreview()**

dtor is trivial as well but should be virtual for a base class

wxDebugReportPreview::Show**bool Show(wxDebugReport& dbg_rpt) const**

Present the report to the user and allow him to modify it by removing some or all of the files and, potentially, adding some notes. Return `true` if the report should be processed or `false` if the user chose to cancel report generation or removed all files from it.

wxDebugReportPreviewStd

`wxDebugReportPreviewStd` is a standard debug report preview window. It displays a `GUIDialog` allowing the user to examine the contents of a debug report, remove files from and add notes to it.

Derived from

wxDebugReportPreview (p. 409)

Include files

<wx/debugrpt.h>

wxDebugReportPreviewStd::wxDebugReportPreviewStd

wxDebugReportPreviewStd()

Trivial default constructor.

wxDebugReportPreviewStd::Show

bool Show(wxDebugReport& *dbgrpt*) const

Show the dialog, see *wxDebugReportPreview::Show()* (p. 409) for more information.

wxDebugReportUpload

This class is used to upload a compressed file using HTTP POST request. As this class derives from `wxDebugReportCompress`, before upload the report is compressed in a single .ZIP file.

Derived from

wxDebugReportCompress (p. 408)

Include files

<wx/debugrpt.h>

wxDebugReportUpload::wxDebugReportUpload

wxDebugReportUpload(const wxString& *url*, const wxString& *input*, const wxString& *action*, const wxString& *curl* = `_T("curl")`)

This class will upload the compressed file created by its base class to an HTML multipart/form-data form at the specified address. The *url* is the upload page address, *input* is the name of the "type=file" control on the form used for the file name and *action* is the value of the form action field. The report is uploaded using *curl* program which should be available, the *curl* parameter may be used to specify the full path to it.

wxDebugReportUpload::OnServerReply

bool OnServerReply(const wxString& WXUNUSED(reply))

This function may be overridden in a derived class to show the output from curl: this may be an HTML page or anything else that the server returned. Value returned by this function becomes the return value of *wxDebugReport::Process()* (p. 408).

wxDelegateRendererNative

wxDelegateRendererNative allows reuse of renderers code by forwarding all the *wxRendererNative* (p. **Error! Bookmark not defined.**) methods to the given object and thus allowing you to only modify some of its methods -- without having to reimplement all of them.

Note that the "normal", inheritance-based approach, doesn't work with the renderers as it is impossible to derive from a class unknown at compile-time and the renderer is only chosen at run-time. So suppose that you want to only add something to the drawing of the tree control buttons but leave all the other methods unchanged -- the only way to do it, considering that the renderer class which you want to customize might not even be written yet when you write your code (it could be written later and loaded from a DLL during run-time), is by using this class.

Except for the constructor, it has exactly the same methods as *wxRendererNative* (p. **Error! Bookmark not defined.**) and their implementation is trivial: they are simply forwarded to the real renderer. Note that the "real" renderer may, in turn, be a *wxDelegateRendererNative* as well and that there may be arbitrarily many levels like this -- but at the end of the chain there must be a real renderer which does the drawing.

Derived from

wxRendererNative (p. **Error! Bookmark not defined.**)

Include files

<wx/renderer.h>

wxDelegateRendererNative::wxDelegateRendererNative

wxDelegateRendererNative()

wxDelegateRendererNative(wxRendererNative& rendererNative)

The default constructor does the same thing as the other one except that it uses the *generic renderer* (p. **Error! Bookmark not defined.**) instead of the user-specified *rendererNative*.

In any case, this sets up the delegate renderer object to follow all calls to the specified real renderer.

Note that this object does *not* take ownership of (i.e. won't delete) *rendererNative*.

wxDelegateRendererNative::DrawXXX

DrawXXX(...)

This class also provides all the virtual methods of *wxRendererNative* (p. **Error! Bookmark not defined.**), please refer to that class documentation for the details.

wxDialog

A dialog box is a window with a title bar and sometimes a system menu, which can be moved around the screen. It can contain controls and other windows and is usually used to allow the user to make some choice or to answer a question.

Derived from

wxTopLevelWindow (p. **Error! Bookmark not defined.**)
wxWindow (p. **Error! Bookmark not defined.**)
wxEvtHandler (p. 490)
wxObject (p. **Error! Bookmark not defined.**)

Include files

<wx/dialog.h>

Remarks

There are two kinds of dialog -- *modal* and *modeless*. A modal dialog blocks program flow and user input on other windows until it is dismissed, whereas a modeless dialog behaves more like a frame in that program flow continues, and input in other windows is still possible. To show a modal dialog you should use the *ShowModal* (p. 421) method while to show a dialog modelessly you simply use *Show* (p. 421), just as with frames.

Note that the modal dialog is one of the very few examples of *wxWindow*-derived objects which may be created on the stack and not on the heap. In other words, although this code snippet:

```
void AskUser()
{
    MyAskDialog *dlg = new MyAskDialog(...);
    if ( dlg->ShowModal() == wxID_OK )
        ...
    //else: dialog was cancelled or some another button
pressed
    dlg->Destroy();
}
```

works, you can also achieve the same result by using a simpler code fragment below:

```
void AskUser()
{
    MyAskDialog dlg(...);
```

```
        if ( dlg.ShowModal() == wxID_OK )
            ...

        // no need to call Destroy() here
    }
```

An application can define a *wxCloseEvent* (p. 157) handler for the dialog to respond to system close events.

Window styles

wxCAPTION	Puts a caption on the dialog box.
wxDEFAULT_DIALOG_STYLE	Equivalent to a combination of wxCAPTION , wxCLOSE_BOX and wxSYSTEM_MENU (the last one is not used under Unix)
wxRESIZE_BORDER	Display a resizable frame around the window.
wxSYSTEM_MENU	Display a system menu.
wxCLOSE_BOX	Displays a close box on the frame.
wxMAXIMIZE_BOX	Displays a maximize box on the dialog.
wxMINIMIZE_BOX	Displays a minimize box on the dialog.
wxTHICK_FRAME	Display a thick frame around the window.
wxSTAY_ON_TOP	The dialog stays on top of all other windows.
wxNO_3D	Under Windows, specifies that the child controls should not have 3D borders unless specified in the control.
wxDIALOG_NO_PARENT	By default, a dialog created with a <code>NULL</code> parent window will be given the <i>application's top level window</i> (p. 39) as parent. Use this style to prevent this from happening and create an orphan dialog. This is not recommended for modal dialogs.
wxDIALOG_EX_CONTEXTHELP	Under Windows, puts a query button on the caption. When pressed, Windows will go into a context-sensitive help mode and <code>wxWidgets</code> will send a <code>wxEVT_HELP</code> event if the user clicked on an application window. <i>Note</i> that this is an extended style and must be set by calling <i>SetExtraStyle</i> (p. Error! Bookmark not defined.) before <i>Create</i> is called (two-step construction).
wxDIALOG_EX_METAL	On Mac OS X, frames with this style will be shown with a metallic look. This is an <i>extra</i> style.

Under Unix or Linux, MWM (the Motif Window Manager) or other window managers recognizing the MHM hints should be running for any of these styles to have an effect.

See also *Generic window styles* (p. **Error! Bookmark not defined.**).

See also

wxDialog overview (p. **Error! Bookmark not defined.**), *wxFrame* (p. 582), *Validator* overview (p. **Error! Bookmark not defined.**)

wxDialog::wxDialog**wxDialog()**

Default constructor.

wxDialog(wxWindow* parent, wxWindowID id, const wxString& title, const wxPoint& pos = wxDefaultPosition, const wxSize& size = wxDefaultSize, long style = wxDEFAULT_DIALOG_STYLE, const wxString& name = "dialogBox")

Constructor.

Parameters*parent*

Can be NULL, a frame or another dialog box.

id

An identifier for the dialog. A value of -1 is taken to mean a default.

title

The title of the dialog.

pos

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

size

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

style

The window style. See *wxDialog* (p. 412).

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. 415)

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. 414) for details.

wxDialog::CreateButtonSizer**wxSizer* CreateButtonSizer(long flags)**

Creates a sizer with standard buttons. *flags* is a bit list of the following flags: wxOK, wxCANCEL, wxYES, wxNO, wxHELP, wxNO_DEFAULT.

The sizer lays out the buttons in a manner appropriate to the platform.

This function simply calls *CreateStdDialogButtonSizer* (p. 415).

wxDialog::CreateStdDialogButtonSizer**wxStdDialogButtonSizer* CreateStdDialogButtonSizer(long flags)**

Creates a *wxStdDialogButtonSizer* (p. **Error! Bookmark not defined.**) with standard buttons. *flags* is a bit list of the following flags: wxOK, wxCANCEL, wxYES, wxNO, wxHELP, wxNO_DEFAULT.

The sizer lays out the buttons in a manner appropriate to the platform.

wxDialog::DoOK**virtual bool DoOK()**

This function is called when the titlebar OK button is pressed (PocketPC only). A

command event for the identifier returned by `GetAffirmativeId` is sent by default. You can override this function. If the function returns false, `wxWidgets` will call `Close()` for the dialog.

`wxDialog::EndModal`

`void EndModal(int retCode)`

Ends a modal dialog, passing a value to be returned from the `wxDialog::ShowModal` (p. 421) invocation.

Parameters

retCode

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

See also

`wxDialog::ShowModal` (p. 421), `wxDialog::GetReturnCode` (p. 417),
`wxDialog::SetReturnCode` (p. 420)

`wxDialog::GetAffirmativeId`

`int GetAffirmativeId() const`

Gets the identifier to be used when the user presses an OK button in a PocketPC titlebar.

See also

`wxDialog::SetAffirmativeId` (p. 419)

`wxDialog::GetEscapId`

`int GetEscapId() const`

Gets the identifier of the button to map presses of `ESC` button to.

See also

`wxDialog::SetEscapId` (p. 419)

`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. 421) returns a code to the application.

See also

wxDialog::SetReturnCode (p. 420), *wxDialog::ShowModal* (p. 421), *wxDialog::EndModal* (p. 416)

wxDialog::GetToolBar**wxToolBar* GetToolBar() const**

On PocketPC, a dialog is automatically provided with an empty toolbar. *GetToolBar* allows you to access the toolbar and add tools to it. Removing tools and adding arbitrary controls are not currently supported.

This function is not available on any other platform.

wxDialog::Iconize**void Iconize(const bool iconize)**

Iconizes or restores the dialog. Windows only.

Parameters

iconize

If true, iconizes the dialog box; if false, shows and restores it.

Remarks

Note that in Windows, iconization has no effect since dialog boxes cannot be iconized. However, applications may need to explicitly restore dialog boxes under Motif which have user-iconizable frames, and under Windows calling *Iconize(false)* will bring the window to the front, as does *Show(true)*.

wxDialog::IsIconized**bool IsIconized() const**

Returns true if the dialog box is iconized. Windows only.

Remarks

Always returns false under Windows since dialogs cannot be iconized.

wxDialog::IsModal**bool IsModal() const**

Returns true if the dialog box is modal, false otherwise.

wxDialog::OnApply

void OnApply(wxCommandEvent& event)

The default handler for the wxID_APPLY identifier.

Remarks

This function calls *wxWindow::Validate* (p. **Error! Bookmark not defined.**) and *wxWindow::TransferDataFromWindow* (p. **Error! Bookmark not defined.**).

See also

wxDialog::OnOK (p. 418), *wxDialog::OnCancel* (p. 418)

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. 418), *wxDialog::OnApply* (p. 418)

wxDialog::OnOK

void OnOK(wxCommandEvent& event)

The default handler for the wxID_OK identifier.

Remarks

The function calls *wxWindow::Validate* (p. **Error! Bookmark not defined.**), then *wxWindow::TransferDataFromWindow* (p. **Error! Bookmark not defined.**). 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. 418), *wxDialog::OnApply* (p. 418)

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 `wxEvt::Skip` to propagate the notification to child windows and controls.

See also

wxSysColourChangedEvent (p. [Error! Bookmark not defined.](#))

wxDialog::SetAffirmativeId

void SetAffirmativeId(int id)

Sets the identifier to be used when the user presses an OK button in a PocketPC titlebar. By default, this is `wxID_OK`.

See also

wxDialog::GetAffirmativeId (p. 416)

wxDialog::SetEscapeld

void SetEscapeld(int id)

Sets the identifier to be used when the user presses `ESC` button in the dialog. By default, this is `wxID_ANY` meaning that the first suitable button is used: if there a `wxID_CANCEL` button, it is activated, otherwise `wxID_OK` button is activated if present. Another possible special value for *id* is `wxID_NONE` meaning that `ESC` presses should be ignored. If another value is given, it is interpreted as the id of the button to map the escape key to.

wxDialog::SetIcon

void SetIcon(const wxIcon& icon)

Sets the icon for this dialog.

Parameters

icon

The icon to associate with this dialog.

See also *wxIcon* (p. 778).

wxDialog::SetIcons

void SetIcons(const wxIconBundle& icons)

Sets the icons for this dialog.

Parameters

icons

The icons to associate with this dialog.

See also *wxIconBundle* (p. 785).

wxDialog::SetModal

void SetModal(const bool flag)

NB: This function is deprecated and doesn't work for all ports, just use *ShowModal* (p. 421) 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. 421) returns a code to the application. The function *wxDialog::EndModal* (p. 416) calls **SetReturnCode**.

See also

wxDialog::GetReturnCode (p. 417), *wxDialog::ShowModal* (p. 421), *wxDialog::EndModal* (p. 416)

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. 416).

wxDialog::ShowModal

int ShowModal()

Shows a modal dialog. Program flow does not return until the dialog has been dismissed with *wxDialog::EndModal* (p. 416).

Return value

The return value is the value set with *wxDialog::SetReturnCode* (p. 420).

See also

wxDialog::EndModal* (p. 416), *wxDialog::GetReturnCode* (p. 417), *wxDialog::SetReturnCode* (p. 420)*wxDialUpEvent

This is the event class for the dialup events sent by *wxDialUpManager* (p. 422).

Derived from

wxEvent (p. 487)

wxObject (p. **Error! Bookmark not defined.**)

Include files

<wx/dialup.h>

wxDialUpEvent::wxDialUpEvent

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

Constructor is only used by *wxDialUpManager* (p. 422).

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 `wxWidgets` classes in that there is at most one instance of this class in the program accessed via `wxDialUpManager::Create()` (p. 423) 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. 422) 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. **Error! Bookmark not defined.**)
wxDialUpEvent (p. 422)

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. 424) 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. 426) 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. 424)

wxDialUpManager::CancelDialing**bool CancelDialing()**

Cancel dialing the number initiated with *Dial* (p. 424) with async parameter equal to `true`.

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

See also

IsDialing (p. 424)

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. 426)) 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. 425)

wxDialUpManager::EnableAutoCheckOnlineStatus

bool EnableAutoCheckOnlineStatus(size_t nSeconds = 60)

Enable automatic checks for the connection status and sending of `wxEVT_DIALUP_CONNECTED/wxEVT_DIALUP_DISCONNECTED` events. The interval parameter is only for Unix where we do the check manually and specifies how often should we repeat the check (each minute by default). Under Windows, the notification about the change of connection status is sent by the system and so we don't do any polling and this parameter is ignored.

Returns `false` if couldn't set up automatic check for online status.

wxDialUpManager::DisableAutoCheckOnlineStatus

void DisableAutoCheckOnlineStatus()

Disable automatic check for connection status change - notice that the `wxEVT_DIALUP_XXX` events won't be sent any more neither.

wxDialUpManager::SetWellKnownHost

void SetWellKnownHost(const wxString& hostname, int portno = 80)

This method is for Unix only.

Under Unix, the value of well-known host is used to check whether we're connected to the internet. It is unused under Windows, but this function is always safe to call. The default value is `www.yahoo.com:80`.

wxDialUpManager::SetConnectCommand

**void SetConnectCommand(const wxString& commandDial = wxT("/usr/bin/pon"),
const wxString& commandHangup = wxT("/usr/bin/poff"))**

This method is for Unix only.

Sets the commands to start up the network and to hang up again.

See also

Dial (p. 424)

wxDir

`wxDir` is a portable equivalent of Unix `open/read/closedir` functions which allow enumerating of the files in a directory. `wxDir` allows enumerate files as well as

directories.

`wxDir` also provides a flexible way to enumerate files recursively using *Traverse* (p. 429) or a simpler *GetAllFiles* (p. 428) function.

Example of use:

```
wxDir dir(wxGetCwd());

if ( !dir.IsOpened() )
{
    // deal with the error here - wxDir would already log an
error message
    // explaining the exact reason of the failure
    return;
}

puts("Enumerating object files in current directory:");

wxString filename;

bool cont = dir.GetFirst(&filename, filespec, flags);
while ( cont )
{
    printf("%s\n", filename.c_str());

    cont = dir.GetNext(&filename);
}
```

Derived from

No base class

Constants

These flags define what kind of filename is included in the list of files enumerated by *GetFirst/GetNext*.

```
enum
{
    wxDIR_FILES      = 0x0001,          // include files
    wxDIR_DIRS       = 0x0002,          // include directories
    wxDIR_HIDDEN     = 0x0004,          // include hidden files
    wxDIR_DOTDOT     = 0x0008,          // include '.' and '..'

    // by default, enumerate everything except '.' and '..'
    wxDIR_DEFAULT    = wxDIR_FILES | wxDIR_DIRS | wxDIR_HIDDEN
}
```

Include files

<wx/dir.h>

wxDir::wxDir

wxDir()

Default constructor, use *Open()* (p. 429) afterwards.

wxDir(const wxString& dir)

Opens the directory for enumeration, use *IsOpened()* (p. 429) to test for errors.

wxDir::~~wxDir**~wxDir()**

Destructor cleans up the associated resources. It is not virtual and so this class is not meant to be used polymorphically.

wxDir::Exists**static bool Exists(const wxString& dir)**

Test for existence of a directory with the given name

wxDir::GetAllFiles**static size_t GetAllFiles(const wxString& dirname, wxArrayString *files, const wxString& filespec = wxEmptyString, int flags = wxDIR_DEFAULT)**

The function appends the names of all the files under directory *dirname* to the array *files* (note that its old content is preserved). Only files matching the *filespec* are taken, with empty spec matching all the files.

The *flags* parameter should always include `wxDIR_FILES` or the array would be unchanged and should include `wxDIR_DIRS` flag to recurse into subdirectories (both flags are included in the value by default).

See also: *Traverse* (p. 429)

wxDir::GetFirst**bool GetFirst(wxString* filename, const wxString& filespec = wxEmptyString, int flags = wxDIR_DEFAULT) const**

Start enumerating all files matching *filespec* (or all files if it is empty) and flags, return true on success.

wxDir::GetName**wxString GetName() const**

Returns the name of the directory itself. The returned string does not have the trailing path separator (slash or backslash).

wxDir::GetNext**bool GetNext(wxString* filename) const**

Continue enumerating files satisfying the criteria specified by the last call to *GetFirst* (p. 428).

wxDir::HasFiles**bool HasFiles(const wxString& filespec = wxEmptyString)**

Returns `true` if the directory contains any files matching the given *filespec*. If *filespec* is empty, look for any files at all. In any case, even hidden files are taken into account.

wxDir::HasSubDirs**bool HasSubDirs(const wxString& dirs spec = wxEmptyString)**

Returns `true` if the directory contains any subdirectories (if a non empty *filespec* is given, only check for directories matching it). The hidden subdirectories are taken into account as well.

wxDir::IsOpened**bool IsOpened() const**

Returns `true` if the directory was successfully opened by a previous call to *Open* (p. 429).

wxDir::Open**bool Open(const wxString& dir)**

Open the directory for enumerating, returns `true` on success or `false` if an error occurred.

wxDir::Traverse**size_t Traverse(wxDirTraverser& sink, const wxString& filespec = wxEmptyString, int flags = wxDIR_DEFAULT)**

Enumerate all files and directories under the given directory recursively calling the element of the provided *wxDirTraverser* (p. 432) object for each of them.

More precisely, the function will really recurse into subdirectories if *flags* contains `wxDIR_DIRS` flag. It will ignore the files (but still possibly recurse into subdirectories) if `wxDIR_FILES` flag is given.

For each found directory, *sink.OnDir()* (p. 433) is called and *sink.OnFile()* (p. 433) is called for every file. Depending on the return value, the enumeration may continue or stop.

The function returns the total number of files found or `(size_t)-1` on error.

See also: *GetAllFiles* (p. 428)

wxDirDialog

This class represents the directory chooser dialog.

Derived from

wxDialog (p. 412)

wxWindow (p. **Error! Bookmark not defined.**)

wxEvtHandler (p. 490)

wxObject (p. **Error! Bookmark not defined.**)

Include files

<wx/dirdlg.h> <wx/generic/dirdlgg.h>

Window styles

wxDD_DEFAULT_STYLE Equivalent to a combination of `wxDEFAULT_DIALOG_STYLE`, `wxDD_NEW_DIR_BUTTON` and `wxRESIZE_BORDER` (the last one is not used under `wxWinCE`).

wxDD_NEW_DIR_BUTTON Add "Create new directory" button and allow directory names to be editable. On Windows the new directory button is only available with recent versions of the common dialogs.

See also *Generic window styles* (p. **Error! Bookmark not defined.**).

See also

wxDirDialog overview (p. **Error! Bookmark not defined.**), *wxFileDialog* (p. 515)

wxDirDialog::wxDirDialog

wxDirDialog(*wxWindow** parent, **const wxString&** message = "Choose a directory", **const wxString&** defaultPath = "", **long** style = `wxDD_DEFAULT_STYLE`, **const wxPoint&** pos = `wxDefaultPosition`, **const wxSize&** size = `wxDefaultSize`, **const wxString&** name = "wxDirCtrl")

Constructor. Use *wxDirDialog::ShowModal* (p. 432) to show the dialog.

Parameters

parent

Parent window.

message

Message to show on the dialog.

defaultPath

The default path, or the empty string.

style

The dialog style. See *wxDirDialog* (p. 429)

pos

Dialog position. Ignored under Windows.

size

Dialog size. Ignored under Windows.

name

The dialog name, not used.

wxDirDialog::~wxDirDialog

~wxDirDialog()

Destructor.

wxDirDialog::GetPath

wxString GetPath() const

Returns the default or user-selected path.

wxDirDialog::GetMessage

wxString GetMessage() const

Returns the message that will be displayed on the dialog.

wxDirDialog::GetStyle

long GetStyle() const

Returns the dialog style.

wxDirDialog::SetMessage

void SetMessage(const wxString& message)

Sets the message that will be displayed on the dialog.

wxDirDialog::SetPath**void SetPath(const wxString& path)**

Sets the default path.

wxDirDialog::SetStyle**void SetStyle(long style)**

Sets the dialog style. This is currently unused.

wxDirDialog::ShowModal**int ShowModal()**

Shows the dialog, returning wxID_OK if the user pressed OK, and wxID_CANCEL otherwise.

wxDirTraverser

wxDirTraverser is an abstract interface which must be implemented by objects passed to *Traverse* (p. 429) function.

Example of use (this works almost like *GetAllFiles* (p. 428)):

```
class wxDirTraverserSimple : public wxDirTraverser
{
public:
    wxDirTraverserSimple(wxArrayString& files) :
m_files(files) { }

    virtual wxDirTraverseResult OnFile(const wxString&
filename)
    {
        m_files.Add(filename);
        return wxDIR_CONTINUE;
    }

    virtual wxDirTraverseResult OnDir(const wxString&
WXUNUSED(dirname))
    {
        return wxDIR_CONTINUE;
    }

private:
    wxArrayString& m_files;
};

// get the names of all files in the array
wxArrayString files;
wxDirTraverserSimple traverser(files);

wxDir dir(dirname);
dir.Traverse(traverser);
```

Derived from

No base class

Constants

The elements of `wxDirTraverseResult` are the possible return values of the callback functions:

```
enum wxDirTraverseResult
{
    wxDIR_IGNORE = -1,          // ignore this directory but continue
    with others
    wxDIR_STOP,                // stop traversing
    wxDIR_CONTINUE              // continue into this directory
};
```

Include files

<wx/dir.h>

wxDirTraverser::OnDir

virtual wxDirTraverseResult OnDir(const wxString& *dirname*)

This function is called for each directory. It may return `wxDIR_STOP` to abort traversing completely, `wxDIR_IGNORE` to skip this directory but continue with others or `wxDIR_CONTINUE` to enumerate all files and subdirectories in this directory.

This is a pure virtual function and must be implemented in the derived class.

wxDirTraverser::OnFile

virtual wxDirTraverseResult OnFile(const wxString& *filename*)

This function is called for each file. It may return `wxDIR_STOP` to abort traversing (for example, if the file being searched is found) or `wxDIR_CONTINUE` to proceed.

This is a pure virtual function and must be implemented in the derived class.

wxOpenErrorTraverser::OnOpenError

virtual wxOpenErrorTraverseResult OnOpenError(const wxString& *openerrorname*)

This function is called for each directory which we failed to open for enumerating. It may return `wxDIR_STOP` to abort traversing completely, `wxDIR_IGNORE` to skip this directory but continue with others or `wxDIR_CONTINUE` to retry opening this directory once again.

The base class version always returns `wxDIR_IGNORE`.

wxDisplay

Determines the sizes and locations of displays connected to the system.

Derived from

None

Include files

<wx/display.h>

See also

wxClientDisplayRect (p. **Error! Bookmark not defined.**), *wxDisplaySize* (p. **Error! Bookmark not defined.**), *wxDisplaySizeMM* (p. **Error! Bookmark not defined.**)

wxDisplay::wxDisplay

wxDisplay(size_t *index* = 0)

Constructor, setting up a wxDisplay instance with the specified display.

Parameters

index

The index of the display to use. This must be non-negative and lower than the value returned by *GetCount()* (p. 435).

wxDisplay::~~wxDisplay

void ~wxDisplay()

Destructor.

wxDisplay::ChangeMode

bool **ChangeMode**(const wxVideoMode& *mode* = wxDefaultVideoMode)

Changes the video mode of this display to the mode specified in the mode parameter.

If wxDefaultVideoMode is passed in as the mode parameter, the defined behaviour is that wxDisplay will reset the video mode to the default mode used by the display. On Windows, the behavior is normal. However, there are differences on other platforms. On Unix variations using X11 extensions it should behave as defined, but some irregularities may occur.

On wxMac passing in wxDefaultVideoMode as the mode parameter does nothing. This happens because carbon no longer has access to DMUseScreenPrefs, an undocumented function that changed the video mode to the system default by using the

system's 'scrn' resource.

wxDisplay::GetClientArea

wxRect GetClientArea() const

Returns the client area of the display. The client area is the part of the display available for the normal (non full screen) windows, usually it is the same as *GetGeometry* (p. 436) but it could be less if there is a taskbar (or equivalent) on this display.

See also:

wxCliëntDisplayRect (p. **Error! Bookmark not defined.**)

wxDisplay::GetCount

static size_t GetCount()

Returns the number of connected displays.

wxDisplay::GetCurrentMode

wxVideoMode GetCurrentMode() const

Returns the current video mode that this display is in.

wxDisplay::GetDepth

int GetDepth() const

Returns the bit depth of the display whose index was passed to the constructor.

wxDisplay::GetFromPoint

static int GetFromPoint(const wxPoint& pt)

Returns the index of the display on which the given point lies. Returns `wxNOT_FOUND` if the point is not on any connected display.

Parameters

pt

The point to locate.

wxDisplay::GetFromWindow

static int GetFromWindow(wxWindow* win)

Returns the index of the display on which the given window lies.

If the window is on more than one display it gets the display that overlaps the window the most.

Returns `wxNOT_FOUND` if the window is not on any connected display.

Parameters

win

The window to locate.

wxDisplay::GetGeometry

wxRect GetGeometry() const

Returns the bounding rectangle of the display whose index was passed to the constructor.

See also:

GetClientArea (p. 434), *wxDisplaySize* (p. **Error! Bookmark not defined.**)

wxDisplay::GetModes

wxArrayVideoModes GetModes(const wxVideoMode& mode = wxDefaultVideoMode) const

Fills and returns an array with all the video modes that are supported by this display, or video modes that are supported by this display and match the mode parameter (if mode is not `wxDefaultVideoMode`).

wxDisplay::GetName

wxString GetName() const

Returns the display's name. A name is not available on all platforms.

wxDisplay::IsPrimary

bool IsPrimary()

Returns true if the display is the primary display. The primary display is the one whose index is 0.

wxDllLoader

Deprecation note: This class is deprecated since version 2.4 and is not compiled in by default in version 2.6 and will be removed in 2.8. Please use *wxDynamicLibrary* (p. 478) instead.

wxDllLoader is a class providing an interface similar to Unix's `dlopen()`. It is used by

the `wxLibrary` framework and manages the actual loading of shared libraries and the resolving of symbols in them. There are no instances of this class, it simply serves as a namespace for its static member functions.

Please note that class `wxDynamicLibrary` (p. 478) provides alternative, friendlier interface to `wxDllLoader`.

The terms *DLL* and *shared library/object* will both be used in the documentation to refer to the same thing: a `.dll` file under Windows or `.so` or `.sl` one under Unix.

Example of using this class to dynamically load the `strlen()` function:

```
#if defined(__WXMSW__)
    static const wxChar *LIB_NAME = _T("kernel32");
    static const wxChar *FUNC_NAME = _T("lstrlenA");
#elif defined(__UNIX__)
    static const wxChar *LIB_NAME = _T("/lib/libc-2.0.7.so");
    static const wxChar *FUNC_NAME = _T("strlen");
#endif

    wxDllType dllHandle = wxDllLoader::LoadLibrary(LIB_NAME);
    if ( !dllHandle )
    {
        ... error ...
    }
    else
    {
        typedef int (*strlenType)(char *);
        strlenType pfnStrlen =
        (strlenType)wxDllLoader::GetSymbol(dllHandle, FUNC_NAME);
        if ( !pfnStrlen )
        {
            ... error ...
        }
        else
        {
            if ( pfnStrlen("foo") != 3 )
            {
                ... error ...
            }
            else
            {
                ... ok! ...
            }
        }

        wxDllLoader::UnloadLibrary(dllHandle);
    }
}
```

Derived from

No base class

Include files

<wx/dynlib.h>

Data structures

This header defines a platform-dependent `wxDllType` typedef which stores a handle to

a loaded DLLs on the given platform.

wxDllLoader::GetDllExt

static wxString GetDllExt()

Returns the string containing the usual extension for shared libraries for the given systems (including the leading dot if not empty).

For example, this function will return ".dll" under Windows or (usually) ".so" under Unix.

wxDllLoader::GetProgramHandle

wxDllType GetProgramHandle()

This function returns a valid handle for the main program itself. Notice that the `NULL` return value is valid for some systems (i.e. doesn't mean that the function failed).

NB: This function is Unix specific. It will always fail under Windows or OS/2.

wxDllLoader::GetSymbol

void * GetSymbol(wxDllType dllHandle, const wxString& name)

This function resolves a symbol in a loaded DLL, such as a variable or function name.

Returned value will be `NULL` if the symbol was not found in the DLL or if an error occurred.

Parameters

dllHandle

Valid handle previously returned by *LoadLibrary* (p. 438)

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. 439).

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. 438) previously.

wxDocChildFrame

The `wxDocChildFrame` class provides a default frame for displaying documents on separate windows. This class can only be used for SDI (not MDI) child frames.

The class is part of the document/view framework supported by `wxWidgets`, and cooperates with the `wxView` (p. **Error! Bookmark not defined.**), `wxDocument` (p. 459), `wxDocManager` (p. 441) and `wxDocTemplate` (p. 454) classes.

See the example application in `samples/docview`.

Derived from

`wxFrame` (p. 582)

`wxWindow` (p. **Error! Bookmark not defined.**)

`wxEvtHandler` (p. 490)

`wxObject` (p. **Error! Bookmark not defined.**)

Include files

`<wx/docview.h>`

See also

Document/view overview (p. **Error! Bookmark not defined.**), `wxFrame` (p. 582)

wxDocChildFrame::m_childDocument

wxDocument* m_childDocument

The document associated with the frame.

wxDocChildFrame::m_childView

wxView* m_childView

The view associated with the frame.

wxDocChildFrame::wxDocChildFrame

wxDocChildFrame(wxDocument* doc, wxView* view, wxFrame* parent, wxWindowID id, const wxString& title, const wxPoint& pos = wxDefaultPosition, const wxSize& size = wxDefaultSize, long style = wxDEFAULT_FRAME_STYLE, const wxString& name = "frame")

Constructor.

wxDocChildFrame::~~wxDocChildFrame

~wxDocChildFrame()

Destructor.

wxDocChildFrame::GetDocument

wxDocument* GetDocument() const

Returns the document associated with this frame.

wxDocChildFrame::GetView

wxView* GetView() const

Returns the view associated with this frame.

wxDocChildFrame::OnActivate

void OnActivate(wxActivateEvent event)

Sets the currently active view to be the frame's view. You may need to override (but still call) this function in order to set the keyboard focus for your subwindow.

wxDocChildFrame::OnCloseWindow

void OnCloseWindow(wxCloseEvent& event)

Closes and deletes the current view and document.

wxDocChildFrame::SetDocument

void SetDocument(wxDocument *doc)

Sets the document for this frame.

wxDocChildFrame::SetView**void SetView**(wxView *view)

Sets the view for this frame.

wxDocManager

The wxDocManager class is part of the document/view framework supported by wxWidgets, and cooperates with the wxView (p. **Error! Bookmark not defined.**), wxDocument (p. 459) and wxDocTemplate (p. 454) classes.

Derived from

wxEvtHandler (p. 490)

wxObject (p. **Error! Bookmark not defined.**)

Include files

<wx/docview.h>

See also

wxDocManager overview (p. **Error! Bookmark not defined.**), wxDocument (p. 459), wxView (p. **Error! Bookmark not defined.**), wxDocTemplate (p. 454), wxFileHistory (p. 520)

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. 520), which manages the history of recently-visited files on the File menu.

wxDocManager::m_maxDocsOpen**int m_maxDocsOpen**

Stores the maximum number of documents that can be opened before existing documents are closed. By default, this is 10,000.

wxDocManager::m_docs**wxList m_docs**

A list of all documents.

wxDocManager::m_flags**long m_flags**

Stores the flags passed to the constructor.

wxDocManager::m_lastDirectory

The directory last selected by the user when opening a file.

wxFileHistory* m_fileHistory**wxDocManager::m_templates****wxList m_templates**

A list of all document templates.

wxDocManager::wxDocManager

wxDocManager(long *flags* = *wxDEFAULT_DOCMAN_FLAGS*, bool *initialize* = *true*)

Constructor. Create a document manager instance dynamically near the start of your application before doing any document or view operations.

flags is currently unused.

If *initialize* is true, the *Initialize* (p. 446) function will be called to create a default history list object. If you derive from *wxDocManager*, you may wish to call the base constructor with false, and then call *Initialize* in your own constructor, to allow your own *Initialize* or *OnCreateFileHistory* functions to be called.

wxDocManager::~wxDocManager**void ~wxDocManager()**

Destructor.

wxDocManager::ActivateView

void ActivateView(wxView* *doc*, bool *activate* = *true*)

Sets the current view.

wxDocManager::AddDocument

void AddDocument(wxDocument *doc)

Adds the document to the list of documents.

wxDocManager::AddFileToHistory

void AddFileToHistory(const wxString& filename)

Adds a file to the file history list, if we have a pointer to an appropriate file menu.

wxDocManager::AssociateTemplate

void AssociateTemplate(wxDocTemplate *temp)

Adds the template to the document manager's template list.

wxDocManager::CloseDocuments

bool CloseDocuments(bool force = true)

Closes all currently opened documents.

wxDocManager::CreateDocument

wxDocument* CreateDocument(const wxString& path, long flags)

Creates a new document in a manner determined by the *flags* parameter, which can be:

- `wxDOC_NEW` Creates a fresh document.
- `wxDOC_SILENT` Silently loads the given document file.

If `wxDOC_NEW` is present, a new document will be created and returned, possibly after asking the user for a template to use if there is more than one document template. If `wxDOC_SILENT` is present, a new document will be created and the given file loaded into it. If neither of these flags is present, the user will be presented with a file selector for the file to load, and the template to use will be determined by the extension (Windows) or by popping up a template choice list (other platforms).

If the maximum number of documents has been reached, this function will delete the oldest currently loaded document before creating a new one.

wxDocManager::CreateView

wxView* CreateView(wxDocument* doc, long flags)

Creates a new view for the given document. If more than one view is allowed for the

document (by virtue of multiple templates mentioning the same document type), a choice of view is presented to the user.

wxDocManager::DisassociateTemplate

void DisassociateTemplate(wxDocTemplate *temp)

Removes the template from the list of templates.

wxDocManager::FileHistoryAddFilesToMenu

void FileHistoryAddFilesToMenu()

Appends the files in the history list, to all menus managed by the file history object.

void FileHistoryAddFilesToMenu(wxMenu* menu)

Appends the files in the history list, to the given menu only.

wxDocManager::FileHistoryLoad

void FileHistoryLoad(wxConfigBase& config)

Loads the file history from a config object.

See also

wxConfig (p. 196)

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. 196)

wxDocManager::FileHistoryUseMenu

void FileHistoryUseMenu(wxMenu* menu)

Use this menu for appending recently-visited document filenames, for convenient

access. Calling this function with a valid menu pointer enables the history list functionality.

Note that you can add multiple menus using this function, to be managed by the file history object.

wxDocManager::FindTemplateForPath

wxDocTemplate * FindTemplateForPath(const wxString& path)

Given a path, try to find template that matches the extension. This is only an approximate method of finding a template for creating a document.

wxDocManager::GetCurrentDocument

wxDocument * GetCurrentDocument()

Returns the document associated with the currently active view (if any).

wxDocManager::GetCurrentView

wxView * GetCurrentView()

Returns the currently active view

wxDocManager::GetDocuments

wxList& GetDocuments()

Returns a reference to the list of documents.

wxDocManager::GetFileHistory

wxFileHistory * GetFileHistory()

Returns a pointer to file history.

wxDocManager::GetLastDirectory

wxString GetLastDirectory() const

Returns the directory last selected by the user when opening a file. Initially empty.

wxDocManager::GetMaxDocsOpen

int GetMaxDocsOpen()

Returns the number of documents that can be open simultaneously.

wxDocManager::GetHistoryFilesCount

size_t GetHistoryFilesCount()

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

wxDocManager::GetTemplates**wxList& GetTemplates()**

Returns a reference to the list of associated templates.

wxDocManager::Initialize**bool Initialize()**

Initializes data; currently just calls `OnCreateFileHistory`. Some data cannot always be initialized in the constructor because the programmer must be given the opportunity to override functionality. If `OnCreateFileHistory` was called from the constructor, an overridden virtual `OnCreateFileHistory` would not be called due to C++'s 'interesting' constructor semantics. In fact `Initialize` is called from the `wxDocManager` constructor, but this can be vetoed by passing false to the second argument, allowing the derived class's constructor to call `Initialize`, possibly calling a different `OnCreateFileHistory` from the default.

The bottom line: if you're not deriving from `Initialize`, forget it and construct `wxDocManager` with no arguments.

wxDocManager::MakeDefaultName**bool MakeDefaultName(const wxString& buf)**

Copies a suitable default name into *buf*. This is implemented by appending an integer counter to the string **unnamed** and incrementing the counter.

wxPerl note: In `wxPerl` this function must return the modified name rather than just modifying the argument.

wxDocManager::OnCreateFileHistory**wxFileHistory * OnCreateFileHistory()**

A hook to allow a derived class to create a different type of file history. Called from *Initialize* (p. 446).

wxDocManager::OnFileClose**void OnFileClose(wxCommandEvent& event)**

Closes and deletes the currently active document.

wxDocManager::OnFileCloseAll

void OnFileCloseAll(wxCommandEvent& event)

Closes and deletes all the currently opened documents.

wxDocManager::OnFileNew

void OnFileNew(wxCommandEvent& event)

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

wxDocManager::OnFileOpen

void OnFileOpen(wxCommandEvent& event)

Creates a new document and reads in the selected file.

wxDocManager::OnFileRevert

void OnFileRevert(wxCommandEvent& event)

Reverts the current document by calling wxDocument::Revert for the current document.

wxDocManager::OnFileSave

void OnFileSave(wxCommandEvent& event)

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

wxDocManager::OnFileSaveAs

void OnFileSaveAs(wxCommandEvent& event)

Calls wxDocument::SaveAs for the current document.

wxDocManager::RemoveDocument

void RemoveDocument(wxDocument *doc)

Removes the document from the list of documents.

wxDocManager::SelectDocumentPath

**wxDocTemplate * SelectDocumentPath(wxDocTemplate **templates, int
noTemplates, wxString& path, long flags, bool save)**

Under Windows, pops up a file selector with a list of filters corresponding to document templates. The wxDocTemplate corresponding to the selected file's extension is returned.

On other platforms, if there is more than one document template a choice list is popped

up, followed by a file selector.

This function is used in `wxDocManager::CreateDocument`.

wxPerl note: In wxPerl `templates` is a reference to a list of templates. If you override this method in your document manager it must return two values, eg:

```
(doctemplate, path) = My::DocManager->SelectDocumentPath( ... );
```

wxDocManager::SelectDocumentType

wxDocTemplate * SelectDocumentType(wxDocTemplate **templates, int noTemplates, bool sort=false)

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

Parameters

templates

Pointer to an array of templates from which to choose a desired template.

noTemplates

Number of templates being pointed to by the *templates* pointer.

sort

If more than one template is passed in in *templates*, then this parameter indicates whether the list of templates that the user will have to choose from is sorted or not when shown the choice box dialog. Default is false.

wxPerl note: In wxPerl `templates` is a reference to a list of templates.

wxDocManager::SelectViewType

wxDocTemplate * SelectViewType(wxDocTemplate **templates, int noTemplates, bool sort=false)

Returns a document template by asking the user (if there is more than one template), displaying a list of valid views. This function is used in `wxDocManager::CreateView`. The dialog normally will not appear because the array of templates only contains those relevant to the document in question, and often there will only be one such.

Parameters

templates

Pointer to an array of templates from which to choose a desired template.

noTemplates

Number of templates being pointed to by the *templates* pointer.

sort

If more than one template is passed in in *templates*, then this parameter indicates whether the list of templates that the user will have to choose from is sorted or not when shown the choice box dialog. Default is false.

wxPerl note: In wxPerl *templates* is a reference to a list of templates.

wxDocManager::SetLastDirectory

void SetLastDirectory(const wxString& dir)

Sets the directory to be displayed to the user when opening a file. Initially this is empty.

wxDocManager::SetMaxDocsOpen

void SetMaxDocsOpen(int n)

Sets the maximum number of documents that can be open at a time. By default, this is 10,000. If you set it to 1, existing documents will be saved and deleted when the user tries to open or create a new one (similar to the behaviour of Windows Write, for example). Allowing multiple documents gives behaviour more akin to MS Word and other Multiple Document Interface applications.

wxDocMDIChildFrame

The `wxDocMDIChildFrame` class provides a default frame for displaying documents on separate windows. This class can only be used for MDI child frames.

The class is part of the document/view framework supported by `wxWidgets`, and cooperates with the `wxView` (p. **Error! Bookmark not defined.**), `wxDocument` (p. 459), `wxDocManager` (p. 441) and `wxDocTemplate` (p. 454) classes.

See the example application in `samples/docview`.

Derived from

`wxMDIChildFrame` (p. **Error! Bookmark not defined.**)

`wxFrame` (p. 582)

`wxWindow` (p. **Error! Bookmark not defined.**)

`wxEvtHandler` (p. 490)

`wxObject` (p. **Error! Bookmark not defined.**)

Include files

<wx/docmdi.h>

See also

Document/view overview (p. **Error! Bookmark not defined.**), `wxMDIChildFrame` (p. **Error! Bookmark not defined.**)

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. **Error! Bookmark not defined.**), wxDocument (p. 459), wxDocManager (p. 441) and wxDocTemplates (p. 454) classes.

See the example application in `samples/docview`.

Derived from

wxMDIParentFrame (p. **Error! Bookmark not defined.**)

wxFrame (p. 582)

wxWindow (p. **Error! Bookmark not defined.**)

wxEvtHandler (p. 490)

wxObject (p. **Error! Bookmark not defined.**)

Include files

<wx/docmdi.h>

See also

Document/view overview (p. **Error! Bookmark not defined.**), wxMDIParentFrame (p. **Error! Bookmark not defined.**)

wxDocMDIParentFrame::wxDocMDIParentFrame

wxDocMDIParentFrame()

wxDocMDIParentFrame(wxDocManager* manager, wxFrame *parent, wxWindowID id, const wxString& title, const wxPoint& pos = wxDefaultPosition, const wxSize& size = wxDefaultSize, long style = wxDEFAULT_FRAME_STYLE, const wxString& name = "frame")

Constructor.

wxDocMDIParentFrame::~wxDocMDIParentFrame

~wxDocMDIParentFrame()

Destructor.

wxDocMDIParentFrame::Create

bool Create(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")

Creates the window.

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. **Error! Bookmark not defined.**), wxDocument (p. 459), wxDocManager (p. 441) and wxDocTemplates (p. 454) classes.

See the example application in `samples/docview`.

Derived from

wxFrame (p. 582)

wxWindow (p. **Error! Bookmark not defined.**)

wxEvtHandler (p. 490)

wxObject (p. **Error! Bookmark not defined.**)

Include files

<wx/docview.h>

See also

Document/view overview (p. **Error! Bookmark not defined.**), *wxFrame* (p. 582)

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. **Error! Bookmark not defined.**)

Include files

<wx/docview.h>

See also

wxDocTemplate overview (p. **Error! Bookmark not defined.**), *wxDocument* (p. 459), *wxView* (p. **Error! Bookmark not defined.**)

wxDocTemplate::m_defaultExt

wxString m_defaultExt

The default extension for files of this type.

wxDocTemplate::m_description

wxString m_description

A short description of this template.

wxDocTemplate::m_directory

wxString m_directory

The default directory for files of this type.

wxDocTemplate::m_docClassInfo

wxClassInfo* m_docClassInfo

Run-time class information that allows document instances to be constructed dynamically.

wxDocTemplate::m_docTypeName

wxString m_docTypeName

The named type of the document associated with this template.

wxDocTemplate::m_documentManager

wxDocTemplate* m_documentManager

A pointer to the document manager for which this template was created.

wxDocTemplate::m_fileFilter**wxString m_fileFilter**

The file filter (such as *.txt) to be used in file selector dialogs.

wxDocTemplate::m_flags**long m_flags**

The flags passed to the constructor.

wxDocTemplate::m_viewClassInfo**wxClassInfo* m_viewClassInfo**

Run-time class information that allows view instances to be constructed dynamically.

wxDocTemplate::m_viewTypeName**wxString m_viewTypeName**

The named type of the view associated with this template.

wxDocTemplate::wxDocTemplate

wxDocTemplate(wxDocManager* manager, const wxString& descr, const wxString& filter, const wxString& dir, const wxString& ext, const wxString& docTypeName, const wxString& viewTypeName, wxClassInfo* docClassInfo = NULL, wxClassInfo* viewClassInfo = NULL, long flags = wxDEFAULT_TEMPLATE_FLAGS)

Constructor. Create instances dynamically near the start of your application after creating a wxDocManager instance, and before doing any document or view operations.

manager is the document manager object which manages this template.

descr is a short description of what the template is for. This string will be displayed in the file filter list of Windows file selectors.

filter is an appropriate file filter such as *.txt.

dir is the default directory to use for file selectors.

ext is the default file extension (such as txt).

docTypeName is a name that should be unique for a given type of document, used for gathering a list of views relevant to a particular document.

viewTypeName is a name that should be unique for a given view.

docClassInfo is a pointer to the run-time document class information as returned by the CLASSINFO macro, e.g. CLASSINFO(MyDocumentClass). If this is not supplied, you

will need to derive a new `wxDocTemplate` class and override the `CreateDocument` member to return a new document instance on demand.

`viewClassInfo` is a pointer to the run-time view class information as returned by the `CLASSINFO` macro, e.g. `CLASSINFO(MyViewClass)`. If this is not supplied, you will need to derive a new `wxDocTemplate` class and override the `CreateView` member to return a new view instance on demand.

`flags` is a bit list of the following:

- `wxTEMPLATE_VISIBLE` The template may be displayed to the user in dialogs.
- `wxTEMPLATE_INVISIBLE` The template may not be displayed to the user in dialogs.
- `wxDEFAULT_TEMPLATE_FLAGS` Defined as `wxTEMPLATE_VISIBLE`.

wxPerl note: In `wxPerl` `docClassInfo` and `viewClassInfo` can be either `Wx::ClassInfo` objects or strings which contain the name of the perl packages which are to be used as `Wx::Document` and `Wx::View` classes (they must have a constructor named `new`):

```
Wx::DocTemplate->new( docmgr, descr, filter, dir, ext, docTypeName,  
                    viewTypeName, docClassInfo,  
                    viewClassInfo, flags ) will construct  
document and view objects from the class  
information
```

```
Wx::DocTemplate->new( docmgr, descr, filter, dir, ext, docTypeName,  
                    viewTypeName, docClassName,  
                    viewClassName, flags ) will construct  
document and view objects from perl packages
```

```
Wx::DocTemplate->new( docmgr, descr, filter, dir, ext, docTypeName,  
                    viewTypeName )  
    Wx::DocTemplate::CreateDocument(  
    ) and Wx::DocTemplate::CreateView(  
    ) must be overridden
```

wxDocTemplate::~~wxDocTemplate

```
void ~wxDocTemplate()
```

Destructor.

wxDocTemplate::CreateDocument

```
wxDocument * CreateDocument(const wxString& path, long flags = 0)
```

Creates a new instance of the associated document class. If you have not supplied a `wxClassInfo` parameter to the template constructor, you will need to override this

function to return an appropriate document instance.

This function calls `wxDocTemplate::InitDocument` which in turns calls `wxDocument::OnCreate`.

wxDocTemplate::CreateView

wxView * CreateView(wxDocument *doc, long flags = 0)

Creates a new instance of the associated view class. If you have not supplied a `wxClassInfo` parameter to the template constructor, you will need to override this function to return an appropriate view instance.

wxDocTemplate::GetDefaultExtension

wxString GetDefaultExtension()

Returns the default file extension for the document data, as passed to the document template constructor.

wxDocTemplate::GetDescription

wxString GetDescription()

Returns the text description of this template, as passed to the document template constructor.

wxDocTemplate::GetDirectory

wxString GetDirectory()

Returns the default directory, as passed to the document template constructor.

wxDocTemplate::GetDocumentManager

wxDocManager * GetDocumentManager()

Returns a pointer to the document manager instance for which this template was created.

wxDocTemplate::GetDocumentName

wxString GetDocumentName()

Returns the document type name, as passed to the document template constructor.

wxDocTemplate::GetFileFilter

wxString GetFileFilter()

Returns the file filter, as passed to the document template constructor.

wxDocTemplate::GetFlags

long GetFlags()

Returns the flags, as passed to the document template constructor.

wxDocTemplate::GetViewName

wxString GetViewName()

Returns the view type name, as passed to the document template constructor.

wxDocTemplate::InitDocument

bool InitDocument(wxDocument* doc, const wxString& path, long flags = 0)

Initialises the document, calling wxDocument::OnCreate. This is called from wxDocTemplate::CreateDocument.

wxDocTemplate::IsVisible

bool IsVisible()

Returns true if the document template can be shown in user dialogs, false otherwise.

wxDocTemplate::SetDefaultExtension

void SetDefaultExtension(const wxString& ext)

Sets the default file extension.

wxDocTemplate::SetDescription

void SetDescription(const wxString& descr)

Sets the template description.

wxDocTemplate::SetDirectory

void SetDirectory(const wxString& dir)

Sets the default directory.

wxDocTemplate::SetDocumentManager

void SetDocumentManager(wxDocManager *manager)

Sets the pointer to the document manager instance for which this template was created.

Should not be called by the application.

wxDocTemplate::SetFileFilter

void SetFileFilter(const wxString& filter)

Sets the file filter.

wxDocTemplate::SetFlags

void SetFlags(long flags)

Sets the internal document template flags (see the constructor description for more details).

wxDocument

The document class can be used to model an application's file-based data. It is part of the document/view framework supported by wxWidgets, and cooperates with the *wxView* (p. **Error! Bookmark not defined.**), *wxDocTemplate* (p. 454) and *wxDocManager* (p. 441) classes.

Derived from

wxEvtHandler (p. 490)

wxObject (p. **Error! Bookmark not defined.**)

Include files

<wx/docview.h>

See also

wxDocument overview (p. **Error! Bookmark not defined.**), *wxView* (p. **Error! Bookmark not defined.**), *wxDocTemplate* (p. 454), *wxDocManager* (p. 441)

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. 189).

`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. 460).

`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. 462)

wxDocument::GetPrintableName

virtual void GetPrintableName(wxString& name) const

Copies a suitable document name into the supplied *name* buffer. The default function uses the title, or if there is no title, uses the filename; or if no filename, the string **unnamed**.

wxPerl note: In wxPerl this function must return the modified name rather than just modifying the argument.

wxDocument::GetTitle

wxString GetTitle() const

Gets the title for this document. The document title is used for an associated frame (if any), and is usually constructed by the framework from the filename.

wxDocument::GetViews

wxList & GetViews() const

Returns the list whose elements are the views on the document.

See also: *GetFirstView* (p. 462)

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. 463).

wxDocument::LoadObject

virtual istream& LoadObject(istream& stream)

virtual wxInputStream& LoadObject(wxInputStream& stream)

Override this function and call it from your own `LoadObject` before streaming your own data. `LoadObject` is called by the framework automatically when the document contents need to be loaded.

Note that only one of these forms exists, depending on how `wxWidgets` was configured.

wxDocument::Modify

virtual void Modify(bool modify)

Call with `true` to mark the document as modified since the last save, `false` otherwise. You may need to override this if your document view maintains its own record of being modified (for example if using `wxTextWindow` to view and edit the document).

See also *IsModified* (p. 462).

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. 189).

wxDocument::OnNewDocument

virtual bool OnNewDocument()

The default implementation calls `OnSaveModified` and `DeleteContents`, makes a default title for the document, and notifies the views that the filename (in fact, the title) has changed.

wxDocument::OnOpenDocument

virtual bool OnOpenDocument(const wxString& filename)

Constructs an input file stream for the given filename (which must not be empty), and calls `LoadObject`. If `LoadObject` returns true, the document is set to unmodified; otherwise, an error message box is displayed. The document's views are notified that the filename has changed, to give windows an opportunity to update their titles. All of the document's views are then updated.

wxDocument::OnSaveDocument

virtual bool OnSaveDocument(const wxString& filename)

Constructs an output file stream for the given filename (which must not be empty), and calls `SaveObject`. If `SaveObject` returns true, the document is set to unmodified; otherwise, an error message box is displayed.

wxDocument::OnSaveModified

virtual bool OnSaveModified()

If the document has been modified, prompts the user to ask if the changes should be changed. If the user replies Yes, the `Save` function is called. If No, the document is marked as unmodified and the function succeeds. If Cancel, the function fails.

wxDocument::RemoveView

virtual bool RemoveView(wxView* view)

Removes the view from the document's list of views, and calls `OnChangedViewList`.

wxDocument::Save

virtual bool Save()

Saves the document by calling `OnSaveDocument` if there is an associated filename, or `SaveAs` if there is no filename.

`wxDocument::SaveAs`

`virtual bool SaveAs()`

Prompts the user for a file to save to, and then calls `OnSaveDocument`.

`wxDocument::SaveObject`

`virtual ostream& SaveObject(ostream& stream)`

`virtual wxOutputStream& SaveObject(wxOutputStream& stream)`

Override this function and call it from your own `SaveObject` before streaming your own data. `SaveObject` is called by the framework automatically when the document contents need to be saved.

Note that only one of these forms exists, depending on how `wxWidgets` was configured.

`wxDocument::SetCommandProcessor`

`virtual void SetCommandProcessor(wxCommandProcessor *processor)`

Sets the command processor to be used for this document. The document will then be responsible for its deletion. Normally you should not call this; override `OnCreateCommandProcessor` instead.

See *wxCommandProcessor* (p. 189).

`wxDocument::SetDocumentName`

`void SetDocumentName(const wxString& name)`

Sets the document type name for this document. See the comment for *documentTypeName* (p. 460).

`wxDocument::SetDocumentTemplate`

`void SetDocumentTemplate(wxDocTemplate* templ)`

Sets the pointer to the template that created the document. Should only be called by the framework.

`wxDocument::SetFilename`

`void SetFilename(const wxString& filename, bool notifyViews = false)`

Sets the filename for this document. Usually called by the framework.

If *notifyViews* is true, `wxView::OnChangeFilename` is called for all views.

wxDocument::SetTitle

void SetTitle(const wxString& title)

Sets the title for this document. The document title is used for an associated frame (if any), and is usually constructed by the framework from the filename.

wxDocument::UpdateAllViews

void UpdateAllViews(wxView* sender = NULL, wxObject* hint = NULL)

Updates all views. If *sender* is non-NULL, does not update this view.

hint represents optional information to allow a view to optimize its update.

wxDragImage

This class is used when you wish to drag an object on the screen, and a simple cursor is not enough.

On Windows, the WIN32 API is used to do achieve smooth dragging. On other platforms, `wxGenericDragImage` is used. Applications may also prefer to use `wxGenericDragImage` on Windows, too.

wxPython note: wxPython uses `wxGenericDragImage` on all platforms, but uses the `wxDragImage` name.

To use this class, when you wish to start dragging an image, create a `wxDragImage` object and store it somewhere you can access it as the drag progresses. Call `BeginDrag` to start, and `EndDrag` to stop the drag. To move the image, initially call `Show` and then `Move`. If you wish to update the screen contents during the drag (for example, highlight an item as in the `dragimag` sample), first call `Hide`, update the screen, call `Move`, and then call `Show`.

You can drag within one window, or you can use full-screen dragging either across the whole screen, or just restricted to one area of the screen to save resources. If you want the user to drag between two windows, then you will need to use full-screen dragging.

If you wish to draw the image yourself, use `wxGenericDragImage` and override `wxDragImage::DoDrawImage` (p. 469) and `wxDragImage::GetImageRect` (p. 469).

Please see `samples/dragimag` for an example.

Derived from

`wxObject` (p. **Error! Bookmark not defined.**)

Include files

<wx/dragimag.h>

<wx/generic/dragimgg.h>

wxDragImage::wxDragImage

wxDragImage()

Default constructor.

wxDragImage(const wxBitmap& image, const wxCursor& cursor = wxNullCursor, const wxPoint& cursorHotspot = wxPoint(0, 0))

Constructs a drag image from a bitmap and optional cursor.

wxDragImage(const wxIcon& image, const wxCursor& cursor = wxNullCursor, const wxPoint& cursorHotspot = wxPoint(0, 0))

Constructs a drag image from an icon and optional cursor.

wxPython note: This constructor is called `wxDragIcon` in wxPython.

wxDragImage(const wxString& text, const wxCursor& cursor = wxNullCursor, const wxPoint& cursorHotspot = wxPoint(0, 0))

Constructs a drag image from a text string and optional cursor.

wxPython note: This constructor is called `wxDragString` in wxPython.

wxDragImage(const wxTreeCtrl& treeCtrl, wxTreeItemId& id)

Constructs a drag image from the text in the given tree control item, and optional cursor.

wxPython note: This constructor is called `wxDragTreeItem` in wxPython.

wxDragImage(const wxListCtrl& treeCtrl, long id)

Constructs a drag image from the text in the given tree control item, and optional cursor.

wxPython note: This constructor is called `wxDragListItem` in wxPython.

wxDragImage(const wxCursor& cursor = wxNullCursor, const wxPoint& cursorHotspot = wxPoint(0, 0))

Constructs a drag image an optional cursor. This constructor is only available for `wxGenericDragImage`, and can be used when the application supplies `wxDragImage::DoDrawImage` (p. 469) and `wxDragImage::GetImageRect` (p. 469).

Parameters

image

Icon or bitmap to be used as the drag image. The bitmap can have a mask.

text

Text used to construct a drag image.

cursor

Optional cursor to combine with the image.

hotspot

This parameter is deprecated.

treeCtrl

Tree control for constructing a tree drag image.

listCtrl

List control for constructing a list drag image.

id

Tree or list control item id.

wxDragImage::BeginDrag

bool BeginDrag(const wxPoint& hotspot, wxWindow* window, bool fullScreen = false, wxRect* rect = NULL)

Start dragging the image, in a window or full screen.

bool BeginDrag(const wxPoint& hotspot, wxWindow* window, wxWindow* boundingWindow)

Start dragging the image, using the first window to capture the mouse and the second to specify the bounding area. This form is equivalent to using the first form, but more convenient than working out the bounding rectangle explicitly.

You need to then call *wxDragImage::Show* (p. 470) and *wxDragImage::Move* (p. 470) to show the image on the screen.

Call *wxDragImage::EndDrag* (p. 469) 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. 469).

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. 469)) 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. 470) and *wxDragImage::Show* (p. 470).

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. 470) has been called previously (for example at the start of the drag).

pt is the position in client coordinates (relative to the window specified in *BeginDrag*).

You can move the image either when the image is hidden or shown, but in general dragging will be smoother if you move the image when it is shown.

wxDragImage::Show

bool Show()

Shows the image. Call this at least once when dragging.

wxDragImage::UpdateBackingFromWindow

bool UpdateBackingFromWindow(wxDC& windowDC, wxMemoryDC& destDC, const wxRect& sourceRect, const wxRect& destRect) const

Override this if you wish to draw the window contents to the backing bitmap yourself. This can be desirable if you wish to avoid flicker by not having to redraw the updated window itself just before dragging, which can cause a flicker just as the drag starts. Instead, paint the drag image's backing bitmap to show the appropriate graphic *minus the objects to be dragged*, and leave the window itself to be updated by the drag image. This can provide eerily smooth, flicker-free drag behaviour.

The default implementation copies the window contents to the backing bitmap. A new implementation will normally copy information from another source, such as from its own backing bitmap if it has one, or directly from internal data structures.

This function is available in *wxGenericDragImage* only.

wxDropFilesEvent

This class is used for drop files events, that is, when files have been dropped onto the window. This functionality is currently only available under Windows. The window must have previously been enabled for dropping by calling *wxWindow::DragAcceptFiles* (p. **Error! Bookmark not defined.**).

Important note: this is a separate implementation to the more general drag and drop implementation documented *here* (p. **Error! Bookmark not defined.**). It uses the older, Windows message-based approach of dropping files.

Derived from

wxEvent (p. 487)

wxObject (p. **Error! Bookmark not defined.**)

Include files

<wx/event.h>

Event table macros

To process a drop files event, use these event handler macros to direct input to a member function that takes a `wxDropFilesEvent` argument.

EVT_DROP_FILES(func) Process a `wxEVT_DROP_FILES` event.

See also

Event handling overview (p. **Error! Bookmark not defined.**)

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. **Error! Bookmark not defined.**) and *wxDataObject overview* (p. **Error! Bookmark not defined.**) for more information.

Derived from

None

Include files

<wx/dnd.h>

Types

wxDragResult is defined as follows:

```
enum wxDragResult
{
    wxDragError,      // error prevented the d&d operation from
completing
    wxDragNone,       // drag target didn't accept the data
    wxDragCopy,       // the data was successfully copied
    wxDragMove,       // the data was successfully moved (MSW only)
    wxDragLink,       // operation is a drag-link
    wxDragCancel      // the operation was cancelled by user (not an
error)
};
```

See also

wxDropTarget (p. 475), *wxTextDropTarget* (p. **Error! Bookmark not defined.**),
wxFileDropTarget (p. 519)

wxDropSource::wxDropSource

wxDropSource(wxWindow* win = NULL, const wxIconOrCursor& iconCopy = wxNullIconOrCursor, const wxIconOrCursor& iconMove = wxNullIconOrCursor, const wxIconOrCursor& iconNone = wxNullIconOrCursor)

```
wxDropSource(wxDataObject& data, wxWindow* win = NULL, const  
wxIconOrCursor& iconCopy = wxNullIconOrCursor, const wxIconOrCursor&  
iconMove = wxNullIconOrCursor, const wxIconOrCursor& iconNone =  
wxNullIconOrCursor)
```

The constructors for wxDataObject.

If you use the constructor without *data* parameter you must call *SetData* (p. 474) 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. **Error! Bookmark not defined.**) 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. 242) associated with the drop source. This will not delete any previously associated data.

wxDropSource::DoDragDrop

```
virtual wxDragResult DoDragDrop(int flags = wxDrag_CopyOnly)
```

Do it (call this in response to a mouse button press, for example). This starts the drag-and-drop operation which will terminate when the user releases the mouse.

Parameters

flags

If `wxDrag_AllowMove` is included in the flags, data may be moved and not only copied (default). If `wxDrag_DefaultMove` is specified (which includes the previous flag), this is even the default operation

.

Return value

Returns the operation requested by the user, may be `wxDragCopy`, `wxDragMove`, `wxDragLink`, `wxDragCancel` or `wxDragNone` if an error occurred.

wxDropSource::GetDataObject

wxDataObject * GetDataObject()

Returns the `wxDataObject` object that has been assigned previously.

wxDropSource::GiveFeedback

virtual bool GiveFeedback(wxDragResult effect)

Overridable: you may give some custom UI feedback during the drag and drop operation in this function. It is called on each mouse move, so your implementation must not be too slow.

Parameters

effect

The effect to implement. One of `wxDragCopy`, `wxDragMove`, `wxDragLink` and `wxDragNone`.

scrolling

true if the window is scrolling. MSW only.

Return value

Return false if you want default feedback, or true if you implement your own feedback. The return values is ignored under GTK.

wxDropSource::SetCursor

void SetCursor(wxDragResult res, const wxCursor& cursor)

Set the icon to use for a certain drag result.

Parameters

res

The drag result to set the icon for.

cursor

The icon to show when this drag result occurs.

wxDropTarget

This class represents a target for a drag and drop operation. A *wxDataObject* (p. 242) 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. 477), possibly many times *OnDragOver* (p. 477), *OnDrop* (p. 476) and finally *OnData* (p. 476).

See *Drag and drop overview* (p. **Error! Bookmark not defined.**) and *wxDataObject overview* (p. **Error! Bookmark not defined.**) for more information.

Derived from

None

Include files

<wx/dnd.h>

Types

wxDragResult is defined as follows:

```
enum wxDragResult
{
    wxDragError,          // error prevented the d&d operation from
completing
    wxDragNone,           // drag target didn't accept the data
    wxDragCopy,           // the data was successfully copied
    wxDragMove,           // the data was successfully moved (MSW only)
    wxDragLink,           // operation is a drag-link
    wxDragCancel          // the operation was cancelled by user (not an
error)
};
```

See also

wxDropSource (p. 472), *wxTextDropTarget* (p. **Error! Bookmark not defined.**),
wxFileDropTarget (p. 519), *wxDataFormat* (p. 237), *wxDataObject* (p. 242)

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. 476). By default, this method copies the data from the drop source to the *wxDataObject* (p. 242) associated with this drop target, calling its *wxDataObject::SetData* (p. 246) method.

wxDropTarget::OnData

virtual wxDragResult OnData(wxCoord x, wxCoord y, wxDragResult def)

Called after *OnDrop* (p. 476) returns true. By default this will usually *GetData* (p. 476) 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. 477).

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. 242) associated with the drop target and deletes any previously associated data object.

wxDynamicLibrary

wxDynamicLibrary is a class representing dynamically loadable library (Windows DLL, shared library under Unix etc.). Just create an object of this class to load a library and don't worry about unloading it -- it will be done in the objects destructor automatically.

Derived from

No base class.

Include files

<wx/dynlib.h>

(only available if `wxUSE_DYNLIB_CLASS` is set to 1)

wxDynamicLibrary::wxDynamicLibrary

wxDynamicLibrary()

wxDynamicLibrary(const wxString& name, int flags = wxDL_DEFAULT)

Constructor. Second form calls *Load* (p. 481).

wxDynamicLibrary::CanonicalizeName

**static wxString CanonicalizeName(const wxString& name,
wxDynamicLibraryCategory cat = wxDL_LIBRARY)**

Returns the platform-specific full name for the library called *name*. E.g. it adds a ".dll" extension under Windows and "lib" prefix and ".so", ".sl" or maybe ".dylib" extension under Unix.

The possible values for *cat* are:

<code>wxDL_LIBRARY</code>	normal library
<code>wxDL_MODULE</code>	a loadable module or plugin

See also

CanonicalizePluginName (p. 479)

wxDynamicLibrary::CanonicalizePluginName

**static wxString CanonicalizePluginName(const wxString& name,
wxPluginCategory cat = wxDL_PLUGIN_GUI)**

This function does the same thing as *CanonicalizeName* (p. 479) but for wxWidgets plugins. The only difference is that compiler and version information are added to the name to ensure that the plugin which is going to be loaded will be compatible with the main program.

The possible values for *cat* are:

<code>wxDL_PLUGIN_GUI</code>	plugin which uses GUI classes (default)
<code>wxDL_PLUGIN_BASE</code>	plugin which only uses wxBase

wxDynamicLibrary::Detach

wxDIIType Detach()

Detaches this object from its library handle, i.e. the object will not unload the library any longer in its destructor but it is now the callers responsibility to do this using *Unload* (p. 481).

wxDynamicLibrary::GetSymbol

void * GetSymbol(const wxString& name) const

Returns pointer to symbol *name* in the library or NULL if the library contains no such symbol.

See also

`wxDYNLIB_FUNCTION` (p. **Error! Bookmark not defined.**)

wxDynamicLibrary::GetSymbolAorW

void * GetSymbolAorW(const wxString& name) const

This function is available only under Windows as it is only useful when dynamically loading symbols from standard Windows DLLs. Such functions have either 'A' (in ANSI build) or 'W' (in Unicode, or wide character build) suffix if they take string parameters. Using this function you can use just the base name of the function and the correct suffix is appended automatically depending on the current build. Otherwise, this method is identical to *GetSymbol* (p. 479).

wxDynamicLibrary::GetProgramHandle

static wxDIIType GetProgramHandle()

Return a valid handle for the main program itself or NULL if symbols from the main program can't be loaded on this platform.

wxDynamicLibrary::HasSymbol

bool HasSymbol(const wxString& name) const

Returns `true` if the symbol with the given *name* is present in the dynamic library, `false` otherwise. Unlike *GetSymbol* (p. 479), this function doesn't log an error message if the symbol is not found.

This function is new since wxWidgets version 2.5.4

wxDynamicLibrary::IsLoaded

bool IsLoaded() const

Returns `true` if the library was successfully loaded, `false` otherwise.

wxDynamicLibrary::ListLoaded

static wxDynamicLibraryDetailsArray ListLoaded()

This static method returns an *array* (p. 57) containing the details of all modules loaded into the address space of the current project, the array elements are object of `wxDynamicLibraryDetails` class. The array will be empty if an error occurred.

This method is currently implemented only under Win32 and Linux and is useful mostly for diagnostics purposes.

wxDynamicLibrary::Load

bool Load(const wxString& name, int flags = wxDL_DEFAULT)

Loads DLL with the given *name* into memory. The *flags* argument can be a combination of the following bits:

<code>wxDL_LAZY</code>	equivalent of <code>RTLD_LAZY</code> under Unix, ignored elsewhere
<code>wxDL_NOW</code>	equivalent of <code>RTLD_NOW</code> under Unix, ignored elsewhere
<code>wxDL_GLOBAL</code>	equivalent of <code>RTLD_GLOBAL</code> under Unix, ignored elsewhere
<code>wxDL_VERBATIM</code>	don't try to append the appropriate extension to the library name (this is done by default).
<code>wxDL_DEFAULT</code>	default flags, same as <code>wxDL_NOW</code> currently

Returns `true` if the library was successfully loaded, `false` otherwise.

wxDynamicLibrary::Unload

void Unload()

static void Unload(wxDllType handle)

Unloads the library from memory. `wxDynamicLibrary` object automatically calls this method from its destructor if it had been successfully loaded.

The second version is only used if you need to keep the library in memory during a longer period of time than the scope of the `wxDynamicLibrary` object. In this case you may call *Detach* (p. 479) and store the handle somewhere and call this static method later to unload it.

wxDynamicLibraryDetails

This class is used for the objects returned by `wxDynamicLibrary::ListLoaded` (p. 480) method and contains the information about a single module loaded into the address space of the current process. A module in this context may be either a dynamic library or the main program itself.

Derived from

No base class.

Include files

<wx/dynlib.h>

(only available if `wxUSE_DYNLIB_CLASS` is set to 1)

wxDynamicLibraryDetails::GetName

wxString GetName() const

Returns the base name of this module, e.g. `kernel32.dll` or `libc-2.3.2.so`.

wxDynamicLibraryDetails::GetPath

wxString GetPath() const

Returns the full path of this module if available, e.g.
`c:\windows\system32\kernel32.dll` or `/lib/libc-2.3.2.so`.

wxDynamicLibraryDetails::GetAddress

bool GetAddress(void **addr, size_t *len) const

Retrieves the load address and the size of this module.

Parameters

addr

the pointer to the location to return load address in, may be `NULL`

len

pointer to the location to return the size of this module in memory in, may be `NULL`

Return value

`true` if the load address and module size were retrieved, `false` if this information is not available.

wxDynamicLibraryDetails::GetVersion

wxString GetVersion() const

Returns the version of this module, e.g. `5.2.3790.0` or `2.3.2`. The returned string is empty if the version information is not available.

wxEncodingConverter

This class is capable of converting strings between two 8-bit encodings/charsets. It can also convert from/to Unicode (but only if you compiled `wxWidgets` with `wxUSE_WCHAR_T` set to 1). Only a limited subset of encodings is supported by `wxEncodingConverter`: `wxFONTENCODING_ISO8859_1`..15, `wxFONTENCODING_CP1250`..1257 and `wxFONTENCODING_KOI8`.

Note

Please use `wxMBConv` classes (p. **Error! Bookmark not defined.**) instead if possible. `wxCSCnv` (p. 229) has much better support for various encodings than `wxEncodingConverter`. `wxEncodingConverter` is useful only if you rely on `wxCONVERT_SUBSTITUTE` mode of operation (see *Init* (p. 483)).

Derived from

`wxObject` (p. **Error! Bookmark not defined.**)

Include files

`<wx/encconv.h>`

See also

`wxFontMapper` (p. 578), `wxMBConv` (p. 923), *Writing non-English applications* (p. **Error! Bookmark not defined.**)

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. 484) will interpret its argument as a string in *input_enc* encoding and will output string in *output_enc* encoding. You must call this method before calling *Convert*. You may call it more than once in order to switch to another conversion. *Method* affects behaviour of *Convert()* in case input character cannot be converted because it does not exist in output encoding:

wxCONVERT_STRICT

follow behaviour of GNU Recode - just copy unconvertible characters to output and don't change them (its integer value will stay the same)

wxCONVERT_SUBSTITUTE

try some (lossy) substitutions - e.g. replace unconvertible latin capitals with acute by ordinary capitals, replace en-dash or em-dash by '-' etc.

Both modes guarantee that output string will have same length as input string.

Return value

false if given conversion is impossible, true otherwise (conversion may be impossible either if you try to convert to Unicode with non-Unicode build of wxWidgets or if input or output encoding is not supported.)

wxEncodingConverter::CanConvert

static bool CanConvert(wxFontEncoding encIn, wxFontEncoding encOut)

Return true if (any text in) multibyte encoding *encIn* can be converted to another one (*encOut*) losslessly.

Do not call this method with wxFONTENCODING_UNICODE as either parameter, it doesn't make sense (always works in one sense and always depends on the text to convert in the other).

wxEncodingConverter::Convert

bool Convert(const char* input, char* output) const

bool Convert(const wchar_t* input, wchar_t* output) const

bool Convert(const char* input, wchar_t* output) const

bool Convert(const wchar_t* input, char* output) const

Convert input string according to settings passed to *Init* (p. 483) and writes the result to

output.

bool Convert(char* str) const

bool Convert(wchar_t* str) const

Convert input string according to settings passed to *Init* (p. 483) in-place, i.e. write the result to the same memory area.

All of the versions above return `true` if the conversion was lossless and `false` if at least one of the characters couldn't be converted and was replaced with '?' in the output. Note that if `wxCONVERT_SUBSTITUTE` was passed to *Init* (p. 483), substitution is considered lossless operation.

wxString Convert(const wxString& input) const

Convert wxString and return new wxString object.

Notes

You must call *Init* (p. 483) before using this method!

wchar_t versions of the method are not available if wxWidgets was compiled with `wxUSE_WCHAR_T` set to 0.

wxEncodingConverter::GetPlatformEquivalents

static wxFontEncodingArray GetPlatformEquivalents(wxFontEncoding enc, int platform = wxPLATFORM_CURRENT)

Return equivalents for given font that are used under given platform. Supported platforms:

- wxPLATFORM_UNIX
- wxPLATFORM_WINDOWS
- wxPLATFORM_OS2
- wxPLATFORM_MAC
- wxPLATFORM_CURRENT

wxPLATFORM_CURRENT means the platform this binary was compiled for.

Examples:

current platform	enc	returned value
unix	CP1250	{ ISO8859_2 }
unix	ISO8859_2	{ ISO8859_2 }
windows	ISO8859_2	{ CP1250 }
unix	CP1252	{ ISO8859_1, ISO8859_15 }

Equivalence is defined in terms of convertibility: two encodings are equivalent if you can

convert text between then without losing information (it may - and will - happen that you lose special chars like quotation marks or em-dashes but you shouldn't lose any diacritics and language-specific characters when converting between equivalent encodings).

Remember that this function does **NOT** check for presence of fonts in system. It only tells you what are most suitable encodings. (It usually returns only one encoding.)

Notes

- Note that argument *enc* itself may be present in the returned array, so that you can, as a side-effect, detect whether the encoding is native for this platform or not.
- *Convert* (p. 484) 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. 485), but this one will return ALL equivalent encodings, regardless of the platform, and including itself.

This platform's encodings are before others in the array. And again, if *enc* is in the array, it is the very first item in it.

wxEraseEvent

An erase event is sent when a window's background needs to be repainted.

On some platforms, such as GTK+, this event is simulated (simply generated just before the paint event) and may cause flicker. It is therefore recommended that you set the text background colour explicitly in order to prevent flicker. The default background colour under GTK+ is grey.

To intercept this event, use the EVT_ERASE_BACKGROUND macro in an event table definition.

You must call wxEraseEvent::GetDC and use the returned device context if it is non-NULL. If it is NULL, create your own temporary wxClientDC object.

Derived from

wxEvent (p. 487)

wxObject (p. **Error! Bookmark not defined.**)

Include files

<wx/event.h>

Event table macros

To process an erase event, use this event handler macro to direct input to a member function that takes a `wxEraseEvent` argument.

EVT_ERASE_BACKGROUND(func) Process a `wxEVT_ERASE_BACKGROUND` event.

Remarks

Use the device context returned by *GetDC* (p. 487) to draw on, don't create a `wxPaintDC` in the event handler.

See also

Event handling overview (p. **Error! Bookmark not defined.**)

`wxEraseEvent::wxEraseEvent`

`wxEraseEvent(int id = 0, wxDC* dc = NULL)`

Constructor.

`wxEraseEvent::GetDC`

`wxDC* GetDC() const`

Returns the device context associated with the erase event to draw on.

`wxEvent`

An event is a structure holding information about an event passed to a callback or member function. **wxEvent** used to be a multipurpose event object, and is an abstract base class for other event classes (see below).

For more information about events, see the *Event handling overview* (p. **Error! Bookmark not defined.**).

wxPerl note: In wxPerl custom event classes should be derived from `Wx::PlEvent` and `Wx::PlCommandEvent`.

Derived from

wxObject (p. **Error! Bookmark not defined.**)

Include files

<wx/event.h>

See also

wxCommandEvent (p. 184), *wxMouseEvent* (p. **Error! Bookmark not defined.**)

wxEvent::wxEvent

wxEvent(int *id* = 0, **wxEventType** *eventType* = *wxEVT_NULL*)

Constructor. Should not need to be used directly by an application.

wxEvent::m_propagationLevel

int m_propagationLevel

Indicates how many levels the event can propagate. This member is protected and should typically only be set in the constructors of the derived classes. It may be temporarily changed by *StopPropagation* (p. 490) and *ResumePropagation* (p. 489) and tested with *ShouldPropagate* (p. 490).

The initial value is set to either *wxEVENT_PROPAGATE_NONE* (by default) meaning that the event shouldn't be propagated at all or to *wxEVENT_PROPAGATE_MAX* (for command events) meaning that it should be propagated as much as necessary.

Any positive number means that the event should be propagated but no more than the given number of times. E.g. the propagation level may be set to 1 to propagate the event to its parent only, but not to its grandparent.

wxEvent::Clone

virtual wxEvent* Clone() const

Returns a copy of the event.

Any event that is posted to the wxWidgets event system for later action (via *wxEvtHandler::AddPendingEvent* (p. 491) or *wxPostEvent* (p. **Error! Bookmark not defined.**)) must implement this method. All wxWidgets events fully implement this method, but any derived events implemented by the user should also implement this method just in case they (or some event derived from them) are ever posted.

All wxWidgets events implement a copy constructor, so the easiest way of implementing the Clone function is to implement a copy constructor for a new event (call it *MyEvent*) and then define the Clone function like this:

```
wxEvent *Clone(void) const { return new MyEvent(*this); }
```

wxEvent::GetEventObject

wxObject* GetEventObject()

Returns the object (usually a window) associated with the event, if any.

wxEvt::GetEventType**WXTYPE GetEventType()**

Returns the identifier of the given event type, such as `wxEVT_TYPE_BUTTON_COMMAND`.

wxEvt::GetId**int GetId() const**

Returns the identifier associated with this event, such as a button command id.

wxEvt::GetSkipped**bool GetSkipped() const**

Returns true if the event handler should be skipped, false otherwise.

wxEvt::GetTimestamp**long GetTimestamp()**

Gets the timestamp for the event.

wxEvt::IsCommandEvent**bool IsCommandEvent() const**

Returns true if the event is or is derived from *wxCommandEvent* (p. 184) else it returns false. Note: Exists only for optimization purposes.

wxEvt::ResumePropagation**void ResumePropagation(int *propagationLevel*)**

Sets the propagation level to the given value (for example returned from an earlier call to *StopPropagation* (p. 490)).

wxEvt::SetEventObject**void SetEventObject(wxObject* *object*)**

Sets the originating object.

wxEvt::SetEventType**void SetEventType(WXTYPE *typ*)**

Sets the event type.

wxEvtHandler::SetId**void SetId(int id)**

Sets the identifier associated with this event, such as a button command id.

wxEvtHandler::SetTimestamp**void SetTimestamp(long timeStamp)**

Sets the timestamp for the event.

wxEvtHandler::ShouldPropagate**bool ShouldPropagate() const**

Test if this event should be propagated or not, i.e. if the propagation level is currently greater than 0.

wxEvtHandler::Skip**void Skip(bool skip = true)**

Called by an event handler, it controls whether additional event handlers bound to this event will be called after the current event handler returns. `Skip(false)` (the default behavior) will prevent additional event handlers from being called and control will be returned to the sender of the event immediately after the current handler has finished. `Skip(true)` will cause the event processing system to continue searching for a handler function for this event.

wxEvtHandler::StopPropagation**int StopPropagation()**

Stop the event from propagating to its parent window.

Returns the old propagation level value which may be later passed to *ResumePropagation* (p. 489) to allow propagating the event again.

wxEvtHandler

A class that can handle events from the windowing system. `wxWindow` (and therefore all window classes) are derived from this class.

When events are received, `wxEvtHandler` invokes the method listed in the event table using itself as the object. When using multiple inheritance it is imperative that the `wxEvtHandler`(-derived) class be the first class inherited such that the "this" pointer for the overall object will be identical to the "this" pointer for the `wxEvtHandler` portion.

Derived from

wxObject (p. **Error! Bookmark not defined.**)

Include files

<wx/event.h>

See also

Event handling overview (p. **Error! Bookmark not defined.**)

wxEvtHandler::wxEvtHandler

wxEvtHandler()

Constructor.

wxEvtHandler::~~wxEvtHandler

~wxEvtHandler()

Destructor. If the handler is part of a chain, the destructor will unlink itself and restore the previous and next handlers so that they point to each other.

wxEvtHandler::AddPendingEvent

void AddPendingEvent(wxEvtHandler::wxEvent& event)

This function posts an event to be processed later.

Parameters

event

Event to add to process queue.

Remarks

The difference between sending an event (using the *ProcessEvent* (p. 495) method) and posting it is that in the first case the event is processed before the function returns, while in the second case, the function returns immediately and the event will be processed sometime later (usually during the next event loop iteration).

A copy of *event* is made by the function, so the original can be deleted as soon as function returns (it is common that the original is created on the stack). This requires that the *wxEvtHandler::Clone* (p. 488) method be implemented by *event* so that it can be duplicated and stored until it gets processed.

This is also the method to call for inter-thread communication---it will post events safely between different threads which means that this method is thread-safe by using critical sections where needed. In a multi-threaded program, you often need to inform the main GUI thread about the status of other working threads and such notification should be

done using this method.

This method automatically wakes up idle handling if the underlying window system is currently idle and thus would not send any idle events. (Waking up idle handling is done calling `::wxWakeUpIdle` (p. **Error! Bookmark not defined.**.)

wxEvtHandler::Connect

void Connect(int *id*, int *lastId*, **wxEventType** *eventType*, **wxObjectEventFunction** *function*, **wxObject*** *userData* = NULL, **wxEvtHandler*** *eventSink* = NULL)

void Connect(int *id*, **wxEventType** *eventType*, **wxObjectEventFunction** *function*, **wxObject*** *userData* = NULL, **wxEvtHandler*** *eventSink* = NULL)

void Connect(**wxEventType** *eventType*, **wxObjectEventFunction** *function*, **wxObject*** *userData* = NULL, **wxEvtHandler*** *eventSink* = NULL)

Connects the given function dynamically with the event handler, *id* and event type. This is an alternative to the use of static event tables. See the 'event' or the old 'dynamic' sample for usage.

Parameters

id

The identifier (or first of the identifier range) to be associated with the event handler function. For the version not taking this argument, it defaults to `wxID_ANY`.

lastId

The second part of the identifier range to be associated with the event handler function.

eventType

The event type to be associated with this event handler.

function

The event handler function. Note that this function should be explicitly converted to the correct type which can be done using a macro called `wxFooHandler` for the handler for any `wxFooEvent`.

userData

Data to be associated with the event table entry.

eventSink

Object whose member function should be called. If this is NULL, *this* will be used.

Example

```
frame->Connect( wxID_EXIT,
                wxEVT_COMMAND_MENU_SELECTED,
```

```
wxCommandEventHandler(MyFrame::OnQuit) );
```

wxPerl note: In wxPerl this function takes 4 arguments: `id`, `lastid`, `type`, `method`; if `method` is `undef`, the handler is disconnected.

wxEvtHandler::Disconnect

bool Disconnect(wxEventType eventType = wxEVT_NULL, wxObjectEventFunction function = NULL, wxObject* userData = NULL, wxEvtHandler* eventSink = NULL)

bool Disconnect(int id = wxID_ANY, wxEventType eventType = wxEVT_NULL, wxObjectEventFunction function = NULL, wxObject* userData = NULL, wxEvtHandler* eventSink = NULL)

bool Disconnect(int id, int lastId = wxID_ANY, wxEventType eventType = wxEVT_NULL, wxObjectEventFunction function = NULL, wxObject* userData = NULL, wxEvtHandler* eventSink = NULL)

Disconnects the given function dynamically from the event handler, using the specified parameters as search criteria and returning true if a matching function has been found and removed. This method can only disconnect functions which have been added using the *wxEvtHandler::Connect* (p. 492) method. There is no way to disconnect functions connected using the (static) event tables.

Parameters

id

The identifier (or first of the identifier range) associated with the event handler function.

lastId

The second part of the identifier range associated with the event handler function.

eventType

The event type associated with this event handler.

function

The event handler function.

userData

Data associated with the event table entry.

eventSink

Object whose member function should be called.

wxPerl note: In wxPerl this function takes 3 arguments: `id`, `lastid`, `type`.

wxEvtHandler::GetClientData

void* GetClientData()

Gets user-supplied client data.

Remarks

Normally, any extra data the programmer wishes to associate with the object should be made available by deriving a new class with new data members.

See also

wxEvtHandler::SetClientData (p. 497)

wxEvtHandler::GetClientObject**wxClientData* GetClientObject() const**

Get a pointer to the user-supplied client data object.

See also

wxEvtHandler::SetClientObject (p. 497), *wxClientData* (p. 152)

wxEvtHandler::GetEventHandlerEnabled**bool GetEventHandlerEnabled()**

Returns true if the event handler is enabled, false otherwise.

See also

wxEvtHandler::SetEventHandlerEnabled (p. 497)

wxEvtHandler::GetNextHandler**wxEvtHandler* GetNextHandler()**

Gets the pointer to the next handler in the chain.

See also

wxEvtHandler::SetNextHandler (p. 498), *wxEvtHandler::GetPreviousHandler* (p. 494), *wxEvtHandler::SetPreviousHandler* (p. 498), *wxWindow::PushEventHandler* (p. **Error! Bookmark not defined.**), *wxWindow::PopEventHandler* (p. **Error! Bookmark not defined.**)

wxEvtHandler::GetPreviousHandler**wxEvtHandler* GetPreviousHandler()**

Gets the pointer to the previous handler in the chain.

See also

wxEvtHandler::SetPreviousHandler (p. 498), *wxEvtHandler::GetNextHandler* (p. 494), *wxEvtHandler::SetNextHandler* (p. 498), *wxWindow::PushEventHandler* (p. **Error! Bookmark not defined.**), *wxWindow::PopEventHandler* (p. **Error! Bookmark not defined.**)

wxEvtHandler::ProcessEvent

virtual bool ProcessEvent(wxEvtHandler& 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 *wxEvtHandler::Skip* (p. 490).

Remarks

Normally, your application would not call this function: it is called in the wxWidgets implementation to dispatch incoming user interface events to the framework (and application).

However, you might need to call it if implementing new functionality (such as a new control) where you define new event types, as opposed to allowing the user to override virtual functions.

An instance where you might actually override the **ProcessEvent** function is where you want to direct event processing to event handlers not normally noticed by wxWidgets. For example, in the document/view architecture, documents and views are potential event handlers. When an event reaches a frame, **ProcessEvent** will need to be called on the associated document and view in case event handler functions are associated with these objects. The property classes library (wxProperty) also overrides **ProcessEvent** for similar reasons.

The normal order of event table searching is as follows:

1. If the object is disabled (via a call to *wxEvtHandler::SetEvtHandlerEnabled* (p. 497)) the function skips to step (6).
2. If the object is a wxWindow, **ProcessEvent** is recursively called on the window's *wxValidator* (p. **Error! Bookmark not defined.**). 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. 496)

wxEvtHandler::SearchEventTable

virtual bool SearchEventTable(wxEventTable& table, wxEvent& event)

Searches the event table, executing an event handler function if an appropriate one is found.

Parameters

table

Event table to be searched.

event

Event to be matched against an event table entry.

Return value

true if a suitable event handler function was found and executed, and the function did not call *wxEvent::Skip* (p. 490).

Remarks

This function looks through the object's event table and tries to find an entry that will match the event.

An entry will match if:

1. The event type matches, and
2. the identifier or identifier range matches, or the event table entry's identifier is zero.

If a suitable function is called but calls *wxEvent::Skip* (p. 490), this function will fail, and searching will continue.

See also

wxEvtHandler::ProcessEvent (p. 495)

wxEvtHandler::SetClientData**void SetClientData(void* data)**

Sets user-supplied client data.

Parameters*data*

Data to be associated with the event handler.

Remarks

Normally, any extra data the programmer wishes to associate with the object should be made available by deriving a new class with new data members. You must not call this method and *SetClientObject* (p. 497) on the same class - only one of them.

See also

wxEvtHandler::GetClientData (p. 493)

wxEvtHandler::SetClientObject**void SetClientObject(wxClientData* data)**

Set the client data object. Any previous object will be deleted.

See also

wxEvtHandler::GetClientObject (p. 494), *wxClientData* (p. 152)

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. 494)

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. 494), *wxEvtHandler::SetPreviousHandler* (p. 498), *wxEvtHandler::GetPreviousHandler* (p. 494), *wxWindow::PushEventHandler* (p. **Error! Bookmark not defined.**), *wxWindow::PopEventHandler* (p. **Error! Bookmark not defined.**)

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. 494), *wxEvtHandler::SetNextHandler* (p. 498), *wxEvtHandler::GetNextHandler* (p. 494), *wxWindow::PushEventHandler* (p. **Error! Bookmark not defined.**), *wxWindow::PopEventHandler* (p. **Error! Bookmark not defined.**)

wxFFile

wxFFile implements buffered file I/O. This is a very small class designed to minimize the overhead of using it - in fact, there is hardly any overhead at all, but using it brings you automatic error checking and hides differences between platforms and compilers. It wraps inside it a `FILE *` handle used by standard C IO library (also known as `stdio`).

Derived from

None.

Include files

<wx/ffile.h>

wxFromStart

Count offset from the start of the file

wxFromCurrent Count offset from the current position of the file pointer

wxFromEnd Count offset from the end of the file (backwards)

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. 501) 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. Note that you should use "b" flag if you use binary files under Windows or the results might be unexpected due to automatic newline conversion done for the text files.

fp

An existing file descriptor, such as `stderr`.

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. 501) will return `false` after call to `Detach()`.

wxFile::fp

FILE * fp() const

Returns the file pointer associated with the file.

wxFile::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. 506) is different as `wxFile::Eof` (p. 509) will return `true` here as soon as the last byte of the file has been read.

Also note that this method may only be called for opened files and may crash if the file is not opened.

See also

`IsOpened` (p. 501)

wxFile::Error

Returns `true` if an error has occurred on this file, similar to the standard `error()` function.

Please note that this method may only be called for opened files and may crash if the file is not opened.

See also

`IsOpened` (p. 501)

wxFile::Flush

bool Flush()

Flushes the file and returns `true` on success.

wxFile::GetKind**wxFileKind GetKind() const**

Returns the type of the file. Possible return values are:

```
enum wxFileKind
{
    wxFILE_KIND_UNKNOWN,
    wxFILE_KIND_DISK,      // a file supporting seeking to arbitrary
offsets
    wxFILE_KIND_TERMINAL, // a tty
    wxFILE_KIND_PIPE      // a pipe
};
```

wxFile::IsOpened**bool IsOpened() const**

Returns `true` if the file is opened. Most of the methods of this class may only be used for an opened file.

wxFile::Length**wxFileOffset Length() const**

Returns the length of the file.

wxFile::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, size_t count)**

Reads the specified number of bytes into a buffer, returning the actual number read.

Parameters

buffer

A buffer to receive the data.

count

The number of bytes to read.

Return value

The number of bytes read.

wxFile::ReadAll

bool ReadAll(wxString * str, wxMBConv& conv = wxConvUTF8)

Reads the entire contents of the file into a string.

Parameters

str

String to read data into.

conv

Conversion object to use in Unicode build; by default supposes that file contents is encoded in UTF-8.

Return value

`true` if file was read successfully, `false` otherwise.

wxFile::Seek

bool Seek(wxFileOffset ofs, wxSeekMode mode = wxFromStart)

Seeks to the specified position and returns `true` on success.

Parameters

ofs

Offset to seek to.

mode

One of `wxFromStart`, `wxFromEnd`, `wxFromCurrent`.

wxFile::SeekEnd

bool SeekEnd(wxFileOffset ofs = 0)

Moves the file pointer to the specified number of bytes before the end of the file and returns `true` on success.

Parameters

ofs

Number of bytes before the end of the file.

wxFile::Tell

wxFileOffset 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, wxMBConv& conv = wxConvUTF8)

Writes the contents of the string to the file, returns `true` on success.

The second argument is only meaningful in Unicode build of wxWidgets when *conv* is used to convert *s* to multibyte representation.

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. 499) whereas *wxFileInputStream* (p. 523) is based in the *wxFile* (p. 506) class.

Note that `SeekI()` (p. 828) can seek beyond the end of the stream (file) and will thus not return `wxInvalidOffset` for that.

Derived from

`wxInputStream` (p. 826)

Include files

`<wx/wfstream.h>`

See also

`wxBufferedInputStream` (p. 118), `wxFFFileOutputStream` (p. 505), `wxFileOutputStream` (p. 541)

wxFFFileInputStream::wxFFFileInputStream

wxFFFileInputStream(const wxString& filename, const wxChar * mode = "rb")

Opens the specified file using its *filename* name using the specified mode.

wxFFFileInputStream(wxFFFile& file)

Initializes a file stream in read-only mode using the file I/O object *file*.

wxFFFileInputStream(FILE * fp)

Initializes a file stream in read-only mode using the specified file pointer *fp*.

wxFFFileInputStream::~~wxFFFileInputStream

~wxFFFileInputStream()

Destructor.

wxFFFileInputStream::Ok

bool Ok() const

Returns true if the stream is initialized and ready.

wxFFFileOutputStream

This class represents data written to a file. There are actually two such groups of classes: this one is based on `wxFFFile` (p. 499) whereas `wxFileInputStream` (p. 504) is based in the `wxFile` (p. 506) class.

Note that `SeekO()` (p. **Error! Bookmark not defined.**) can seek beyond the end of the stream (file) and will thus not return `wxInvalidOffset` for that.

Derived from

wxOutputStream (p. **Error! Bookmark not defined.**)

Include files

<wx/wfstream.h>

See also

wxBufferedOutputStream (p. 119), *wxFFileInputStream* (p. 504), *wxFileInputStream* (p. 523)

wxFFileOutputStream::wxFFileOutputStream

wxFFileOutputStream(const wxString& filename, const wxChar * mode="w+b")

Opens the file with the given *filename* name in the specified mode.

wxFFileOutputStream(wxFFile& file)

Initializes a file stream in write-only mode using the file I/O object *file*.

wxFFileOutputStream(FILE * fp)

Initializes a file stream in write-only mode using the file descriptor *fp*.

wxFFileOutputStream::~~wxFFileOutputStream

~wxFFileOutputStream()

Destructor.

wxFFileOutputStream::Ok

bool Ok() const

Returns true if the stream is initialized and ready.

wxFFileStream**Derived from**

wxFFileOutputStream (p. 505), *wxFFileInputStream* (p. 504)

Include files

<wx/wfstream.h>

See also

wxStreamBuffer (p. **Error! Bookmark not defined.**)

wxFFileStream::wxFFileStream

wxFFileStream(const **wxString&** *iofileName*)

Initializes a new file stream in read-write mode using the specified *iofilename* name.

wxFile

A **wxFile** performs raw file I/O. This is a very small class designed to minimize the overhead of using it - in fact, there is hardly any overhead at all, but using it brings you automatic error checking and hides differences between platforms and compilers. **wxFile** also automatically closes the file in its destructor making it unnecessary to worry about forgetting to do it. **wxFile** is a wrapper around `file descriptor`. - see also *wxFFile* (p. 499) for a wrapper around `FILE` structure.

`wxFileOffset` is used by the **wxFile** functions which require offsets as parameter or return them. If the platform supports it, `wxFileOffset` is a typedef for a native 64 bit integer, else a 32 bit integer is used for `wxFileOffset`.

Derived from

None.

Include files

<wx/file.h>

Constants

wx/file.h defines the following constants:

```
#define wxS_IRUSR 00400
#define wxS_IWUSR 00200
#define wxS_IXUSR 00100

#define wxS_IRGRP 00040
#define wxS_IWGRP 00020
#define wxS_IXGRP 00010

#define wxS_IROTH 00004
#define wxS_IWOTH 00002
#define wxS_IXOTH 00001

// default mode for the new files: corresponds to umask 022
#define wxS_DEFAULT (wxS_IRUSR | wxS_IWUSR | wxS_IRGRP |
wxS_IWGRP | wxS_IROTH | wxS_IWOTH)
```

These constants define the file access rights and are used with *wxFile::Create* (p. 509) and *wxFile::Open* (p. 510).

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. 508) function.

<code>wxFile::read</code>	Open file for reading or test if it can be opened for reading with <code>Access()</code>
<code>wxFile::write</code>	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>
<code>wxFile::read_write</code>	Open file for reading and writing; can not be used with <code>Access()</code>
<code>wxFile::write_append</code>	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 <code>wxFile::write</code> if the file doesn't exist.
<code>wxFile::write_excl</code>	Open the file securely for writing (Uses <code>O_EXCL O_CREAT</code>). Will fail if the file already exists, else create and open it atomically. Useful for opening temporary files without being vulnerable to race exploits.

Other constants defined elsewhere but used by `wxFile` functions are `wxInvalidOffset` which represents an invalid value of type `wxFileOffset` and is returned by functions returning `wxFileOffset` on error and the seek mode constants used with `Seek()` (p. 511):

<code>wxFromStart</code>	Count offset from the start of the file
<code>wxFromCurrent</code>	Count offset from the current position of the file pointer
<code>wxFromEnd</code>	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. 510) 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. 509) for the list of predefined descriptors)

wxFile::~~wxFile

~wxFile()

Destructor will close the file.

NB: it is not virtual so you should not use **wxFile** polymorphically.

wxFile::Access

static bool Access(const char * name, OpenMode mode)

This function verifies if we may access the given file in specified mode. Only values of **wxFile::read** or **wxFile::write** really make sense here.

wxFile::Attach

void Attach(int fd)

Attaches an existing file descriptor to the **wxFile** object. Example of predefined file descriptors are 0, 1 and 2 which correspond to **stdin**, **stdout** and **stderr** (and have symbolic names of **wxFile::fd_stdin**, **wxFile::fd_stdout** and **wxFile::fd_stderr**).

The descriptor should be already opened and it will be closed by **wxFile** object.

wxFile::Close

void Close()

Closes the file.

wxFile::Create

bool Create(const char* filename, bool overwrite = false, int access = wxS_DEFAULT)

Creates a file for writing. If the file already exists, setting **overwrite** to true will ensure it is overwritten.

wxFile::Detach

void Detach()

Get back a file descriptor from wxFile object - the caller is responsible for closing the file if this descriptor is opened. *IsOpened()* (p. 510) 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 *wxFile* (p. 499) is different as *wxFile::Eof* (p. 500) 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. 511) repeatedly and tests its return condition instead of using *Eof()* as this will not work for special files under Unix.

wxFile::Exists

static bool Exists(const char* filename)

Returns true if the given name specifies an existing regular file (not a directory or a link)

wxFile::Flush

bool Flush()

Flushes the file descriptor.

Note that *wxFile::Flush* is not implemented on some Windows compilers due to a missing *fsync* function, which reduces the usefulness of this function (it can still be called but it will do nothing on unsupported compilers).

wxFile::GetKind

wxFileKind GetKind() const

Returns the type of the file. Possible return values are:

```
enum wxFileKind
{
    wxFILE_KIND_UNKNOWN,
    wxFILE_KIND_DISK,      // a file supporting seeking to arbitrary
offsets
    wxFILE_KIND_TERMINAL, // a tty
    wxFILE_KIND_PIPE      // a pipe
};
```

wxFile::IsOpened**bool IsOpened() const**

Returns true if the file has been opened.

wxFile::Length**wxFileOffset Length() const**

Returns the length of the file.

wxFile::Open**bool Open(const char* filename, wxFile::OpenMode mode = wxFile::read)**

Opens the file, returning true if successful.

Parameters

filename

The filename.

mode

The mode in which to open the file. May be one of **wxFile::read**, **wxFile::write** and **wxFile::read_write**.

wxFile::Read**size_t Read(void* buffer, size_t count)**

Reads the specified number of bytes into a buffer, returning the actual number read.

Parameters

buffer

A buffer to receive the data.

count

The number of bytes to read.

Return value

The number of bytes read, or the symbol **wxInvalidOffset** (-1) if there was an error.

wxFile::Seek

wxFileOffset Seek(**wxFileOffset** ofs, **wxSeekMode** mode = wxFromStart)

Seeks to the specified position.

Parameters

ofs

Offset to seek to.

mode

One of **wxFromStart**, **wxFromEnd**, **wxFromCurrent**.

Return value

The actual offset position achieved, or **wxInvalidOffset** on failure.

wxFile::SeekEnd

wxFileOffset SeekEnd(**wxFileOffset** ofs = 0)

Moves the file pointer to the specified number of bytes relative to the end of the file. For example, **SeekEnd**(-5) would position the pointer 5bytes before the end.

Parameters

ofs

Number of bytes before the end of the file.

Return value

The actual offset position achieved, or **wxInvalidOffset** on failure.

wxFile::Tell

wxFileOffset Tell() const

Returns the current position or **wxInvalidOffset** if file is not opened or if another error occurred.

wxFile::Write

size_t Write(const void* buffer, 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

the number of bytes actually written

wxFile::Write

bool Write(const wxString& s, wxMBConv& conv = wxConvUTF8)

Writes the contents of the string to the file, returns true on success.

The second argument is only meaningful in Unicode build of wxWidgets when *conv* is used to convert *s* to multibyte representation.

Note that this method only works with NUL-terminated strings, if you want to write data with embedded NULs to the file you should use the other *Write() overload* (p. 512).

wxFileConfig

wxFileConfig implements *wxConfigBase* (p. 196) interface for storing and retrieving configuration information using plain text files. The files have a simple format reminiscent of Windows INI files with lines of the form `key = value` defining the keys and lines of special form `[group]` indicating the start of each group.

This class is used by default for wxConfig on Unix platforms but may also be used explicitly if you want to use files and not the registry even under Windows.

Derived from

wxConfigBase (p. 196)

Include files

<wx/fileconf.h>

wxFileConfig::wxFileConfig

wxFileConfig(wxInputStream& is, wxMBConv& conv = wxConvUTF8)

Read the config data from the specified stream instead of the associated file, as usual.

See also

Save (p. 513)

wxFileConfig::Save

bool **Save**(**wxOutputStream&** *os*, **wxMBConv&** *conv* = *wxConvUTF8*)

Saves all config data to the given stream, returns `true` if data was saved successfully or `false` on error.

Note the interaction of this function with the internal "dirty flag": the data is saved unconditionally, i.e. even if the object is not dirty. However after saving it successfully, the dirty flag is reset so no changes will be written back to the file this object is associated with until you change its contents again.

See also

Flush (p. 204)

wxFileConfig::SetUmask

void **SetUmask**(**int** *mode*)

Allows to set the mode to be used for the config file creation. For example, to create a config file which is not readable by other users (useful if it stores some sensitive information, such as passwords), you could use `SetUmask(0077)`.

This function doesn't do anything on non-Unix platforms.

See also

`wxCHANGE_UMASK` (p. **Error! Bookmark not defined.**)

wxFileDataObject

`wxFileDataObject` is a specialization of `wxDataObject` (p. 242) for file names. The program works with it just as if it were a list of absolute file names, but internally it uses the same format as Explorer and other compatible programs under Windows or GNOME/KDE filemanager under Unix which makes it possible to receive files from them using this class.

Warning: Under all non-Windows platforms this class is currently "input-only", i.e. you can receive the files from another application, but copying (or dragging) file(s) from a `wxWidgets` application is not currently supported. PS: GTK2 should work as well.

Virtual functions to override

None.

Derived from

wxDataObjectSimple (p. 247)
wxDataObject (p. 242)

Include files

<wx/dataobj.h>

See also

wxDataObject (p. 242), *wxDataObjectSimple* (p. 247), *wxTextDataObject* (p. **Error! Bookmark not defined.**), *wxBitmapDataObject* (p. 103), *wxDataObject* (p. 242)

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. 70) of file names.

wxFileDialog

This class represents the file chooser dialog.

Derived from

wxDialog (p. 412)
wxWindow (p. **Error! Bookmark not defined.**)
wxEvtHandler (p. 490)
wxObject (p. **Error! Bookmark not defined.**)

Include files

<wx/filedlg.h>

See also

wxFileDialog overview (p. **Error! Bookmark not defined.**), *wxFileSelector* (p. **Error!**

Bookmark not defined.)

Remarks

Pops up a file selector box. In Windows and GTK2.4+, this is the common file selector dialog. In X, this is a file selector box with somewhat less functionality. The path and filename are distinct elements of a full file pathname. If path is "", the current directory will be used. If filename is "", no default filename will be supplied. The wildcard determines what files are displayed in the file selector, and file extension supplies a type extension for the required filename. Flags may be a combination of wxOPEN, wxSAVE, wxOVERWRITE_PROMPT, wxHIDE_READONLY, wxFILE_MUST_EXIST, wxMULTIPLE, wxCHANGE_DIR or 0.

Both the X and Windows versions implement a wildcard filter. Typing a filename containing wildcards (*, ?) in the filename text item, and clicking on Ok, will result in only those files matching the pattern being displayed. The wildcard may be a specification for multiple types of file with a description for each, such as:

```
"BMP and GIF files (*.bmp;*.gif)|*.bmp;*.gif|PNG files  
(*.png)|*.png"
```

It must be noted that wildcard support in the native Motif file dialog is quite limited: only one alternative is supported, and it is displayed without the descriptive test; "BMP files (*.bmp)|*.bmp" is displayed as "*.bmp", and both "BMP files (*.bmp)|*.bmp|GIF files (*.gif)|*.gif" and "Image files|*.bmp;*.gif" are errors.

wxFileDialog::wxFileDialog

wxFileDialog(wxWindow* parent, const wxString& message = "Choose a file", const wxString& defaultDir = "", const wxString& defaultFile = "", const wxString& wildcard = ".*", long style = 0, const wxPoint& pos = wxDefaultPosition)

Constructor. Use *wxFileDialog::ShowModal* (p. 519) to show the dialog.

Parameters

parent

Parent window.

message

Message to show on the dialog.

defaultDir

The default directory, or the empty string.

defaultFile

The default filename, or the empty string.

wildcard

A wildcard, such as `"*.*" or "BMP files (*.bmp)|*.bmp|GIF files (*.gif)|*.gif"`.

Note that the native Motif dialog has some limitations with respect to wildcards; see the Remarks section above.

style

A dialog style. A bitlist of:

wxOPEN	This is an open dialog.
wxSAVE	This is a save dialog.
wxOVERWRITE_PROMPT	For save dialog only: prompt for a confirmation if a file will be overwritten.
wxHIDE_READONLY	Do not display the checkbox to toggle display of read-only files. Deprecated in 2.6; the checkbox is never shown.
wxFILE_MUST_EXIST	The user may only select files that actually exist.
wxMULTIPLE	For open dialog only: allows selecting multiple files.
wxCHANGE_DIR	Change the current working directory to the directory where the file(s) chosen by the user are.

pos

Dialog position. Not implemented.

NB: Previous versions of `wxWidgets` used `wxCHANGE_DIR` by default under MS Windows which allowed the program to simply remember the last directory where user selected the files to open/save. This (desired) functionality must be implemented in the program itself now (manually remember the last path used and pass it to the dialog the next time it is called) or by using this flag.

wxFileDialog::~wxFileDialog

~wxFileDialog()

Destructor.

wxFileDialog::GetDirectory

wxString GetDirectory() const

Returns the default directory.

wxFileDialog::GetFilename

wxString GetFilename() const

Returns the default filename.

wxFileDialog::GetFileNames**void GetFileNames(wxArrayString& *filenames*) const**

Fills the array *filenames* with the names of the files chosen. This function should only be used with the dialogs which have `wxMULTIPLE` style, use *GetFilename* (p. 517) for the others.

Note that under Windows, if the user selects shortcuts, the filenames include paths, since the application cannot determine the full path of each referenced file by appending the directory containing the shortcuts to the filename.

wxFileDialog::GetFilterIndex**int GetFilterIndex() const**

Returns the index into the list of filters supplied, optionally, in the wildcard parameter. Before the dialog is shown, this is the index which will be used when the dialog is first displayed. After the dialog is shown, this is the index selected by the user.

wxFileDialog::GetMessage**wxString GetMessage() const**

Returns the message that will be displayed on the dialog.

wxFileDialog::GetPath**wxString GetPath() const**

Returns the full path (directory and filename) of the selected file.

wxFileDialog::GetPaths**void GetPaths(wxArrayString& *paths*) const**

Fills the array *paths* with the full paths of the files chosen. This function should only be used with the dialogs which have `wxMULTIPLE` style, use *GetPath* (p. 518) for the others.

wxFileDialog::GetStyle**long GetStyle() const**

Returns the dialog style.

wxFileDialog::GetWildcard**wxString GetWildcard() const**

Returns the file dialog wildcard.

wxFileDialog::SetDirectory**void SetDirectory(const wxString& *directory*)**

Sets the default directory.

wxFileDialog::SetFilename**void SetFilename(const wxString& *setfilename*)**

Sets the default filename.

wxFileDialog::SetFilterIndex**void SetFilterIndex(int *filterIndex*)**

Sets the default filter index, starting from zero.

wxFileDialog::SetMessage**void SetMessage(const wxString& *message*)**

Sets the message that will be displayed on the dialog.

wxFileDialog::SetPath**void SetPath(const wxString& *path*)**

Sets the path (the combined directory and filename that will be returned when the dialog is dismissed).

wxFileDialog::SetStyle**void SetStyle(long *style*)**

Sets the dialog style. See *wxFileDialog::wxFileDialog* (p. 516) for details.

wxFileDialog::SetWildcard**void SetWildcard(const wxString& *wildCard*)**

Sets the wildcard, which can contain multiple file types, for example:

"BMP files (*.bmp)|*.bmp|GIF files (*.gif)|*.gif"

Note that the native Motif dialog has some limitations with respect to wildcards; see the Remarks section above.

wxFileDialog::ShowModal

int ShowModal()

Shows the dialog, returning `wxID_OK` if the user pressed OK, and `wxID_CANCEL` otherwise.

wxFileDropTarget

This is a *drop target* (p. 475) which accepts files (dragged from File Manager or Explorer).

Derived from

wxDropTarget (p. 475)

Include files

`<wx/dnd.h>`

See also

Drag and drop overview (p. **Error! Bookmark not defined.**), *wxDropSource* (p. 472), *wxDropTarget* (p. 475), *wxTextDropTarget* (p. **Error! Bookmark not defined.**)

wxFileDropTarget::wxFileDropTarget

wxFileDropTarget()

Constructor.

wxFileDropTarget::OnDrop

virtual bool OnDrop(long x, long y, const void *data, size_t size)

See *wxDropTarget::OnDrop* (p. 476). This function is implemented appropriately for files, and calls *wxFileDropTarget::OnDropFiles* (p. 520).

wxFileDropTarget::OnDropFiles

virtual bool OnDropFiles(wxCoord x, wxCoord y, const wxString& filenames)

Override this function to receive dropped files.

Parameters

x

The x coordinate of the mouse.

y

The y coordinate of the mouse.

filenames

An array of filenames.

Return value

Return true to accept the data, false to veto the operation.

wxFileHistory

The wxFileHistory encapsulates a user interface convenience, the list of most recently visited files as shown on a menu (usually the File menu).

wxFileHistory can manage one or more file menus. More than one menu may be required in an MDI application, where the file history should appear on each MDI child menu as well as the MDI parent frame.

Derived from

wxObject (p. **Error! Bookmark not defined.**)

Include files

<wx/docview.h>

See also

wxFileHistory overview (p. **Error! Bookmark not defined.**), *wxDocManager* (p. 441)

wxFileHistory::m_fileHistory

char m_fileHistory**

A character array of strings corresponding to the most recently opened files.

wxFileHistory::m_fileHistoryN

size_t m_fileHistoryN

The number of files stored in the history array.

wxFileHistory::m_fileMaxFiles

size_t m_fileMaxFiles

The maximum number of files to be stored and displayed on the menu.

wxFileHistory::m_fileMenu**wxMenu* m_fileMenu**

The file menu used to display the file history list (if enabled).

wxFileHistory::wxFileHistory

wxFileHistory(size_t maxFiles = 9, wxWindowID idBase = wxID_FILE1)

Constructor. Pass the maximum number of files that should be stored and displayed.

idBase defaults to `wxID_FILE1` and represents the id given to the first history menu item. Since menu items can't share the same ID you should change *idBase* (To one of your own defined IDs) when using more than one `wxFileHistory` in your application.

wxFileHistory::~wxFileHistory

~wxFileHistory()

Destructor.

wxFileHistory::AddFileToHistory

void AddFileToHistory(const wxString& filename)

Adds a file to the file history list, if the object has a pointer to an appropriate file menu.

wxFileHistory::AddFilesToMenu

void AddFilesToMenu()

Appends the files in the history list, to all menus managed by the file history object.

void AddFilesToMenu(wxMenu* menu)

Appends the files in the history list, to the given menu only.

wxFileHistory::GetCount

size_t GetCount() const

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

wxFileHistory::GetHistoryFile

wxString GetHistoryFile(size_t 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::GetMenus

const wxList& GetMenus() const

Returns the list of menus that are managed by this file history object.

See also

wxFileHistory::UseMenu (p. 523)

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. 196)

wxFileHistory::RemoveFileFromHistory

void RemoveFileFromHistory(size_t i)

Removes the specified file from the history.

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. 196)

wxFileHistory::UseMenu**void UseMenu(wxMenu* menu)**

Adds this menu to the list of those menus that are managed by this file history object. Also see *AddFilesToMenu()* (p. 522) for initializing the menu with filenames that are already in the history when this function is called, as this is not done automatically.

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. 506) whereas *wxFFileInputStream* (p. 504) is based in the *wxFFile* (p. 499) class.

Note that *SeekI()* (p. 828) can seek beyond the end of the stream (file) and will thus not return *wxInvalidOffset* for that.

Derived from

wxInputStream (p. 826)

Include files

<wx/wfstream.h>

See also

wxBufferedInputStream (p. 118), *wxFileOutputStream* (p. 541), *wxFFileOutputStream* (p. 505)

wxFileInputStream::wxFileInputStream**wxFileInputStream(const wxString& ifilename)**

Opens the specified file using its *ifilename* name in read-only mode.

wxFileInputStream(wxFile& file)

Initializes a file stream in read-only mode using the file I/O object *file*.

wxFileInputStream(int fd)

Initializes a file stream in read-only mode using the specified file descriptor.

wxFileInputStream::~~wxFileInputStream**~wxFileInputStream()**

Destructor.

wxFileInputStream::Ok

bool Ok() const

Returns true if the stream is initialized and ready.

wxFileName

wxFileName encapsulates a file name. This class serves two purposes: first, it provides the functions to split the file names into components and to recombine these components in the full file name which can then be passed to the OS file functions (and *wxWidgets functions* (p. **Error! Bookmark not defined.**) wrapping them). Second, it includes the functions for working with the files itself. Note that to change the file data you should use *wxFile* (p. 506) class instead. wxFileName provides functions for working with the file attributes.

Derived from

No base class

Include files

<wx/filename.h>

Data structures

Many wxFileName methods accept the path format argument which is by `wxPATH_NATIVE` by default meaning to use the path format native for the current platform.

The path format affects the operation of wxFileName functions in several ways: first and foremost, it defines the path separator character to use, but it also affects other things such as whether the path has the drive part or not.

```
enum wxPathFormat
{
    wxPATH_NATIVE = 0,          // the path format for the current
platform
    wxPATH_UNIX,
    wxPATH_BEOS = wxPATH_UNIX,
    wxPATH_MAC,
    wxPATH_DOS,
    wxPATH_WIN = wxPATH_DOS,
    wxPATH_OS2 = wxPATH_DOS,
    wxPATH_VMS,

    wxPATH_MAX // Not a valid value for specifying path format
}
```

File name format

wxFileName currently supports the file names in the Unix, DOS/Windows, Mac OS and

VMS formats. Although these formats are quite different, `wxFileName` tries to treat them all in the same generic way. It supposes that all file names consist of the following parts: the volume (also known as drive under Windows or device under VMS), the path which is a sequence of directory names separated by the *path separators* (p. 533) and the full filename itself which, in turn, is composed from the base file name and the extension. All of the individual components of the file name may be empty and, for example, the volume name is always empty under Unix, but if they are all empty simultaneously, the filename object is considered to be in an invalid state and `IsOk` (p. 534) returns `false` for it.

File names can be case-sensitive or not, the function `IsCaseSensitive` (p. 534) allows to determine this.

The rules for determining if the file name is absolute or relative also depends on the file name format and the only portable way to answer to this question is to use `IsAbsolute` (p. 534) method. To ensure that the filename is absolute you may use `MakeAbsolute` (p. 535). There is also an inverse function `MakeRelativeTo` (p. 536) which undoes what `Normalize(wxPATH_NORM_DOTS)` (p. 537) does.

Other functions returning information about the file format provided by this class are `GetVolumeSeparator` (p. 534), `IsPathSeparator` (p. 535).

`IsRelative` (p. 535)

File name construction

TODO.

File tests

Before doing the other tests you should use `IsOk` (p. 534) to verify that the filename is well defined. If it is, `FileExists` (p. 530) can be used to test if a file with such name exists and `DirExists` (p. 530) - if a directory with this name exists.

File names should be compared using `SameAs` (p. 538) method or `==` (p. 540).

File name components

These functions allow to examine and modify the individual directories of the path:

`AppendDir` (p. 528)

`InsertDir` (p. 534)

`GetDirCount` (p. 531) `PrependDir` (p. 537)

`RemoveDir` (p. 537)

`RemoveLastDir` (p. 538)

To change the components of the file name individually you can use the following functions:

`GetExt` (p. 531)

`GetName` (p. 532)

GetVolume (p. 533)
HasExt (p. 534)
HasName (p. 534)
HasVolume (p. 534)
SetExt (p. 538)
ClearExt (p. 529)
SetEmptyExt (p. 538)
SetName (p. 539)
SetVolume (p. 539)

Operations

These methods allow to work with the file creation, access and modification times. Note that not all filesystems under all platforms implement these times in the same way. For example, the access time under Windows has a resolution of one day (so it is really the access date and not time). The access time may be updated when the file is executed or not depending on the platform.

GetModificationTime (p. 532)
GetTimes (p. 533)
SetTimes (p. 539)
Touch (p. 540)

Other file system operations functions are:

Mkdir (p. 536)
Rmdir (p. 538)

wxFileName::wxFileName

wxFileName()

Default constructor.

wxFileName(const wxFileName& filename)

Copy constructor.

wxFileName(const wxString& fullpath, wxPathFormat format = wxPATH_NATIVE)

Constructor taking a full filename. If it terminates with a '/', a directory path is constructed (the name will be empty), otherwise a file name and extension are extracted from it.

wxFileName(const wxString& path, const wxString& name, wxPathFormat format = wxPATH_NATIVE)

Constructor from a directory name and a file name.

wxFileName(const wxString& path, const wxString& name, const wxString& ext, wxPathFormat format = wxPATH_NATIVE)

Constructor from a directory name, base file name and extension.

wxFileName(const wxString& volume, const wxString& path, const wxString& name, const wxString& ext, wxPathFormat format = wxPATH_NATIVE)

Constructor from a volume name, a directory name, base file name and extension.

wxFileName::AppendDir

void AppendDir(const wxString& dir)

Appends a directory component to the path. This component should contain a single directory name level, i.e. not contain any path or volume separators nor should it be empty, otherwise the function does nothing (and generates an assert failure in debug build).

wxFileName::Assign

void Assign(const wxFileName& filepath)

void Assign(const wxString& fullpath, wxPathFormat format = wxPATH_NATIVE)

void Assign(const wxString& volume, const wxString& path, const wxString& name, const wxString& ext, bool hasExt, wxPathFormat format = wxPATH_NATIVE)

void Assign(const wxString& volume, const wxString& path, const wxString& name, const wxString& ext, wxPathFormat format = wxPATH_NATIVE)

void Assign(const wxString& path, const wxString& name, wxPathFormat format = wxPATH_NATIVE)

void Assign(const wxString& path, const wxString& name, const wxString& ext, wxPathFormat format = wxPATH_NATIVE)

Creates the file name from various combinations of data.

wxFileName::AssignCwd

static void AssignCwd(const wxString& volume = wxEmptyString)

Makes this object refer to the current working directory on the specified volume (or current volume if *volume* is empty).

See also

GetCwd (p. 530)

wxFileName::AssignDir

void AssignDir(const wxString& dir, wxPathFormat format = wxPATH_NATIVE)

Sets this file name object to the given directory name. The name and extension will be

empty.

wxFileName::AssignHomeDir

void AssignHomeDir()

Sets this file name object to the home directory.

wxFileName::AssignTempFileName

void AssignTempFileName(const wxString& prefix, wxFile *fileTemp = NULL)

The function calls *CreateTempFileName* (p. 529) to create a temporary file and sets this object to the name of the file. If a temporary file couldn't be created, the object is put into the *invalid* (p. 534) state.

wxFileName::Clear

void Clear()

Reset all components to default, uninitialized state.

wxFileName::ClearExt

void SetClearExt()

Removes the extension from the file name resulting in a file name with no trailing dot.

See also

SetExt (p. 538) *SetEmptyExt* (p. 538)

wxFileName::CreateTempFileName

static wxString CreateTempFileName(const wxString& prefix, wxFile *fileTemp = NULL)

Returns a temporary file name starting with the given *prefix*. If the *prefix* is an absolute path, the temporary file is created in this directory, otherwise it is created in the default system directory for the temporary files or in the current directory.

If the function succeeds, the temporary file is actually created. If *fileTemp* is not `NULL`, this file will be opened using the name of the temporary file. When possible, this is done in an atomic way ensuring that no race condition occurs between the temporary file name generation and opening it which could often lead to security compromise on the multiuser systems. If *fileTemp* is `NULL`, the file is only created, but not opened.

Under Unix, the temporary file will have read and write permissions for the owner only to minimize the security problems.

Parameters

prefix

Prefix to use for the temporary file name construction

fileTemp

The file to open or `NULL` to just get the name

Return value

The full temporary file name or an empty string on error.

wxFileName::DirExists

bool DirExists() const

static bool DirExists(const wxString& dir)

Returns `true` if the directory with this name exists.

wxFileName::DirName

static wxFileName DirName(const wxString& dir, wxPathFormat format = wxPATH_NATIVE)

Returns the object corresponding to the directory with the given name. The *dir* parameter may have trailing path separator or not.

wxFileName::FileExists

bool FileExists() const

static bool FileExists(const wxString& file)

Returns `true` if the file with this name exists.

See also

DirExists (p. 530)

wxFileName::FileName

static wxFileName FileName(const wxString& file, wxPathFormat format = wxPATH_NATIVE)

Returns the file name object corresponding to the given *file*. This function exists mainly for symmetry with *DirName* (p. 530).

wxFileName::GetCwd

static wxString GetCwd(const wxString& volume = "")

Retrieves the value of the current working directory on the specified volume. If the volume is empty, the program's current working directory is returned for the current volume.

Return value

The string containing the current working directory or an empty string on error.

See also

AssignCwd (p. 528)

wxFileName::GetDirCount

size_t GetDirCount() const

Returns the number of directories in the file name.

wxFileName::GetDirs

const wxArrayString& GetDirs() const

Returns the directories in string array form.

wxFileName::GetExt

wxString GetExt() const

Returns the file name extension.

wxFileName::GetForbiddenChars

static wxString GetForbiddenChars(wxPathFormat format = wxPATH_NATIVE)

Returns the characters that can't be used in filenames and directory names for the specified format.

wxFileName::GetFormat

static wxPathFormat GetFormat(wxPathFormat format = wxPATH_NATIVE)

Returns the canonical path format for this platform.

wxFileName::GetFullName

wxString GetFullName() const

Returns the full name (including extension but excluding directories).

wxFileName::GetFullPath

wxString GetFullPath(wxPathFormat format = wxPATH_NATIVE) const

Returns the full path with name and extension.

wxFileName::GetHomeDir

static wxString GetHomeDir()

Returns the home directory.

wxFileName::GetLongPath

wxString GetLongPath() const

Return the long form of the path (returns identity on non-Windows platforms)

wxFileName::GetModificationTime

wxDateTime GetModificationTime() const

Returns the last time the file was last modified.

wxFileName::GetName

wxString GetName() const

Returns the name part of the filename (without extension).

See also

GetFullName (p. 531)

wxFileName::GetPath

wxString GetPath(int flags = wxPATH_GET_VOLUME, wxPathFormat format = wxPATH_NATIVE) const

Returns the path part of the filename (without the name or extension). The possible flags values are:

wxPATH_GET_VOLUME Return the path with the volume (does nothing for the filename formats without volumes), otherwise the path without volume part is returned.

wxPATH_GET_SEPARATOR Return the path with the trailing separator, if this flag is not given there will be no separator at the end of the path.

wxFileName::GetPathSeparator

static wxChar GetPathSeparator(wxPathFormat format = wxPATH_NATIVE)

Returns the usually used path separator for this format. For all formats but `wxPATH_DOS` there is only one path separator anyhow, but for DOS there are two of them and the native one, i.e. the backslash is returned by this method.

See also

GetPathSeparators (p. 533)

wxFileName::GetPathSeparators

static wxString GetPathSeparators(wxPathFormat format = wxPATH_NATIVE)

Returns the string containing all the path separators for this format. For all formats but `wxPATH_DOS` this string contains only one character but for DOS and Windows both `'/'` and `'\'` may be used as separators.

See also

GetPathSeparator (p. 532)

wxFileName::GetPathTerminators

static wxString GetPathTerminators(wxPathFormat format = wxPATH_NATIVE)

Returns the string of characters which may terminate the path part. This is the same as *GetPathSeparators* (p. 533) except for VMS path format where `]` is used at the end of the path part.

wxFileName::GetShortPath

wxString GetShortPath() const

Return the short form of the path (returns identity on non-Windows platforms).

wxFileName::GetTimes

**bool GetTimes(wxDateTime* dtAccess, wxDateTime* dtMod, wxDateTime* dtCreate)
const**

Returns the last access, last modification and creation times. The last access time is updated whenever the file is read or written (or executed in the case of Windows), last modification time is only changed when the file is written to. Finally, the creation time is indeed the time when the file was created under Windows and the inode change time under Unix (as it is impossible to retrieve the real file creation time there anyhow) which can also be changed by many operations after the file creation.

Any of the pointers may be `NULL` if the corresponding time is not needed.

Return value

`true` on success, `false` if we failed to retrieve the times.

wxFileName::GetVolume**wxString GetVolume() const**

Returns the string containing the volume for this file name, empty if it doesn't have one or if the file system doesn't support volumes at all (for example, Unix).

wxFileName::GetVolumeSeparator**static wxString GetVolumeSeparator(wxPathFormat *format* = wxPATH_NATIVE)**

Returns the string separating the volume from the path for this format.

wxFileName::HasExt**bool HasExt() const**

Returns `true` if an extension is present.

wxFileName::HasName**bool HasName() const**

Returns `true` if a name is present.

wxFileName::HasVolume**bool HasVolume() const**

Returns `true` if a volume specifier is present.

wxFileName::InsertDir**void InsertDir(size_t *before*, const wxString& *dir*)**

Inserts a directory component before the zero-based position in the directory list. Please see *AppendDir* (p. 528) for important notes.

wxFileName::IsAbsolute**bool IsAbsolute(wxPathFormat *format* = wxPATH_NATIVE)**

Returns `true` if this filename is absolute.

wxFileName::IsCaseSensitive**static bool IsCaseSensitive(wxPathFormat *format* = wxPATH_NATIVE)**

Returns `true` if the file names of this type are case-sensitive.

wxFileName::IsOk**bool IsOk() const**

Returns `true` if the filename is valid, `false` if it is not initialized yet. The assignment functions and *Clear* (p. 529) may reset the object to the uninitialized, invalid state (the former only do it on failure).

wxFileName::IsPathSeparator**static bool IsPathSeparator(wxChar ch, wxPathFormat format = wxPATH_NATIVE)**

Returns `true` if the char is a path separator for this format.

wxFileName::IsRelative**bool IsRelative(wxPathFormat format = wxPATH_NATIVE)**

Returns `true` if this filename is not absolute.

wxFileName::IsDir**bool IsDir() const**

Returns `true` if this object represents a directory, `false` otherwise (i.e. if it is a file). Note that this method doesn't test whether the directory or file really exists, you should use *DirExists* (p. 530) or *FileExists* (p. 530) for this.

wxFileName::MacFindDefaultTypeAndCreator**static bool MacFindDefaultTypeAndCreator(const wxString& ext, wxUint32* type, wxUint32* creator)**

On Mac OS, gets the common type and creator for the given extension.

wxFileName::MacRegisterDefaultTypeAndCreator**static void MacRegisterDefaultTypeAndCreator(const wxString& ext, wxUint32 type, wxUint32 creator)**

On Mac OS, registers application defined extensions and their default type and creator.

wxFileName::MacSetDefaultTypeAndCreator**bool MacSetDefaultTypeAndCreator()**

On Mac OS, looks up the appropriate type and creator from the registration and then sets it.

wxFileName::MakeAbsolute

bool MakeAbsolute(const wxString& cwd = wxEmptyString, wxPathFormat format = wxPATH_NATIVE)

Make the file name absolute. This is a shortcut for `Normalize (p. 537) (wxPATH_NORM_DOTS | wxPATH_NORM_ABSOLUTE | wxPATH_NORM_TILDE, cwd, format)`.

See also

MakeRelativeTo (p. 536), *Normalize* (p. 537), *IsAbsolute* (p. 534)

wxFileName::MakeRelativeTo

bool MakeRelativeTo(const wxString& pathBase = wxEmptyString, wxPathFormat format = wxPATH_NATIVE)

This function tries to put this file name in a form relative to *pathBase*. In other words, it returns the file name which should be used to access this file if the current directory were *pathBase*.

pathBase

the directory to use as root, current directory is used by default

format

the file name format, native by default

Return value

`true` if the file name has been changed, `false` if we failed to do anything with it (currently this only happens if the file name is on a volume different from the volume specified by *pathBase*).

See also

Normalize (p. 537)

wxFileName::Mkdir

bool Mkdir(int perm = 0777, int flags = 0)

static bool Mkdir(const wxString& dir, int perm = 0777, int flags = 0)

dir

the directory to create

perm

the permissions for the newly created directory

flags

if the flags contain `wxPATH_MKDIR_FULL` flag, try to create each directory in the path and also don't return an error if the target directory already exists.

Return value

Returns `true` if the directory was successfully created, `false` otherwise.

wxFileName::Normalize

bool Normalize(int *flags* = `wxPATH_NORM_ALL`, const **wxString&** *cwd* = `wxEmptyString`, **wxPathFormat** *format* = `wxPATH_NATIVE`)

Normalize the path. With the default flags value, the path will be made absolute, without any `".."` and `"."` and all environment variables will be expanded in it.

flags

The kind of normalization to do with the file name. It can be any or-combination of the following constants:

wxPATH_NORM_ENV_VARS replace env vars with their values

wxPATH_NORM_DOTS squeeze all `..` and `.` and prepend `cwd`

wxPATH_NORM_TILDE Unix only: replace `~` and `~user`

wxPATH_NORM_CASE if filesystem is case insensitive, transform to lower case

wxPATH_NORM_ABSOLUTE make the path absolute

wxPATH_NORM_LONG make the path the long form

wxPATH_NORM_SHORTCUT resolve if it is a shortcut (Windows only)

wxPATH_NORM_ALL all of previous flags except `wxPATH_NORM_CASE`

cwd

If not empty, this directory will be used instead of current working directory in normalization.

format

The file name format, native by default.

wxFileName::PrependDir

void PrependDir(const **wxString&** *dir*)

Prepends a directory to the file path. Please see *AppendDir* (p. 528) for important notes.

wxFileName::RemoveDir

void RemoveDir(size_t pos)

Removes the specified directory component from the path.

See also

GetDirCount (p. 531)

wxFileName::RemoveLastDir

void RemoveLastDir()

Removes last directory component from the path.

wxFileName::Rmdir

bool Rmdir()

static bool Rmdir(const wxString& dir)

Deletes the specified directory from the file system.

wxFileName::SameAs

bool SameAs(const wxFileName& filepath, wxPathFormat format = wxPATH_NATIVE) const

Compares the filename using the rules of this platform.

wxFileName::SetCwd

bool SetCwd()

static bool SetCwd(const wxString& cwd)

Changes the current working directory.

wxFileName::SetExt

void SetExt(const wxString& ext)

Sets the extension of the file name. Setting an empty string as the extension will remove the extension resulting in a file name without a trailing dot, unlike a call to *SetEmptyExt* (p. 538).

See also

SetEmptyExt (p. 538) *ClearExt* (p. 529)

wxFileName::SetEmptyExt

void SetEmptyExt()

Sets the extension of the file name to be an empty extension. This is different from having no extension at all as the file name will have a trailing dot after a call to this method.

See also

SetExt (p. 538) *ClearExt* (p. 529)

wxFileName::SetFullName**void SetFullName(const wxString& fullname)**

The full name is the file name and extension (but without the path).

wxFileName::SetName**void SetName(const wxString& name)**

Sets the name part (without extension).

See also

SetFullName (p. 539)

wxFileName::SetTimes**bool SetTimes(const wxDateTime* dtAccess, const wxDateTime* dtMod, const wxDateTime* dtCreate)**

Sets the file creation and last access/modification times (any of the pointers may be NULL).

wxFileName::SetVolume**void SetVolume(const wxString& volume)**

Sets the volume specifier.

wxFileName::SplitPath

static void SplitPath(const wxString& fullpath, wxString* volume, wxString* path, wxString* name, wxString* ext, bool *hasExt = NULL, wxPathFormat format = wxPATH_NATIVE)

static void SplitPath(const wxString& fullpath, wxString* volume, wxString* path, wxString* name, wxString* ext, wxPathFormat format = wxPATH_NATIVE)

static void SplitPath(const wxString& fullpath, wxString* path, wxString* name, wxString* ext, wxPathFormat format = wxPATH_NATIVE)

This function splits a full file name into components: the volume (with the first version) path (including the volume in the second version), the base name and the extension. Any of the output parameters (*volume*, *path*, *name* or *ext*) may be `NULL` if you are not interested in the value of a particular component. Also, *fullpath* may be empty on entry.

On return, *path* contains the file path (without the trailing separator), *name* contains the file name and *ext* contains the file extension without leading dot. All three of them may be empty if the corresponding component is. The old contents of the strings pointed to by these parameters will be overwritten in any case (if the pointers are not `NULL`).

Note that for a filename "foo." the extension is present, as indicated by the trailing dot, but empty. If you need to cope with such cases, you should use *hasExt* instead of relying on testing whether *ext* is empty or not.

wxFileName::SplitVolume

static void SplitVolume(const wxString& fullpath, wxString* volume, wxString* path, wxPathFormat format = wxPATH_NATIVE)

Splits the given *fullpath* into the volume part (which may be empty) and the pure path part, not containing any volume.

See also

SplitPath (p. 539)

wxFileName::Touch

bool Touch()

Sets the access and modification times to the current moment.

wxFileName::operator=

wxFileName& operator operator=(const wxFileName& filename)

wxFileName& operator operator=(const wxString& filename)

Assigns the new value to this filename object.

wxFileName::operator==

bool operator operator==(const wxFileName& filename) const

bool operator operator==(const wxString& filename) const

Returns `true` if the filenames are equal. The string *filenames* is interpreted as a path in the native filename format.

wxFileName::operator!=

bool operator operator!=(const wxFileName& filename) const

bool operator operator!=(const wxString& filename) const

Returns `true` if the filenames are different. The string *filename* is interpreted as a path in the native filename format.

wxFileOutputStream

This class represents data written to a file. There are actually two such groups of classes: this one is based on *wxFile* (p. 506) whereas *wxFileInputStream* (p. 504) is based in the *wxFile* (p. 499) class.

Note that *SeekO()* (p. **Error! Bookmark not defined.**) can seek beyond the end of the stream (file) and will thus not return *wxInvalidOffset* for that.

Derived from

wxOutputStream (p. **Error! Bookmark not defined.**)

Include files

<wx/wfstream.h>

See also

wxBufferedOutputStream (p. 119), *wxFileInputStream* (p. 523), *wxFileInputStream* (p. 504)

wxFileOutputStream::wxFileOutputStream

wxFileOutputStream(const wxString& ofFileName)

Creates a new file with *ofFileName* 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. 541), *wxFileInputStream* (p. 523)

Include files

<wx/wfstream.h>

See also

wxStreamBuffer (p. **Error! Bookmark not defined.**)

wxFileStream::wxFileStream

wxFileStream(const wxString& iofilename)

Initializes a new file stream in read-write mode using the specified *iofilename* name.

wxFileSystem

This class provides an interface for opening files on different file systems. It can handle absolute and/or local filenames. It uses a system of *handlers* (p. 545) to provide access to user-defined virtual file systems.

Derived from

wxObject (p. **Error! Bookmark not defined.**)

Include files

<wx/filesys.h>

See Also

wxFileSystemHandler (p. 545), *wxFSFile* (p. 593), *Overview* (p. **Error! Bookmark not defined.**)

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. 545) 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. 544) is relative to this path.

Caution! Unless *is_dir* is true the *location* parameter is not the directory name but the name of the file in this directory. All these commands change the path to "dir/subdir/":

```
ChangePathTo("dir/subdir/xh.htm");  
ChangePathTo("dir/subdir", true);  
ChangePathTo("dir/subdir/", true);
```

Parameters

location

the new location. Its meaning depends on the value of *is_dir*

is_dir

if true *location* is new directory. If false (default) *location* is **file in** the new directory.

Example

```
f = fs -> OpenFile("hello.htm"); // opens file 'hello.htm'  
fs -> ChangePathTo("subdir/folder", true);  
f = fs -> OpenFile("hello.htm"); // opens file  
'subdir/folder/hello.htm' !!
```

wxFileSystem::GetPath

wxString GetPath()

Returns actual path (set by *ChangePathTo* (p. 543)).

wxFileSystem::FileNameToURL

static wxString FileNameToURL(wxFileName filename)

Converts filename into URL.

See also

wxFileSystem::URLToFileName (p. 544), *wxFileName* (p. 524)

wxFileSystem::FindFirst

wxString FindFirst(const wxString& wildcard, int flags = 0)

Works like *wxFindFirstFile* (p. **Error! Bookmark not defined.**). Returns name of the first filename (within filesystem's current path) that matches *wildcard*. *flags* may be one of *wxFILE* (only files), *wxDIR* (only directories) or 0 (both).

wxFileSystem::FindNext

wxString FindNext()

Returns the next filename that matches parameters passed to *FindFirst* (p. 544).

wxFileSystem::OpenFile

wxFSFile* OpenFile(const wxString& location)

Opens the file and returns a pointer to a *wxFSFile* (p. 593) object or NULL if failed. It first tries to open the file in relative scope (based on value passed to *ChangePathTo()* method) and then as an absolute path. Note that the user is responsible for deleting the returned *wxFSFile*.

wxFileSystem::URLToFileName

static wxFileName URLToFileName(const wxString& url)

Converts URL into a well-formed filename. The URL must use the `file` protocol.

See also

wxFileSystem::FileNameToURL (p. 544), *wxFileName* (p. 524)

wxFileSystemHandler

Classes derived from *wxFileSystemHandler* are used to access virtual file systems. Its public interface consists of two methods: *CanOpen* (p. 545) and *OpenFile* (p. 547). 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. **Error! Bookmark not defined.**) if you don't know how locations are constructed.

Also consult *list of available handlers* (p. **Error! Bookmark not defined.**).

wxPerl note: In wxPerl, you need to derive your file system handler class from `Wx::PIFileSystemHandler`.

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. **Error! Bookmark not defined.**)

Include files

`<wx/filesys.h>`

See also

`wxFileSystem` (p. 542), `wxFSFile` (p. 593), *Overview* (p. **Error! Bookmark not defined.**)

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. 594) 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. **Error! Bookmark not defined.**). Returns name of the first filename (within filesystem's current path) that matches *wildcard*. *flags* may be one of `wxFILE` (only files), `wxDIR` (only directories) or 0 (both).

This method is only called if `CanOpen` (p. 545) returns true.

wxFileSystemHandler::FindNext

virtual wxString FindNext()

Returns next filename that matches parameters passed to *FindFirst* (p. 544).

This method is only called if *CanOpen* (p. 545) 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. **Error! Bookmark not defined.**) and *wxMimeTypesManager::GetFileTypeFromExtension* (p. **Error! Bookmark not defined.**) 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. **Error! Bookmark not defined.**)

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. 550) 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. **Error! Bookmark not defined.**) 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& mimeTypeypes)

Same as *GetMimeType* (p. 549) 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. **Error! Bookmark not defined.**) function in the first place.

wxFileType::GetIcon

bool GetIcon(wxIconLocation * iconLoc)

If the function returns `true`, the `iconLoc` is filled with the location of the icon for this MIME type. A *wxIcon* (p. 778) may be created from *iconLoc* later.

Windows: The function returns the icon shown by Explorer for the files of the specified type.

Mac: This function is not implemented and always returns `false`.

Unix: MIME manager gathers information about icons from GNOME and KDE settings and thus *GetIcon*'s success depends on availability of these desktop environments.

wxFileType::GetDescription

bool GetDescription(wxString* desc)

If the function returns `true`, the string pointed to by *desc* is filled with a brief description for this file type: for example, "text document" for the "text/plain" MIME type.

wxFileType::GetOpenCommand

bool GetOpenCommand(wxString* command, MessageParameters& params)

wxString GetOpenCommand(const wxString& filename)

With the first version of this method, if the `true` is returned, the string pointed to by *command* is filled with the command which must be executed (see *wxExecute* (p. **Error! Bookmark not defined.**)) in order to open the file of the given type. In this case, the name of the file as well as any other parameters is retrieved from *MessageParameters* (p. 548) class.

In the second case, only the filename is specified and the command to be used to open this kind of file is returned directly. An empty string is returned to indicate that an error occurred (typically meaning that there is no standard way to open this kind of files).

wxFileType::GetPrintCommand

bool GetPrintCommand(wxString* command, MessageParameters& params)

If the function returns `true`, the string pointed to by *command* is filled with the command which must be executed (see *wxExecute* (p. **Error! Bookmark not defined.**)) in order to print the file of the given type. The name of the file is retrieved from *MessageParameters* (p. 548) class.

wxFileType::ExpandCommand

**static wxString ExpandCommand(const wxString& command,
MessageParameters& params)**

This function is primarily intended for GetOpenCommand and GetPrintCommand usage but may be also used by the application directly if, for example, you want to use some non default command to open the file.

The function replaces all occurrences of

format specification	with
%s	the full file name
%t	the MIME type
%{param}	the value of the parameter <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.

wxFilterInputStream

A filter stream has the capability of a normal stream but it can be placed on top of another stream. So, for example, it can uncompress or decrypt the data which are read from another stream and pass it to the requester.

Derived from

wxInputStream (p. 826)
wxStreamBase (p. **Error! Bookmark not defined.**)

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. **Error! Bookmark not defined.**)

wxStreamBase (p. **Error! Bookmark not defined.**)

Include files

<wx/stream.h>

Note

The use of this class is exactly the same as of *wxOutputStream*. Only a constructor differs and it is documented below.

wxFilterOutputStream::wxFilterOutputStream

wxFilterOutputStream(*wxOutputStream*& *stream*)

Initializes a "filter" stream.

wxFindDialogEvent

wxFindReplaceDialog events

Derived from

wxCommandEvent (p. 184)

Include files

<wx/fdreplg.h>

Event table macros

To process a command event from *wxFindReplaceDialog* (p. 556), use these event handler macros to direct input to member functions that take a *wxFindDialogEvent* argument. The *id* parameter is the identifier of the find dialog and you may usually specify -1 for it unless you plan to have several find dialogs sending events to the same owner window simultaneously.

EVT_FIND(id, func)

Find button was pressed in the dialog.

EVT_FIND_NEXT(id, func)	Find next button was pressed in the dialog.
EVT_FIND_REPLACE(id, func)	Replace button was pressed in the dialog.
EVT_FIND_REPLACE_ALL(id, func)	Replace all button was pressed in the dialog.
EVT_FIND_CLOSE(id, func)	The dialog is being destroyed, any pointers to it cannot be used any longer.

wxFindDialogEvent::wxFindDialogEvent

wxFindDialogEvent(wxEventType commandType = wxEVT_NULL, int id = 0)

Constructor used by wxWidgets only.

wxFindDialogEvent::GetFlags

int GetFlags() const

Get the currently selected flags: this is the combination of `wxFR_DOWN`, `wxFR_WHOLEWORD` and `wxFR_MATCHCASE` flags.

wxFindDialogEvent::GetFindString

wxString GetFindString() const

Return the string to find (never empty).

wxFindDialogEvent::GetReplaceString

const wxString& GetReplaceString() const

Return the string to replace the search string with (only for replace and replace all events).

wxFindDialogEvent::GetDialog

wxFindReplaceDialog* GetDialog() const

Return the pointer to the dialog which generated this event.

wxFindReplaceData

`wxFindReplaceData` holds the data for `wxFindReplaceDialog` (p. 556). It is used to initialize the dialog with the default values and will keep the last values from the dialog when it is closed. It is also updated each time a `wxFindDialogEvent` (p. 553) is generated so instead of using the `wxFindDialogEvent` methods you can also directly query this object.

Note that all `SetXXX()` methods may only be called before showing the dialog and calling them has no effect later.

Include files

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

Derived from

wxObject (p. **Error! Bookmark not defined.**)

Data structures

Flags used by *wxFindReplaceData::GetFlags()* (p. 555)
and *wxFindDialogEvent::GetFlags()* (p. 553):

```
enum wxFindReplaceFlags
{
    // downward search/replace selected (otherwise - upwards)
    wxFR_DOWN = 1,

    // whole word search/replace selected
    wxFR_WHOLEWORD = 2,

    // case sensitive search/replace selected (otherwise - case
    insensitive)
    wxFR_MATCHCASE = 4
}
```

These flags can be specified in *wxFindReplaceDialog* ctor (p. 556) or *Create()* (p. 556):

```
enum wxFindReplaceDialogStyles
{
    // replace dialog (otherwise find dialog)
    wxFR_REPLACEDIALOG = 1,

    // don't allow changing the search direction
    wxFR_NOUPDOWN = 2,

    // don't allow case sensitive searching
    wxFR_NOMATCHCASE = 4,

    // don't allow whole word searching
    wxFR_NOWHOLEWORD = 8
}
```

wxFindReplaceData::wxFindReplaceData

wxFindReplaceData(wxUint32 flags = 0)

Constructor initializes the flags to default value (0).

wxFindReplaceData::GetFindString

const wxString& GetFindString()

Get the string to find.

wxFindReplaceData::GetReplaceString

const wxString& GetReplaceString()

Get the replacement string.

wxFindReplaceData::GetFlags

int GetFlags() const

Get the combination of `wxFindReplaceFlags` values.

wxFindReplaceData::SetFlags

void SetFlags(wxUint32 flags)

Set the flags to use to initialize the controls of the dialog.

wxFindReplaceData::SetFindString

void SetFindString(const wxString& str)

Set the string to find (used as initial value by the dialog).

wxFindReplaceData::SetReplaceString

void SetReplaceString(const wxString& str)

Set the replacement string (used as initial value by the dialog).

wxFindReplaceDialog

`wxFindReplaceDialog` is a standard modeless dialog which is used to allow the user to search for some text (and possibly replace it with something else). The actual searching is supposed to be done in the owner window which is the parent of this dialog. Note that it means that unlike for the other standard dialogs this one **must** have a parent window. Also note that there is no way to use this dialog in a modal way; it is always, by design and implementation, modeless.

Please see the dialogs sample for an example of using it.

Include files

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

Derived from

`wxDialog` (p. 412)

wxFindReplaceDialog::wxFindReplaceDialog

wxFindReplaceDialog()

wxFindReplaceDialog(wxWindow * parent, wxFindReplaceData* data, const wxString& title, int style = 0)

After using default constructor *Create()* (p. 556) must be called.

The *parent* and *data* parameters must be non-NULL.

wxFindReplaceDialog::~~wxFindReplaceDialog

~wxFindReplaceDialog()

Destructor.

wxFindReplaceDialog::Create

bool Create(wxWindow * parent, wxFindReplaceData* data, const wxString& title, int style = 0)

Creates the dialog; use *Show* (p. **Error! Bookmark not defined.**) to show it on screen.

The *parent* and *data* parameters must be non-NULL.

wxFindReplaceDialog::GetData

const wxFindReplaceData* GetData() const

Get the *wxFindReplaceData* (p. 554) object used by this dialog.

wxFlexGridSizer

A flex grid sizer is a sizer which lays out its children in a two-dimensional table with all table fields in one row having the same height and all fields in one column having the same width, but all rows or all columns are not necessarily the same height or width as in the *wxGridSizer* (p. 682).

Since wxWidgets 2.5.0, *wxFlexGridSizer* can also size items equally in one direction but unequally ("flexibly") in the other. If the sizer is only flexible in one direction (this can be changed using *SetFlexibleDirection* (p. 559)), it needs to be decided how the sizer should grow in the other ("non flexible") direction in order to fill the available space. The *SetNonFlexibleGrowMode* (p. 559) method serves this purpose.

Derived from

wxGridSizer (p. 682)

wxSizer (p. **Error! Bookmark not defined.**)

wxObject (p. **Error! Bookmark not defined.**)

Include files

<wx/sizer.h>

See also

wxSizer (p. **Error! Bookmark not defined.**), *Sizer overview* (p. **Error! Bookmark not defined.**)

wxFlexGridSizer::wxFlexGridSizer

wxFlexGridSizer(int *rows*, int *cols*, int *vgap*, int *hgap*)

wxFlexGridSizer(int *cols*, int *vgap* = 0, int *hgap* = 0)

Constructor for a *wxGridSizer*. *rows* and *cols* determine the number of columns and rows in the sizer - if either of the parameters is zero, it will be calculated to form the total number of children in the sizer, thus making the sizer grow dynamically. *vgap* and *hgap* define extra space between all children.

wxFlexGridSizer::AddGrowableCol

void AddGrowableCol(size_t *idx*, int *proportion* = 0)

Specifies that column *idx* (starting from zero) should be grown if there is extra space available to the sizer.

The *proportion* parameter has the same meaning as the stretch factor for the *sizers* (p. **Error! Bookmark not defined.**) except that if all proportions are 0, then all columns are resized equally (instead of not being resized at all).

wxFlexGridSizer::AddGrowableRow

void AddGrowableRow(size_t *idx*, int *proportion* = 0)

Specifies that row *idx* (starting from zero) should be grown if there is extra space available to the sizer.

See *AddGrowableCol* (p. 558) for the description of *proportion* parameter.

wxFlexGridSizer::GetFlexibleDirection

int GetFlexibleDirection() const

Returns a *wxOrientation* value that specifies whether the sizer flexibly resizes its columns, rows, or both (default).

Return value

One of the following values:

<code>wxVERTICAL</code>	Rows are flexibly sized.
<code>wxHORIZONTAL</code>	Columns are flexibly sized.
<code>wxBOTH</code>	Both rows and columns are flexibly sized (this is the default value).

See also

SetFlexibleDirection (p. 559)

`wxFlexGridSizer::GetNonFlexibleGrowMode`

`int GetNonFlexibleGrowMode() const`

Returns the value that specifies how the sizer grows in the "non flexible" direction if there is one.

Return value

One of the following values:

<code>wxFLEX_GROWMODE_NONE</code>	Sizer doesn't grow in the non flexible direction.
<code>wxFLEX_GROWMODE_SPECIFIED</code>	Sizer honors growable columns/rows set with <i>AddGrowableCol</i> (p. 558) and <i>AddGrowableRow</i> (p. 558). In this case equal sizing applies to minimum sizes of columns or rows (this is the default value).
<code>wxFLEX_GROWMODE_ALL</code>	Sizer equally stretches all columns or rows in the non flexible direction, whether they are growable or not in the flexible direction.

See also

SetFlexibleDirection (p. 559), *SetNonFlexibleGrowMode* (p. 559)

`wxFlexGridSizer::RemoveGrowableCol`

`void RemoveGrowableCol(size_t idx)`

Specifies that column `idx` is no longer growable.

`wxFlexGridSizer::RemoveGrowableRow`

`void RemoveGrowableRow(size_t idx)`

Specifies that row `idx` is no longer growable.

`wxFlexGridSizer::SetFlexibleDirection`

void SetFlexibleDirection(int direction)

Specifies whether the sizer should flexibly resize its columns, rows, or both. Argument *direction* can be `wxVERTICAL`, `wxHORIZONTAL` or `wxBOTH` (which is the default value). Any other value is ignored. See *GetFlexibleDirection()* (p. 558) for the explanation of these values.

Note that this method does not trigger relayout.

wxFlexGridSizer::SetNonFlexibleGrowMode**void SetNonFlexibleGrowMode(wxFlexSizerGrowMode mode)**

Specifies how the sizer should grow in the non flexible direction if there is one (so *SetFlexibleDirection()* (p. 559) must have been called previously). Argument *mode* can be one of those documented in *GetNonFlexibleGrowMode* (p. 558), please see there for their explanation.

**Note that this method does not trigger
relayout.wxFocusEvent**

A focus event is sent when a window's focus changes. The window losing focus receives a "kill focus" event while the window gaining it gets a "set focus" one.

Notice that the set focus event happens both when the user gives focus to the window (whether using the mouse or keyboard) and when it is done from the program itself using *SetFocus* (p. **Error! Bookmark not defined.**).

Derived from

wxEvent (p. 487)

wxObject (p. **Error! Bookmark not defined.**)

Include files

<wx/event.h>

Event table macros

To process a focus event, use these event handler macros to direct input to a member function that takes a `wxFocusEvent` argument.

EVT_SET_FOCUS(func) Process a `wxEVT_SET_FOCUS` event.

EVT_KILL_FOCUS(func) Process a `wxEVT_KILL_FOCUS` event.

See also

Event handling overview (p. **Error! Bookmark not defined.**)

wxFocusEvent::wxFocusEvent

wxFocusEvent(WXTYPE *eventType* = 0, int *id* = 0)

Constructor.

wxFocusEvent::GetWindow

Returns the window associated with this event, that is the window which had the focus before for the `wxEVT_SET_FOCUS` event and the window which is going to receive focus for the `wxEVT_KILL_FOCUS` one.

Warning: the window pointer may be NULL!

wxFont

A font is an object which determines the appearance of text. Fonts are used for drawing text to a device context, and setting the appearance of a window's text.

You can retrieve the current system font settings with `wxSystemSettings` (p. **Error! Bookmark not defined.**).

`wxSystemSettings` (p. **Error! Bookmark not defined.**)

Derived from

`wxGDIObject` (p. 609)

`wxObject` (p. **Error! Bookmark not defined.**)

Include files

<wx/font.h>

Constants

The possible values for the *family* parameter of `wxFont` constructor (p. 563) are (the old names are for compatibility only):

```
enum wxFontFamily
{
    wxFONTFAMILY_DEFAULT = wxDEFAULT,
    wxFONTFAMILY_DECORATIVE = wxDECORATIVE,
    wxFONTFAMILY_ROMAN = wxROMAN,
    wxFONTFAMILY_SCRIPT = wxSCRIPT,
    wxFONTFAMILY_SWISS = wxSWISS,
    wxFONTFAMILY_MODERN = wxMODERN,
    wxFONTFAMILY_TELETYPE = wxTELETYPE,
    wxFONTFAMILY_MAX
};
```

The possible values for the *weight* parameter are (the old names are for compatibility only):

```
enum wxFontWeight
{
    wxFONTWEIGHT_NORMAL = wxNORMAL,
    wxFONTWEIGHT_LIGHT = wxLIGHT,
```

```

    wxFONTWEIGHT_BOLD = wxBOLD,
    wxFONTWEIGHT_MAX
};

```

The font flags which can be used during the font creation are:

```

enum
{
    // no special flags: font with default weight/slant/anti-
    aliasing
    wxFONTFLAG_DEFAULT          = 0,

    // slant flags (default: no slant)
    wxFONTFLAG_ITALIC           = 1 << 0,
    wxFONTFLAG_SLANT            = 1 << 1,

    // weight flags (default: medium)
    wxFONTFLAG_LIGHT            = 1 << 2,
    wxFONTFLAG_BOLD             = 1 << 3,

    // anti-aliasing flag: force on or off (default: the current
    system default)
    wxFONTFLAG_ANTIALIASED      = 1 << 4,
    wxFONTFLAG_NOT_ANTIALIASED  = 1 << 5,

    // underlined/strikethrough flags (default: no lines)
    wxFONTFLAG_UNDERLINED       = 1 << 6,
    wxFONTFLAG_STRIKETHROUGH    = 1 << 7,
};

```

The known font encodings are:

```

enum wxFontEncoding
{
    wxFONTENCODING_SYSTEM = -1,        // system default
    wxFONTENCODING_DEFAULT,             // current default encoding

    // ISO8859 standard defines a number of single-byte charsets
    wxFONTENCODING_ISO8859_1,           // West European (Latin1)
    wxFONTENCODING_ISO8859_2,           // Central and East European
    (Latin2)
    wxFONTENCODING_ISO8859_3,           // Esperanto (Latin3)
    wxFONTENCODING_ISO8859_4,           // Baltic (old) (Latin4)
    wxFONTENCODING_ISO8859_5,           // Cyrillic
    wxFONTENCODING_ISO8859_6,           // Arabic
    wxFONTENCODING_ISO8859_7,           // Greek
    wxFONTENCODING_ISO8859_8,           // Hebrew
    wxFONTENCODING_ISO8859_9,           // Turkish (Latin5)
    wxFONTENCODING_ISO8859_10,          // Variation of Latin4
    (Latin6)
    wxFONTENCODING_ISO8859_11,           // Thai
    wxFONTENCODING_ISO8859_12,           // doesn't exist currently,
    but put it
    // here anyhow to make all
    ISO8859
    // consecutive numbers
    wxFONTENCODING_ISO8859_13,           // Baltic (Latin7)
    wxFONTENCODING_ISO8859_14,           // Latin8
    wxFONTENCODING_ISO8859_15,           // Latin9 (a.k.a. Latin0,
    includes euro)
    wxFONTENCODING_ISO8859_MAX,

    // Cyrillic charset soup (see

```

```

http://czyborra.com/charsets/cyrillic.html)
wxFONTENCODING_KOI8,           // we don't support any of
KOI8 variants
    wxFONTENCODING_ALTERNATIVE, // same as MS-DOS CP866
    wxFONTENCODING_BULGARIAN,   // used under Linux in
Bulgaria

    // what would we do without Microsoft? They have their own
encodings
    // for DOS
    wxFONTENCODING_CP437,       // original MS-DOS codepage
    wxFONTENCODING_CP850,       // CP437 merged with Latin1
    wxFONTENCODING_CP852,       // CP437 merged with Latin2
    wxFONTENCODING_CP855,       // another cyrillic encoding
    wxFONTENCODING_CP866,       // and another one
    // and for Windows
    wxFONTENCODING_CP874,       // WinThai
    wxFONTENCODING_CP1250,      // WinLatin2
    wxFONTENCODING_CP1251,      // WinCyrillic
    wxFONTENCODING_CP1252,      // WinLatin1
    wxFONTENCODING_CP1253,      // WinGreek (8859-7)
    wxFONTENCODING_CP1254,      // WinTurkish
    wxFONTENCODING_CP1255,      // WinHebrew
    wxFONTENCODING_CP1256,      // WinArabic
    wxFONTENCODING_CP1257,      // WinBaltic (same as Latin 7)
    wxFONTENCODING_CP12_MAX,

    wxFONTENCODING_UTF7,        // UTF-7 Unicode encoding
    wxFONTENCODING_UTF8,        // UTF-8 Unicode encoding

    wxFONTENCODING_UNICODE,     // Unicode - currently used
only by

                                // wxEncodingConverter class

    wxFONTENCODING_MAX
};

```

Predefined objects

Objects:

wxNullFont

Pointers:

wxNORMAL_FONT

wxSMALL_FONT

wxITALIC_FONT

wxSWISS_FONT

See also

wxFont overview (p. **Error! Bookmark not defined.**), *wxDC::SetFont* (p. 389),
wxDC::DrawText (p. 380), *wxDC::GetTextExtent* (p. 384), *wxFontDialog* (p. 574),
wxSystemSettings (p. **Error! Bookmark not defined.**)

wxFont::wxFont

wxFont()

Default constructor.

```
wxFont(int pointSize, wxFontFamily family, int style, wxFontWeight weight, const  
bool underline = false, const wxString& faceName = "", wxFontEncoding encoding =  
wxFONTENCODING_DEFAULT)
```

```
wxFont(const wxSize& pixelSize, wxFontFamily family, int style, wxFontWeight  
weight, const bool underline = false, const wxString& faceName = "",  
wxFontEncoding encoding = wxFONTENCODING_DEFAULT)
```

Creates a font object with the specified attributes.

Parameters

pointSize

Size in points.

pixelSize

Size in pixels: this is directly supported only under MSW currently where this constructor can be used directly, under other platforms a font with the closest size to the given one is found using binary search and the static *New* (p. 567) method must be used.

family

Font family, a generic way of referring to fonts without specifying actual facename. One of:

wxFONTFAMILY_DEFAULT Chooses a default font.

wxFONTFAMILY_DECORATIVE A decorative font.

wxFONTFAMILY_ROMAN A formal, serif font.

wxFONTFAMILY_SCRIPT A handwriting font.

wxFONTFAMILY_SWISS A sans-serif font.

wxFONTFAMILY_MODERN A fixed pitch font.

wxFONTFAMILY_TELETYPE A teletype font.

style

One of **wxFONTSTYLE_NORMAL**, **wxFONTSTYLE_SLANT** and **wxFONTSTYLE_ITALIC**.

weight

Font weight, sometimes also referred to as font boldness. One of:

wxFONTWEIGHT_NORMAL Normal font.

wxFONTWEIGHT_LIGHT Light font.

wxFONTWEIGHT_BOLD Bold font.

underline

The value can be true or false. At present this has an effect on Windows and Motif 2.x only.

faceName

An optional string specifying the actual typeface to be used. If it is an empty string, a default typeface will be chosen based on the family.

encoding

An encoding which may be one of **wxFONTENCODING_SYSTEM** Default system encoding.

wxFONTENCODING_DEFAULT Default application encoding: this is the encoding set by calls to *SetDefaultEncoding* (p. 568) and which may be set to, say, KOI8 to create all fonts by default with KOI8 encoding. Initially, the default application encoding is the same as default system encoding.

wxFONTENCODING_ISO8859_1...15 ISO8859 encodings.

wxFONTENCODING_KOI8 The standard Russian encoding for Internet.

wxFONTENCODING_CP1250...1252 Windows encodings similar to ISO8859 (but not identical).

If the specified encoding isn't available, no font is created (see also *font encoding overview* (p. **Error! Bookmark not defined.**)).

Remarks

If the desired font does not exist, the closest match will be chosen. Under Windows, only scalable TrueType fonts are used.

See also *wxDC::SetFont* (p. 389), *wxDC::DrawText* (p. 380) and *wxDC::GetTextExtent* (p. 384).

wxFont::~~wxFont

~wxFont()

Destructor.

Remarks

The destructor may not delete the underlying font object of the native windowing system, since `wxFont` uses a reference counting system for efficiency.

Although all remaining fonts are deleted when the application exits, the application should try to clean up all fonts itself. This is because `wxWidgets` cannot know if a pointer to the font object is stored in an application data structure, and there is a risk of double deletion.

wxFont::IsFixedWidth

bool IsFixedWidth() const

Returns `true` if the font is a fixed width (or monospaced) font, `false` if it is a proportional one or font is invalid.

wxFont::GetDefaultEncoding

static wxFontEncoding GetDefaultEncoding()

Returns the current application's default encoding.

See also

Font encoding overview (p. **Error! Bookmark not defined.**), *SetDefaultEncoding* (p. 568)

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. 568)

wxFont::GetFamily

wxFontFamily GetFamily() const

Gets the font family. See *wxFont::SetFamily* (p. 569) for a list of valid family identifiers.

See also

wxFont::SetFamily (p. 569)

wxFont::GetNativeFontInfoDesc

wxString GetNativeFontInfoDesc() const

Returns the platform-dependent string completely describing this font or an empty string

if the font wasn't constructed using the native font description.

See also

wxFont::SetNativeFontInfo (p. 569)

wxFont::GetPointSize

int GetPointSize() const

Gets the point size.

See also

wxFont::SetPointSize (p. 569)

wxFont::GetStyle

int GetStyle() const

Gets the font style. See *wxFont::wxFont* (p. 563) for a list of valid styles.

See also

wxFont::SetStyle (p. 570)

wxFont::GetUnderlined

bool GetUnderlined() const

Returns true if the font is underlined, false otherwise.

See also

wxFont::SetUnderlined (p. 570)

wxFont::GetWeight

wxFontWeight GetWeight() const

Gets the font weight. See *wxFont::wxFont* (p. 563) for a list of valid weight identifiers.

See also

wxFont::SetWeight (p. 570)

wxFont::New

static wxFont * New(int pointSize, wxFontFamily family, int style, wxFontWeight weight, const bool underline = false, const wxString& faceName = "", wxFontEncoding encoding = wxFONTENCODING_DEFAULT)

```
static wxFont * New(int pointSize, wxFontFamily family, int flags =  
wxFONTFLAG_DEFAULT, const wxString& faceName = "", wxFontEncoding encoding  
= wxFONTENCODING_DEFAULT)
```

```
static wxFont * New(const wxSize& pixelSize, wxFontFamily family, int style,  
wxFontWeight weight, const bool underline = false, const wxString& faceName = "",  
wxFontEncoding encoding = wxFONTENCODING_DEFAULT)
```

```
static wxFont * New(const wxSize& pixelSize, wxFontFamily family, int flags =  
wxFONTFLAG_DEFAULT, const wxString& faceName = "", wxFontEncoding encoding  
= wxFONTENCODING_DEFAULT)
```

These functions take the same parameters as *wxFont* constructor (p. 563) and return a new font object allocated on the heap.

Using `New()` is currently the only way to directly create a font with the given size in pixels on platforms other than wxMSW.

wxFont::Ok

```
bool Ok() const
```

Returns `true` if this object is a valid font, `false` otherwise.

wxFont::SetDefaultEncoding

```
static void SetDefaultEncoding(wxFontEncoding encoding)
```

Sets the default font encoding.

See also

Font encoding overview (p. **Error! Bookmark not defined.**), *GetDefaultEncoding* (p. 566)

wxFont::SetFaceName

```
void SetFaceName(const wxString& faceName)
```

Sets the facename for the font.

Parameters

faceName

A valid facename, which should be on the end-user's system.

Remarks

To avoid portability problems, don't rely on a specific face, but specify the font family instead or as well. A suitable font will be found on the end-user's system. If both the family and the facename are specified, wxWidgets will first search for the specific face, and then for a font belonging to the same family.

See also

wxFont::GetFaceName (p. 566), *wxFont::SetFamily* (p. 569)

wxFont::SetFamily

void SetFamily(wxFontFamily *family*)

Sets the font family.

Parameters

family

One of:

wxFONTFAMILY_DEFAULT Chooses a default font.

wxFONTFAMILY_DECORATIVE A decorative font.

wxFONTFAMILY_ROMAN A formal, serif font.

wxFONTFAMILY_SCRIPT A handwriting font.

wxFONTFAMILY_SWISS A sans-serif font.

wxFONTFAMILY_MODERN A fixed pitch font.

wxFONTFAMILY_TELETYPE A teletype font.

See also

wxFont::GetFamily (p. 566), *wxFont::SetFaceName* (p. 568)

wxFont::SetNativeFontInfo

void SetNativeFontInfo(const wxString& *info*)

Creates the font corresponding to the given native font description string which must have been previously returned by *GetNativeFontInfoDesc* (p. 566). If the string is invalid, font is unchanged.

wxFont::SetPointSize

void SetPointSize(int *pointSize*)

Sets the point size.

Parameters

pointSize

Size in points.

See also

wxFont::GetPointSize (p. 567)

wxFont::SetStyle

void SetStyle(int *style*)

Sets the font style.

Parameters

style

One of **wxFONTSTYLE_NORMAL**, **wxFONTSTYLE_SLANT** and **wxFONTSTYLE_ITALIC**.

See also

wxFont::GetStyle (p. 567)

wxFont::SetUnderlined

void SetUnderlined(const bool *underlined*)

Sets underlining.

Parameters

underlining

true to underline, false otherwise.

See also

wxFont::GetUnderlined (p. 567)

wxFont::SetWeight

void SetWeight(wxFontWeight *weight*)

Sets the font weight.

Parameters

weight

One of:

wxFONTWEIGHT_NORMAL Normal font.

wxFONTWEIGHT_LIGHT Light font.

wxFONTWEIGHT_BOLD Bold font.

See also

wxFont::GetWeight (p. 567)

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. **Error! Bookmark not defined.**)

This class holds a variety of information related to font dialogs.

Derived from

wxObject (p. **Error! Bookmark not defined.**)

Include files

<wx/cmndata.h>

See also

Overview (p. **Error! Bookmark not defined.**), *wxFont* (p. 561), *wxFontDialog* (p. 574)

wxFontData::wxFontData

wxFontData()

Constructor. Initializes *fontColour* to black, *showHelp* to black, *allowSymbols* to true,

enableEffects to true, *minSize* to 0 and *maxSize* to 0.

wxFontData::EnableEffects

void EnableEffects(bool *enable*)

Enables or disables 'effects' under MS Windows or generic only. This refers to the controls for manipulating colour, strikeout and underline properties.

The default value is true.

wxFontData::GetAllowSymbols

bool GetAllowSymbols()

Under MS Windows, returns a flag determining whether symbol fonts can be selected. Has no effect on other platforms.

The default value is true.

wxFontData::GetColour

wxColour& GetColour()

Gets the colour associated with the font dialog.

The default value is black.

wxFontData::GetChosenFont

wxFont GetChosenFont()

Gets the font chosen by the user if the user pressed OK (`wxFontDialog::ShowModal` returned `wxID_OK`).

wxFontData::GetEnableEffects

bool GetEnableEffects()

Determines whether 'effects' are enabled under Windows. This refers to the controls for manipulating colour, strikeout and underline properties.

The default value is true.

wxFontData::GetInitialFont

wxFont GetInitialFont()

Gets the font that will be initially used by the font dialog. This should have previously been set by the application.

wxFontData::GetShowHelp**bool GetShowHelp()**

Returns true if the Help button will be shown (Windows only).

The default value is false.

wxFontData::SetAllowSymbols**void SetAllowSymbols(bool allowSymbols)**

Under MS Windows, determines whether symbol fonts can be selected. Has no effect on other platforms.

The default value is true.

wxFontData::SetChosenFont**void SetChosenFont(const wxFont& font)**

Sets the font that will be returned to the user (for internal use only).

wxFontData::SetColour**void SetColour(const wxColour& colour)**

Sets the colour that will be used for the font foreground colour.

The default colour is black.

wxFontData::SetInitialFont**void SetInitialFont(const wxFont& font)**

Sets the font that will be initially used by the font dialog.

wxFontData::SetRange**void SetRange(int min, int max)**

Sets the valid range for the font point size (Windows only).

The default is 0, 0 (unrestricted range).

wxFontData::SetShowHelp**void SetShowHelp(bool showHelp)**

Determines whether the Help button will be displayed in the font dialog (Windows only).

The default value is `false`.

wxFontData::operator =

void operator =(const wxFontData& data)

Assignment operator for the font data.

wxFontDialog

This class represents the font chooser dialog.

Derived from

wxDialog (p. 412)

wxWindow (p. **Error! Bookmark not defined.**)

wxEvtHandler (p. 490)

wxObject (p. **Error! Bookmark not defined.**)

Include files

<wx/fontdlg.h>

See also

Overview (p. **Error! Bookmark not defined.**),

wxFontData (p. 571),

wxGetFontFromUser (p. **Error! Bookmark not defined.**)

wxFontDialog::wxFontDialog

wxFontDialog()

wxFontDialog(wxWindow* parent)

wxFontDialog(wxWindow* parent, const wxFontData& data)

Constructor. Pass a parent window, and optionally the *font data* (p. 571) object to be used to initialize the dialog controls. If the default constructor is used, *Create()* (p. 574) must be called before the dialog can be shown.

wxFontDialog::Create

bool Create(wxWindow* parent)

bool Create(wxWindow* parent, const wxFontData& data)

Creates the dialog if the *wxFontDialog* object had been initialized using the default constructor. Returns `true` on success and `false` if an error occurred.

wxFontDialog::GetFontData**const wxFontData& GetFontData() const****wxFontData& GetFontData()**

Returns the *font data* (p. 571) associated with the font dialog.

wxFontDialog::ShowModal**int ShowModal()**

Shows the dialog, returning `wxID_OK` if the user pressed Ok, and `wxID_CANCEL` otherwise.

If the user cancels the dialog (`ShowModal` returns `wxID_CANCEL`), no font will be created. If the user presses OK, a new `wxFont` will be created and stored in the font dialog's `wxFontData` structure.

wxFontEnumerator

`wxFontEnumerator` enumerates either all available fonts on the system or only the ones with given attributes - either only fixed-width (suited for use in programs such as terminal emulators and the like) or the fonts available in the given *encoding* (p. **Error! Bookmark not defined.**).

To do this, you just have to call one of `EnumerateXXX()` functions - either *EnumerateFacenames* (p. 576) or *EnumerateEncodings* (p. 576) and the corresponding callback (*OnFacename* (p. 577) or *OnFontEncoding* (p. 577)) 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. 577) or *OnFontEncoding* (p. 577) should be overridden depending on whether you plan to call *EnumerateFacenames* (p. 576) or *EnumerateEncodings* (p. 576). 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. **Error! Bookmark not defined.**), *Font sample* (p. **Error! Bookmark not defined.**), *wxFont* (p. 561), *wxFontMapper* (p. 578)

wxFontEnumerator::EnumerateFacenames

virtual bool EnumerateFacenames(wxFontEncoding encoding = wxFONTENCODING_SYSTEM, bool fixedWidthOnly = false)

Call *OnFacename* (p. 577) 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. 577) 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. 576). This is convenience function. It is based on default implementation of *OnFontEncoding* (p. 577) 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. 576). This is convenience function. It is based on default implementation of *OnFacename* (p. 577) so don't expect it to work if you overwrite that method.

wxFontEnumerator::OnFacename

virtual bool OnFacename(const wxString& font)

Called by *EnumerateFacenames* (p. 576) 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. 576) 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. 851)

wxObject (p. **Error! Bookmark not defined.**)

Include files

<wx/gdicmn.h>

See also

wxFont (p. 561)

wxFontList::wxFontList

wxFontList()

Constructor. The application should not construct its own font list: use the object pointer **wxTheFontList**.

wxFontList::AddFont

void AddFont(wxFont *font)

Used by wxWidgets to add a font to the list, called in the font constructor.

wxFontList::FindOrCreateFont

wxFont * FindOrCreateFont(int point_size, int family, int style, int weight, bool underline = false, const wxString& facename = NULL, wxFontEncoding encoding = wxFONTENCODING_DEFAULT)

Finds a font of the given specification, or creates one and adds it to the list. See the *wxFont constructor* (p. 563) for details of the arguments.

wxFontList::RemoveFont

void RemoveFont(wxFont *font)

Used by wxWidgets to remove a font from the list.

wxFontMapper

wxFontMapper manages user-definable correspondence between logical font names and the fonts present on the machine.

The default implementations of all functions will ask the user if they are not capable of finding the answer themselves and store the answer in a config file (configurable via SetConfigXXX functions). This behaviour may be disabled by giving the value of false to "interactive" parameter.

However, the functions will always consult the config file to allow the user-defined values override the default logic and there is no way to disable this - which shouldn't be ever needed because if "interactive" was never true, the config file is never created anyhow.

In case everything else fails (i.e. there is no record in config file and "interactive" is false or user denied to choose any replacement), the class queries *wxEncodingConverter* (p. 482) for "equivalent" encodings (e.g. iso8859-2 and cp1250) and tries them.

Using wxFontMapper in conjunction with wxMBConv classes

If you need to display text in encoding which is not available at host system (see *IsEncodingAvailable* (p. 581)), you may use these two classes to find font in some similar encoding (see *GetAltForEncoding* (p. 580)) and convert the text to this encoding (*wxMBConv* classes (p. **Error! Bookmark not defined.**)).

Following code snippet demonstrates it:

```
if (!wxFontMapper::Get()->IsEncodingAvailable(enc, facename))
{
    wxFontEncoding alternative;
    if (wxFontMapper::Get()->GetAltForEncoding(enc, &alternative,
                                                facename, false))
    {
        wxCSConv convFrom(wxFontMapper::Get()-
>GetEncodingName(enc));
        wxCSConv convTo(wxFontMapper::Get()-
>GetEncodingName(alternative));
        text = wxString(text.mb_str(convFrom), convTo);
    }
    else
        ...failure (or we may try iso8859-1/7bit ASCII)...
}
...display text...
```

Derived from

No base class

Include files

<wx/fontmap.h>

See also

wxEncodingConverter (p. 482), *Writing non-English applications* (p. **Error! Bookmark not defined.**)

wxFontMapper::wxFontMapper**wxFontMapper()**

Default ctor.

Note

The preferred way of creating a `wxFontMapper` instance is to call `wxFontMapper::Get` (p. 580).

wxFontMapper::~~wxFontMapper**~wxFontMapper()**

Virtual dtor for a base class.

wxFontMapper::CharsetToEncoding**wxFontEncoding CharsetToEncoding(const wxString& charset, bool interactive = true)**

Returns the encoding for the given charset (in the form of RFC 2046) or `wxFONTENCODING_SYSTEM` if couldn't decode it.

Be careful when using this function with *interactive* set to `true` (default value) as the function then may show a dialog box to the user which may lead to unexpected reentrancies and may also take a significantly longer time than a simple function call. For these reasons, it is almost always a bad idea to call this function from the event handlers for repeatedly generated events such as `EVT_PAINT`.

wxFontMapper::Get**static wxFontMapper * Get()**

Get the current font mapper object. If there is no current object, creates one.

See also*wxFontMapper::Set* (p. 582)**wxFontMapper::GetAllEncodingNames****static const wxChar** GetAllEncodingNames(wxFontEncoding encoding)**

Returns the array of all possible names for the given encoding. The array is `NULL`-terminated. IF it isn't empty, the first name in it is the canonical encoding name, i.e. the same string as returned by *GetEncodingName*() (p. 581).

wxFontMapper::GetAltForEncoding

bool GetAltForEncoding(wxFontEncoding encoding, wxNativeEncodingInfo* info, const wxString& facename = wxEmptyString, bool interactive = true)

bool GetAltForEncoding(wxFontEncoding encoding, wxFontEncoding* alt_encoding, const wxString& facename = wxEmptyString, bool interactive = true)

Find an alternative for the given encoding (which is supposed to not be available on this system). If successful, return true and fill info structure with the parameters required to create the font, otherwise return false.

The first form is for wxWidgets' internal use while the second one is better suitable for general use -- it returns wxFontEncoding which can consequently be passed to wxFont constructor.

wxFontMapper::GetEncoding

static wxFontEncoding GetEncoding(size_t n)

Returns the *n*-th supported encoding. Together with *GetSupportedEncodingsCount()* (p. 581) this method may be used to get all supported encodings.

wxFontMapper::GetEncodingDescription

static wxString GetEncodingDescription(wxFontEncoding encoding)

Return user-readable string describing the given encoding.

wxFontMapper::GetEncodingFromName

static wxFontEncoding GetEncodingFromName(const wxString& encoding)

Return the encoding corresponding to the given internal name. This function is the inverse of *GetEncodingName* (p. 581) and is intentionally less general than *CharsetToEncoding* (p. 579), i.e. it doesn't try to make any guesses nor ever asks the user. It is meant just as a way of restoring objects previously serialized using *GetEncodingName* (p. 581).

wxFontMapper::GetEncodingName

static wxString GetEncodingName(wxFontEncoding encoding)

Return internal string identifier for the encoding (see also *GetEncodingDescription()* (p. 580))

See also

GetEncodingFromName (p. 581)

wxFontMapper::GetSupportedEncodingsCount

static size_t GetSupportedEncodingsCount()

Returns the number of the font encodings supported by this class. Together with *GetEncoding* (p. 580) this method may be used to get all supported encodings.

wxFontMapper::IsEncodingAvailable

bool IsEncodingAvailable(wxFontEncoding encoding, const wxString& facename = wxEmptyString)

Check whether given encoding is available in given face or not. If no facename is given, find *any* font in this encoding.

wxFontMapper::SetDialogParent

void SetDialogParent(wxWindow* parent)

The parent window for modal dialogs.

wxFontMapper::SetDialogTitle

void SetDialogTitle(const wxString& title)

The title for the dialogs (note that default is quite reasonable).

wxFontMapper::Set

static wxFontMapper * Set(wxFontMapper *mapper)

Set the current font mapper object and return previous one (may be NULL). This method is only useful if you want to plug-in an alternative font mapper into wxWidgets.

See also

wxFontMapper::Get (p. 580)

wxFontMapper::SetConfig

void SetConfig(wxConfigBase* config)

Set the config object to use (may be NULL to use default).

By default, the global one (from *wxConfigBase::Get()* will be used) and the default root path for the config settings is the string returned by *GetDefaultConfigPath()*.

wxFontMapper::SetConfigPath

void SetConfigPath(const wxString& prefix)

Set the root config path to use (should be an absolute path).

wxFrame

A frame is a window whose size and position can (usually) be changed by the user. It usually has thick borders and a title bar, and can optionally contain a menu bar, toolbar and status bar. A frame can contain any window that is not a frame or dialog.

A frame that has a status bar and toolbar created via the `CreateStatusBar/CreateToolBar` functions manages these windows, and adjusts the value returned by `GetClientSize` to reflect the remaining size available to application windows.

Derived from

wxTopLevelWindow (p. **Error! Bookmark not defined.**)

wxWindow (p. **Error! Bookmark not defined.**)

wxEvtHandler (p. 490)

wxObject (p. **Error! Bookmark not defined.**)

Include files

<wx/frame.h>

Window styles

wxDEFAULT_FRAME_STYLE	Defined as wxMINIMIZE_BOX wxMAXIMIZE_BOX wxRESIZE_BORDER wxSYSTEM_MENU wxCAPTION wxCLOSE_BOX wxCLIP_CHILDREN.
wxICONIZE	Display the frame iconized (minimized). Windows only.
wxCAPTION	Puts a caption on the frame.
wxMINIMIZE	Identical to wxICONIZE . Windows only.
wxMINIMIZE_BOX	Displays a minimize box on the frame.
wxMAXIMIZE	Displays the frame maximized. Windows only.
wxMAXIMIZE_BOX	Displays a maximize box on the frame.
wxCLOSE_BOX	Displays a close box on the frame.
wxSTAY_ON_TOP	Stay on top of all other windows, see also wxFRAME_FLOAT_ON_PARENT .
wxSYSTEM_MENU	Displays a system menu.
wxRESIZE_BORDER	Displays a resizable border around the window.
wxFRAME_TOOL_WINDOW	Causes a frame with a small titlebar to be created; the frame does not appear in the taskbar under Windows or GTK+.

- wxFRAME_NO_TASKBAR** Creates an otherwise normal frame but it does not appear in the taskbar under Windows or GTK+ (note that it will minimize to the desktop window under Windows which may seem strange to the users and thus it might be better to use this style only without `wxMINIMIZE_BOX` style). In `wxGTK`, the flag is respected only if GTK+ is at least version 2.2 and the window manager supports `_NET_WM_STATE_SKIP_TASKBAR` (<http://freedesktop.org/Standards/wm-spec/1.3/ar01s05.html>) hint. Has no effect under other platforms.
- wxFRAME_FLOAT_ON_PARENT** The frame will always be on top of its parent (unlike `wxSTAY_ON_TOP`). A frame created with this style must have a non-NULL parent.
- wxFRAME_EX_CONTEXTHELP** Under Windows, puts a query button on the caption. When pressed, Windows will go into a context-sensitive help mode and `wxWidgets` will send a `wxEVT_HELP` event if the user clicked on an application window. *Note* that this is an extended style and must be set by calling `SetExtraStyle` (p. **Error! Bookmark not defined.**) before `Create` is called (two-step construction). You cannot use this style together with `wxMAXIMIZE_BOX` or `wxMINIMIZE_BOX`, so you should use `wxDEFAULT_FRAME_STYLE & ~ (wxMINIMIZE_BOX | wxMAXIMIZE_BOX)` for the frames having this style (the dialogs don't have a minimize or a maximize box by default)
- wxFRAME_SHAPED** Windows with this style are allowed to have their shape changed with the `SetShape` (p. **Error! Bookmark not defined.**) method.
- wxFRAME_EX_METAL** On Mac OS X, frames with this style will be shown with a metallic look. This is an *extra* style.

The default frame style is for normal, 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_BORDER | wxRESIZE_BOX | wxMAXIMIZE_BOX)`. See also *window styles overview* (p. **Error! Bookmark not defined.**).

Default event processing

`wxFrame` processes the following events:

`wxEVT_SIZE` (p. **Error! Bookmark not defined.**) If the frame has exactly one child window, not counting the status and toolbar, this child is resized to take the entire frame client area. If two or more windows are present, they should be laid out explicitly either by manually handling `wxEVT_SIZE` or using `sizers` (p. **Error! Bookmark not defined.**)

`wxEVT_MENU_HIGHLIGHT` (p. **Error! Bookmark not defined.**) The default implementation displays the *help string* (p. **Error! Bookmark not defined.**) associated with the selected item in the first pane of the status bar, if there is one.

Remarks

An application should normally define an `wxCloseEvent` (p. 157) 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. **Error! Bookmark not defined.**), `wxMDIChildFrame` (p. **Error! Bookmark not defined.**), `wxMiniFrame` (p. **Error! Bookmark not defined.**), `wxDialog` (p. 412)

wxFrame::wxFrame

wxFrame()

Default constructor.

wxFrame(wxWindow* parent, wxWindowID id, const wxString& title, const wxPoint& pos = wxDefaultPosition, const wxSize& size = wxDefaultSize, long style = wxDEFAULT_FRAME_STYLE, const wxString& name = "frame")

Constructor, creating the window.

Parameters

parent

The window parent. This may be NULL. If it is non-NULL, the frame will always be displayed on top of the parent window on Windows.

id

The window identifier. It may take a value of -1 to indicate a default value.

title

The caption to be displayed on the frame's title bar.

pos

The window position. A value of (-1, -1) indicates a default position, chosen by either the windowing system or wxWidgets, depending on platform.

size

The window size. A value of (-1, -1) indicates a default size, chosen by either the

windowing system or wxWidgets, depending on platform.

style

The window style. See *wxFrame* (p. 582).

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. 586)

wxFrame::~~wxFrame

void ~wxFrame()

Destructor. Destroys all child windows and menu bar if present.

wxFrame::Centre

void Centre(int direction = wxBOTH)

Centres the frame on the display.

Parameters

direction

The parameter may be `wxHORIZONTAL`, `wxVERTICAL` or `wxBOTH`.

wxFrame::Create

bool Create(wxWindow* parent, wxWindowID id, const wxString& title, const wxPoint& pos = wxDefaultPosition, const wxSize& size = wxDefaultSize, long style = wxDEFAULT_FRAME_STYLE, const wxString& name = "frame")

Used in two-step frame construction. See *wxFrame::wxFrame* (p. 584) 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. **Error! Bookmark not defined.**) for a list of valid styles.

id

The status bar window identifier. If -1, an identifier will be chosen by wxWidgets.

name

The status bar window name.

Return value

A pointer to the status bar if it was created successfully, NULL otherwise.

Remarks

The width of the status bar is the whole width of the frame (adjusted automatically when resizing), and the height and text size are chosen by the host windowing system.

By default, the status bar is an instance of *wxStatusBar*. To use a different class, override *wxFrame::OnCreateStatusBar* (p. 589).

Note that you can put controls and other windows on the status bar if you wish.

See also

wxFrame::SetStatusText (p. 592), *wxFrame::OnCreateStatusBar* (p. 589),
wxFrame::GetStatusBar (p. 588)

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. **Error! Bookmark not defined.**) for a list of valid styles.

id

The toolbar window identifier. If -1, an identifier will be chosen by wxWidgets.

name

The toolbar window name.

Return value

A pointer to the toolbar if it was created successfully, NULL otherwise.

Remarks

By default, the toolbar is an instance of wxToolBar (which is defined to be a suitable toolbar class on each platform, such as wxToolBar95). To use a different class, override *wxFrame::OnCreateToolBar* (p. 590).

When a toolbar has been created with this function, or made known to the frame with *wxFrame::SetToolBar* (p. 593), the frame will manage the toolbar position and adjust the return value from *wxWindow::GetClientSize* (p. **Error! Bookmark not defined.**) to reflect the available space for application windows.

Under Pocket PC, you should *always* use this function for creating the toolbar to be managed by the frame, so that wxWidgets can use a combined menubar and toolbar. Where you manage your own toolbars, create a wxToolBar as usual.

See also

wxFrame::CreateStatusBar (p. 586), *wxFrame::OnCreateToolBar* (p. 590), *wxFrame::SetToolBar* (p. 593), *wxFrame::GetToolBar* (p. 589)

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. 591), *wxMenuBar* (p. **Error! Bookmark not defined.**), *wxMenu* (p. **Error! Bookmark not defined.**)

wxFrame::GetStatusBar

wxStatusBar* GetStatusBar() const

Returns a pointer to the status bar currently associated with the frame (if any).

See also

wxFrame::CreateStatusBar (p. 586), *wxStatusBar* (p. **Error! Bookmark not defined.**)

wxFrame::GetStatusBarPane

int GetStatusBarPane()

Returns the status bar pane used to display menu and toolbar help.

See also

wxFrame::SetStatusBarPane (p. 592)

wxFrame::GetToolBar

wxToolBar* GetToolBar() const

Returns a pointer to the toolbar currently associated with the frame (if any).

See also

wxFrame::CreateToolBar (p. 587), *wxToolBar* (p. **Error! Bookmark not defined.**),
wxFrame::SetToolBar (p. 593)

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. 586).

Parameters

number

The number of fields to create.

style

The window style. See *wxStatusBar* (p. **Error! Bookmark not defined.**) for a list of valid styles.

id

The window identifier. If -1, an identifier will be chosen by wxWidgets.

name

The window name.

Return value

A status bar object.

Remarks

An application can override this function to return a different kind of status bar. The default implementation returns an instance of *wxStatusBar* (p. **Error! Bookmark not defined.**).

See also

wxFrame::CreateStatusBar (p. 586), *wxStatusBar* (p. **Error! Bookmark not defined.**).

wxFrame::OnCreateToolBar

virtual wxToolBar* OnCreateToolBar(long style, wxWindowID id, const wxString& name)

Virtual function called when a toolbar is requested by *wxFrame::CreateToolBar* (p. 587).

Parameters

style

The toolbar style. See *wxToolBar* (p. **Error! Bookmark not defined.**) for a list of valid styles.

id

The toolbar window identifier. If -1, an identifier will be chosen by wxWidgets.

name

The toolbar window name.

Return value

A toolbar object.

Remarks

An application can override this function to return a different kind of toolbar. The default implementation returns an instance of *wxToolBar* (p. **Error! Bookmark not defined.**).

See also

wxFrame::CreateToolBar (p. 587), *wxToolBar* (p. **Error! Bookmark not defined.**).

wxFrame::ProcessCommand

void ProcessCommand(int id)

Simulate a menu command.

Parameters

id

The identifier for a menu item.

wxFrame::SendSizeEvent

void SendSizeEvent()

This function sends a dummy *size event* (p. **Error! Bookmark not defined.**) to the frame forcing it to reevaluate its children positions. It is sometimes useful to call this function after adding or deleting a children after the frame creation or if a child size changes.

Note that if the frame is using either sizers or constraints for the children layout, it is enough to call *Layout()* (p. **Error! Bookmark not defined.**) directly and this function should not be used in this case.

wxFrame::SetMenuBar

void SetMenuBar(wxMenuBar* menuBar)

Tells the frame to show the given menu bar.

Parameters

menuBar

The menu bar to associate with the frame.

Remarks

If the frame is destroyed, the menu bar and its menus will be destroyed also, so do not delete the menu bar explicitly (except by resetting the frame's menu bar to another frame or NULL).

Under Windows, a size event is generated, so be sure to initialize data members properly before calling **SetMenuBar**.

Note that on some platforms, it is not possible to call this function twice for the same frame object.

See also

wxFrame::GetMenuBar (p. 588), *wxMenuBar* (p. **Error! Bookmark not defined.**), *wxMenu* (p. **Error! Bookmark not defined.**).

wxFrame::SetStatusBar

void SetStatusBar(wxStatusBar* statusBar)

Associates a status bar with the frame.

See also

wxFrame::CreateStatusBar (p. 586), *wxStatusBar* (p. **Error! Bookmark not defined.**),
wxFrame::GetStatusBar (p. 588)

wxFrame::SetStatusBarPane

void SetStatusBarPane(int *n*)

Set the status bar pane used to display menu and toolbar help. Using -1 disables help display.

wxFrame::SetStatusText

virtual void SetStatusText(const wxString& *text*, int *number* = 0)

Sets the status bar text and redraws the status bar.

Parameters

text

The text for the status field.

number

The status field (starting from zero).

Remarks

Use an empty string to clear the status bar.

See also

wxFrame::CreateStatusBar (p. 586), *wxStatusBar* (p. **Error! Bookmark not defined.**)

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. 586).

widths

Must contain an array of *n* integers, each of which is a status field width in pixels. A value of -1 indicates that the field is variable width; at least one field must be -1. You should delete this array after calling **SetStatusWidths**.

Remarks

The widths of the variable fields are calculated from the total width of all fields, minus the sum of widths of the non-variable fields, divided by the number of variable fields.

wxPython note: Only a single parameter is required, a Python list of integers.

wxPerl note: In wxPerl this method takes the field widths as parameters.

wxFrame::SetToolBar

void SetToolBar(wxToolBar* *toolBar*)

Associates a toolbar with the frame.

See also

wxFrame::CreateToolBar (p. 587), *wxToolBar* (p. **Error! Bookmark not defined.**),
wxFrame::GetToolBar (p. 589)

wxFSFile

This class represents a single file opened by *wxFileSystem* (p. 542). It provides more information than *wxWindow*'s input stream (*stream*, *filename*, *mime type*, *anchor*).

Note: Any pointer returned by a method of *wxFSFile* is valid only as long as the *wxFSFile* object exists. For example a call to *GetStream()* doesn't *create* the stream but only returns the pointer to it. In other words after 10 calls to *GetStream()* you will obtain ten identical pointers.

Derived from

wxObject (p. **Error! Bookmark not defined.**)

Include files

<wx/filesys.h>

See Also

wxFileSystemHandler (p. 545), *wxFileSystem* (p. 542), *Overview* (p. **Error! Bookmark not defined.**)

wxFSFile::wxFSFile

wxFSFile(wxInputStream* *stream*, const wxString& *loc*, const wxString& *mimetype*, const wxString& *anchor*, wxDateTime *modif*)

Constructor. You probably won't use it. See Notes for details.

Parameters

stream

The input stream that will be used to access data

location

The full location (aka filename) of the file

mimetype

MIME type of this file. Mime type is either extension-based or HTTP Content-Type

anchor

Anchor. See *GetAnchor()* (p. 594) for details.

If you are not sure of the meaning of these params, see the description of the *GetXXXX()* functions.

Notes

It is seldom used by the application programmer but you will need it if you are writing your own virtual FS. For example you may need something similar to *wxMemoryInputStream*, but because *wxMemoryInputStream* doesn't free the memory when destroyed and thus passing a memory stream pointer into *wxFSFile* constructor would lead to memory leaks, you can write your own class derived from *wxFSFile*:

```
class wxMyFSFile : public wxFSFile
{
    private:
        void *m_Mem;
    public:
        wxMyFSFile(.....)
        ~wxMyFSFile() {free(m_Mem);}
        // of course dtor is virtual ;-)
};
```

wxFSFile::GetAnchor**const wxString& GetAnchor() const**

Returns anchor (if present). The term of **anchor** can be easily explained using few examples:

```
index.htm#anchor           /* 'anchor' is anchor */
index/wx001.htm           /* NO anchor here!    */
archive/main.zip#zip:index.htm#global /* 'global'          */
archive/main.zip#zip:index.htm /* NO anchor here!    */
```

Usually an anchor is presented only if the MIME type is 'text/html'. But it may have some meaning with other files; for example *myanim.avi#200* may refer to position in animation or *reality.wrl#MyView* may refer to a predefined view in VRML.

wxFSFile::GetLocation

const wxString& GetLocation() const

Returns full location of the file, including path and protocol. Examples :

```
http://www.wxwidgets.org
http://www.ms.mff.cuni.cz/~vsla8348/wxhtml/archive.zip#zip:info.txt
file:/home/vasek/index.htm
relative-file.htm
```

wxFSFile::GetMimeType**const wxString& GetMimeType() const**

Returns the MIME type of the content of this file. It is either extension-based (see `wxMimeTypesManager`) or extracted from HTTP protocol Content-Type header.

wxFSFile::GetModificationTime**wxDateTime GetModificationTime() const**

Returns time when this file was modified.

wxFSFile::GetStream**wxInputStream* GetStream() const**

Returns pointer to the stream. You can use the returned stream to directly access data. You may suppose that the stream provide `Seek` and `GetSize` functionality (even in the case of the HTTP protocol which doesn't provide this by default. `wxHtml` uses local cache to work around this and to speed up the connection).

wxFTP

`wxFTP` can be used to establish a connection to an FTP server and perform all the usual operations. Please consult the RFC 959 for more details about the FTP protocol.

To use a commands which doesn't involve file transfer (i.e. directory oriented commands) you just need to call a corresponding member function or use the generic *SendCommand* (p. 597) method. However to actually transfer files you just get or give a stream to or from this class and the actual data are read or written using the usual stream methods.

Example of using `wxFTP` for file downloading:

```
wxFTP ftp;

// if you don't use these lines anonymous login will be used
ftp.SetUser("user");
ftp.SetPassword("password");

if ( !ftp.Connect("ftp.wxwindows.org") )
{
```

```
        wxLogError("Couldn't connect");
        return;
    }

    ftp.ChDir("/pub");
    wxInputStream *in = ftp.GetInputStream("wxWidgets-
4.2.0.tar.gz");
    if ( !in )
    {
        wxLogError("Coudln't get file");
    }
    else
    {
        size_t size = in->GetSize();
        char *data = new char[size];
        if ( !in->Read(data, size) )
        {
            wxLogError("Read error");
        }
        else
        {
            // file data is in the buffer
            ...
        }

        delete [] data;
        delete in;
    }
}
```

To upload a file you would do (assuming the connection to the server was opened successfully):

```
wxOutputStream *out = ftp.GetOutputStream("filename");
if ( out )
{
    out->Write(...); // your data
    delete out;
}
```

Constants

wxFTP defines constants corresponding to the two supported transfer modes:

```
enum TransferMode
{
    ASCII,
    BINARY
};
```

Derived from

wxProtocol (p. **Error! Bookmark not defined.**)

Include files

<wx/protocol/ftp.h>

See also

wxSocketBase (p. **Error! Bookmark not defined.**)

wxFTP::wxFTP**wxFTP()**

Default constructor.

wxFTP::~~wxFTP**~wxFTP()**

Destructor will close the connection if connected.

wxFTP::Abort**bool Abort()**

Aborts the download currently in process, returns `true` if ok, `false` if an error occurred.

wxFTP::CheckCommand**bool CheckCommand(const wxString& *command*, char *ret*)**

Send the specified *command* to the FTP server. *ret* specifies the expected result.

Return value

`true` if the command has been sent successfully, else `false`.

wxFTP::SendCommand**char SendCommand(const wxString& *command*)**

Send the specified *command* to the FTP server and return the first character of the return code.

wxFTP::GetLastResult**const wxString& GetLastResult()**

Returns the last command result, i.e. the full server reply for the last command.

wxFTP::ChDir**bool ChDir(const wxString& *dir*)**

Change the current FTP working directory. Returns `true` if successful.

wxFTP::MkDir**bool MkDir(const wxString& dir)**

Create the specified directory in the current FTP working directory. Returns true if successful.

wxFTP::RmDir**bool RmDir(const wxString& dir)**

Remove the specified directory from the current FTP working directory. Returns true if successful.

wxFTP::Pwd**wxString Pwd()**

Returns the current FTP working directory.

wxFTP::Rename**bool Rename(const wxString& src, const wxString& dst)**

Rename the specified *src* element to *dst*. Returns true if successful.

wxFTP::RmFile**bool RmFile(const wxString& path)**

Delete the file specified by *path*. Returns true if successful.

wxFTP::SetAscii**bool SetAscii()**

Sets the transfer mode to ASCII. It will be used for the next transfer.

wxFTP::SetBinary**bool SetBinary()**

Sets the transfer mode to binary (IMAGE). It will be used for the next transfer.

wxFTP::SetPassive**void SetPassive(bool pasv)**

If *pasv* is `true`, passive connection to the FTP server is used. This is the default as it works with practically all firewalls. If the server doesn't support passive move, you may

call this function with `false` argument to use active connection.

wxFTP::SetTransferMode

bool SetTransferMode(TransferMode mode)

Sets the transfer mode to the specified one. It will be used for the next transfer.

If this function is never called, binary transfer mode is used by default.

wxFTP::SetUser

void SetUser(const wxString& user)

Sets the user name to be sent to the FTP server to be allowed to log in.

Default value

The default value of the user name is "anonymous".

Remark

This parameter can be included in a URL if you want to use the URL manager. For example, you can use: "ftp://a_user:a_password@a.host:service/a_directory/a_file" to specify a user and a password.

wxFTP::SetPassword

void SetPassword(const wxString& passwd)

Sets the password to be sent to the FTP server to be allowed to log in.

Default value

The default value of the user name is your email address. For example, it could be "username@userhost.domain". This password is built by getting the current user name and the host name of the local machine from the system.

Remark

This parameter can be included in a URL if you want to use the URL manager. For example, you can use: "ftp://a_user:a_password@a.host:service/a_directory/a_file" to specify a user and a password.

wxFTP::FileExists

bool FileExists(const wxString& filename)

Returns `true` if the given remote file exists, `false` otherwise.

wxFTP::GetFileSize

int GetFileSize(const wxString& filename)

Returns the file size in bytes or -1 if the file doesn't exist or the size couldn't be determined. Notice that this size can be approximative size only and shouldn't be used for allocating the buffer in which the remote file is copied, for example.

wxFTP::GetDirList**bool GetDirList(wxArrayString& files, const wxString& wildcard = "")**

The GetList function is quite low-level. It returns the list of the files in the current directory. The list can be filtered using the *wildcard* string. If *wildcard* is empty (default), it will return all files in directory.

The form of the list can change from one peer system to another. For example, for a UNIX peer system, it will look like this:

```
-r--r--r--  1 guilhem  lavaux      12738 Jan 16 20:17 cmndata.cpp
-r--r--r--  1 guilhem  lavaux      10866 Jan 24 16:41 config.cpp
-rw-rw-rw-  1 guilhem  lavaux      29967 Dec 21 19:17 cwlex.yy.c
-rw-rw-rw-  1 guilhem  lavaux      14342 Jan 22 19:51 cwy_tab.c
-r--r--r--  1 guilhem  lavaux      13890 Jan 29 19:18 date.cpp
-r--r--r--  1 guilhem  lavaux       3989 Feb  8 19:18 datstrm.cpp
```

But on Windows system, it will look like this:

```
winamp~1 exe      520196 02-25-1999 19:28 winamp204.exe
1 file(s)                520 196 bytes
```

Return value: true if the file list was successfully retrieved, false otherwise.

See also

GetFilesList (p. 600)

wxFTP::GetFilesList**bool GetFilesList(wxArrayString& files, const wxString& wildcard = "")**

This function returns the computer-parsable list of the files in the current directory (optionally only of the files matching the *wildcard*, all files by default). This list always has the same format and contains one full (including the directory path) file name per line.

Return value: true if the file list was successfully retrieved, false otherwise.

wxFTP::GetOutputStream**wxOutputStream * GetOutputStream(const wxString& file)**

Initializes an output stream to the specified *file*. The returned stream has all but the seek functionality of wxStreams. When the user finishes writing data, he has to delete the stream to close it.

Return value

An initialized write-only stream.

See also

wxOutputStream (p. **Error! Bookmark not defined.**)

wxFTP::GetInputStream

wxInputStream * GetInputStream(const wxString& path)

Creates a new input stream on the specified path. You can use all but the seek functionality of *wxStream*. Seek isn't available on all streams. For example, HTTP or FTP streams do not deal with it. Other functions like Tell are not available for this sort of stream, at present. You will be notified when the EOF is reached by an error.

Return value

Returns NULL if an error occurred (it could be a network failure or the fact that the file doesn't exist).

Returns the initialized stream. You will have to delete it yourself when you don't need it anymore. The destructor closes the DATA stream connection but will leave the COMMAND stream connection opened. It means that you can still send new commands without reconnecting.

Example of a standalone connection (without wxURL)

```
wxFTP ftp;
wxInputStream *in_stream;
char *data;

ftp.Connect("a.host.domain");
ftp.ChDir("a_directory");
in_stream = ftp.GetInputStream("a_file_to_get");

data = new char[in_stream->GetSize()];

in_stream->Read(data, in_stream->GetSize());
if (in_stream->LastError() != wxStream_NOERROR) {
    // Do something.
}

delete in_stream; /* Close the DATA connection */

ftp.Close(); /* Close the COMMAND connection */
```

See also

wxInputStream (p. 826)

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. 218)

wxWindow (p. **Error! Bookmark not defined.**)

wxEvtHandler (p. 490)

wxObject (p. **Error! Bookmark not defined.**)

Include files

<wx/gauge.h>

Window styles

wxGA_HORIZONTAL Creates a horizontal gauge.

wxGA_VERTICAL Creates a vertical gauge.

wxGA_SMOOTH Creates smooth progress bar with one pixel wide update step (not supported by all platforms).

See also *window styles overview* (p. **Error! Bookmark not defined.**).

Event handling

wxGauge is read-only so generates no events.

See also

wxSlider (p. **Error! Bookmark not defined.**), *wxScrollBar* (p. **Error! Bookmark not defined.**)

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. 601).

name

Window name.

See also

wxGauge::Create (p. 603)

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. 602) for further details.

wxGauge::GetBezelFace

int GetBezelFace() const

Returns the width of the 3D bezel face.

Remarks

This method is not implemented (returns 0) for most platforms.

See also

wxGauge::SetBezelFace (p. 604)

wxGauge::GetRange**int GetRange() const**

Returns the maximum position of the gauge.

Remarks

This method is not implemented (doesn't do anything) for most platforms.

See also

wxGauge::SetRange (p. 605)

wxGauge::GetShadowWidth**int GetShadowWidth() const**

Returns the 3D shadow margin width.

Remarks

This method is not implemented (returns 0) for most platforms.

See also

wxGauge::SetShadowWidth (p. 605)

wxGauge::GetValue**int GetValue() const**

Returns the current position of the gauge.

See also

wxGauge::SetValue (p. 605)

wxGauge::IsVertical**bool IsVertical() const**

Returns `true` if the gauge is vertical (has `wxGA_VERTICAL` style) and `false` otherwise.

wxGauge::SetBezelFace**void SetBezelFace(int width)**

Sets the 3D bezel face width.

Remarks

This method is not implemented (doesn't do anything) for most platforms.

See also

wxGauge::GetBezelFace (p. 603)

wxGauge::SetRange

void SetRange(int range)

Sets the range (maximum value) of the gauge.

See also

wxGauge::GetRange (p. 604)

wxGauge::SetShadowWidth

void SetShadowWidth(int width)

Sets the 3D shadow width.

Remarks

This method is not implemented (doesn't do anything) for most platforms.

wxGauge::SetValue

void SetValue(int pos)

Sets the position of the gauge.

Parameters

pos

Position for the gauge level.

See also

wxGauge::GetValue (p. 604)

wxGBPosition

This class represents the position of an item in a virtual grid of rows and columns managed by a *wxGridBagSizer* (p. 657).

Derived from

No base class

Include files

<wx/gbsizer.h>

wxGBPosition::wxGBPosition**wxGBPosition()****wxGBPosition(int row, int col)**

Construct a new wxGBPosition, optionally setting the row and column. The default is (0,0).

wxGBPosition::GetCol**int GetCol() const**

Get the current column value.

wxGBPosition::GetRow**int GetRow() const**

Get the current row value.

wxGBPosition::SetCol**void SetCol(int col)**

Set a new column value.

wxGBPosition::SetRow**void SetRow(int row)**

Set a new row value.

wxGBPosition::operator!**bool operator!(const wxGBPosition& p) const**

Is the wxGBPosition valid? (An invalid wxGBPosition is (-1,-1).)

wxGBPosition::operator==**bool operator==(const wxGBPosition& p) const**

Compare equality of two wxGBPositions.

wxGBSizerItem

The `wxGBSizerItem` class is used by the `wxGridBagSizer` (p. 657) for tracking the items in the sizer. It adds grid position and spanning information to the normal `wxSizerItem` (p. **Error! Bookmark not defined.**) by adding `wxGBPosition` (p. 605) and `wxGBSpan` (p. 608) attributes. Most of the time you will not need to use a `wxGBSizerItem` directly in your code, but there are a couple of cases where it is handy.

Derived from

`wxSizerItem` (p. **Error! Bookmark not defined.**)

Include files

<wx/gbsizer.h>

`wxGBSizerItem::wxGBSizerItem`

`wxGBSizerItem(int width, int height, const wxGBPosition& pos, const wxGBSpan& span, int flag, int border, wxObject* userData)`

Construct a sizer item for tracking a spacer.

`wxGBSizerItem(wxWindow* window, const wxGBPosition& pos, const wxGBSpan& span, int flag, int border, wxObject* userData)`

Construct a sizer item for tracking a window.

`wxGBSizerItem(wxSizer* sizer, const wxGBPosition& pos, const wxGBSpan& span, int flag, int border, wxObject* userData)`

Construct a sizer item for tracking a subsizer.

`wxGBSizerItem::GetEndPos`

`void GetEndPos(int& row, int& col)`

Get the row and column of the endpoint of this item

`wxGBSizerItem::GetPos`

`wxGBPosition GetPos() const`

`void GetPos(int& row, int& col) const`

Get the grid position of the item.

`wxGBSizerItem::GetSpan`

`wxGBSpan GetSpan() const`

`void GetSpan(int& rowspan, int& colspan) const`

Get the row and column spanning of the item.

wxGBSizerItem::Intersects

bool Intersects(const wxGBSizerItem& other)

Returns true if this item and the other item intersect

bool Intersects(const wxGBPosition& pos, const wxGBSpan& span)

Returns true if the given pos/span would intersect with this item.

wxGBSizerItem::SetPos

bool SetPos(const wxGBPosition& pos)

If the item is already a member of a sizer then first ensure that there is no other item that would intersect with this one at the new position, then set the new position. Returns true if the change is successful and after the next Layout the item will be moved.

wxGBSizerItem::SetSpan

bool SetSpan(const wxGBSpan& span)

If the item is already a member of a sizer then first ensure that there is no other item that would intersect with this one with its new spanning size, then set the new spanning. Returns true if the change is successful and after the next Layout the item will be resized.

wxGBSpan

This class is used to hold the row and column spanning attributes of items in a *wxGridBagSizer* (p. 657).

Derived from

No base class

Include files

<wx/gbsizer.h>

wxGBSpan::wxGBSpan

wxGBSpan()

wxGBSpan(int rowspan, int colspan)

Construct a new wxGBSpan, optionally setting the rowspan and colspan. The default is

(1,1). (Meaning that the item occupies one cell in each direction.)

wxGBSpan::GetColspan

int GetColspan() const

Get the current colspan value.

wxGBSpan::GetRowspan

int GetRowspan() const

Get the current rowspan value.

wxGBSpan::SetColspan

void SetColspan(int colspan)

Set a new colspan value.

wxGBSpan::SetRowspan

void SetRowspan(int rowspan)

Set a new rowspan value.

wxGBSpan::operator!

bool operator!(const wxGBSpan& o) const

Is the wxGBSpan valid? (An invalid wxGBSpan is (-1,-1).)

wxGBSpan::operator==

bool operator==(const wxGBSpan& o) const

Compare equality of two wxGBSpans.

wxGDIObject

This class allows platforms to implement functionality to optimise GDI objects, such as wxPen, wxBrush and wxFont. On Windows, the 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. **Error! Bookmark not defined.**)

Include files

<wx/gdiobj.h>

See also

wxPen (p. **Error! Bookmark not defined.**), *wxBrush* (p. 108), *wxFont* (p. 561)

wxGDIObject::wxGDIObject

wxGDIObject()

Default constructor.

wxGenericDirCtrl

This control can be used to place a directory listing (with optional files) on an arbitrary window.

The control contains a *wxTreeCtrl* (p. **Error! Bookmark not defined.**) window representing the directory hierarchy, and optionally, a *wxChoice* (p. 145) window containing a list of filters.

Derived from

wxControl (p. 218)

wxWindow (p. **Error! Bookmark not defined.**)

wxEvtHandler (p. 490)

wxObject (p. **Error! Bookmark not defined.**)

Include files

<wx/dirctrl.h>

Window styles

<code>wxDIRCTRL_DIR_ONLY</code>	Only show directories, and not files.
<code>wxDIRCTRL_3D_INTERNAL</code>	Use 3D borders for internal controls.
<code>wxDIRCTRL_SELECT_FIRST</code>	When setting the default path, select the first file in the directory.
<code>wxDIRCTRL_SHOW_FILTERS</code>	Show the drop-down filter list.
<code>wxDIRCTRL_EDIT_LABELS</code>	Allow the folder and file labels to be editable.

See also *Generic window styles* (p. **Error! Bookmark not defined.**).

Data structures

wxGenericDirCtrl::wxGenericDirCtrl

wxGenericDirCtrl()

Default constructor.

wxGenericDirCtrl(wxWindow* parent, const wxWindowID id = -1, const wxString& dir = wxDirDialogDefaultFolderStr, const wxPoint& pos = wxDefaultPosition, const wxSize& size = wxDefaultSize, long style = wxDIRCTRL_3D_INTERNAL|wxSUNKEN_BORDER, const wxString& filter = wxEmptyString, int defaultFilter = 0, const wxString& name = wxTreeCtrlNameStr)

Main constructor.

Parameters

parent

Parent window.

id

Window identifier.

dir

Initial folder.

pos

Position.

size

Size.

style

Window style. Please see *wxGenericDirCtrl* (p. 610) for a list of possible styles.

filter

A filter string, using the same syntax as that for *wxFileDialog* (p. 515). This may be empty if filters are not being used.

Example: "All files (*.*)|*.jpg|JPEG files (*.jpg)|*.jpg"

defaultFilter

The zero-indexed default filter setting.

name

The window name.

wxGenericDirCtrl::~~wxGenericDirCtrl

~wxGenericDirCtrl()

Destructor.

wxGenericDirCtrl::Create

bool Create(wxWindow* parent, const wxWindowID id = -1, const wxString& dir = wxDirDialogDefaultFolderStr, const wxPoint& pos = wxDefaultPosition, const wxSize& size = wxDefaultSize, long style = wxDIRCTRL_3D_INTERNAL|wxSUNKEN_BORDER, const wxString& filter = wxEmptyString, int defaultFilter = 0, const wxString& name = wxTreeCtrlNameStr)

Create function for two-step construction. See *wxGenericDirCtrl::wxGenericDirCtrl* (p. 611) for details.

wxGenericDirCtrl::Init

void Init()

Initializes variables.

wxGenericDirCtrl::CollapseTree

void CollapseTree()

Collapses the entire tree.

wxGenericDirCtrl::ExpandPath

bool ExpandPath(const wxString& path)

Tries to expand as much of the given path as possible, so that the filename or directory is visible in the tree control.

wxGenericDirCtrl::GetDefaultPath

wxString GetDefaultPath() const

Gets the default path.

wxGenericDirCtrl::GetPath

wxString GetPath() const

Gets the currently-selected directory or filename.

wxGenericDirCtrl::GetFilePath**wxString GetFilePath() const**

Gets selected filename path only (else empty string).

This function doesn't count a directory as a selection.

wxGenericDirCtrl::GetFilter**wxString GetFilter() const**

Returns the filter string.

wxGenericDirCtrl::GetFilterIndex**int GetFilterIndex() const**

Returns the current filter index (zero-based).

wxGenericDirCtrl::GetFilterListCtrl**wxDirFilterListCtrl* GetFilterListCtrl() const**

Returns a pointer to the filter list control (if present).

wxGenericDirCtrl::GetRootId**wxTreeItemId GetRootId()**

Returns the root id for the tree control.

wxGenericDirCtrl::GetTreeCtrl**wxTreeCtrl* GetTreeCtrl() const**

Returns a pointer to the tree control.

wxGenericDirCtrl::ReCreateTree**void ReCreateTree()**

Collapse and expand the tree, thus re-creating it from scratch. May be used to update the displayed directory content.

wxGenericDirCtrl::SetDefaultPath**void SetDefaultPath(const wxString& path)**

Sets the default path.

wxGenericDirCtrl::SetFilter

void SetFilter(const wxString& filter)

Sets the filter string.

wxGenericDirCtrl::SetFilterIndex

void SetFilterIndex(int n)

Sets the current filter index (zero-based).

wxGenericDirCtrl::SetPath

void SetPath(const wxString& path)

Sets the current path.

wxGenericValidator

wxGenericValidator performs data transfer (but not validation or filtering) for the following basic controls: wxButton, wxCheckBox, wxListBox, wxStaticText, wxRadioButton, wxRadioBox, wxChoice, wxComboBox, wxGauge, wxSlider, wxScrollBar, wxSpinButton, wxTextCtrl, wxCheckListBox.

It checks the type of the window and uses an appropriate type for that window. For example, wxButton and wxTextCtrl transfer data to and from a wxString variable; wxListBox uses a wxArrayInt; wxCheckBox uses a bool.

For more information, please see *Validator overview* (p. **Error! Bookmark not defined.**).

Derived from

wxValidator (p. **Error! Bookmark not defined.**)

wxEvtHandler (p. 490)

wxObject (p. **Error! Bookmark not defined.**)

Include files

<wx/valgen.h>

See also

Validator overview (p. **Error! Bookmark not defined.**), *wxValidator* (p. **Error! Bookmark not defined.**), *wxTextValidator* (p. **Error! Bookmark not defined.**)

wxGenericValidator::wxGenericValidator**wxGenericValidator(const wxGenericValidator& validator)**

Copy constructor.

wxGenericValidator(bool* valPtr)

Constructor taking a bool pointer. This will be used for wxCheckBox and wxRadioButton.

wxGenericValidator(wxString* valPtr)

Constructor taking a wxString pointer. This will be used for wxButton, wxComboBox, wxStaticText, wxTextCtrl.

wxGenericValidator(int* valPtr)

Constructor taking an integer pointer. This will be used for wxGauge, wxScrollBar, wxRadioBox, wxSpinButton, wxChoice.

wxGenericValidator(wxArrayInt* valPtr)

Constructor taking a wxArrayInt pointer. This will be used for wxListBox, wxCheckListBox.

Parameters*validator*

Validator to copy.

valPtr

A pointer to a variable that contains the value. This variable should have a lifetime equal to or longer than the validator lifetime (which is usually determined by the lifetime of the window).

wxGenericValidator::~~wxGenericValidator**~wxGenericValidator()**

Destructor.

wxGenericValidator::Clone**virtual wxValidator* Clone() const**

Clones the generic validator using the copy constructor.

wxGenericValidator::TransferFromWindow**virtual bool TransferFromWindow()**

Transfers the value from the window to the appropriate data type.

wxGenericValidator::TransferToWindow

virtual bool TransferToWindow()

Transfers the value to the window.

wxGLCanvas

wxGLCanvas is a class for displaying OpenGL graphics. There are wrappers for OpenGL on Windows, and GTK+ and Motif.

To use this class, create a wxGLCanvas window, call *wxGLCanvas::SetCurrent* (p. 619) to direct normal OpenGL commands to the window, and then call *wxGLCanvas::SwapBuffers* (p. 619) to show the OpenGL buffer on the window.

To set up the attributes for the rendering context (number of bits for the depth buffer, number of bits for the stencil buffer and so on) you should set up the correct values of the *attribList* parameter. The values that should be set up and their meanings will be described below.

To switch wxGLCanvas support on under Windows, edit *setup.h* and set *wxUSE_GL_CANVAS* to 1. You may also need to have to add *opengl32.lib* to the list of libraries your program is linked with. On Unix, pass *--with-opengl* to configure to compile using OpenGL or Mesa.

Derived from

wxWindow (p. **Error! Bookmark not defined.**)

wxEvtHandler (p. 490)

wxObject (p. **Error! Bookmark not defined.**)

Include files

<wx/glcanvas.h>

Window styles

There are no specific window styles for this class.

See also *window styles overview* (p. **Error! Bookmark not defined.**).

Constants

The generic GL implementation doesn't support many of these options, such as stereo, auxiliary buffers, alpha channel, and accum buffer. Other implementations may support them.

WX_GL_RGBA Use true colour

WX_GL_BUFFER_SIZE Bits for buffer if not WX_GL_RGBA

WX_GL_LEVEL	0 for main buffer, >0 for overlay, <0 for underlay
WX_GL_DOUBLEBUFFER	Use doublebuffer
WX_GL_STEREO	Use stereoscopic display
WX_GL_AUX_BUFFERS	Number of auxiliary buffers (not all implementation support this option)
WX_GL_MIN_RED	Use red buffer with most bits (> MIN_RED bits)
WX_GL_MIN_GREEN	Use green buffer with most bits (> MIN_GREEN bits)
WX_GL_MIN_BLUE	Use blue buffer with most bits (> MIN_BLUE bits)
WX_GL_MIN_ALPHA	Use alpha buffer with most bits (> MIN_ALPHA bits)
WX_GL_DEPTH_SIZE	Bits for Z-buffer (0,16,32)
WX_GL_STENCIL_SIZE	Bits for stencil buffer
WX_GL_MIN_ACCUM_RED	Use red accum buffer with most bits (> MIN_ACCUM_RED bits)
WX_GL_MIN_ACCUM_GREEN	Use green buffer with most bits (> MIN_ACCUM_GREEN bits)
WX_GL_MIN_ACCUM_BLUE	Use blue buffer with most bits (> MIN_ACCUM_BLUE bits)
WX_GL_MIN_ACCUM_ALPHA	Use blue buffer with most bits (> MIN_ACCUM_ALPHA bits)

See also

wxGLContext (p. 619)

wxGLCanvas::wxGLCanvas

void wxGLCanvas(wxWindow* parent, wxWindowID id = -1, const wxPoint& pos = wxDefaultPosition, const wxSize& size = wxDefaultSize, long style=0, const wxString& name="GLCanvas", int* attribList = 0, const wxPalette& palette = wxNullPalette)

void wxGLCanvas(wxWindow* parent, wxGLContext* sharedContext, wxWindowID id = -1, const wxPoint& pos = wxDefaultPosition, const wxSize& size = wxDefaultSize, long style=0, const wxString& name="GLCanvas", int* attribList = 0, const wxPalette& palette = wxNullPalette)

void wxGLCanvas(wxWindow* parent, wxGLCanvas* sharedCanvas, wxWindowID id = -1, const wxPoint& pos = wxDefaultPosition, const wxSize& size = wxDefaultSize, long style=0, const wxString& name="GLCanvas", int* attribList = 0, const

wxPalette& *palette* = *wxNullPalette*)

Constructor.

parent

Pointer to a parent window.

sharedContext

Context to share object resources with.

id

Window identifier. If -1, will automatically create an identifier.

pos

Window position. *wxDefaultPosition* is (-1, -1) which indicates that *wxWidgets* should generate a default position for the window.

size

Window size. *wxDefaultSize* is (-1, -1) which indicates that *wxWidgets* should generate a default size for the window. If no suitable size can be found, the window will be sized to 20x20 pixels so that the window is visible but obviously not correctly sized.

style

Window style.

name

Window name.

attribList

Array of int. With this parameter you can set the device context attributes associated to this window. This array is zero-terminated: it should be set up with constants described in the table above. If a constant should be followed by a value, put it in the next array position. For example, the *WX_GL_DEPTH_SIZE* should be followed by the value that indicates the number of bits for the depth buffer, so:

```
attribList[index]= WX_GL_DEPTH_SIZE;  
attribList[index+1]=32;  
and so on.
```

palette

If the window has the palette, it should by pass this value. Note: *palette* and *WX_GL_RGBA* are mutually exclusive.

wxGLCanvas::GetContext

wxGLContext* GetContext()

Obtains the context that is associated with this canvas.

wxGLCanvas::SetCurrent**void SetCurrent()**

Sets this canvas as the current recipient of OpenGL calls. Each canvas contains an OpenGL device context that has been created during the creation of this window. So this call sets the current device context as the target device context for OpenGL operations.

Note that this function may only be called after the window has been shown.

wxGLCanvas::SetColour**void SetColour(const char* colour)**

Sets the current colour for this window, using the wxWidgets colour database to find a named colour.

wxGLCanvas::SwapBuffers**void SwapBuffers()**

Displays the previous OpenGL commands on the window.

wxGLContext

wxGLContext is a class for sharing OpenGL data resources, such as display lists, with another *wxGLCanvas* (p. 616).

By sharing data resources, you can prevent code duplication, save memory, and ultimately help optimize your application.

As an example, let's say you want to draw a ball on two different windows that is identical on each one, but the dimensions of one is slightly different than the other one. What you would do is create your display lists in the shared context, and then translate each ball on the individual canvas's context. This way the actual data for the ball is only created once (in the shared context), and you won't have to duplicate your development efforts on the second ball.

Note that some wxGLContext features are extremely platform-specific - its best to check your native platform's glcanvas header (on windows include/wx/msw/glcanvas.h) to see what features your native platform provides.

Derived from

wxObject (p. **Error! Bookmark not defined.**)

Include files

<wx/glcanvas.h>

See also

wxGLCanvas (p. 616)

wxGLContext::wxGLContext

void wxGLContext(bool *isRGB*, wxGLCanvas* *win*, const wxPalette& *palette* = wxNullPalette)

void wxGLContext(bool *isRGB*, wxGLCanvas* *win*, const wxPalette& *palette* = wxNullPalette, const wxGLContext* *other*)

win

Canvas to associate this shared context with.

other

Context to share data resources with.

wxGLContext::GetWindow

const wxWindow* GetWindow()

Obtains the window that is associated with this context.

wxGLContext::SetCurrent

void SetCurrent()

Sets this context as the current recipient of OpenGL calls. Each context contains an OpenGL device context that has been created during the creation of this window. So this call sets the current device context as the target device context for OpenGL operations.

wxGLContext::SetColour

void SetColour(const char* *colour*)

Sets the current colour for this context, using the wxWidgets colour database to find a named colour.

wxGLContext::SwapBuffers

void SwapBuffers()

Displays the previous OpenGL commands on the associated window.

wxGrid

wxGrid and its related classes are used for displaying and editing tabular data. They provide a rich set of features for display, editing, and interacting with a variety of data sources. For simple applications, and to help you get started, wxGrid is the only class you need to refer to directly. It will set up default instances of the other classes and manage them for you. For more complex applications you can derive your own classes for custom grid views, grid data tables, cell editors and renderers. The *wxGrid classes overview* (p. **Error! Bookmark not defined.**) has examples of simple and more complex applications, explains the relationship between the various grid classes and has a summary of the keyboard shortcuts and mouse functions provided by wxGrid.

wxGrid has been greatly expanded and redesigned for wxWidgets 2.2 onwards. If you have been using the old wxGrid class you will probably want to have a look at the *wxGrid classes overview* (p. **Error! Bookmark not defined.**) to see how things have changed. The new grid classes are reasonably backward-compatible but there are some exceptions. There are also easier ways of doing many things compared to the previous implementation.

Derived from

wxScrolledWindow (p. **Error! Bookmark not defined.**)

wxWindow (p. **Error! Bookmark not defined.**)

wxEvtHandler (p. 490)

wxObject (p. **Error! Bookmark not defined.**)

Include files

<wx/grid.h>

Window styles

There are presently no specific window styles for wxGrid.

Event handling

The event handler for the following functions takes a *wxGridEvent* (p. 667) parameter. The ..._CMD_... variants also take a window identifier.

EVT_GRID_CELL_LEFT_CLICK(func) The user clicked a cell with the left mouse button. Processes a wxEVT_GRID_CELL_LEFT_CLICK.

EVT_GRID_CELL_RIGHT_CLICK(func) The user clicked a cell with the right mouse button. Processes a wxEVT_GRID_CELL_RIGHT_CLICK.

EVT_GRID_CELL_LEFT_DCLICK(func) The user double-clicked a cell with the left mouse button. Processes a wxEVT_GRID_CELL_LEFT_DCLICK.

EVT_GRID_CELL_RIGHT_DCLICK(func) The user double-clicked a cell with the right mouse button. Processes a

`wxEVT_GRID_CELL_RIGHT_DCLICK`.

EVT_GRID_LABEL_LEFT_CLICK(func) The user clicked a label with the left mouse button. Processes a `wxEVT_GRID_LABEL_LEFT_CLICK`.

EVT_GRID_LABEL_RIGHT_CLICK(func) The user clicked a label with the right mouse button. Processes a `wxEVT_GRID_LABEL_RIGHT_CLICK`.

EVT_GRID_LABEL_LEFT_DCLICK(func) The user double-clicked a label with the left mouse button. Processes a `wxEVT_GRID_LABEL_LEFT_DCLICK`.

EVT_GRID_LABEL_RIGHT_DCLICK(func) The user double-clicked a label with the right mouse button. Processes a `wxEVT_GRID_LABEL_RIGHT_DCLICK`.

EVT_GRID_CELL_CHANGE(func) The user changed the data in a cell. Processes a `wxEVT_GRID_CELL_CHANGE`.

EVT_GRID_SELECT_CELL(func) The user moved to, and selected a cell. Processes a `wxEVT_GRID_SELECT_CELL`.

EVT_GRID_EDITOR_HIDDEN(func) The editor for a cell was hidden. Processes a `wxEVT_GRID_EDITOR_HIDDEN`.

EVT_GRID_EDITOR_SHOWN(func) The editor for a cell was shown. Processes a `wxEVT_GRID_EDITOR_SHOWN`.

EVT_GRID_CMD_CELL_LEFT_CLICK(id, func) The user clicked a cell with the left mouse button; variant taking a window identifier. Processes a `wxEVT_GRID_CELL_LEFT_CLICK`.

EVT_GRID_CMD_CELL_RIGHT_CLICK(id, func) The user clicked a cell with the right mouse button; variant taking a window identifier. Processes a `wxEVT_GRID_CELL_RIGHT_CLICK`.

EVT_GRID_CMD_CELL_LEFT_DCLICK(id, func) The user double-clicked a cell with the left mouse button; variant taking a window identifier. Processes a `wxEVT_GRID_CELL_LEFT_DCLICK`.

EVT_GRID_CMD_CELL_RIGHT_DCLICK(id, func) The user double-clicked a cell with the right mouse button; variant taking a window identifier. Processes a `wxEVT_GRID_CELL_RIGHT_DCLICK`.

EVT_GRID_CMD_LABEL_LEFT_CLICK(id, func) The user clicked a label with the left mouse button; variant taking a window identifier. Processes a

`wxEVT_GRID_LABEL_LEFT_CLICK`.

EVT_GRID_CMD_LABEL_RIGHT_CLICK(id, func) The user clicked a label with the right mouse button; variant taking a window identifier. Processes a `wxEVT_GRID_LABEL_RIGHT_CLICK`.

EVT_GRID_CMD_LABEL_LEFT_DCLICK(id, func) The user double-clicked a label with the left mouse button; variant taking a window identifier. Processes a `wxEVT_GRID_LABEL_LEFT_DCLICK`.

EVT_GRID_CMD_LABEL_RIGHT_DCLICK(id, func) The user double-clicked a label with the right mouse button; variant taking a window identifier. Processes a `wxEVT_GRID_LABEL_RIGHT_DCLICK`.

EVT_GRID_CMD_CELL_CHANGE(id, func) The user changed the data in a cell; variant taking a window identifier. Processes a `wxEVT_GRID_CELL_CHANGE`.

EVT_GRID_CMD_SELECT_CELL(id, func) The user moved to, and selected a cell; variant taking a window identifier. Processes a `wxEVT_GRID_SELECT_CELL`.

EVT_GRID_CMD_EDITOR_HIDDEN(id, func) The editor for a cell was hidden; variant taking a window identifier. Processes a `wxEVT_GRID_EDITOR_HIDDEN`.

EVT_GRID_CMD_EDITOR_SHOWN(id, func) The editor for a cell was shown; variant taking a window identifier. Processes a `wxEVT_GRID_EDITOR_SHOWN`.

The event handler for the following functions takes a `wxGridSizeEvent` (p. 673) parameter. The `..._CMD_...` variants also take a window identifier.

EVT_GRID_COL_SIZE(func) The user resized a column by dragging it. Processes a `wxEVT_GRID_COL_SIZE`.

EVT_GRID_ROW_SIZE(func) The user resized a row by dragging it. Processes a `wxEVT_GRID_ROW_SIZE`.

EVT_GRID_CMD_COL_SIZE(func) The user resized a column by dragging it; variant taking a window identifier. Processes a `wxEVT_GRID_COL_SIZE`.

EVT_GRID_CMD_ROW_SIZE(func) The user resized a row by dragging it; variant taking a window identifier. Processes a `wxEVT_GRID_ROW_SIZE`.

The event handler for the following functions takes a `wxGridRangeSelectEvent` (p. 671) parameter. The `..._CMD_...` variant also takes a window identifier.

EVT_GRID_RANGE_SELECT(func) The user selected a group of contiguous cells. Processes a `wxEVT_GRID_RANGE_SELECT`.

EVT_GRID_CMD_RANGE_SELECT(id, func) The user selected a group of contiguous cells; variant taking a window identifier. Processes a `wxEVT_GRID_RANGE_SELECT`.

The event handler for the following functions takes a `wxGridEditorCreatedEvent` (p. 666) parameter. The `..._CMD_...` variant also takes a window identifier.

EVT_GRID_EDITOR_CREATED(func) The editor for a cell was created. Processes a `wxEVT_GRID_EDITOR_CREATED`.

EVT_GRID_CMD_EDITOR_CREATED(id, func) The editor for a cell was created; variant taking a window identifier. Processes a `wxEVT_GRID_EDITOR_CREATED`.

See also

wxGrid overview (p. [Error! Bookmark not defined.](#))

Constructors and initialization

wxGrid (p. 624)
~wxGrid (p. 625)
CreateGrid (p. 628)
SetTable (p. 653)

Display format

Selection functions

wxGrid::ClearSelection (p. 628)
wxGrid::IsSelection (p. 641)
wxGrid::SelectAll (p. 644)
wxGrid::SelectBlock (p. 644)
wxGrid::SelectCol (p. 644)
wxGrid::SelectRow (p. 644)

wxGrid::wxGrid

wxGrid()

Default constructor

wxGrid(wxWindow* parent, wxWindowID id, const wxPoint& pos = wxDefaultPosition, const wxSize& size = wxDefaultSize, long style = wxWANTS_CHARS, const wxString& name = wxPanelNameStr)

Constructor to create a grid object. Call either *wxGrid::CreateGrid* (p. 628) or *wxGrid::SetTable* (p. 653) directly after this to initialize the grid before using it.

wxGrid::~wxGrid**~wxGrid()**

Destructor. This will also destroy the associated grid table unless you passed a table object to the grid and specified that the grid should not take ownership of the table (see *wxGrid::SetTable* (p. 653)).

wxGrid::AppendCols

bool AppendCols(int numCols = 1, bool updateLabels = true)

Appends one or more new columns to the right of the grid and returns true if successful. The updateLabels argument is not used at present.

If you are using a derived grid table class you will need to override *wxGridTableBase::AppendCols* (p. 681). See *wxGrid::InsertCols* (p. 640) for further information.

wxGrid::AppendRows

bool AppendRows(int numRows = 1, bool updateLabels = true)

Appends one or more new rows to the bottom of the grid and returns true if successful. The updateLabels argument is not used at present.

If you are using a derived grid table class you will need to override *wxGridTableBase::AppendRows* (p. 680). See *wxGrid::InsertRows* (p. 640) for further information.

wxGrid::AutoSize**void AutoSize()**

Automatically sets the height and width of all rows and columns to fit their contents.

Note

wxGrid sets up arrays to store individual row and column sizes when non-default sizes are used. The memory requirements for this could become prohibitive if your grid is very large.

wxGrid::AutoSizeColOrRow

void AutoSizeColOrRow(int n, bool setAsMin, bool column)

Common part of *AutoSizeColumn/Row()* or row?

wxGrid::AutoSizeColumn

void AutoSizeColumn(int col, bool setAsMin = true)

Automatically sizes the column to fit its contents. If `setAsMin` is true the calculated width will also be set as the minimal width for the column.

Note

`wxGrid` sets up arrays to store individual row and column sizes when non-default sizes are used. The memory requirements for this could become prohibitive if your grid is very large.

wxGrid::AutoSizeColumns

void AutoSizeColumns(bool *setAsMin* = true)

Automatically sizes all columns to fit their contents. If `setAsMin` is true the calculated widths will also be set as the minimal widths for the columns.

Note

`wxGrid` sets up arrays to store individual row and column sizes when non-default sizes are used. The memory requirements for this could become prohibitive if your grid is very large.

wxGrid::AutoSizeRow

void AutoSizeRow(int *row*, bool *setAsMin* = true)

Automatically sizes the row to fit its contents. If `setAsMin` is true the calculated height will also be set as the minimal height for the row.

Note

`wxGrid` sets up arrays to store individual row and column sizes when non-default sizes are used. The memory requirements for this could become prohibitive if your grid is very large.

wxGrid::AutoSizeRows

void AutoSizeRows(bool *setAsMin* = true)

Automatically sizes all rows to fit their contents. If `setAsMin` is true the calculated heights will also be set as the minimal heights for the rows.

Note

`wxGrid` sets up arrays to store individual row and column sizes when non-default sizes are used. The memory requirements for this could become prohibitive if your grid is very large.

wxGrid::BeginBatch

void BeginBatch()

Increments the grid's batch count. When the count is greater than zero repainting of the grid is suppressed. Each call to `BeginBatch` must be matched by a later call to `wxGrid::EndBatch` (p. 630). Code that does a lot of grid modification can be enclosed

between `BeginBatch` and `EndBatch` calls to avoid screen flicker. The final `EndBatch` will cause the grid to be repainted.

`wxGrid::BlockToDeviceRect`

`wxRect BlockToDeviceRect(const wxGridCellCoords & topLeft, const wxGridCellCoords & bottomRight)`

This function returns the rectangle that encloses the block of cells limited by `TopLeft` and `BottomRight` cell in device coords and clipped to the client size of the grid window.

`wxGrid::CanDragColSize`

`bool CanDragColSize()`

Returns true if columns can be resized by dragging with the mouse. Columns can be resized by dragging the edges of their labels. If grid line dragging is enabled they can also be resized by dragging the right edge of the column in the grid cell area (see `wxGrid::EnableDragGridSize` (p. 630)).

`wxGrid::CanDragRowSize`

`bool CanDragRowSize()`

Returns true if rows can be resized by dragging with the mouse. Rows can be resized by dragging the edges of their labels. If grid line dragging is enabled they can also be resized by dragging the lower edge of the row in the grid cell area (see `wxGrid::EnableDragGridSize` (p. 630)).

`wxGrid::CanDragGridSize`

`bool CanDragGridSize()`

Return true if the dragging of grid lines to resize rows and columns is enabled or false otherwise.

`wxGrid::CanEnableCellControl`

`bool CanEnableCellControl() const`

Returns true if the in-place edit control for the current grid cell can be used and false otherwise (e.g. if the current cell is read-only).

`wxGrid::CanHaveAttributes`

`bool CanHaveAttributes()`

Do we have some place to store attributes in?

`wxGrid::CellToRect`

wxRect CellToRect(int row, int col)

wxRect CellToRect(const wxGridCellCoords& coords)

Return the rectangle corresponding to the grid cell's size and position in logical coordinates.

wxGrid::ClearGrid

void ClearGrid()

Clears all data in the underlying grid table and repaints the grid. The table is not deleted by this function. If you are using a derived table class then you need to override `wxGridTableBase::Clear` (p. 680) for this function to have any effect.

wxGrid::ClearSelection

void ClearSelection()

Deselects all cells that are currently selected.

wxGrid::CreateGrid

bool CreateGrid(int numRows, int numCols, wxGrid::wxGridSelectionMode selmode = wxGrid::wxGridSelectCells)

Creates a grid with the specified initial number of rows and columns. Call this directly after the grid constructor. When you use this function `wxGrid` will create and manage a simple table of string values for you. All of the grid data will be stored in memory.

For applications with more complex data types or relationships, or for dealing with very large datasets, you should derive your own grid table class and pass a table object to the grid with `wxGrid::SetTable` (p. 653).

wxGrid::DeleteCols

bool DeleteCols(int pos = 0, int numCols = 1, bool updateLabels = true)

Deletes one or more columns from a grid starting at the specified position and returns true if successful. The `updateLabels` argument is not used at present.

If you are using a derived grid table class you will need to override `wxGridTableBase::DeleteCols` (p. 681). See `wxGrid::InsertCols` (p. 640) for further information.

wxGrid::DeleteRows

bool DeleteRows(int pos = 0, int numRows = 1, bool updateLabels = true)

Deletes one or more rows from a grid starting at the specified position and returns true if successful. The `updateLabels` argument is not used at present.

If you are using a derived grid table class you will need to override `wxGridTableBase::DeleteRows` (p. 680). See `wxGrid::InsertRows` (p. 640) for further information.

wxGrid::DisableCellEditControl

void DisableCellEditControl()

Disables in-place editing of grid cells. Equivalent to calling `EnableCellEditControl(false)`.

wxGrid::DisableDragColSize

void DisableDragColSize()

Disables column sizing by dragging with the mouse. Equivalent to passing false to `wxGrid::EnableDragColSize` (p. 630).

wxGrid::DisableDragGridSize

void DisableDragGridSize()

Disable mouse dragging of grid lines to resize rows and columns. Equivalent to passing false to `wxGrid::EnableDragGridSize` (p. 630)

wxGrid::DisableDragRowSize

void DisableDragRowSize()

Disables row sizing by dragging with the mouse. Equivalent to passing false to `wxGrid::EnableDragRowSize` (p. 630).

wxGrid::EnableCellEditControl

void EnableCellEditControl(bool enable = true)

Enables or disables in-place editing of grid cell data. The grid will issue either a `wxEVT_GRID_EDITOR_SHOWN` or `wxEVT_GRID_EDITOR_HIDDEN` event.

wxGrid::EnableDragColSize

void EnableDragColSize(bool enable = true)

Enables or disables column sizing by dragging with the mouse.

wxGrid::EnableDragGridSize

void EnableDragGridSize(bool enable = true)

Enables or disables row and column resizing by dragging gridlines with the mouse.

wxGrid::EnableDragRowSize**void EnableDragRowSize**(bool *enable* = true)

Enables or disables row sizing by dragging with the mouse.

wxGrid::EnableEditing**void EnableEditing**(bool *edit*)

If the edit argument is false this function sets the whole grid as read-only. If the argument is true the grid is set to the default state where cells may be editable. In the default state you can set single grid cells and whole rows and columns to be editable or read-only via `wxGridCellAttribute::SetReadOnly` (p. 655). For single cells you can also use the shortcut function `wxGrid::SetReadOnly` (p. 650).

For more information about controlling grid cell attributes see the `wxGridCellAttr` (p. 654) cell attribute class and the `wxGrid` classes overview (p. **Error! Bookmark not defined.**).

wxGrid::EnableGridLines**void EnableGridLines**(bool *enable* = true)

Turns the drawing of grid lines on or off.

wxGrid::EndBatch**void EndBatch**()

Decrements the grid's batch count. When the count is greater than zero repainting of the grid is suppressed. Each previous call to `wxGrid::BeginBatch` (p. 627) must be matched by a later call to `EndBatch`. Code that does a lot of grid modification can be enclosed between `BeginBatch` and `EndBatch` calls to avoid screen flicker. The final `EndBatch` will cause the grid to be repainted.

wxGrid::Fit**void Fit**()

Overridden wxWindow method.

wxGrid::ForceRefresh**void ForceRefresh**()

Causes immediate repainting of the grid. Use this instead of the usual `wxWindow::Refresh`.

wxGrid::GetBatchCount

int GetBatchCount()

Returns the number of times that *wxGrid::BeginBatch* (p. 627) has been called without (yet) matching calls to *wxGrid::EndBatch* (p. 630). While the grid's batch count is greater than zero the display will not be updated.

wxGrid::GetCellAlignment

void GetCellAlignment(int row, int col, int* horiz, int* vert)

Sets the arguments to the horizontal and vertical text alignment values for the grid cell at the specified location.

Horizontal alignment will be one of *wxALIGN_LEFT*, *wxALIGN_CENTRE* or *wxALIGN_RIGHT*.

Vertical alignment will be one of *wxALIGN_TOP*, *wxALIGN_CENTRE* or *wxALIGN_BOTTOM*.

wxPerl note: This method only takes the parameters *row* and *col* and returns a 2-element list (*horiz*, *vert*).

wxGrid::GetCellBackgroundColour

wxColour GetCellBackgroundColour(int row, int col)

Returns the background colour of the cell at the specified location.

wxGrid::GetCellEditor

wxGridCellEditor* GetCellEditor(int row, int col)

Returns a pointer to the editor for the cell at the specified location.

See *wxGridCellEditor* (p. 661) and the *wxGrid overview* (p. **Error! Bookmark not defined.**) for more information about cell editors and renderers.

wxGrid::GetCellFont

wxFont GetCellFont(int row, int col)

Returns the font for text in the grid cell at the specified location.

wxGrid::GetCellRenderer

wxGridCellRenderer* GetCellRenderer(int row, int col)

Returns a pointer to the renderer for the grid cell at the specified location.

See *wxGridCellRenderer* (p. 677) and the *wxGrid overview* (p. **Error! Bookmark not defined.**) for more information about cell editors and renderers.

wxGrid::GetCellTextColour**wxColour GetCellTextColour**(int row, int col)

Returns the text colour for the grid cell at the specified location.

wxGrid::GetCellValue**wxString GetCellValue**(int row, int col)**wxString GetCellValue**(const wxGridCellCoords&coords)

Returns the string contained in the cell at the specified location. For simple applications where a grid object automatically uses a default grid table of string values you use this function together with *wxGrid::SetCellValue* (p. 646) to access cell values.

For more complex applications where you have derived your own grid table class that contains various data types (e.g. numeric, boolean or user-defined custom types) then you only use this function for those cells that contain string values.

See *wxGridTableBase::CanGetValueAs* (p. 679) and the *wxGrid overview* (p. **Error! Bookmark not defined.**) for more information.

wxGrid::GetColLeft**int GetColLeft**(int col) const**wxGrid::GetColLabelAlignment****void GetColLabelAlignment**(int* horiz, int* vert)

Sets the arguments to the current column label alignment values.

Horizontal alignment will be one of wxALIGN_LEFT, wxALIGN_CENTRE or wxALIGN_RIGHT.

Vertical alignment will be one of wxALIGN_TOP, wxALIGN_CENTRE or wxALIGN_BOTTOM.

wxPerl note: This method takes no parameters and returns a 2-element list (horiz, vert).

wxGrid::GetColLabelSize**int GetColLabelSize**()

Returns the current height of the column labels.

wxGrid::GetColLabelValue**wxString GetColLabelValue**(int col)

Returns the specified column label. The default grid table class provides column labels of the form A,B...Z,AA,AB...ZZ,AAA... If you are using a custom grid table you can override `wxGridTableBase::GetColLabelValue` (p. 681) to provide your own labels.

wxGrid::GetColMinimalAcceptableWidth

int GetColMinimalAcceptableWidth()

This returns the value of the lowest column width that can be handled correctly. See member `SetColMinimalAcceptableWidth` (p. 648) for details.

wxGrid::GetColMinimalWidth

int GetColMinimalWidth(int col) const

Get the minimal width of the given column/row.

wxGrid::GetColRight

int GetColRight(int col) const

wxGrid::GetColSize

int GetColSize(int col)

Returns the width of the specified column.

wxGrid::GetDefaultCellAlignment

void GetDefaultCellAlignment(int* horiz, int* vert)

Sets the arguments to the current default horizontal and vertical text alignment values.

Horizontal alignment will be one of `wxALIGN_LEFT`, `wxALIGN_CENTRE` or `wxALIGN_RIGHT`.

Vertical alignment will be one of `wxALIGN_TOP`, `wxALIGN_CENTRE` or `wxALIGN_BOTTOM`.

wxGrid::GetDefaultCellBackgroundColour

wxColour GetDefaultCellBackgroundColour()

Returns the current default background colour for grid cells.

wxGrid::GetDefaultCellFont

wxFont GetDefaultCellFont()

Returns the current default font for grid cell text.

wxGrid::GetDefaultCellTextColour**wxColour GetDefaultCellTextColour()**

Returns the current default colour for grid cell text.

wxGrid::GetDefaultColLabelSize**int GetDefaultColLabelSize()**

Returns the default height for column labels.

wxGrid::GetDefaultColSize**int GetDefaultColSize()**

Returns the current default width for grid columns.

wxGrid::GetDefaultEditor**wxGridCellEditor* GetDefaultEditor() const**

Returns a pointer to the current default grid cell editor.

See *wxGridCellEditor* (p. 661) and the *wxGrid overview* (p. **Error! Bookmark not defined.**) for more information about cell editors and renderers.

wxGrid::GetDefaultEditorForCell**wxGridCellEditor* GetDefaultEditorForCell(int row, int col) const****wxGridCellEditor* GetDefaultEditorForCell(const wxGridCellCoords& c) const****wxGrid::GetDefaultEditorForType****wxGridCellEditor* GetDefaultEditorForType(const wxString& typeName) const****wxGrid::GetDefaultRenderer****wxGridCellRenderer* GetDefaultRenderer() const**

Returns a pointer to the current default grid cell renderer.

See *wxGridCellRenderer* (p. 677) and the *wxGrid overview* (p. **Error! Bookmark not defined.**) for more information about cell editors and renderers.

wxGrid::GetDefaultRendererForCell**wxGridCellRenderer* GetDefaultRendererForCell(int row, int col) const**

wxGrid::GetDefaultRendererForType

wxGridCellRenderer* GetDefaultRendererForType(const wxString& *typeName*)
const

wxGrid::GetDefaultRowLabelSize

int GetDefaultRowLabelSize()

Returns the default width for the row labels.

wxGrid::GetDefaultRowSize

int GetDefaultRowSize()

Returns the current default height for grid rows.

wxGrid::GetGridCursorCol

int GetGridCursorCol()

Returns the current grid cell column position.

wxGrid::GetGridCursorRow

int GetGridCursorRow()

Returns the current grid cell row position.

wxGrid::GetGridLineColour

wxColour GetGridLineColour()

Returns the colour used for grid lines.

wxGrid::GridLinesEnabled

bool GridLinesEnabled()

Returns true if drawing of grid lines is turned on, false otherwise.

wxGrid::GetLabelBackgroundColour

wxColour GetLabelBackgroundColour()

Returns the colour used for the background of row and column labels.

wxGrid::GetLabelFont

wxFont GetLabelFont()

Returns the font used for row and column labels.

wxGrid::GetLabelTextColour

wxColour GetLabelTextColour()

Returns the colour used for row and column label text.

wxGrid::GetNumberCols

int GetNumberCols()

Returns the total number of grid columns (actually the number of columns in the underlying grid table).

wxGrid::GetNumberRows

int GetNumberRows()

Returns the total number of grid rows (actually the number of rows in the underlying grid table).

wxGrid::GetOrCreateCellAttr

wxGridCellAttr* GetOrCreateCellAttr(int row, int col) const

wxGrid::GetRowMinimalAcceptableHeight

int GetRowMinimalAcceptableHeight()

This returns the value of the lowest row width that can be handled correctly. See member *SetRowMinimalAcceptableHeight* (p. 651) for details.

wxGrid::GetRowMinimalHeight

int GetRowMinimalHeight(int col) const

wxGrid::GetRowLabelAlignment

void GetRowLabelAlignment(int* horiz, int* vert)

Sets the arguments to the current row label alignment values.

Horizontal alignment will be one of wxLEFT, wxCENTRE or wxRIGHT.
Vertical alignment will be one of wxTOP, wxCENTRE or wxBOTTOM.

wxPerl note: This method takes no parameters and returns a 2-element list (*horiz*, *vert*).

wxGrid::GetRowLabelSize

int GetRowLabelSize()

Returns the current width of the row labels.

wxGrid::GetRowLabelValue**wxString GetRowLabelValue(int row)**

Returns the specified row label. The default grid table class provides numeric row labels. If you are using a custom grid table you can override *wxGridTableBase::GetRowLabelValue* (p. 681) to provide your own labels.

wxGrid::GetRowSize**int GetRowSize(int row)**

Returns the height of the specified row.

wxGrid::GetScrollLineX**int GetScrollLineX() const**

Returns the number of pixels per horizontal scroll increment. The default is 15.

See also

wxGrid::GetScrollLineY (p. 638), *wxGrid::SetScrollLineX* (p. 652),
wxGrid::SetScrollLineY (p. 652)

wxGrid::GetScrollLineY**int GetScrollLineY() const**

Returns the number of pixels per vertical scroll increment. The default is 15.

See also

wxGrid::GetScrollLineX (p. 638), *wxGrid::SetScrollLineX* (p. 652),
wxGrid::SetScrollLineY (p. 652)

wxGrid::GetSelectionMode**wxGrid::wxGridSelectionModes GetSelectionMode() const**

Returns the current selection mode, see *wxGrid::SetSelectionMode* (p. 653).

wxGrid::GetSelectedCells**wxGridCellCoordsArray GetSelectedCells() const**

Returns an array of singly selected cells.

wxGrid::GetSelectedCols**wxArrayInt GetSelectedCols() const**

Returns an array of selected cols.

wxGrid::GetSelectedRows**wxArrayInt GetSelectedRows() const**

Returns an array of selected rows.

wxGrid::GetSelectionBackground**wxColour GetSelectionBackground() const**

Access or update the selection fore/back colours

wxGrid::GetSelectionBlockTopLeft**wxGridCellCoordsArray GetSelectionBlockTopLeft() const**

Returns an array of the top left corners of blocks of selected cells, see *wxGrid::GetSelectionBlockBottomRight* (p. 639).

wxGrid::GetSelectionBlockBottomRight**wxGridCellCoordsArray GetSelectionBlockBottomRight() const**

Returns an array of the bottom right corners of blocks of selected cells, see *wxGrid::GetSelectionBlockTopLeft* (p. 639).

wxGrid::GetSelectionForeground**wxColour GetSelectionForeground() const****wxGrid::GetTable****wxGridTableBase * GetTable() const**

Returns a base pointer to the current table object.

wxGrid::GetViewWidth**int GetViewWidth()**

Returned number of whole cols visible.

wxGrid::HideCellEditControl

void HideCellEditControl()

Hides the in-place cell edit control.

wxGrid::InitColWidths**void InitColWidths()**

Init the m_colWidths/Rights arrays

wxGrid::InitRowHeights**void InitRowHeights()**

NB: *never* access m_row/col arrays directly because they are created on demand, *always* use accessor functions instead!

Init the m_rowHeights/Bottoms arrays with default values.

wxGrid::InsertCols**bool InsertCols(int pos = 0, int numCols = 1, bool updateLabels = true)**

Inserts one or more new columns into a grid with the first new column at the specified position and returns true if successful. The updateLabels argument is not used at present.

The sequence of actions begins with the grid object requesting the underlying grid table to insert new columns. If this is successful the table notifies the grid and the grid updates the display. For a default grid (one where you have called *wxGrid::CreateGrid* (p. 628)) this process is automatic. If you are using a custom grid table (specified with *wxGrid::SetTable* (p. 653)) then you must override *wxGridTableBase::InsertCols* (p. 681) in your derived table class.

wxGrid::InsertRows**bool InsertRows(int pos = 0, int numRows = 1, bool updateLabels = true)**

Inserts one or more new rows into a grid with the first new row at the specified position and returns true if successful. The updateLabels argument is not used at present.

The sequence of actions begins with the grid object requesting the underlying grid table to insert new rows. If this is successful the table notifies the grid and the grid updates the display. For a default grid (one where you have called *wxGrid::CreateGrid* (p. 628)) this process is automatic. If you are using a custom grid table (specified with *wxGrid::SetTable* (p. 653)) then you must override *wxGridTableBase::InsertRows* (p. 680) in your derived table class.

wxGrid::IsCellEditControlEnabled**bool IsCellEditControlEnabled() const**

Returns true if the in-place edit control is currently enabled.

wxGrid::IsCurrentCellReadOnly

bool IsCurrentCellReadOnly() const

Returns true if the current cell has been set to read-only (see *wxGrid::SetReadOnly* (p. 650)).

wxGrid::IsEditable

bool IsEditable()

Returns false if the whole grid has been set as read-only or true otherwise. See *wxGrid::EnableEditing* (p. 630) for more information about controlling the editing status of grid cells.

wxGrid::IsInSelection

bool IsInSelection(int row, int col) const

bool IsInSelection(const wxGridCellCoords& coords) const

Is this cell currently selected.

wxGrid::IsReadOnly

bool IsReadOnly(int row, int col) const

Returns true if the cell at the specified location can't be edited. See also *wxGrid::IsReadOnly* (p. 641).

wxGrid::IsSelection

bool IsSelection()

Returns true if there are currently rows, columns or blocks of cells selected.

wxGrid::IsVisible

bool IsVisible(int row, int col, bool wholeCellVisible = true)

bool IsVisible(const wxGridCellCoords& coords, bool wholeCellVisible = true)

Returns true if a cell is either wholly visible (the default) or at least partially visible in the grid window.

wxGrid::MakeCellVisible

void MakeCellVisible(int row, int col)

void MakeCellVisible(const wxGridCellCoords& coords)

Brings the specified cell into the visible grid cell area with minimal scrolling. Does nothing if the cell is already visible.

wxGrid::MoveCursorDown

bool MoveCursorDown(bool expandSelection)

Moves the grid cursor down by one row. If a block of cells was previously selected it will expand if the argument is true or be cleared if the argument is false.

Keyboard

This function is called for Down cursor key presses or Shift+Down to expand a selection.

wxGrid::MoveCursorLeft

bool MoveCursorLeft(bool expandSelection)

Moves the grid cursor left by one column. If a block of cells was previously selected it will expand if the argument is true or be cleared if the argument is false.

Keyboard

This function is called for Left cursor key presses or Shift+Left to expand a selection.

wxGrid::MoveCursorRight

bool MoveCursorRight(bool expandSelection)

Moves the grid cursor right by one column. If a block of cells was previously selected it will expand if the argument is true or be cleared if the argument is false.

Keyboard

This function is called for Right cursor key presses or Shift+Right to expand a selection.

wxGrid::MoveCursorUp

bool MoveCursorUp(bool expandSelection)

Moves the grid cursor up by one row. If a block of cells was previously selected it will expand if the argument is true or be cleared if the argument is false.

Keyboard

This function is called for Up cursor key presses or Shift+Up to expand a selection.

wxGrid::MoveCursorDownBlock

bool MoveCursorDownBlock(bool expandSelection)

Moves the grid cursor down in the current column such that it skips to the beginning or end of a block of non-empty cells. If a block of cells was previously selected it will

expand if the argument is true or be cleared if the argument is false.

Keyboard

This function is called for the Ctrl+Down key combination. Shift+Ctrl+Down expands a selection.

wxGrid::MoveCursorLeftBlock

bool MoveCursorLeftBlock(bool *expandSelection*)

Moves the grid cursor left in the current row such that it skips to the beginning or end of a block of non-empty cells. If a block of cells was previously selected it will expand if the argument is true or be cleared if the argument is false.

Keyboard

This function is called for the Ctrl+Left key combination. Shift+Ctrl+left expands a selection.

wxGrid::MoveCursorRightBlock

bool MoveCursorRightBlock(bool *expandSelection*)

Moves the grid cursor right in the current row such that it skips to the beginning or end of a block of non-empty cells. If a block of cells was previously selected it will expand if the argument is true or be cleared if the argument is false.

Keyboard

This function is called for the Ctrl+Right key combination. Shift+Ctrl+Right expands a selection.

wxGrid::MoveCursorUpBlock

bool MoveCursorUpBlock(bool *expandSelection*)

Moves the grid cursor up in the current column such that it skips to the beginning or end of a block of non-empty cells. If a block of cells was previously selected it will expand if the argument is true or be cleared if the argument is false.

Keyboard

This function is called for the Ctrl+Up key combination. Shift+Ctrl+Up expands a selection.

wxGrid::MovePageDown

bool MovePageDown()

Moves the grid cursor down by some number of rows so that the previous bottom visible row becomes the top visible row.

Keyboard

This function is called for PgDn keypresses.

wxGrid::MovePageUp**bool MovePageUp()**

Moves the grid cursor up by some number of rows so that the previous top visible row becomes the bottom visible row.

Keyboard

This function is called for PgUp keypresses.

wxGrid::RegisterDataType

void RegisterDataType(const wxString& typeName, wxGridCellRenderer* renderer, wxGridCellEditor* editor)

Methods for a registry for mapping data types to Renderers/Editors

wxGrid::SaveEditControlValue**void SaveEditControlValue()**

Sets the value of the current grid cell to the current in-place edit control value. This is called automatically when the grid cursor moves from the current cell to a new cell. It is also a good idea to call this function when closing a grid since any edits to the final cell location will not be saved otherwise.

wxGrid::SelectAll**void SelectAll()**

Selects all cells in the grid.

wxGrid::SelectBlock

void SelectBlock(int topRow, int leftCol, int bottomRow, int rightCol, bool addToSelected = false)

void SelectBlock(const wxGridCellCoords& topLeft, const wxGridCellCoords& bottomRight, bool addToSelected = false)

Selects a rectangular block of cells. If addToSelected is false then any existing selection will be deselected; if true the column will be added to the existing selection.

wxGrid::SelectCol

void SelectCol(int col, bool addToSelected = false)

Selects the specified column. If addToSelected is false then any existing selection will be deselected; if true the column will be added to the existing selection.

wxGrid::SelectionToDeviceRect**wxRect SelectionToDeviceRect()**

This function returns the rectangle that encloses the selected cells in device coords and clipped to the client size of the grid window.

wxGrid::SelectRow**void SelectRow(int row, bool addToSelected = false)**

Selects the specified row. If `addToSelected` is false then any existing selection will be deselected; if true the row will be added to the existing selection.

wxGrid::SetCellAlignment**void SetCellAlignment(int row, int col, int horiz, int vert)****void SetCellAlignment(int align, int row, int col)**

Sets the horizontal and vertical alignment for grid cell text at the specified location.

Horizontal alignment should be one of `wxALIGN_LEFT`, `wxALIGN_CENTRE` or `wxALIGN_RIGHT`.

Vertical alignment should be one of `wxALIGN_TOP`, `wxALIGN_CENTRE` or `wxALIGN_BOTTOM`.

wxGrid::SetCellBackgroundColour**void SetCellBackgroundColour(int row, int col, const wxColour& colour)****wxGrid::SetCellEditor****void SetCellEditor(int row, int col, wxGridCellEditor* editor)**

Sets the editor for the grid cell at the specified location. The grid will take ownership of the pointer.

See *wxGridCellEditor* (p. 661) and the *wxGrid overview* (p. **Error! Bookmark not defined.**) for more information about cell editors and renderers.

wxGrid::SetCellFont**void SetCellFont(int row, int col, const wxFont& font)**

Sets the font for text in the grid cell at the specified location.

wxGrid::SetCellRenderer**void SetCellRenderer(int row, int col, wxGridCellRenderer* renderer)**

Sets the renderer for the grid cell at the specified location. The grid will take ownership of the pointer.

See *wxGridCellRenderer* (p. 677) and the *wxGrid overview* (p. **Error! Bookmark not defined.**) for more information about cell editors and renderers.

wxGrid::SetCellTextColour

void SetCellTextColour(int row, int col, const wxColour& colour)

void SetCellTextColour(const wxColour& val, int row, int col)

void SetCellTextColour(const wxColour& colour)

Sets the text colour for the grid cell at the specified location.

wxGrid::SetCellValue

void SetCellValue(int row, int col, const wxString& s)

void SetCellValue(const wxGridCellCoords& coords, const wxString& s)

void SetCellValue(const wxString& val, int row, int col)

Sets the string value for the cell at the specified location. For simple applications where a grid object automatically uses a default grid table of string values you use this function together with *wxGrid::GetCellValue* (p. 632) to access cell values.

For more complex applications where you have derived your own grid table class that contains various data types (e.g. numeric, boolean or user-defined custom types) then you only use this function for those cells that contain string values.

The last form is for backward compatibility only.

See *wxGridTableBase::CanSetValueAs* (p. 679) and the *wxGrid overview* (p. **Error! Bookmark not defined.**) for more information.

wxGrid::SetColAttr

void SetColAttr(int col, wxGridCellAttr* attr)

Sets the cell attributes for all cells in the specified column.

For more information about controlling grid cell attributes see the *wxGridCellAttr* (p. 654) cell attribute class and the *wxGrid classes overview* (p. **Error! Bookmark not defined.**).

wxGrid::SetColFormatBool

void SetColFormatBool(int col)

Sets the specified column to display boolean values. *wxGrid* displays boolean values with a checkbox.

wxGrid::SetColFormatNumber**void SetColFormatNumber(int col)**

Sets the specified column to display integer values.

wxGrid::SetColFormatFloat**void SetColFormatFloat(int col, int width = -1, int precision = -1)**

Sets the specified column to display floating point values with the given width and precision.

wxGrid::SetColFormatCustom**void SetColFormatCustom(int col, const wxString& typeName)**

Sets the specified column to display data in a custom format. See the *wxGrid overview* (p. **Error! Bookmark not defined.**) for more information on working with custom data types.

wxGrid::SetColLabelAlignment**void SetColLabelAlignment(int horiz, int vert)**

Sets the horizontal and vertical alignment of column label text.

Horizontal alignment should be one of wxALIGN_LEFT, wxALIGN_CENTRE or wxALIGN_RIGHT.

Vertical alignment should be one of wxALIGN_TOP, wxALIGN_CENTRE or wxALIGN_BOTTOM.

wxGrid::SetColLabelSize**void SetColLabelSize(int height)**

Sets the height of the column labels.

wxGrid::SetColLabelValue**void SetColLabelValue(int col, const wxString& value)**

Set the value for the given column label. If you are using a derived grid table you must override *wxGridTableBase::SetColLabelValue* (p. 681) for this to have any effect.

wxGrid::SetColMinimalWidth**void SetColMinimalWidth(int col, int width)**

Sets the minimal width for the specified column. This should normally be called when

creating the grid because it will not resize a column that is already narrower than the minimal width. The width argument must be higher than the minimal acceptable column width, see `wxGrid::GetColMinimalAcceptableWidth` (p. 633).

wxGrid::SetColMinimalAcceptableWidth

void SetColMinimalAcceptableWidth(int width)

This modifies the minimum column width that can be handled correctly. Specifying a low value here allows smaller grid cells to be dealt with correctly. Specifying a value here which is much smaller than the actual minimum size will incur a performance penalty in the functions which perform grid cell index lookup on the basis of screen coordinates. This should normally be called when creating the grid because it will not resize existing columns with sizes smaller than the value specified here.

wxGrid::SetColSize

void SetColSize(int col, int width)

Sets the width of the specified column.

This function does not refresh the grid. If you are calling it outside of a BeginBatch / EndBatch block you can use `wxGrid::ForceRefresh` (p. 631) to see the changes.

Automatically sizes the column to fit its contents. If `setAsMin` is true the calculated width will also be set as the minimal width for the column.

Note

`wxGrid` sets up arrays to store individual row and column sizes when non-default sizes are used. The memory requirements for this could become prohibitive if your grid is very large.

wxGrid::SetDefaultCellAlignment

void SetDefaultCellAlignment(int horiz, int vert)

Sets the default horizontal and vertical alignment for grid cell text.

Horizontal alignment should be one of `wxALIGN_LEFT`, `wxALIGN_CENTRE` or `wxALIGN_RIGHT`.

Vertical alignment should be one of `wxALIGN_TOP`, `wxALIGN_CENTRE` or `wxALIGN_BOTTOM`.

wxGrid::SetDefaultCellBackgroundColour

void SetDefaultCellBackgroundColour(const wxColour& colour)

Sets the default background colour for grid cells.

wxGrid::SetDefaultCellFont

void SetDefaultCellFont(const wxFont& font)

Sets the default font to be used for grid cell text.

wxGrid::SetDefaultCellTextColour

void SetDefaultCellTextColour(const wxColour& colour)

Sets the current default colour for grid cell text.

wxGrid::SetDefaultEditor

void SetDefaultEditor(wxGridCellEditor* editor)

Sets the default editor for grid cells. The grid will take ownership of the pointer.

See *wxGridCellEditor* (p. 661) and the *wxGrid overview* (p. **Error! Bookmark not defined.**) for more information about cell editors and renderers.

wxGrid::SetDefaultRenderer

void SetDefaultRenderer(wxGridCellRenderer* renderer)

Sets the default renderer for grid cells. The grid will take ownership of the pointer.

See *wxGridCellRenderer* (p. 677) and the *wxGrid overview* (p. **Error! Bookmark not defined.**) for more information about cell editors and renderers.

wxGrid::SetDefaultColSize

void SetDefaultColSize(int width, bool resizeExistingCols = false)

Sets the default width for columns in the grid. This will only affect columns subsequently added to the grid unless *resizeExistingCols* is true.

wxGrid::SetDefaultRowSize

void SetDefaultRowSize(int height, bool resizeExistingRows = false)

Sets the default height for rows in the grid. This will only affect rows subsequently added to the grid unless *resizeExistingRows* is true.

wxGrid::SetGridCursor

void SetGridCursor(int row, int col)

Set the grid cursor to the specified cell. This function calls *wxGrid::MakeCellVisible* (p. 641).

wxGrid::SetGridLineColour

void SetGridLineColour(const wxColour& colour)

Sets the colour used to draw grid lines.

wxGrid::SetLabelBackgroundColour

void SetLabelBackgroundColour(const wxColour& colour)

Sets the background colour for row and column labels.

wxGrid::SetLabelFont

void SetLabelFont(const wxFont& font)

Sets the font for row and column labels.

wxGrid::SetLabelTextColour

void SetLabelTextColour(const wxColour& colour)

Sets the colour for row and column label text.

wxGrid::SetMargins

void SetMargins(int extraWidth, int extraHeight)

A grid may occupy more space than needed for its rows/columns. This function allows to set how big this extra space is

wxGrid::SetOrCalcColumnSizes

int SetOrCalcColumnSizes(bool calcOnly, bool setAsMin = true)

Common part of AutoSizeColumn/Row() and GetBestSize()

wxGrid::SetOrCalcRowSizes

int SetOrCalcRowSizes(bool calcOnly, bool setAsMin = true)

wxGrid::SetReadOnly

void SetReadOnly(int row, int col, bool isReadOnly = true)

Makes the cell at the specified location read-only or editable. See also *wxGrid::IsReadOnly* (p. 641).

wxGrid::SetRowAttr

void SetRowAttr(int row, wxGridCellAttr* attr)

Sets the cell attributes for all cells in the specified row. See the *wxGridCellAttr* (p. 654) class for more information about controlling cell attributes.

wxGrid::SetRowLabelAlignment

void SetRowLabelAlignment(int horiz, int vert)

Sets the horizontal and vertical alignment of row label text.

Horizontal alignment should be one of *wxALIGN_LEFT*, *wxALIGN_CENTRE* or *wxALIGN_RIGHT*.

Vertical alignment should be one of *wxALIGN_TOP*, *wxALIGN_CENTRE* or *wxALIGN_BOTTOM*.

wxGrid::SetRowLabelSize

void SetRowLabelSize(int width)

Sets the width of the row labels.

wxGrid::SetRowLabelValue

void SetRowLabelValue(int row, const wxString& value)

Set the value for the given row label. If you are using a derived grid table you must override *wxGridTableBase::SetRowLabelValue* (p. 681) for this to have any effect.

wxGrid::SetRowMinimalHeight

void SetRowMinimalHeight(int row, int height)

Sets the minimal height for the specified row. This should normally be called when creating the grid because it will not resize a row that is already shorter than the minimal height. The height argument must be higher than the minimal acceptable row height, see *wxGrid::GetRowMinimalAcceptableHeight* (p. 637).

wxGrid::SetRowMinimalAcceptableHeight

void SetRowMinimalAcceptableHeight(int height)

This modifies the minimum row width that can be handled correctly. Specifying a low value here allows smaller grid cells to be dealt with correctly. Specifying a value here which is much smaller than the actual minimum size will incur a performance penalty in the functions which perform grid cell index lookup on the basis of screen coordinates. This should normally be called when creating the grid because it will not resize existing rows with sizes smaller than the value specified here.

wxGrid::SetRowSize

void SetRowSize(int row, int height)

Sets the height of the specified row.

This function does not refresh the grid. If you are calling it outside of a `BeginBatch / EndBatch` block you can use `wxGrid::ForceRefresh` (p. 631) to see the changes.

Automatically sizes the column to fit its contents. If `setAsMin` is true the calculated width will also be set as the minimal width for the column.

Note

`wxGrid` sets up arrays to store individual row and column sizes when non-default sizes are used. The memory requirements for this could become prohibitive if your grid is very large.

wxGrid::SetScrollLineX

void SetScrollLineX(int x)

Sets the number of pixels per horizontal scroll increment. The default is 15. Sometimes `wxGrid` has trouble setting the scrollbars correctly due to rounding errors: setting this to 1 can help.

See also

`wxGrid::GetScrollLineX` (p. 638), `wxGrid::GetScrollLineY` (p. 638),
`wxGrid::SetScrollLineY` (p. 652)

wxGrid::SetScrollLineY

void SetScrollLineY(int y)

Sets the number of pixels per vertical scroll increment. The default is 15. Sometimes `wxGrid` has trouble setting the scrollbars correctly due to rounding errors: setting this to 1 can help.

See also

`wxGrid::GetScrollLineX` (p. 638), `wxGrid::GetScrollLineY` (p. 638),
`wxGrid::SetScrollLineX` (p. 652)

wxGrid::SetSelectionBackground

void SetSelectionBackground(const wxColour& c)

wxGrid::SetSelectionForeground

void SetSelectionForeground(const wxColour& c)

wxGrid::SetSelectionMode

void SetSelectionMode(wxGrid::wxGridSelectionModes selmode)

Set the selection behaviour of the grid.

Parameters

wxGrid::wxGridSelectCells

The default mode where individual cells are selected.

wxGrid::wxGridSelectRows

Selections will consist of whole rows.

wxGrid::wxGridSelectColumns

Selections will consist of whole columns.

wxGrid::SetTable

**bool SetTable(wxGridTableBase* table, bool takeOwnership = false,
wxGrid::wxGridSelectionModes selmode = wxGrid::wxGridSelectCells)**

Passes a pointer to a custom grid table to be used by the grid. This should be called after the grid constructor and before using the grid object. If takeOwnership is set to true then the table will be deleted by the wxGrid destructor.

Use this function instead of *wxGrid::CreateGrid* (p. 628) when your application involves complex or non-string data or data sets that are too large to fit wholly in memory.

wxGrid::ShowCellEditControl

void ShowCellEditControl()

Displays the in-place cell edit control for the current cell.

wxGrid::XToCol

int XToCol(int x)

Returns the grid column that corresponds to the logical x coordinate. Returns `wxNOT_FOUND` if there is no column at the x position.

wxGrid::XToEdgeOfCol

int XToEdgeOfCol(int x)

Returns the column whose right hand edge is close to the given logical x position. If no column edge is near to this position `wxNOT_FOUND` is returned.

wxGrid::YToEdgeOfRow

int YToEdgeOfRow(int y)

Returns the row whose bottom edge is close to the given logical y position. If no row edge is near to this position `wxNOT_FOUND` is returned.

wxGrid::YToRow**int YToRow(int y)**

Returns the grid row that corresponds to the logical y coordinate. Returns `wxNOT_FOUND` if there is no row at the y position.

wxGridCellAttr

This class can be used to alter the cells' appearance in the grid by changing their colour/font/... from default. An object of this class may be returned by `wxGridTable::GetAttr()`.

Derived from

No base class

Include files

<wx/grid.h>

wxGridCellAttr::wxGridCellAttr**wxGridCellAttr()**

Default constructor.

wxGridCellAttr(const wxColour& colText, const wxColour& colBack, const wxFont& font, int hAlign, int vAlign)

VZ: considering the number of members `wxGridCellAttr` has now, this ctor seems to be pretty useless... may be we should just remove it?

wxGridCellAttr::Clone**wxGridCellAttr* Clone() const**

Creates a new copy of this object.

wxGridCellAttr::IncRef**void IncRef()**

This class is ref counted: it is created with ref count of 1, so calling `DecRef()` once will

delete it. Calling `IncRef()` allows to lock it until the matching `DecRef()` is called

wxGridCellAttr::DecRef

void DecRef()

wxGridCellAttr::SetTextColour

void SetTextColour(const wxColour& colText)

Sets the text colour.

wxGridCellAttr::SetBackgroundColour

void SetBackgroundColour(const wxColour& colBack)

Sets the background colour.

wxGridCellAttr::SetFont

void SetFont(const wxFont& font)

Sets the font.

wxGridCellAttr::SetAlignment

void SetAlignment(int hAlign, int vAlign)

Sets the alignment.

wxGridCellAttr::SetReadOnly

void SetReadOnly(bool isReadOnly = true)

wxGridCellAttr::SetRenderer

void SetRenderer(wxGridCellRenderer* renderer)

takes ownership of the pointer

wxGridCellAttr::SetEditor

void SetEditor(wxGridCellEditor* editor)

wxGridCellAttr::HasTextColour

bool HasTextColour() const

accessors

wxGridCellAttr::HasBackgroundColour

bool HasBackgroundColour() const

wxGridCellAttr::HasFont

bool HasFont() const

wxGridCellAttr::HasAlignment

bool HasAlignment() const

wxGridCellAttr::HasRenderer

bool HasRenderer() const

wxGridCellAttr::HasEditor

bool HasEditor() const

wxGridCellAttr::GetTextColour

const wxColour& GetTextColour() const

wxGridCellAttr::GetBackgroundColour

const wxColour& GetBackgroundColour() const

wxGridCellAttr::GetFont

const wxFont& GetFont() const

wxGridCellAttr::GetAlignment

void GetAlignment(int* hAlign, int* vAlign) const

wxPerl note: This method takes no parameters and returns a 2-element list (*hAlign*, *vAlign*).

wxGridCellAttr::GetRenderer

wxGridCellRenderer* GetRenderer(wxGrid* grid, int row, int col) const

wxGridCellAttr::GetEditor

wxGridCellEditor* GetEditor(wxGrid* grid, int row, int col) const

wxGridCellAttr::IsReadOnly**bool IsReadOnly() const****wxGridCellAttr::SetDefAttr****void SetDefAttr(wxGridCellAttr* defAttr)****wxGridBagSizer**

A *wxSizer* (p. **Error! Bookmark not defined.**) that can lay out items in a virtual grid like a *wxFlexGridSizer* (p. 557) but in this case explicit positioning of the items is allowed using *wxGBPosition* (p. 605), and items can optionally span more than one row and/or column using *wxGBSpan* (p. 608).

Derived from*wxFlexGridSizer* (p. 557)*wxGridSizer* (p. 682)*wxSizer* (p. **Error! Bookmark not defined.**)*wxObject* (p. **Error! Bookmark not defined.**)**Include files**

<wx/gbsizer.h>

wxGridBagSizer::wxGridBagSizer**wxGridBagSizer(int vgap = 0, int hgap = 0)**

Constructor, with optional parameters to specify the gap between the rows and columns.

wxGridBagSizer::Add**wxSizerItem* Add(wxWindow* window, const wxGBPosition& pos, const wxGBSpan& span = wxDefaultSpan, int flag = 0, int border = 0, wxObject* userData = NULL)****wxSizerItem* Add(wxSizer* sizer, const wxGBPosition& pos, const wxGBSpan& span = wxDefaultSpan, int flag = 0, int border = 0, wxObject* userData = NULL)****wxSizerItem* Add(int width, int height, const wxGBPosition& pos, const wxGBSpan& span = wxDefaultSpan, int flag = 0, int border = 0, wxObject* userData = NULL)****wxSizerItem* Add(wxGBSizerItem* item)**

The Add methods return a valid pointer if the item was successfully placed at the given position, NULL if something was already there.

wxGridBagSizer::CalcMin**wxSize CalcMin()**

Called when the managed size of the sizer is needed or when layout needs done.

wxGridBagSizer::CheckForIntersection**bool CheckForIntersection(wxGBSizerItem* item, wxGBSizerItem* excludeItem = NULL)****bool CheckForIntersection(const wxGBPosition& pos, const wxGBSpan& span, wxGBSizerItem* excludeItem = NULL)**

Look at all items and see if any intersect (or would overlap) the given item. Returns true if so, false if there would be no overlap. If an excludeItem is given then it will not be checked for intersection, for example it may be the item we are checking the position of.

wxGridBagSizer::FindItem**wxGBSizerItem* FindItem(wxWindow* window)****wxGBSizerItem* FindItem(wxSizer* sizer)**

Find the sizer item for the given window or subsizer, returns NULL if not found. (non-recursive)

wxGridBagSizer::FindItemAtPoint**wxGBSizerItem* FindItemAtPoint(const wxPoint& pt)**

Return the sizer item located at the point given in pt, or NULL if there is no item at that point. The (x,y) coordinates in pt correspond to the client coordinates of the window using the sizer for layout. (non-recursive)

wxGridBagSizer::FindItemAtPosition**wxGBSizerItem* FindItemAtPosition(const wxGBPosition& pos)**

Return the sizer item for the given grid cell, or NULL if there is no item at that position. (non-recursive)

wxGridBagSizer::FindItemWithData**wxGBSizerItem* FindItemWithData(const wxObject* userData)**

Return the sizer item that has a matching user data (it only compares pointer values) or NULL if not found. (non-recursive)

wxGridBagSizer::GetCellSize

wxSize GetCellSize(int row, int col) const

Get the size of the specified cell, including hgap and vgap. Only valid after a Layout.

wxGridBagSizer::GetEmptyCellSize

wxSize GetEmptyCellSize() const

Get the size used for cells in the grid with no item.

wxGridBagSizer::GetItemPosition

wxGBPosition GetItemPosition(wxWindow* window)

wxGBPosition GetItemPosition(wxSizer* sizer)

wxGBPosition GetItemPosition(size_t index)

Get the grid position of the specified item.

wxGridBagSizer::GetItemSpan

wxGBSpan GetItemSpan(wxWindow* window)

wxGBSpan GetItemSpan(wxSizer* sizer)

wxGBSpan GetItemSpan(size_t index)

Get the row/col spanning of the specified item

wxGridBagSizer::RecalcSizes

void RecalcSizes()

Called when the managed size of the sizer is needed or when layout needs done.

wxGridBagSizer::SetEmptyCellSize

void SetEmptyCellSize(const wxSize& sz)

Set the size used for cells in the grid with no item.

wxGridBagSizer::SetItemPosition

bool SetItemPosition(wxWindow* window, const wxGBPosition& pos)

bool SetItemPosition(wxSizer* sizer, const wxGBPosition& pos)

bool SetItemPosition(size_t index, const wxGBPosition& pos)

Set the grid position of the specified item. Returns true on success. If the move is not allowed (because an item is already there) then false is returned.

wxGridBagSizer::SetItemSpan**bool SetItemSpan(wxWindow* window, const wxGBSpan& span)****bool SetItemSpan(wxSizer* sizer, const wxGBSpan& span)****bool SetItemSpan(size_t index, const wxGBSpan& span)**

Set the row/col spanning of the specified item. Returns true on success. If the move is not allowed (because an item is already there) then false is returned.

wxGridCellBoolEditor

The editor for boolean data.

Derived from

wxGridCellEditor (p. 661)

See also

wxGridCellEditor (p. 661), *wxGridCellFloatEditor* (p. 663), *wxGridCellNumberEditor* (p. 664), *wxGridCellTextEditor* (p. 665), *wxGridCellChoiceEditor* (p. 660)

Include files

<wx/grid.h>

wxGridCellBoolEditor::wxGridCellBoolEditor**wxGridCellBoolEditor()**

Default constructor.

wxGridCellChoiceEditor

The editor for string data allowing to choose from a list of strings.

Derived from

wxGridCellEditor (p. 661)

See also

wxGridCellEditor (p. 661), *wxGridCellFloatEditor* (p. 663), *wxGridCellBoolEditor* (p. 660), *wxGridCellTextEditor* (p. 665), *wxGridCellNumberEditor* (p. 664)

wxGridCellChoiceEditor::wxGridCellChoiceEditor

wxGridCellChoiceEditor(*size_t count* = 0, **const wxString** choices[] = NULL, **bool allowOthers** = false)

wxGridCellChoiceEditor(**const wxArrayString&** choices, **bool allowOthers** = false)
count

Number of strings from which the user can choose.

choices

An array of strings from which the user can choose.

allowOthers

If *allowOthers* is true, the user can type a string not in choices array.

wxGridCellChoiceEditor::SetParameters

void SetParameters(**const wxString&** *params*)

Parameters string format is "item1[,item2[...itemN]]"

wxGridCellEditor

This class is responsible for providing and manipulating the in-place edit controls for the grid. Instances of `wxGridCellEditor` (actually, instances of derived classes since it is an abstract class) can be associated with the cell attributes for individual cells, rows, columns, or even for the entire grid.

Derived from

`wxGridCellWorker`

See also

`wxGridCellTextEditor` (p. 665), `wxGridCellFloatEditor` (p. 663), `wxGridCellBoolEditor` (p. 660), `wxGridCellNumberEditor` (p. 664), `wxGridCellChoiceEditor` (p. 660)

Include files

<wx/grid.h>

wxGridCellEditor::wxGridCellEditor

wxGridCellEditor()

wxGridCellEditor::IsCreated

bool IsCreated()

wxGridCellEditor::Create

void Create(wxWindow* *parent*, wxWindowID *id*, wxEvtHandler* *evtHandler*)

Creates the actual edit control.

wxGridCellEditor::SetSize

void SetSize(const wxRect& *rect*)

Size and position the edit control.

wxGridCellEditor::Show

void Show(bool *show*, wxGridCellAttr* *attr* = NULL)

Show or hide the edit control, use the specified attributes to set colours/fonts for it.

wxGridCellEditor::PaintBackground

void PaintBackground(const wxRect& *rectCell*, wxGridCellAttr* *attr*)

Draws the part of the cell not occupied by the control: the base class version just fills it with background colour from the attribute.

wxGridCellEditor::BeginEdit

void BeginEdit(int *row*, int *col*, wxGrid* *grid*)

Fetch the value from the table and prepare the edit control to begin editing. Set the focus to the edit control.

wxGridCellEditor::EndEdit

bool EndEdit(int *row*, int *col*, wxGrid* *grid*)

Complete the editing of the current cell. Returns true if the value has changed. If necessary, the control may be destroyed.

wxGridCellEditor::Reset

void Reset()

Reset the value in the control back to its starting value.

wxGridCellEditor::StartingKey

void StartingKey(wxKeyEvent& *event*)

If the editor is enabled by pressing keys on the grid, this will be called to let the editor do something about that first key if desired.

wxGridCellEditor::StartingClick

void StartingClick()

If the editor is enabled by clicking on the cell, this method will be called.

wxGridCellEditor::HandleReturn

void HandleReturn(wxKeyEvent& event)

Some types of controls on some platforms may need some help with the Return key.

wxGridCellEditor::Destroy

void Destroy()

Final cleanup.

wxGridCellEditor::Clone

wxGridCellEditor* Clone() const

Create a new object which is the copy of this one.

wxGridCellEditor::~~wxGridCellEditor

~wxGridCellEditor()

The dtor is private because only DecRef() can delete us.

wxGridCellFloatEditor

The editor for floating point numbers data.

Derived from

wxGridCellTextEditor (p. 665)

wxGridCellEditor (p. 661)

See also

wxGridCellEditor (p. 661), *wxGridCellNumberEditor* (p. 664), *wxGridCellBoolEditor* (p. 660), *wxGridCellTextEditor* (p. 665), *wxGridCellChoiceEditor* (p. 660)

Include files

<wx/grid.h>

wxGridCellFloatEditor::wxGridCellFloatEditor**wxGridCellFloatEditor**(int *width* = -1, int *precision* = -1)*width*

Minimum number of characters to be shown.

precision

Number of digits after the decimal dot.

wxGridCellFloatEditor::SetParameters**void SetParameters**(const wxString& *params*)

Parameters string format is "width,precision"

wxGridCellNumberEditor

The editor for numeric integer data.

Derived from*wxGridCellTextEditor* (p. 665)*wxGridCellEditor* (p. 661)**See also***wxGridCellEditor* (p. 661), *wxGridCellFloatEditor* (p. 663), *wxGridCellBoolEditor* (p. 660), *wxGridCellTextEditor* (p. 665), *wxGridCellChoiceEditor* (p. 660)**Include files**

<wx/grid.h>

wxGridCellNumberEditor::wxGridCellNumberEditor**wxGridCellNumberEditor**(int *min* = -1, int *max* = -1)

Allows to specify the range for acceptable data; if min == max == -1, no range checking is done

wxGridCellNumberEditor::GetString**wxString GetString**() const

String representation of the value.

wxGridCellNumberEditor::HasRange

bool HasRange() const

If the return value is true, the editor uses a wxSpinCtrl to get user input, otherwise it uses a wxTextCtrl.

wxGridCellNumberEditor::SetParameters

void SetParameters(const wxString& params)

Parameters string format is "min,max".

wxGridCellTextEditor

The editor for string/text data.

Derived from

wxGridCellEditor (p. 661)

See also

wxGridCellEditor (p. 661), *wxGridCellFloatEditor* (p. 663), *wxGridCellBoolEditor* (p. 660), *wxGridCellNumberEditor* (p. 664), *wxGridCellChoiceEditor* (p. 660)

Include files

<wx/grid.h>

wxGridCellTextEditor::wxGridCellTextEditor

wxGridCellTextEditor()

Default constructor.

wxGridCellTextEditor::SetParameters

void SetParameters(const wxString& params)

The parameters string format is "n" where n is a number representing the maximum width.

wxGridEditorCreatedEvent

Derived from

wxCommandEvent (p. 184)

wxEvent (p. 487)

wxObject (p. **Error! Bookmark not defined.**)

Event handling

The event handler for the following functions takes a *wxGridEditorCreatedEvent* (p. 666) parameter. The `..._CMD_...` variants also take a window identifier.

EVT_GRID_EDITOR_CREATED(func) The editor for a cell was created. Processes a `wxEVT_GRID_EDITOR_CREATED`.

EVT_GRID_CMD_EDITOR_CREATED(id, func) The editor for a cell was created; variant taking a window identifier. Processes a `wxEVT_GRID_EDITOR_CREATED`.

Include files

`<wx/grid.h>`

wxGridEditorCreatedEvent::wxGridEditorCreatedEvent

wxGridEditorCreatedEvent()

Default constructor.

wxGridEditorCreatedEvent(int id, wxEventType type, wxObject* obj, int row, int col, wxControl* ctrl)

wxGridEditorCreatedEvent::GetCol

int GetCol()

Returns the column at which the event occurred.

wxGridEditorCreatedEvent::GetControl

wxControl* GetControl()

Returns the edit control.

wxGridEditorCreatedEvent::GetRow

int GetRow()

Returns the row at which the event occurred.

wxGridEditorCreatedEvent::SetCol

void SetCol(int col)

Sets the column at which the event occurred.

wxGridEditorCreatedEvent::SetControl

void SetControl(wxControl* *ctrl*)

Sets the edit control.

wxGridEditorCreatedEvent::SetRow

void SetRow(int *row*)

Sets the row at which the event occurred.

wxGridEvent

This event class contains information about various grid events.

Derived from

wxNotifyEvent (p. **Error! Bookmark not defined.**)

wxCommandEvent (p. 184)

wxEvent (p. 487)

wxObject (p. **Error! Bookmark not defined.**)

Include files

<wx/grid.h>

Event handling

The event handler for the following functions takes a *wxGridEvent* (p. 667) parameter. The ..._CMD_... variants also take a window identifier.

EVT_GRID_CELL_LEFT_CLICK(func) The user clicked a cell with the left mouse button. Processes a wxEVT_GRID_CELL_LEFT_CLICK.

EVT_GRID_CELL_RIGHT_CLICK(func) The user clicked a cell with the right mouse button. Processes a wxEVT_GRID_CELL_RIGHT_CLICK.

EVT_GRID_CELL_LEFT_DCLICK(func) The user double-clicked a cell with the left mouse button. Processes a wxEVT_GRID_CELL_LEFT_DCLICK.

EVT_GRID_CELL_RIGHT_DCLICK(func) The user double-clicked a cell with the right mouse button. Processes a wxEVT_GRID_CELL_RIGHT_DCLICK.

EVT_GRID_LABEL_LEFT_CLICK(func) The user clicked a label with the left mouse button. Processes a

`wxEVT_GRID_LABEL_LEFT_CLICK`.

EVT_GRID_LABEL_RIGHT_CLICK(func) The user clicked a label with the right mouse button. Processes a `wxEVT_GRID_LABEL_RIGHT_CLICK`.

EVT_GRID_LABEL_LEFT_DCLICK(func) The user double-clicked a label with the left mouse button. Processes a `wxEVT_GRID_LABEL_LEFT_DCLICK`.

EVT_GRID_LABEL_RIGHT_DCLICK(func) The user double-clicked a label with the right mouse button. Processes a `wxEVT_GRID_LABEL_RIGHT_DCLICK`.

EVT_GRID_CELL_CHANGE(func) The user changed the data in a cell. Processes a `wxEVT_GRID_CELL_CHANGE`.

EVT_GRID_SELECT_CELL(func) The user moved to, and selected a cell. Processes a `wxEVT_GRID_SELECT_CELL`.

EVT_GRID_EDITOR_HIDDEN(func) The editor for a cell was hidden. Processes a `wxEVT_GRID_EDITOR_HIDDEN`.

EVT_GRID_EDITOR_SHOWN(func) The editor for a cell was shown. Processes a `wxEVT_GRID_EDITOR_SHOWN`.

EVT_GRID_CMD_CELL_LEFT_CLICK(id, func) The user clicked a cell with the left mouse button; variant taking a window identifier. Processes a `wxEVT_GRID_CELL_LEFT_CLICK`.

EVT_GRID_CMD_CELL_RIGHT_CLICK(id, func) The user clicked a cell with the right mouse button; variant taking a window identifier. Processes a `wxEVT_GRID_CELL_RIGHT_CLICK`.

EVT_GRID_CMD_CELL_LEFT_DCLICK(id, func) The user double-clicked a cell with the left mouse button; variant taking a window identifier. Processes a `wxEVT_GRID_CELL_LEFT_DCLICK`.

EVT_GRID_CMD_CELL_RIGHT_DCLICK(id, func) The user double-clicked a cell with the right mouse button; variant taking a window identifier. Processes a `wxEVT_GRID_CELL_RIGHT_DCLICK`.

EVT_GRID_CMD_LABEL_LEFT_CLICK(id, func) The user clicked a label with the left mouse button; variant taking a window identifier. Processes a `wxEVT_GRID_LABEL_LEFT_CLICK`.

EVT_GRID_CMD_LABEL_RIGHT_CLICK(id, func) The user clicked a label with the right mouse button; variant taking a window

identifier. Processes a
wxEVT_GRID_LABEL_RIGHT_CLICK.

EVT_GRID_CMD_LABEL_LEFT_DCLICK(id, func) The user double-clicked a label with the left mouse button; variant taking a window identifier. Processes a wxEVT_GRID_LABEL_LEFT_DCLICK.

EVT_GRID_CMD_LABEL_RIGHT_DCLICK(id, func) The user double-clicked a label with the right mouse button; variant taking a window identifier. Processes a wxEVT_GRID_LABEL_RIGHT_DCLICK.

EVT_GRID_CMD_CELL_CHANGE(id, func) The user changed the data in a cell; variant taking a window identifier. Processes a wxEVT_GRID_CELL_CHANGE.

EVT_GRID_CMD_SELECT_CELL(id, func) The user moved to, and selected a cell; variant taking a window identifier. Processes a wxEVT_GRID_SELECT_CELL.

EVT_GRID_CMD_EDITOR_HIDDEN(id, func) The editor for a cell was hidden; variant taking a window identifier. Processes a wxEVT_GRID_EDITOR_HIDDEN.

EVT_GRID_CMD_EDITOR_SHOWN(id, func) The editor for a cell was shown; variant taking a window identifier. Processes a wxEVT_GRID_EDITOR_SHOWN.

wxGridEvent::wxGridEvent

wxGridEvent()

Default constructor.

wxGridEvent(int id, wxEventType type, wxObject* obj, int row = -1, int col = -1, int x = -1, int y = -1, bool sel = true, bool control = false, bool shift = false, bool alt = false, bool meta = false)

Parameters

wxGridEvent::AltDown

bool AltDown()

Returns true if the Alt key was down at the time of the event.

wxGridEvent::ControlDown

bool ControlDown()

Returns true if the Control key was down at the time of the event.

wxGridEvent::GetCol

int GetCol()

Column at which the event occurred.

wxGridEvent::GetPosition

wxPoint GetPosition()

Position in pixels at which the event occurred.

wxGridEvent::GetRow

int GetRow()

Row at which the event occurred.

wxGridEvent::MetaDown

bool MetaDown()

Returns true if the Meta key was down at the time of the event.

wxGridEvent::Selecting

bool Selecting()

Returns true if the user deselected a cell, false if the user deselected a cell.

wxGridEvent::ShiftDown

bool ShiftDown()

Returns true if the Shift key was down at the time of the event.

wxGridRangeSelectEvent

Derived from

wxNotifyEvent (p. **Error! Bookmark not defined.**)

wxCommandEvent (p. 184)

wxEvent (p. 487)

wxObject (p. **Error! Bookmark not defined.**)

Event handling

The event handler for the following functions takes a *wxGridRangeSelectEvent* (p. 671)

parameter. The ..._CMD_... variants also take a window identifier.

EVT_GRID_RANGE_SELECT(func) The user selected a group of contiguous cells. Processes a wxEVT_GRID_RANGE_SELECT.

EVT_GRID_CMD_RANGE_SELECT(func) The user selected a group of contiguous cells; variant taking a window identifier. Processes a wxEVT_GRID_RANGE_SELECT.

Include files

<wx/grid.h>

wxGridRangeSelectEvent::wxGridRangeSelectEvent

wxGridRangeSelectEvent()

Default constructor.

wxGridRangeSelectEvent(int id, wxEventType type, wxObject* obj, const wxGridCellCoords& topLeft, const wxGridCellCoords& bottomRight, bool sel = true, bool control = false, bool shift = false, bool alt = false, bool meta = false)

wxGridRangeSelectEvent::AltDown

bool AltDown()

Returns true if the Alt key was down at the time of the event.

wxGridRangeSelectEvent::ControlDown

bool ControlDown()

Returns true if the Control key was down at the time of the event.

wxGridRangeSelectEvent::GetBottomRightCoords

wxGridCellCoords GetBottomRightCoords()

Top left corner of the rectangular area that was (de)selected.

wxGridRangeSelectEvent::GetBottomRow

int GetBottomRow()

Bottom row of the rectangular area that was (de)selected.

wxGridRangeSelectEvent::GetLeftCol

int GetLeftCol()

Left column of the rectangular area that was (de)selected.

wxGridRangeSelectEvent::GetRightCol**int GetRightCol()**

Right column of the rectangular area that was (de)selected.

wxGridRangeSelectEvent::GetTopLeftCoords**wxGridCellCoords GetTopLeftCoords()**

Top left corner of the rectangular area that was (de)selected.

wxGridRangeSelectEvent::GetTopRow**int GetTopRow()**

Top row of the rectangular area that was (de)selected.

wxGridRangeSelectEvent::MetaDown**bool MetaDown()**

Returns true if the Meta key was down at the time of the event.

wxGridRangeSelectEvent::Selecting**bool Selecting()**

Returns true if the area was selected, false otherwise.

wxGridRangeSelectEvent::ShiftDown**bool ShiftDown()**

Returns true if the Shift key was down at the time of the event.

wxGridSizeEvent

This event class contains information about a row/column resize event.

Derived from

wxNotifyEvent (p. **Error! Bookmark not defined.**)

wxCommandEvent (p. 184)

wxEvent (p. 487)

wxObject (p. **Error! Bookmark not defined.**)

Include files

<wx/grid.h>

Event handling

The event handler for the following functions takes a *wxGridSizeEvent* (p. 673) parameter. The *..._CMD_...* variants also take a window identifier.

EVT_GRID_COL_SIZE(func)	The user resized a column by dragging it. Processes a <i>wxEVT_GRID_COL_SIZE</i> .
EVT_GRID_ROW_SIZE(func)	The user resized a row by dragging it. Processes a <i>wxEVT_GRID_ROW_SIZE</i> .
EVT_GRID_CMD_COL_SIZE(func)	The user resized a column by dragging it; variant taking a window identifier. Processes a <i>wxEVT_GRID_COL_SIZE</i> .
EVT_GRID_CMD_ROW_SIZE(func)	The user resized a row by dragging it; variant taking a window identifier. Processes a <i>wxEVT_GRID_ROW_SIZE</i> .

wxGridSizeEvent::wxGridSizeEvent

wxGridSizeEvent()

Default constructor.

wxGridSizeEvent(int id, wxEventType type, wxObject* obj, int rowOrCol = -1, int x = -1, int y = -1, bool control = false, bool shift = false, bool alt = false, bool meta = false)

wxGridSizeEvent::AltDown

bool AltDown()

Returns true if the Alt key was down at the time of the event.

wxGridSizeEvent::ControlDown

bool ControlDown()

Returns true if the Control key was down at the time of the event.

wxGridSizeEvent::GetPosition

wxPoint GetPosition()

Position in pixels at which the event occurred.

wxGridSizeEvent::GetRowOrCol

int GetRowOrCol()

Row or column at that was resized.

wxGridSizeEvent::MetaDown

bool MetaDown()

Returns true if the Meta key was down at the time of the event.

wxGridSizeEvent::ShiftDown

bool ShiftDown()

Returns true if the Shift key was down at the time of the event.

wxGridCellBoolRenderer

This class may be used to format boolean data in a cell. for string cells.

Derived from

wxGridCellRenderer (p. 677)

See also

wxGridCellRenderer (p. 677), *wxGridCellStringRenderer* (p. 678),
wxGridCellFloatRenderer (p. 675), *wxGridCellNumberRenderer* (p. 676)

Include files

<wx/grid.h>

wxGridCellBoolRenderer::wxGridCellBoolRenderer

wxGridCellBoolRenderer()

Default constructor

wxGridCellFloatRenderer

This class may be used to format floating point data in a cell.

Derived from

wxGridCellStringRenderer (p. 678)
wxGridCellRenderer (p. 677)

See also

wxGridCellRenderer (p. 677), *wxGridCellNumberRenderer* (p. 676),

wxGridCellStringRenderer (p. 678), *wxGridCellBoolRenderer* (p. 674)

Include files

<wx/grid.h>

wxGridCellFloatRenderer::wxGridCellFloatRenderer

wxGridCellFloatRenderer(int *width* = -1, int *precision* = -1)

width

Minimum number of characters to be shown.

precision

Number of digits after the decimal dot.

wxGridCellFloatRenderer::GetPrecision

int GetPrecision() const

Returns the precision (see *wxGridCellFloatRenderer* (p. 675)).

wxGridCellFloatRenderer::GetWidth

int GetWidth() const

Returns the width (see *wxGridCellFloatRenderer* (p. 675)).

wxGridCellFloatRenderer::SetParameters

void SetParameters(const wxString& *params*)

Parameters string format is "width[,precision]".

wxGridCellFloatRenderer::SetPrecision

void SetPrecision(int *precision*)

Sets the precision (see *wxGridCellFloatRenderer* (p. 675)).

wxGridCellFloatRenderer::SetWidth

void SetWidth(int *width*)

Sets the width (see *wxGridCellFloatRenderer* (p. 675))

wxGridCellNumberRenderer

This class may be used to format integer data in a cell.

Derived from

wxGridCellStringRenderer (p. 678)
wxGridCellRenderer (p. 677)

See also

wxGridCellRenderer (p. 677), *wxGridCellStringRenderer* (p. 678),
wxGridCellFloatRenderer (p. 675), *wxGridCellBoolRenderer* (p. 674)

Include files

<wx/grid.h>

wxGridCellNumberRenderer::wxGridCellNumberRenderer

wxGridCellNumberRenderer()

Default constructor

wxGridCellRenderer

This class is responsible for actually drawing the cell in the grid. You may pass it to the *wxGridCellAttr* (below) to change the format of one given cell or to *wxGrid::SetDefaultRenderer()* to change the view of all cells. This is an abstract class, and you will normally use one of the predefined derived classes or derive your own class from it.

Derived from

wxGridCellWorker

See also

wxGridCellStringRenderer (p. 678), *wxGridCellNumberRenderer* (p. 676),
wxGridCellFloatRenderer (p. 675), *wxGridCellBoolRenderer* (p. 674)

Include files

<wx/grid.h>

wxGridCellRenderer::Draw

void Draw(*wxGrid& grid*, *wxGridCellAttr& attr*, *wxDC& dc*, **const** *wxRect& rect*, **int**

row, **int** *col*, **bool** *isSelected*)

Draw the given cell on the provided DC inside the given rectangle using the style specified by the attribute and the default or selected state corresponding to the *isSelected* value.

This pure virtual function has a default implementation which will prepare the DC using the given attribute: it will draw the rectangle with the background colour from *attr* and set the text colour and font.

wxGridCellRenderer::GetBestSize

wxSize **GetBestSize**(**wxGrid&** *grid*, **wxGridCellAttr&** *attr*, **wxDC&** *dc*, **int** *row*, **int** *col*)

Get the preferred size of the cell for its contents.

wxGridCellRenderer::Clone

wxGridCellRenderer* **Clone**() **const**

wxGridCellStringRenderer

This class may be used to format string data in a cell; it is the default for string cells.

Derived from

wxGridCellRenderer (p. 677)

See also

wxGridCellRenderer (p. 677), *wxGridCellNumberRenderer* (p. 676),
wxGridCellFloatRenderer (p. 675), *wxGridCellBoolRenderer* (p. 674)

Include files

<wx/grid.h>

wxGridCellStringRenderer::wxGridCellStringRenderer

wxGridCellStringRenderer()

Default constructor

wxGridTableBase

Grid table classes.

Derived from

wxObject (p. **Error! Bookmark not defined.**)

Include files

<wx/grid.h>

wxGridTableBase::wxGridTableBase

wxGridTableBase()

wxGridTableBase::~~wxGridTableBase

~wxGridTableBase()

wxGridTableBase::GetNumberRows

int GetNumberRows()

You must override these functions in a derived table class.

wxGridTableBase::GetNumberCols

int GetNumberCols()

wxGridTableBase::IsEmptyCell

bool IsEmptyCell(int row, int col)

wxGridTableBase::GetValue

wxString GetValue(int row, int col)

wxGridTableBase::SetValue

void SetValue(int row, int col, const wxString& value)

wxGridTableBase::GetTypeName

wxString GetTypeName(int row, int col)

Data type determination and value access.

wxGridTableBase::CanGetValueAs

bool CanGetValueAs(int row, int col, const wxString& typeName)

wxGridTableBase::CanSetValueAs

bool CanSetValueAs(int *row*, int *col*, const wxString& *typeName*)

wxGridTableBase::GetValueAsLong

long GetValueAsLong(int *row*, int *col*)

wxGridTableBase::GetValueAsDouble

double GetValueAsDouble(int *row*, int *col*)

wxGridTableBase::GetValueAsBool

bool GetValueAsBool(int *row*, int *col*)

wxGridTableBase::SetValueAsLong

void SetValueAsLong(int *row*, int *col*, long *value*)

wxGridTableBase::SetValueAsDouble

void SetValueAsDouble(int *row*, int *col*, double *value*)

wxGridTableBase::SetValueAsBool

void SetValueAsBool(int *row*, int *col*, bool *value*)

wxGridTableBase::GetValueAsCustom

void* GetValueAsCustom(int *row*, int *col*, const wxString& *typeName*)

For user defined types

wxGridTableBase::SetValueAsCustom

void SetValueAsCustom(int *row*, int *col*, const wxString& *typeName*, void* *value*)

wxGridTableBase::SetView

void SetView(wxGrid* *grid*)

Overriding these is optional

wxGridTableBase::GetView

wxGrid * GetView() const

wxGridTableBase::Clear

void Clear()

wxGridTableBase::InsertRows

bool InsertRows(size_t pos = 0, size_t numRows = 1)

wxGridTableBase::AppendRows

bool AppendRows(size_t numRows = 1)

wxGridTableBase::DeleteRows

bool DeleteRows(size_t pos = 0, size_t numRows = 1)

wxGridTableBase::InsertCols

bool InsertCols(size_t pos = 0, size_t numCols = 1)

wxGridTableBase::AppendCols

bool AppendCols(size_t numCols = 1)

wxGridTableBase::DeleteCols

bool DeleteCols(size_t pos = 0, size_t numCols = 1)

wxGridTableBase::GetRowLabelValue

wxString GetRowLabelValue(int row)

wxGridTableBase::GetColLabelValue

wxString GetColLabelValue(int col)

wxGridTableBase::SetRowLabelValue

void SetRowLabelValue(int WXUNUSED(row), const wxString&)

wxGridTableBase::SetColLabelValue

void SetColLabelValue(int WXUNUSED(col), const wxString&)

wxGridTableBase::SetAttrProvider

void SetAttrProvider(wxGridCellAttrProvider* attrProvider)

Attribute handling give us the attr provider to use - we take ownership of the pointer

wxGridTableBase::GetAttrProvider**wxGridCellAttrProvider* GetAttrProvider() const**

get the currently used attr provider (may be NULL)

wxGridTableBase::CanHaveAttributes**bool CanHaveAttributes()**

Does this table allow attributes? Default implementation creates a wxGridCellAttrProvider if necessary.

wxGridTableBase::UpdateAttrRows**void UpdateAttrRows(size_t pos, int numRows)**

change row/col number in attribute if needed

wxGridTableBase::UpdateAttrCols**void UpdateAttrCols(size_t pos, int numCols)****wxGridTableBase::GetAttr****wxGridCellAttr* GetAttr(int row, int col)**

by default forwarded to wxGridCellAttrProvider if any. May be overridden to handle attributes directly in the table.

wxGridTableBase::SetAttr**void SetAttr(wxGridCellAttr* attr, int row, int col)**

these functions take ownership of the pointer

wxGridTableBase::SetRowAttr**void SetRowAttr(wxGridCellAttr* attr, int row)****wxGridTableBase::SetColAttr****void SetColAttr(wxGridCellAttr* attr, int col)****wxGridSizer**

A grid sizer is a sizer which lays out its children in a two-dimensional table with all table fields having the same size, i.e. the width of each field is the width of the widest child, the height of each field is the height of the tallest child.

Derived from

wxSizer (p. **Error! Bookmark not defined.**)

wxObject (p. **Error! Bookmark not defined.**)

Include files

<wx/sizer.h>

See also

wxSizer (p. **Error! Bookmark not defined.**), *Sizer overview* (p. **Error! Bookmark not defined.**)

wxGridSizer::wxGridSizer

wxGridSizer(int *rows*, int *cols*, int *vgap*, int *hgap*)

wxGridSizer(int *cols*, int *vgap* = 0, int *hgap* = 0)

Constructor for a *wxGridSizer*. *rows* and *cols* determine the number of columns and rows in the sizer - if either of the parameters is zero, it will be calculated to form the total number of children in the sizer, thus making the sizer grow dynamically. *vgap* and *hgap* define extra space between all children.

wxGridSizer::GetCols

int **GetCols**()

Returns the number of columns in the sizer.

wxGridSizer::GetHGap

int **GetHGap**()

Returns the horizontal gap (in pixels) between cells in the sizer.

wxGridSizer::GetRows

int **GetRows**()

Returns the number of rows in the sizer.

wxGridSizer::GetVGap

int **GetVGap**()

Returns the vertical gap (in pixels) between the cells in the sizer.

wxGridSizer::SetCols

void SetCols(int cols)

Sets the number of columns in the sizer.

wxGridSizer::SetHGap

void SetHGap(int gap)

Sets the horizontal gap (in pixels) between cells in the sizer.

wxGridSizer::SetRows

void SetRows(int rows)

Sets the number of rows in the sizer.

wxGridSizer::SetVGap

void SetVGap(int gap)

Sets the vertical gap (in pixels) between the cells in the sizer.

wxHashMap

This is a simple, type-safe, and reasonably efficient hash map class, whose interface is a subset of the interface of STL containers. In particular, the interface is modeled after `std::map`, and the various, non standard, `std::hash_map`.

Example

```
class MyClass { /* ... */ };

// declare a hash map with string keys and int values
WX_DECLARE_STRING_HASH_MAP( int, MyHash5 );
// same, with int keys and MyClass* values
WX_DECLARE_HASH_MAP( int, MyClass*, wxIntegerHash,
wxIntegerEqual, MyHash1 );
// same, with wxString keys and int values
WX_DECLARE_STRING_HASH_MAP( int, MyHash3 );
// same, with wxString keys and values
WX_DECLARE_STRING_HASH_MAP( wxString, MyHash2 );

MyHash1 h1;
MyHash2 h2;

// store and retrieve values
h1[1] = new MyClass( 1 );
h1[100000000] = NULL;
h1[50000] = new MyClass( 2 );
h2["Bill"] = "ABC";
wxString tmp = h2["Bill"];
// since element with key "Joe" is not present, this will
return
// the default value, which is an empty string in the case of
wxString
MyClass tmp2 = h2["Joe"];
```

```
// iterate over all the elements in the class
MyHash2::iterator it;
for( it = h2.begin(); it != h2.end(); ++it )
{
    wxString key = it->first, value = it->second;
    // do something useful with key and value
}
```

Declaring new hash table types

```
WX_DECLARE_STRING_HASH_MAP( VALUE_T,          // type of the values
                             CLASSNAME ); // name of the class
```

Declares a hash map class named CLASSNAME, with wxString keys and VALUE_T values.

```
WX_DECLARE_VOIDPTR_HASH_MAP( VALUE_T,          // type of the
values                                     CLASSNAME ); // name of the class
```

Declares a hash map class named CLASSNAME, with void* keys and VALUE_T values.

```
WX_DECLARE_HASH_MAP( KEY_T,          // type of the keys
                     VALUE_T,        // type of the values
                     HASH_T,         // hasher
                     KEY_EQ_T,       // key equality predicate
                     CLASSNAME); // name of the class
```

The HASH_T and KEY_EQ_T are the types used for the hashing function and key comparison. wxWidgets provides three predefined hashing functions: wxIntegerHash for integer types (int, long, short, and their unsigned counterparts), wxStringHash for strings (wxString, wxChar*, char*), and wxPointerHash for any kind of pointer. Similarly three equality predicates: wxIntegerEqual, wxStringEqual, wxPointerEqual are provided.

Using this you could declare a hash map mapping int values to wxString like this:

```
WX_DECLARE_HASH_MAP( int,
                     wxString,
                     wxIntegerHash,
                     wxIntegerEqual,
                     MyHash );

// using an user-defined class for keys
class MyKey { /* ... */ };

// hashing function
class MyKeyHash
{
public:
    MyKeyHash() { }

    unsigned long operator()( const MyKey& k ) const
    { /* compute the hash */ }

    MyKeyHash& operator=(const MyKeyHash&) { return *this; }
};
```

```
// comparison operator
class MyKeyEqual
{
public:
    MyKeyEqual() { }
    bool operator()( const MyKey& a, const MyKey& b ) const
    { /* compare for equality */ }

    MyKeyEqual& operator=(const MyKeyEqual&) { return *this; }
};

WX_DECLARE_HASH_MAP( MyKey,          // type of the keys
                     SOME_TYPE,     // any type you like
                     MyKeyHash,     // hasher
                     MyKeyEqual,    // key equality predicate
                     CLASSNAME);    // name of the class
```

In the documentation below you should replace `wxHashMap` with the name you used in the class declaration.

<code>wxHashMap::key_type</code>	Type of the hash keys
<code>wxHashMap::mapped_type</code>	Type of the values stored in the hash map
<code>wxHashMap::value_type</code>	Equivalent to <code>struct { key_type first; mapped_type second };</code>
<code>wxHashMap::iterator</code>	Used to enumerate all the elements in a hash map; it is similar to a <code>value_type*</code>
<code>wxHashMap::const_iterator</code>	Used to enumerate all the elements in a constant hash map; it is similar to a <code>const value_type*</code>
<code>wxHashMap::size_type</code>	Used for sizes
<code>wxHashMap::Insert_Result</code>	The return value for <code>insert()</code> (p. 688)

Iterators

An iterator is similar to a pointer, and so you can use the usual pointer operations: `++it` (and `it++`) to move to the next element, `*it` to access the element pointed to, `it->first(it->second)` to access the key (value) of the element pointed to. Hash maps provide forward only iterators, this means that you can't use `--it`, `it + 3`, `it1 - it2`.

Include files

`<wx/hashmap.h>`

`wxHashMap::wxHashMap`

`wxHashMap(size_type size = 10)`

The size parameter is just a hint, the table will resize automatically to preserve performance.

wxHashMap(const wxHashMap& map)

Copy constructor.

wxHashMap::begin

const_iterator begin() const

iterator begin()

Returns an iterator pointing at the first element of the hash map. Please remember that hash maps do not guarantee ordering.

wxHashMap::clear

void clear()

Removes all elements from the hash map.

wxHashMap::count

size_type count(const key_type& key) const

Counts the number of elements with the given key present in the map. This function returns only 0 or 1.

wxHashMap::empty

bool empty() const

Returns true if the hash map does not contain any elements, false otherwise.

wxHashMap::end

const_iterator end() const

iterator end()

Returns an iterator pointing at the one-after-the-last element of the hash map. Please remember that hash maps do not guarantee ordering.

wxHashMap::erase

size_type erase(const key_type& key)

Erases the element with the given key, and returns the number of elements erased (either 0 or 1).

void erase(iterator it)

void erase(const_iterator it)

Erases the element pointed to by the iterator. After the deletion the iterator is no longer valid and must not be used.

wxHashMap::find

iterator find(const key_type& key)

const_iterator find(const key_type& key) const

If an element with the given key is present, the functions returns an iterator pointing at that element, otherwise an invalid iterator is returned (i.e. `hashmap.find(non_existent_key) == hashmap.end()`).

wxHashMap::insert

Insert_Result insert(const value_type& v)

Inserts the given value in the hash map. The return value is equivalent to a `std::pair<wxHashMap::iterator, bool>`; the iterator points to the inserted element, the boolean value is `true` if `v` was actually inserted.

wxHashMap::operator[]

mapped_type& operator[](const key_type& key)

Use the key as an array subscript. The only difference is that if the given key is not present in the hash map, an element with the default `value_type()` is inserted in the table.

wxHashMap::size

size_type size() const

Returns the number of elements in the map.

wxHashSet

This is a simple, type-safe, and reasonably efficient hash set class, whose interface is a subset of the interface of STL containers. In particular, the interface is modeled after `std::set`, and the various, non standard, `std::hash_map`.

Example

```
class MyClass { /* ... */ };

// same, with MyClass* keys (only uses pointer equality!)
WX_DECLARE_HASH_SET( MyClass*, wxPointerHash, wxPointerEqual,
```



```
MySet1 );
    // same, with int keys
    WX_DECLARE_HASH_SET( int, wxIntegerHash, wxIntegerEqual,
MySet2 );
    // declare a hash set with string keys
    WX_DECLARE_HASH_SET( wxString, wxStringHash, wxStringEqual,
MySet3 );

    MySet1 h1;
    MySet2 h1;
    MySet3 h3;

    // store and retrieve values
    h1.insert( new MyClass( 1 ) );

    h3.insert( "foo" );
    h3.insert( "bar" );
    h3.insert( "baz" );

    int size = h3.size(); // now is three
    bool has_foo = h3.find( "foo" ) != h3.end();

    h3.insert( "bar" ); // still has size three

    // iterate over all the elements in the class
    MySet3::iterator it;
    for( it = h3.begin(); it != h3.end(); ++it )
    {
        wxString key = *it;
        // do something useful with key
    }
```

Declaring new hash set types

```
WX_DECLARE_HASH_SET( KEY_T,          // type of the keys
                     HASH_T,        // hasher
                     KEY_EQ_T,      // key equality predicate
                     CLASSNAME); // name of the class
```

The `HASH_T` and `KEY_EQ_T` are the types used for the hashing function and key comparison. `wxWidgets` provides three predefined hashing functions: `wxIntegerHash` for integer types (`int`, `long`, `short`, and their unsigned counterparts), `wxStringHash` for strings (`wxString`, `wxChar*`, `char*`), and `wxPointerHash` for any kind of pointer. Similarly three equality predicates: `wxIntegerEqual`, `wxStringEqual`, `wxPointerEqual` are provided.

Using this you could declare a hash set using `int` values like this:

```
WX_DECLARE_HASH_SET( int,
                    wxIntegerHash,
                    wxIntegerEqual,
                    MySet );

// using an user-defined class for keys
class MyKey { /* ... */ };

// hashing function
class MyKeyHash
{
public:
    MyKeyHash() { }
```

```
        unsigned long operator()( const MyKey& k ) const
        { /* compute the hash */ }

        MyKeyHash& operator=(const MyKeyHash&) { return *this; }
};

// comparison operator
class MyKeyEqual
{
public:
    MyKeyEqual() { }
    bool operator()( const MyKey& a, const MyKey& b ) const
    { /* compare for equality */ }

    MyKeyEqual& operator=(const MyKeyEqual&) { return *this; }
};

WX_DECLARE_HASH_SET( MyKey,          // type of the keys
                    MyKeyHash,      // hasher
                    MyKeyEqual,     // key equality predicate
                    CLASSNAME);    // name of the class
```

In the documentation below you should replace `wxHashSet` with the name you used in the class declaration.

<code>wxHashSet::key_type</code>	Type of the hash keys
<code>wxHashSet::mapped_type</code>	Type of hash keys
<code>wxHashSet::value_type</code>	Type of hash keys
<code>wxHashSet::iterator</code>	Used to enumerate all the elements in a hash set; it is similar to a <code>value_type*</code>
<code>wxHashSet::const_iterator</code>	Used to enumerate all the elements in a constant hash set; it is similar to a <code>const value_type*</code>
<code>wxHashSet::size_type</code>	Used for sizes
<code>wxHashSet::Insert_Result</code>	The return value for <code>insert()</code> (p. 692)

Iterators

An iterator is similar to a pointer, and so you can use the usual pointer operations: `++it` (and `it++`) to move to the next element, `*it` to access the element pointed to, `*it` to access the value of the element pointed to. Hash sets provide forward only iterators, this means that you can't use `--it`, `it + 3`, `it1 - it2`.

Include files

`<wx/hashset.h>`

`wxHashSet::wxHashSet`

wxHashSet(size_type size = 10)

The size parameter is just a hint, the table will resize automatically to preserve performance.

wxHashSet(const wxHashSet& set)

Copy constructor.

wxHashSet::begin

const_iterator begin() const

iterator begin()

Returns an iterator pointing at the first element of the hash set. Please remember that hash sets do not guarantee ordering.

wxHashSet::clear

void clear()

Removes all elements from the hash set.

wxHashSet::count

size_type count(const key_type& key) const

Counts the number of elements with the given key present in the set. This function returns only 0 or 1.

wxHashSet::empty

bool empty() const

Returns true if the hash set does not contain any elements, false otherwise.

wxHashSet::end

const_iterator end() const

iterator end()

Returns an iterator pointing at the one-after-the-last element of the hash set. Please remember that hash sets do not guarantee ordering.

wxHashSet::erase

size_type erase(const key_type& key)

Erases the element with the given key, and returns the number of elements erased

(either 0 or 1).

void erase(iterator it)

void erase(const_iterator it)

Erases the element pointed to by the iterator. After the deletion the iterator is no longer valid and must not be used.

wxHashSet::find

iterator find(const key_type& key)

const_iterator find(const key_type& key) const

If an element with the given key is present, the functions returns an iterator pointing at that element, otherwise an invalid iterator is returned (i.e. `hashset.find(non_existent_key) == hashset.end()`).

wxHashSet::insert

Insert_Result insert(const value_type& v)

Inserts the given value in the hash set. The return value is equivalent to a `std::pair<wxHashMap::iterator, bool>`; the iterator points to the inserted element, the boolean value is `true` if `v` was actually inserted.

wxHashSet::size

size_type size() const

Returns the number of elements in the set.

wxHashTable

Please note that this class is retained for backward compatibility reasons; you should use *wxHashMap* (p. 684).

This class provides hash table functionality for `wxWidgets`, and for an application if it wishes. Data can be hashed on an integer or string key.

Derived from

wxObject (p. **Error! Bookmark not defined.**)

Include files

<wx/hash.h>

Example

Below is an example of using a hash table.

```
wxHashTable table(wxKEY_STRING);

wxPoint *point = new wxPoint(100, 200);
table.Put("point 1", point);

....

wxPoint *found_point = (wxPoint *)table.Get("point 1");
```

A hash table is implemented as an array of pointers to lists. When no data has been stored, the hash table takes only a little more space than this array (default size is 1000). When a data item is added, an integer is constructed from the integer or string key that is within the bounds of the array. If the array element is NULL, a new (keyed) list is created for the element. Then the data object is appended to the list, storing the key in case other data objects need to be stored in the list also (when a 'collision' occurs).

Retrieval involves recalculating the array index from the key, and searching along the keyed list for the data object whose stored key matches the passed key. Obviously this is quicker when there are fewer collisions, so hashing will become inefficient if the number of items to be stored greatly exceeds the size of the hash table.

See also

wxList (p. 851)

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 hash table constructor was used).

wxHashTable::MakeKey**long MakeKey(const wxString& string)**

Makes an integer key out of a string. An application may wish to make a key explicitly (for instance when combining two data values to form a key).

wxHashTable::Next**wxHashTable::Node * Next()**

If the application wishes to iterate through all the data in the hash table, it can call *BeginFind* and then loop on *Next*. This function returns a **wxHashTable::Node** pointer (or NULL if there are no more nodes). The return value is functionally equivalent to **wxNode** but might not be implemented as a **wxNode**. The user will probably only wish to use the **GetData** method to retrieve the data; the node may also be deleted.

wxHashTable::Put**void Put(long key, wxObject *object)****void Put(const char* key, wxObject *object)**

Inserts data into the hash table, using an integer or string key (depending on which hash table constructor was used). The key string is copied and stored by the hash table implementation.

wxHashTable::GetCount

size_t GetCount() const

Returns the number of elements in the hash table.

wxHelpController

This is a family of classes by which applications may invoke a help viewer to provide on-line help.

A help controller allows an application to display help, at the contents or at a particular topic, and shut the help program down on termination. This avoids proliferation of many instances of the help viewer whenever the user requests a different topic via the application's menus or buttons.

Typically, an application will create a help controller instance when it starts, and immediately call **Initialize** to associate a filename with it. The help viewer will only get run, however, just before the first call to display something.

Most help controller classes actually derive from `wxHelpControllerBase` and have names of the form `wxXXXHelpController` or `wxHelpControllerXXX`. An appropriate class is aliased to the name `wxHelpController` for each platform, as follows:

- On desktop Windows, `wxCHMHelpController` is used (MS HTML Help).
- On Windows CE, `wxWinceHelpController` is used.
- On all other platforms, `wxHtmlHelpController` is used if `wxHTML` is compiled into `wxWidgets`; otherwise `wxExtHelpController` is used (for invoking an external browser).

The remaining help controller classes need to be named explicitly by an application that wishes to make use of them.

There are currently the following help controller classes defined:

- `wxWinHelpController`, for controlling Windows Help.
- `wxCHMHelpController`, for controlling MS HTML Help. To use this, you need to set `wxUSE_MS_HTML_HELP` to 1 in `setup.h` and have `htmlhelp.h` header from Microsoft's HTML Help kit (you don't need VC++ specific `htmlhelp.lib` because `wxWidgets` loads necessary DLL at runtime and so it works with all compilers).
- `wxBestHelpController`, for controlling MS HTML Help or, if Microsoft's runtime is not available, *wxHtmlHelpController* (p. 721). You need to provide **both** CHM and HTB versions of the help file. For 32bit Windows only.
- `wxExtHelpController`, for controlling external browsers under Unix. The default browser is Netscape Navigator. The 'help' sample shows its use.
- `wxWinceHelpController`, for controlling a simple `.htm` help controller for

Windows CE applications.

- *wxHtmlHelpController* (p. 721), a sophisticated help controller using *wxHTML* (p. **Error! Bookmark not defined.**), in a similar style to the Microsoft HTML Help viewer and using some of the same files. Although it has an API compatible with other help controllers, it has more advanced features, so it is recommended that you use the specific API for this class instead. Note that if you use .zip or .htb formats for your books, you must add this line to your application initialization: `wxFileSystem::AddHandler(new wxZipFSHandler);` or nothing will be shown in your help window.

Derived from

wxHelpControllerBase

wxObject (p. **Error! Bookmark not defined.**)

Include files

<wx/help.h> (*wxWidgets* chooses the appropriate help controller class)

<wx/helpbase.h> (*wxHelpControllerBase* class)

<wx/helpwin.h> (Windows Help controller)

<wx/msw/helpchm.h> (MS HTML Help controller)

<wx/generic/helpext.h> (external HTML browser controller)

<wx/html/helpctrl.h> (*wxHTML* based help controller: *wxHtmlHelpController*)

See also

wxHtmlHelpController (p. 721), *wxHTML* (p. **Error! Bookmark not defined.**)

wxHelpController::wxHelpController

wxHelpController(*wxWindow** *parentWindow* = *NULL*)

Constructs a help instance object, but does not invoke the help viewer.

If you provide a window, it will be used by some help controller classes, such as *wxCHMHelpController*, *wxWinHelpController* and *wxHtmlHelpController*, as the parent for the help window instead of the value of *wxApp::GetTopWindow* (p. 39). You can also change the parent window later with *wxHelpController::SetParentWindow* (p. 699).

wxHelpController::~~wxHelpController

~wxHelpController()

Destroys the help instance, closing down the viewer if it is running.

wxHelpController::Initialize

virtual bool Initialize(const *wxString&* *file*)

virtual bool Initialize(const wxString& file, int server)

Initializes the help instance with a help filename, and optionally a server socket number if using wxHelp (now obsolete). Does not invoke the help viewer. This must be called directly after the help instance object is created and before any attempts to communicate with the viewer.

You may omit the file extension and a suitable one will be chosen. For wxHtmlHelpController, the extensions zip, htb and hhp will be appended while searching for a suitable file. For WinHelp, the hlp extension is appended.

wxHelpController::DisplayBlock**virtual bool DisplayBlock(long blockNo)**

If the help viewer is not running, runs it and displays the file at the given block number.

WinHelp: Refers to the context number.

MS HTML Help: Refers to the context number.

External HTML help: the same as for wxHelpController::DisplaySection (p. 697).

wxHtmlHelpController: sectionNo is an identifier as specified in the .hhc file. See *Help files format* (p. **Error! Bookmark not defined.**).

This function is for backward compatibility only, and applications should use wxHelpController (p. 697) instead.

wxHelpController::DisplayContents**virtual bool DisplayContents()**

If the help viewer is not running, runs it and displays the contents.

wxHelpController::DisplayContextPopup**virtual bool DisplayContextPopup(int contextId)**

Displays the section as a popup window using a context id.

Returns false if unsuccessful or not implemented.

wxHelpController::DisplaySection**virtual bool DisplaySection(const wxString& section)**

If the help viewer is not running, runs it and displays the given section.

The interpretation of *section* differs between help viewers. For most viewers, this call is equivalent to KeywordSearch. For MS HTML Help, wxHTML help and external HTML help, if *section* has a .htm or .html extension, that HTML file will be displayed; otherwise

a keyword search is done.

virtual bool DisplaySection(int sectionNo)

If the help viewer is not running, runs it and displays the given section.

WinHelp, *MS HTML Help* *sectionNo* is a context id.

External HTML help: *wxExtHelpController* implements *sectionNo* as an id in a map file, which is of the form:

```
0  wx.html           ; Index
1  wx34.html#classref ; Class reference
2  wx204.html        ; Function reference
```

wxHtmlHelpController: *sectionNo* is an identifier as specified in the .hhc file. See *Help files format* (p. **Error! Bookmark not defined.**).

See also the help sample for notes on how to specify section numbers for various help file formats.

wxHelpController::DisplayTextPopup

virtual bool DisplayTextPopup(const wxString& text, const wxPoint& pos)

Displays the text in a popup window, if possible.

Returns false if unsuccessful or not implemented.

wxHelpController::GetFrameParameters

virtual wxFrame * GetFrameParameters(const wxSize * size = NULL, const wxPoint * pos = NULL, bool *newFrameEachTime = NULL)

wxHtmlHelpController returns the frame, size and position.

For all other help controllers, this function does nothing and just returns NULL.

Parameters

viewer

This defaults to "netscape" for *wxExtHelpController*.

flags

This defaults to *wxHELP_NETSCAPE* for *wxExtHelpController*, indicating that the viewer is a variant of Netscape Navigator.

wxHelpController::GetParentWindow

virtual bool GetParentWindow() const

Returns the window to be used as the parent for the help window. This window is used

by `wxCHMHelpController`, `wxWinHelpController` and `wxHtmlHelpController`.

wxHelpController::KeywordSearch

virtual bool KeywordSearch(const wxString& keyWord, wxHelpSearchMode mode = wxHELP_SEARCH_ALL)

If the help viewer is not running, runs it, and searches for sections matching the given keyword. If one match is found, the file is displayed at this section. The optional parameter allows the search the index (`wxHELP_SEARCH_INDEX`) but this currently only supported by the `wxHtmlHelpController`.

WinHelp, MS HTML Help: If more than one match is found, the first topic is displayed.

External HTML help, simple wxHTML help: If more than one match is found, a choice of topics is displayed.

wxHtmlHelpController: see `wxHtmlHelpController::KeywordSearch` (p. 725).

wxHelpController::LoadFile

virtual bool LoadFile(const wxString& file = "")

If the help viewer is not running, runs it and loads the given file. If the filename is not supplied or is empty, the file specified in **Initialize** is used. If the viewer is already displaying the specified file, it will not be reloaded. This member function may be used before each display call in case the user has opened another file.

`wxHtmlHelpController` ignores this call.

wxHelpController::OnQuit

virtual bool OnQuit()

Overridable member called when this application's viewer is quit by the user.

This does not work for all help controllers.

wxHelpController::SetFrameParameters

virtual void SetFrameParameters(const wxString & title, const wxSize & size, const wxPoint & pos = wxDefaultPosition, bool newFrameEachTime = false)

For `wxHtmlHelpController`, the title is set (again with %s indicating the page title) and also the size and position of the frame if the frame is already open. *newFrameEachTime* is ignored.

For all other help controllers this function has no effect.

wxHelpController::SetParentWindow

virtual void SetParentWindow(wxWindow* parentWindow)

Sets the window to be used as the parent for the help window. This is used by wxCHMHelpController, wxWinHelpController and wxHtmlHelpController.

wxHelpController::SetViewer

virtual void SetViewer(const wxString& viewer, long flags)

Sets detailed viewer information. So far this is only relevant to wxExtHelpController.

Some examples of usage:

```
m_help.SetViewer("kdehelp");  
m_help.SetViewer("gnome-help-browser");  
m_help.SetViewer("netscape", wxHELP_NETSCAPE);
```

wxHelpController::Quit

virtual bool Quit()

If the viewer is running, quits it by disconnecting.

For Windows Help, the viewer will only close if no other application is using it.

wxHelpControllerHelpProvider

wxHelpControllerHelpProvider is an implementation of wxHelpProvider which supports both context identifiers and plain text help strings. If the help text is an integer, it is passed to wxHelpController::DisplayContextPopup. Otherwise, it shows the string in a tooltip as per wxSimpleHelpProvider. If you use this with a wxCHMHelpController instance on windows, it will use the native style of tip window instead of wxTipWindow (p. **Error! Bookmark not defined.**).

You can use the convenience function **wxContextId** to convert an integer context id to a string for passing to wxWindow::SetHelpText (p. **Error! Bookmark not defined.**).

Derived from

wxSimpleHelpProvider (p. **Error! Bookmark not defined.**)
wxHelpProvider (p. 702)

Include files

<wx/cshelp.h>

See also

wxHelpProvider (p. 702), wxSimpleHelpProvider (p. **Error! Bookmark not defined.**),
wxContextHelp (p. 215), wxWindow::SetHelpText (p. **Error! Bookmark not defined.**),
wxWindow::GetHelpText (p. **Error! Bookmark not defined.**)

wxHelpControllerHelpProvider::wxHelpControllerHelpProvider**wxHelpControllerHelpProvider(wxHelpControllerBase* hc = NULL)**

Note that the instance doesn't own the help controller. The help controller should be deleted separately.

wxHelpControllerHelpProvider::SetHelpController**void SetHelpController(wxHelpControllerBase* hc)**

Sets the help controller associated with this help provider.

wxHelpControllerHelpProvider::GetHelpController**wxHelpControllerBase* GetHelpController() const**

Returns the help controller associated with this help provider.

wxHelpEvent

A help event is sent when the user has requested context-sensitive help. This can either be caused by the application requesting context-sensitive help mode via *wxContextHelp* (p. 215), or (on MS Windows) by the system generating a WM_HELP message when the user pressed F1 or clicked on the query button in a dialog caption.

A help event is sent to the window that the user clicked on, and is propagated up the window hierarchy until the event is processed or there are no more event handlers. The application should call *wxEvent::GetId* to check the identity of the clicked-on window, and then either show some suitable help or call *wxEvent::Skip* if the identifier is unrecognised. Calling *Skip* is important because it allows *wxWidgets* to generate further events for ancestors of the clicked-on window. Otherwise it would be impossible to show help for container windows, since processing would stop after the first window found.

Derived from*wxCommandEvent* (p. 184)*wxEvent* (p. 487)*wxObject* (p. **Error! Bookmark not defined.**)**Include files**

<wx/event.h>

Event table macros

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

EVT_HELP(id, func)	Process a wxEVT_HELP event.
EVT_HELP_RANGE(id1, id2, func)	Process a wxEVT_HELP event for a range of ids.

See also

wxContextHelp (p. 215), *wxDialog* (p. 412), *Event handling overview* (p. **Error! Bookmark not defined.**)

wxHelpEvent::wxHelpEvent

wxHelpEvent(WXTYPE eventType = 0, wxWindowID id = 0, const wxPoint& point)

Constructor.

wxHelpEvent::GetPosition

const wxPoint& GetPosition() const

Returns the left-click position of the mouse, in screen coordinates. This allows the application to position the help appropriately.

wxHelpEvent::SetPosition

void SetPosition(const wxPoint& pt)

Sets the left-click position of the mouse, in screen coordinates.

wxHelpProvider

wxHelpProvider is an abstract class used by a program implementing context-sensitive help to show the help text for the given window.

The current help provider must be explicitly set by the application using wxHelpProvider::Set().

Derived from

No base class

Include files

<wx/cshelp.h>

See also

wxContextHelp (p. 215), *wxContextHelpButton* (p. 216), *wxSimpleHelpProvider* (p. **Error! Bookmark not defined.**), *wxHelpControllerHelpProvider* (p. 700), *wxWindow::SetHelpText* (p. **Error! Bookmark not defined.**), *wxWindow::GetHelpText* (p. **Error! Bookmark not defined.**)

wxHelpProvider::~wxHelpProvider**~wxHelpProvider()**

Virtual destructor for any base class.

wxHelpProvider::AddHelp**void AddHelp(wxWindowBase* window, const wxString& text)**

Associates the text with the given window or id. Although all help providers have these functions to allow making *wxWindow::SetHelpText* (p. **Error! Bookmark not defined.**) work, not all of them implement the functions.

wxHelpProvider::Get**wxHelpProvider* Get()**

Unlike some other classes, the help provider is not created on demand. This must be explicitly done by the application.

wxHelpProvider::GetHelp**wxString GetHelp(const wxWindowBase* window)**

Gets the help string for this window. Its interpretation is dependent on the help provider except that empty string always means that no help is associated with the window.

void AddHelp(wxWindowID id, const wxString& text)

This version associates the given text with all windows with this id. May be used to set the same help string for all Cancel buttons in the application, for example.

wxHelpProvider::RemoveHelp**void RemoveHelp(wxWindowBase* window)**

Removes the association between the window pointer and the help text. This is called by the *wxWindow* destructor. Without this, the table of help strings will fill up and when window pointers are reused, the wrong help string will be found.

wxHelpProvider::Set**wxHelpProvider* Set(wxHelpProvider* helpProvider)**

Get/set the current, application-wide help provider. Returns the previous one.

wxHelpProvider::ShowHelp

bool ShowHelp(wxWindowBase* window)

Shows help for the given window. Uses *GetHelp* (p. 703) internally if applicable.

Returns true if it was done, or false if no help was available for this window.

wxHtmlCell

Internal data structure. It represents fragments of parsed HTML page, the so-called **cell** - a word, picture, table, horizontal line and so on. It is used by *wxHtmlWindow* (p. 751) and *wxHtmlWinParser* (p. 760) 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. **Error! Bookmark not defined.**)

Include files

<wx/html/htmlcell.h>

See Also

Cells Overview (p. **Error! Bookmark not defined.**), *wxHtmlContainerCell* (p. 710)

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. 704) when the cell is certainly out of the screen (and thus invisible). This is not nonsense - some tags (like *wxHtmlColourCell* (p. 709) 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.

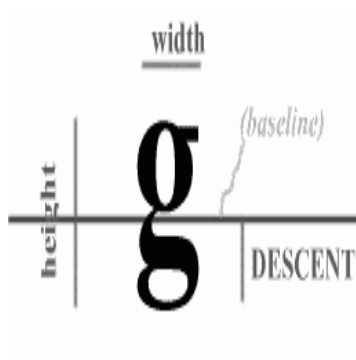
wxHTML_COND_USER

User-defined conditions start from this number.

wxHtmlCell::GetDescent

int GetDescent() const

Returns descent value of the cell (m_Descent member). See explanation:



wxHtmlCell::GetFirstChild

wxHtmlCell* GetFirstChild()

Returns pointer to the first cell in the list. You can then use child's *GetNext* (p. 707) 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. 705) method instead.

wxHtmlCell::GetHeight

int GetHeight() const

Returns height of the cell (m_Height member).

wxHtmlCell::GetId

virtual wxString GetId() const

Returns unique cell identifier if there is any, empty string otherwise.

wxHtmlCell::GetLink

virtual wxHtmlLinkInfo* GetLink(int x = 0, int y = 0) const

Returns hypertext link if associated with this cell or NULL otherwise. See *wxHtmlLinkInfo* (p. 735). (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. 707) 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. 707) 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 OnMouseClicked(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. 751) this method of that cell is called. Default behavior is that it calls *wxHtmlWindow::LoadPage* (p. 754).

Note

If you need more "advanced" event handling you should use *wxHtmlBinderCell* instead.

Parameters

parent

parent window (always *wxHtmlWindow*!)

x, y

coordinates of mouse click (this is relative to cell's origin)

left, middle, right

boolean flags for mouse buttons. true if the left/middle/right button is pressed, false otherwise

wxHtmlCell::SetId

void SetId(const wxString& id)

Sets unique cell identifier. Default value is no identifier, i.e. empty string.

wxHtmlCell::SetLink

void SetLink(const wxHtmlLinkInfo& link)

Sets the hypertext link associated with this cell. (Default value is *wxHtmlLinkInfo* (p. 735)("", "")) (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. 711).

wxHtmlCell::SetParent**void SetParent(wxHtmlContainerCell *p)**

Sets parent container of this cell. This is called from *wxHtmlContainerCell::InsertCell* (p. 711).

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. 704)

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. 704)

Include files

<wx/html/htmlcell.h>

See Also

Cells Overview (p. [Error! Bookmark not defined.](#))

wxHtmlContainerCell::wxHtmlContainerCell

wxHtmlContainerCell(wxHtmlContainerCell *parent)

Constructor. *parent* is pointer to parent container or NULL.

wxHtmlContainerCell::GetAlignHor

int GetAlignHor() const

Returns container's horizontal alignment.

wxHtmlContainerCell::GetAlignVer

int GetAlignVer() const

Returns container's vertical alignment.

wxHtmlContainerCell::GetBackgroundColour

wxColour GetBackgroundColour()

Returns the background colour of the container or `wxNullColour` if no background colour is set.

wxHtmlContainerCell::GetIndent

int GetIndent(int ind) const

Returns the indentation. *ind* is one of the **wxHTML_INDENT_*** constants.

Note: You must call *GetIndentUnits* (p. 711) 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. 711) and *SetAlignVer* (p. 711).

wxHtmlContainerCell::SetAlignHor

void SetAlignHor(int al)

Sets the container's *horizontal alignment*. During *Layout* (p. 707) each line is aligned according to *al* value.

Parameters

al

new horizontal alignment. May be one of these values:

wxHTML_ALIGN_LEFT	lines are left-aligned (default)
wxHTML_ALIGN_JUSTIFY	lines are justified
wxHTML_ALIGN_CENTER	lines are centered
wxHTML_ALIGN_RIGHT	lines are right-aligned

wxHtmlContainerCell::SetAlignVer

void SetAlignVer(int al)

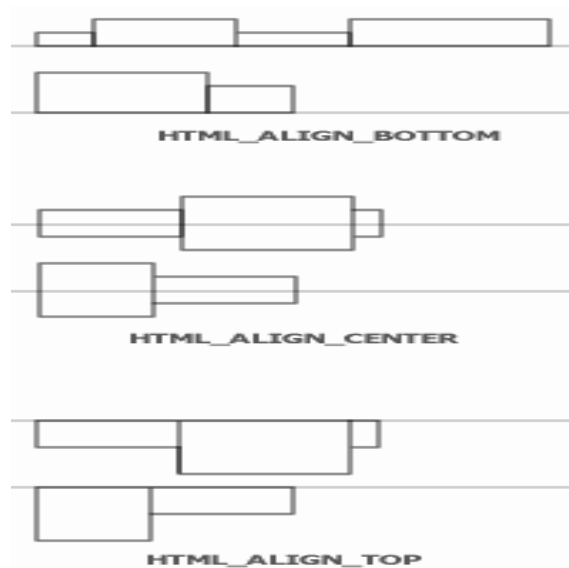
Sets the container's *vertical alignment*. This is per-line alignment!

Parameters

al

new vertical alignment. May be one of these values:

wxHTML_ALIGN_BOTTOM	cells are over the line (default)
wxHTML_ALIGN_CENTER	cells are centered on line
wxHTML_ALIGN_TOP	cells are under the line



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

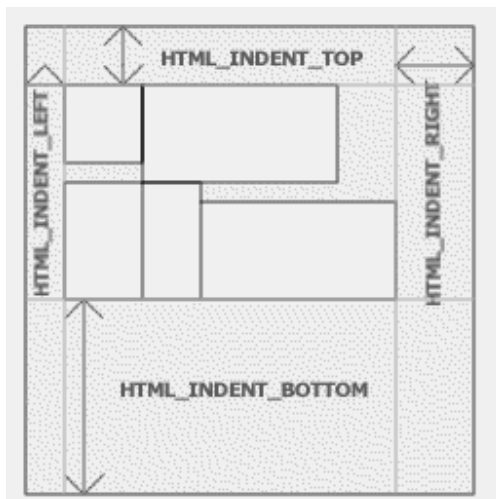
i

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>i</i> is number of pixels
wxHTML_UNITS_PERCENT	<i>i</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. 707) 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 *unitsinfo* 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. 717) or *wxHtmlPrintout* (p. 743) is strongly recommended.

Derived from

wxObject (p. **Error! Bookmark not defined.**)

Include files

<wx/html/htmprint.h>

wxHtmlDCRenderer::wxHtmlDCRenderer

wxHtmlDCRenderer()

Constructor.

wxHtmlDCRenderer::SetDC

void SetDC(wxDC* dc, double pixel_scale = 1.0)

Assign DC instance to the renderer.

pixel_scale can be used when rendering to high-resolution DCs (e.g. printer) to adjust size of pixel metrics. (Many dimensions in HTML are given in pixels -- e.g. image sizes. 300x300 image would be only one inch wide on typical printer. With *pixel_scale* = 3.0 it would be 3 inches.)

wxHtmlDCRenderer::SetFont

void SetFont(const wxString& normal_face, const wxString& fixed_face, const int *sizes = NULL)

Sets fonts. See *wxHtmlWindow::SetFont* (p. 758) for detailed description.

See also *SetSize* (p. 715).

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. 716)! (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. 716) 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. 542) 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. 715))

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. 715)
- `SetSize` (p. 715)
- `SetHtmlText` (p. 715)

`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. 715)) is smaller than total height and thus the page cannot fit into it. In that case you're supposed to call `Render` (p. 716) 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. **Error! Bookmark not defined.**)

Include files

`<wx/html/htmprint.h>`

`wxHtmlEasyPrinting::wxHtmlEasyPrinting`

`wxHtmlEasyPrinting(const wxString& name = "Printing", wxWindow* parentWindow = NULL)`

Constructor.

Parameters

name

Name of the printing object. Used by preview frames and setup dialogs.

parentWindow

pointer to the window that will own the preview frame and setup dialogs. May be NULL.

wxHtmlEasyPrinting::PreviewFile

bool PreviewFile(const wxString& *htmlfile*)

Preview HTML file.

Returns false in case of error -- call *wxPrinter::GetLastError* (p. **Error! Bookmark not defined.**) 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. **Error! Bookmark not defined.**) 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. **Error! Bookmark not defined.**) 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. **Error! Bookmark not defined.**) to get detailed information about the kind of the error.

Parameters

htmltext

HTML text.

basepath

base directory (html string would be stored there if it was in file). It is used to determine path for loading images, for example.

wxHtmlEasyPrinting::PageSetup

void PageSetup()

Display page setup dialog and allows the user to modify settings.

wxHtmlEasyPrinting::SetFont

void SetFont(const wxString& *normal_face*, const wxString& *fixed_face*, const int **sizes* = NULL)

Sets fonts. See *wxHtmlWindow::SetFont* (p. 758) for detailed description.

wxHtmlEasyPrinting::SetHeader

void SetHeader(const wxString& *header*, int *pg* = wxPAGE_ALL)

Set page header.

Parameters

header

HTML text to be used as header. You can use macros in it:

- @PAGENUM@ is replaced by page number
- @PAGESCNT@ is replaced by total number of pages

pg

one of wxPAGE_ODD, wxPAGE_EVEN and wxPAGE_ALL constants.

wxHtmlEasyPrinting::SetFooter

void SetFooter(const wxString& footer, int pg = wxPAGE_ALL)

Set page footer.

Parameters

footer

HTML text to be used as footer. You can use macros in it:

- @PAGENUM@ is replaced by page number
- @PAGESCNT@ is replaced by total number of pages

pg

one of wxPAGE_ODD, wxPAGE_EVEN and wxPAGE_ALL constants.

wxHtmlEasyPrinting::GetPrintData

wxPrintData* GetPrintData()

Returns pointer to *wxPrintData* (p. **Error! Bookmark not defined.**) instance used by this class. You can set its parameters (via SetXXXX methods).

wxHtmlEasyPrinting::GetPageSetupData

wxPageSetupDialogData* GetPageSetupData()

Returns a pointer to *wxPageSetupDialogData* (p. **Error! Bookmark not defined.**) instance used by this class. You can set its parameters (via SetXXXX methods).

wxHtmlFilter

This class is the parent class of input filters for *wxHtmlWindow* (p. 751). It allows you to read and display files of different file formats.

Derived from

wxObject (p. **Error! Bookmark not defined.**)

Include files

<wx/html/htmlfilt.h>

See Also

Overview (p. **Error! Bookmark not defined.**)

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

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. 723)). A book is a logical section of documentation (for example "User's Guide" or "Programmer's Guide" or "C++ Reference" or "wxWidgets Reference"). The help controller can handle as many books as you want.

Although this class has an API compatible with other wxWidgets help controllers as documented by *wxHelpController* (p. 694), it is recommended that you use the enhanced capabilities of wxHtmlHelpController's API.

wxHTML uses Microsoft's HTML Help Workshop project files (.hhp, .hhk, .hhc) as its native format. The file format is described *here* (p. **Error! Bookmark not defined.**). 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. The commercial tool HelpBlocks (www.helpblocks.com) can also create these files.

Note

It is strongly recommended to use preprocessed **.hhp.cached** version of projects. It can be either created on-the-fly (see *SetTempDir* (p. 725)) or you can use **hhp2cached** utility from *utils/hhp2cached* to create it and distribute the cached version together with helpfiles. See *samples/html/help* sample for demonstration of its use.

See also

Information about wxBestHelpController (p. 694), *wxHtmlHelpFrame* (p. 729), *wxHtmlHelpDialog* (p. 727), *wxHtmlHelpWindow* (p. 730), *wxHtmlModalHelp* (p. 734)

Derived from

wxHelpControllerBase

Include files

<wx/html/helpctrl.h>

wxHtmlHelpController::wxHtmlHelpController

wxHtmlHelpController(int *style* = wxHF_DEFAULT_STYLE, **wxWindow*** *parentWindow* = NULL)

Constructor.

Parameters

style is a combination of these flags:

wxHF_TOOLBAR	The help window has a toolbar.
wxHF_FLAT_TOOLBAR	The help window has a toolbar with flat buttons (aka coolbar).
wxHF_CONTENTS	The help window has a contents panel.
wxHF_INDEX	The help window has an index panel.
wxHF_SEARCH	The help window has a search panel.
wxHF_BOOKMARKS	The help window has bookmarks controls.
wxHF_OPEN_FILES	Allows user to open arbitrary HTML document.
wxHF_PRINT	The toolbar contains "print" button.
wxHF_MERGE_BOOKS	The contents pane does not show book nodes. All books are merged together and appear as single book to the user.
wxHF_ICONS_BOOK	All nodes in contents pane have a book icon. This is how Microsoft's HTML help viewer behaves.

wxHF_ICONS_FOLDER	Book nodes in contents pane have a book icon, book's sections have a folder icon. This is the default.
wxHF_ICONS_BOOK_CHAPTER	Both book nodes and nodes of top-level sections of a book (i.e. chapters) have a book icon, all other sections (sections, subsections, ...) have a folder icon.
wxHF_EMBEDDED	Specifies that the help controller controls an embedded window of class <i>wxHtmlHelpWindow</i> (p. 730) that should not be destroyed when the controller is destroyed.
wxHF_DIALOG	Specifies that the help controller should create a dialog containing the help window.
wxHF_FRAME	Specifies that the help controller should create a frame containing the help window. This is the default if neither wxHF_DIALOG nor wxHF_EMBEDDED is specified.
wxHF_MODAL	Specifies that the help controller should create a modal dialog containing the help window (used with the wxHF_DIALOG style).
wxHF_DEFAULT_STYLE	<div style="display: flex; align-items: center;"> <div style="flex: 1;"> wxHF_TOOLBAR wxHF_CONTENTS wxHF_INDEX wxHF_SEARCH wxHF_BOOKMARKS wxHF_PRINT </div> </div>

parentWindow is an optional window to be used as the parent for the help window.

wxHtmlHelpController::AddBook

bool AddBook(const wxString& bookFile, bool showWaitMsg)

bool AddBook(const wxString& bookUrl, bool showWaitMsg)

Adds book (*.hhp file* (p. **Error! Bookmark not defined.**) - HTML Help Workshop project file) into the list of loaded books. This must be called at least once before displaying any help.

bookFile or *bookUrl* may be either .hhp file or ZIP archive that contains arbitrary number of .hhp files in top-level directory. This ZIP archive must have .zip or .htb extension (the latter stands for "HTML book"). In other words, `AddBook(wxFileName("help.zip"))` is possible and is the recommended way.

Parameters

showWaitMsg

If true then a decoration-less window with progress message is displayed.

bookFile

Help book filename. It is recommended to use this prototype instead of the one taking URL, because it is less error-prone.

bookUrl

Help book URL (note that syntax of filename and URL is different on most platforms)

Note

Don't forget to install wxFileSystem ZIP handler
with `wxFileSystem::AddHandler(new wxZipFSHandler);` before calling this method on a .zip or .htb file!

wxHtmlHelpController::CreateHelpDialog

virtual wxHtmlHelpDialog* CreateHelpDialog(wxHtmlHelpData * data)

This protected virtual method may be overridden so that when specifying the wxHF_DIALOG style, the controller uses a different dialog.

wxHtmlHelpController::CreateHelpFrame

virtual wxHtmlHelpFrame* CreateHelpFrame(wxHtmlHelpData * data)

This protected virtual method may be overridden so that the controller uses a different frame.

wxHtmlHelpController::Display

void Display(const wxString& x)

Displays page x. This is THE important function - it is used to display the help in application.

You can specify the page in many ways:

- as direct filename of HTML document
- as chapter name (from contents) or as a book name
- as some word from index
- even as any word (will be searched)

Looking for the page runs in these steps:

1. try to locate file named x (if x is for example "doc/howto.htm")
2. try to open starting page of book named x
3. try to find x in contents (if x is for example "How To ...")

4. try to find *x* in index (if *x* is for example "How To ...")
5. switch to Search panel and start searching

void Display(const int *id*)

This alternative form is used to search help contents by numeric IDs.

wxPython note: The second form of this method is named `DisplayId` in wxPython.

wxHtmlHelpController::DisplayContents

void DisplayContents()

Displays help window and focuses contents panel.

wxHtmlHelpController::DisplayIndex

void DisplayIndex()

Displays help window and focuses index panel.

wxHtmlHelpController::KeywordSearch

bool KeywordSearch(const wxString& *keyword*, wxHelpSearchMode *mode* = wxHELP_SEARCH_ALL)

Displays help window, focuses search panel and starts searching. Returns true if the keyword was found. Optionally it searches through the index (*mode* = wxHELP_SEARCH_INDEX), default the content (*mode* = wxHELP_SEARCH_ALL).

Important: `KeywordSearch` searches only pages listed in .hhc file(s). You should list all pages in the contents file.

wxHtmlHelpController::ReadCustomization

void ReadCustomization(wxConfigBase* *cfg*, wxString *path* = wxEmptyString)

Reads the controller's setting (position of window, etc.)

wxHtmlHelpController::SetTempDir

void SetTempDir(const wxString& *path*)

Sets the path for storing temporary files - cached binary versions of index and contents files. These binary forms are much faster to read. Default value is empty string (empty string means that no cached data are stored). Note that these files are *not* deleted when program exits.

Once created these cached files will be used in all subsequent executions of your application. If cached files become older than corresponding .hhp file (e.g. if you regenerate documentation) it will be refreshed.

wxHtmlHelpController::SetTitleFormat**void SetTitleFormat(const wxString& format)**

Sets format of title of the frame. Must contain exactly one "%s" (for title of displayed HTML page).

wxHtmlHelpController::UseConfig**void UseConfig(wxConfigBase* config, const wxString& rootpath = wxEmptyString)**

Associates *config* object with the controller.

If there is associated config object, wxHtmlHelpController automatically reads and writes settings (including wxHtmlWindow's settings) when needed.

The only thing you must do is create wxConfig object and call UseConfig.

If you do not use *UseConfig*, wxHtmlHelpController will use default wxConfig object if available (for details see *wxConfigBase::Get* (p. 205) and *wxConfigBase::Set* (p. 209)).

wxHtmlHelpController::WriteCustomization**void WriteCustomization(wxConfigBase* cfg, wxString path = wxEmptyString)**

Stores controllers setting (position of window etc.)

wxHtmlHelpData

This class is used by *wxHtmlHelpController* (p. 721) and *wxHtmlHelpFrame* (p. 729) to access HTML help items. It is internal class and should not be used directly - except for the case you're writing your own HTML help controller.

Derived from

wxObject (p. **Error! Bookmark not defined.**)

Include files

<wx/html/helpdata.h>

wxHtmlHelpData::wxHtmlHelpData**wxHtmlHelpData()**

Constructor.

wxHtmlHelpData::AddBook

bool AddBook(const wxString& book_url)

Adds new book. *book* is URL (not filename!) of HTML help project (hhp) or ZIP file that contains arbitrary number of .hhp projects (this zip file can have either .zip or .htb extension, htb stands for "html book"). Returns success.

wxHtmlHelpData::FindPageById

wxString FindPageById(int id)

Returns page's URL based on integer ID stored in project.

wxHtmlHelpData::FindPageByName

wxString FindPageByName(const wxString& page)

Returns page's URL based on its (file)name.

wxHtmlHelpData::GetBookRecArray

const wxHtmlBookRecArray& GetBookRecArray()

Returns array with help books info.

wxHtmlHelpData::GetContentsArray

const wxHtmlHelpDataItems& GetContentsArray()

Returns reference to array with contents entries.

wxHtmlHelpData::GetIndexArray

const wxHtmlHelpDataItems& GetIndexArray()

Returns reference to array with index entries.

wxHtmlHelpData::SetTempDir

void SetTempDir(const wxString& path)

Sets temporary directory where binary cached versions of MS HTML Workshop files will be stored. (This is turned off by default and you can enable this feature by setting non-empty temp dir.)

wxHtmlHelpDialog

This class is used by *wxHtmlHelpController* (p. 721) 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. 582)

Include files

<wx/html/helpdlg.h>

wxHtmlHelpDialog::wxHtmlHelpDialog

wxHtmlHelpDialog(wxHtmlHelpData* *data* = *NULL*)

wxHtmlHelpDialog(wxWindow* *parent*, int *wxWindowID*, const wxString& *title* = *wxEmptyString*, int *style* = *wxHF_DEFAULT_STYLE*, wxHtmlHelpData* *data* = *NULL*)

Constructor. For the values of *style*, please see the documentation for *wxHtmlHelpController* (p. 721).

wxHtmlHelpDialog::AddToolBarButtons

virtual void AddToolBarButtons(wxToolBar* *toolBar*, int *style*)

You may override this virtual method to add more buttons to the help window's toolbar. *toolBar* is a pointer to the toolbar and *style* is the style flag as passed to the *Create* method.

wxToolBar::Realize is called immediately after returning from this function.

wxHtmlHelpDialog::Create

bool Create(wxWindow* *parent*, wxWindowID *id*, const wxString& *title* = *wxEmptyString*, int *style* = *wxHF_DEFAULT_STYLE*)

Creates the dialog. See *the constructor* (p. 728) for a description of the parameters.

wxHtmlHelpDialog::GetController

wxHtmlHelpController* GetController() const

Returns the help controller associated with the dialog.

wxHtmlHelpDialog::ReadCustomization

void ReadCustomization(wxConfigBase* *cfg*, const wxString& *path* = *wxEmptyString*)

Reads the user's settings for this dialog see *wxHtmlHelpController::ReadCustomization* (p. 725))

wxHtmlHelpDialog::SetController**void SetController(wxHtmlHelpController* controller)**

Sets the help controller associated with the dialog.

wxHtmlHelpDialog::SetTitleFormat**void SetTitleFormat(const wxString& format)**

Sets the dialog's title format. *format* must contain exactly one "%s" (it will be replaced by the page title).

wxHtmlHelpDialog::WriteCustomization**void WriteCustomization(wxConfigBase* cfg, const wxString& path = wxEmptyString)**

Saves the user's settings for this dialog (see *wxHtmlHelpController::WriteCustomization* (p. 726)).

wxHtmlHelpFrame

This class is used by *wxHtmlHelpController* (p. 721) 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. 582)

Include files

<wx/html/helpfrm.h>

wxHtmlHelpFrame::wxHtmlHelpFrame**wxHtmlHelpFrame(wxHtmlHelpData* data = NULL)****wxHtmlHelpFrame(wxWindow* parent, int windowID, const wxString& title = wxEmptyString, int style = wxHF_DEFAULT_STYLE, wxHtmlHelpData* data = NULL)**

Constructor. For the values of *style*, please see the documentation for *wxHtmlHelpController* (p. 721).

wxHtmlHelpFrame::AddToolBarButtons**virtual void AddToolBarButtons(wxToolBar* toolBar, int style)**

You may override this virtual method to add more buttons to the help window's toolbar. *toolBar* is a pointer to the toolbar and *style* is the style flag as passed to the `Create` method.

`wxToolBar::Realize` is called immediately after returning from this function.

wxHtmlHelpFrame::Create

bool Create(wxWindow* parent, wxWindowID id, const wxString& title = wxEmptyString, int style = wxHF_DEFAULT_STYLE)

Creates the frame. See *the constructor* (p. 729) for a description of the parameters.

wxHtmlHelpFrame::GetController

wxHtmlHelpController* GetController() const

Returns the help controller associated with the frame.

wxHtmlHelpFrame::ReadCustomization

void ReadCustomization(wxConfigBase* cfg, const wxString& path = wxEmptyString)

Reads the user's settings for this frame see *wxHtmlHelpController::ReadCustomization* (p. 725))

wxHtmlHelpFrame::SetController

void SetController(wxHtmlHelpController* controller)

Sets the help controller associated with the frame.

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::WriteCustomization

void WriteCustomization(wxConfigBase* cfg, const wxString& path = wxEmptyString)

Saves the user's settings for this frame (see *wxHtmlHelpController::WriteCustomization* (p. 726)).

wxHtmlHelpWindow

This class is used by *wxHtmlHelpController* (p. 721) to display help within a frame or dialog, but you can use it yourself to create an embedded HTML help window.

For example:

```
// m_embeddedHelpWindow is a wxHtmlHelpWindow
// m_embeddedHtmlHelp is a wxHtmlHelpController

// Create embedded HTML Help window
m_embeddedHelpWindow = new wxHtmlHelpWindow;
m_embeddedHtmlHelp.UseConfig(config, rootPath); // Set your
own config object here
m_embeddedHtmlHelp.SetHelpWindow(m_embeddedHelpWindow);
m_embeddedHelpWindow->Create(this,
    wxID_ANY, wxDefaultPosition, GetClientSize(),
    wxTAB_TRAVERSAL|wxNO_BORDER, wxHF_DEFAULT_STYLE);
m_embeddedHtmlHelp.AddBook(wxFileName(_T("doc.zip")));
```

You should pass the style `wxHF_EMBEDDED` to the style parameter of *wxHtmlHelpController* to allow the embedded window to be destroyed independently of the help controller.

Derived from

wxWindow (p. **Error! Bookmark not defined.**)

Include files

<wx/html/helpwnd.h>

wxHtmlHelpWindow::wxHtmlHelpWindow

wxHtmlHelpWindow(wxHtmlHelpData* data = NULL)

wxHtmlHelpWindow(wxWindow* parent, int wxWindowID, int style = wxHF_DEFAULT_STYLE, wxHtmlHelpData* data = NULL)

Constructor.

Constructor. For the values of *style*, please see the documentation for *wxHtmlHelpController* (p. 721).

wxHtmlHelpWindow::Create

bool Create(wxWindow* parent, wxWindowID id, const wxString& title = wxEmptyString, int style = wxHF_DEFAULT_STYLE)

Creates the frame. See *the constructor* (p. 731) for a description of the parameters.

wxHtmlHelpWindow::CreateContents

void CreateContents()

Creates contents panel. (May take some time.)

Protected.

wxHtmlHelpWindow::CreateIndex

void CreateIndex()

Creates index panel. (May take some time.)

Protected.

wxHtmlHelpWindow::CreateSearch

void CreateSearch()

Creates search panel.

wxHtmlHelpWindow::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.

wxHtmlHelpWindow::DisplayContents

bool DisplayContents()

Displays contents panel.

wxHtmlHelpWindow::DisplayIndex

bool DisplayIndex()

Displays index panel.

wxHtmlHelpWindow::GetData**wxHtmlHelpData* GetData()**

Returns the wxHtmlHelpData object, which is usually a pointer to the controller's data.

wxHtmlHelpWindow::KeywordSearch**bool KeywordSearch(const wxString& keyword, wxHelpSearchMode mode = wxHELP_SEARCH_ALL)**

Search for given keyword. Optionally it searches through the index (mode = wxHELP_SEARCH_INDEX), default the content (mode = wxHELP_SEARCH_ALL).

wxHtmlHelpWindow::ReadCustomization**void ReadCustomization(wxConfigBase* cfg, const wxString& path = wxEmptyString)**

Reads the user's settings for this window (see *wxHtmlHelpController::ReadCustomization* (p. 725))

wxHtmlHelpWindow::RefreshLists**void RefreshLists()**

Refresh all panels. This is necessary if a new book was added.

Protected.

wxHtmlHelpWindow::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).

wxHtmlHelpWindow::UseConfig**void UseConfig(wxConfigBase* config, const wxString& rootpath = wxEmptyString)**

Associates a wxConfig object with the help window. It is recommended that you use *wxHtmlHelpController::UseConfig* (p. 726) instead.

wxHtmlHelpWindow::WriteCustomization**void WriteCustomization(wxConfigBase* cfg, const wxString& path = wxEmptyString)**

Saves the user's settings for this window(see *wxHtmlHelpController::WriteCustomization* (p. 726)).

wxHtmlHelpWindow::AddToolBarButtons

virtual void AddToolBarButtons(wxToolBar *toolBar, int style)

You may override this virtual method to add more buttons to the help window's toolbar. *toolBar* is a pointer to the toolbar and *style* is the style flag as passed to the Create method.

wxToolBar::Realize is called immediately after returning from this function.

See *samples/html/helpview* for an example.

wxHtmlModalHelp

This class uses *wxHtmlHelpController* (p. 721) to display help in a modal dialog. This is useful on platforms such as wxMac where if you display help from a modal dialog, the help window must itself be a modal dialog.

Create objects of this class on the stack, for example:

```
// The help can be browsed during the lifetime of this object;  
when the user quits  
// the help, program execution will continue.  
wxHtmlModalHelp help(parent, wxT("help"), wxT("My topic"));
```

Derived from

None

Include files

<wx/html/helpctrl.h>

wxHtmlModalHelp::wxHtmlModalHelp

wxHtmlModalHelp(wxWindow* parent, const wxString& helpFile, const wxString& topic = wxEmptyString, int style = wxHF_DEFAULT_STYLE | wxHF_DIALOG | wxHF_MODAL)

Parameters

parent is the parent of the dialog.

helpFile is the HTML help file to show.

topic is an optional topic. If this is empty, the help contents will be shown.

style is a combination of the flags described in the *wxHtmlHelpController* (p. 721) documentation.

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. **Error! Bookmark not defined.**)

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. 756), NULL otherwise.

wxHtmlLinkInfo::GetHtmlCell

const wxHtmlCell * GetHtmlCell()

Return pointer to the cell that was clicked. Valid only within *wxHtmlWindow::OnLinkClicked* (p. 756), 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. 735) opened).

wxHtmlListBox

wxHtmlListBox is an implementation of *wxVListBox* (p. **Error! Bookmark not defined.**) which shows HTML content in the listbox rows. This is still an abstract base class and you will need to derive your own class from it (see *htmlbox* sample for the example) but you will only need to override a single *OnGetItem()* (p. 738) function.

Derived from

wxVListBox (p. **Error! Bookmark not defined.**)

Include files

<wx/htmlbox.h>

wxHtmlListBox::wxHtmlListBox

wxHtmlListBox(wxWindow* *parent*, wxWindowID *id* = wxID_ANY, const wxPoint& *pos* = wxDefaultPosition, const wxSize& *size* = wxDefaultSize, long *style* = 0, const wxString& *name* = wxVListBoxNameStr)

Normal constructor which calls *Create()* (p. 736) internally.

wxHtmlListBox()

Default constructor, you must call *Create()* (p. 736) later.

wxHtmlListBox::~~wxHtmlListBox

~wxHtmlListBox()

Destructor cleans up whatever resources we use.

wxHtmlListBox::Create

bool Create(wxWindow* *parent*, wxWindowID *id* = wxID_ANY, const wxPoint& *pos* = wxDefaultPosition, const wxSize& *size* = wxDefaultSize, long *style* = 0, const wxString& *name* = wxVListBoxNameStr)

Creates the control and optionally sets the initial number of items in it (it may also be set or changed later with *SetItemCount()* (p. **Error! Bookmark not defined.**)).

There are no special styles defined for wxHtmlListBox, in particular the wxListBox styles can not be used here.

Returns *true* on success or *false* if the control couldn't be created

wxHtmlListBox::GetFileSystem**wxFileSystem& GetFileSystem()****const wxFileSystem& GetFileSystem() const**

Returns the *wxFileSystem* (p. 542) used by the HTML parser of this object. The file system object is used to resolve the paths in HTML fragments displayed in the control and you should use *wxFileSystem::ChangePathTo* (p. 543) if you use relative paths for the images or other resources embedded in your HTML.

wxHtmlListBox::GetSelectedTextBgColour**wxColour GetSelectedTextBgColour(const wxColour& colBg) const**

This virtual function may be overridden to change the appearance of the background of the selected cells in the same way as *GetSelectedTextColour* (p. 737).

It should be rarely, if ever, used because *SetSelectionBackground* (p. **Error! Bookmark not defined.**) allows to change the selection background for all cells at once and doing anything more fancy is probably going to look strangely.

See also

GetSelectedTextColour (p. 737)

wxHtmlListBox::GetSelectedTextColour**wxColour GetSelectedTextColour(const wxColour& colFg) const**

This virtual function may be overridden to customize the appearance of the selected cells. It is used to determine how the colour *colFg* is going to look inside selection. By default all original colours are completely ignored and the standard, system-dependent, selection colour is used but the program may wish to override this to achieve some custom appearance.

See also

GetSelectedTextBgColour (p. 737),
SetSelectionBackground (p. **Error! Bookmark not defined.**),
wxSystemSettings::GetColour (p. **Error! Bookmark not defined.**)

wxHtmlListBox::OnGetItem**wxString OnGetItem(size_t n) const**

This method must be implemented in the derived class and should return the body (i.e. without `<html>` nor `<body>` tags) of the HTML fragment for the given item.

wxHtmlListBox::OnGetItemMarkup**wxString OnGetItemMarkup(size_t n) const**

This function may be overridden to decorate HTML returned by *OnGetItem()* (p. 738).

wxHtmlParser

Classes derived from this handle the **generic** parsing of HTML documents: it scans the document and divide it into blocks of tags (where one block consists of beginning and ending tag and of text between these two tags).

It is independent from wxHtmlWindow and can be used as stand-alone parser (Julian Smart's idea of speech-only HTML viewer or wget-like utility - see InetGet sample for example).

It uses system of tag handlers to parse the HTML document. Tag handlers are not statically shared by all instances but are created for each wxHtmlParser instance. The reason is that the handler may contain document-specific temporary data used during parsing (e.g. complicated structures like tables).

Typically the user calls only the *Parse* (p. 741) method.

Derived from

wxObject

Include files

<wx/html/htmlpars.h>

See also

Cells Overview (p. **Error! Bookmark not defined.**), *Tag Handlers Overview* (p. **Error! Bookmark not defined.**), *wxHtmlTag* (p. 745)

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. 745) 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. 739) each time a part of text is parsed. *txt* is NOT only one word, it is substring of input. It is not formatted or preprocessed (so white spaces are unmodified).

wxHtmlParser::DoParsing**void DoParsing(int begin_pos, int end_pos)****void DoParsing()**

Parses the m_Source from begin_pos to end_pos-1. (in noparams version it parses whole m_Source)

wxHtmlParser::DoneParser**virtual void DoneParser()**

This must be called after DoParsing().

wxHtmlParser::GetFS**wxFileSystem* GetFS() const**

Returns pointer to the file system. Because each tag handler has reference to it is parent parser it can easily request the file by calling

```
wxFSFile *f = m_Parser -> GetFS() -> OpenFile("image.jpg");
```

wxHtmlParser::GetProduct**virtual wxObject* GetProduct()**

Returns product of parsing. Returned value is result of parsing of the document. The

type of this result depends on internal representation in derived parser (but it must be derived from `wxObject!`).

See `wxHtmlWinParser` for details.

wxHtmlParser::GetSource

wxString* GetSource()

Returns pointer to the source being parsed.

wxHtmlParser::InitParser

virtual void InitParser(const wxString& source)

Setups the parser for parsing the *source* string. (Should be overridden in derived class)

wxHtmlParser::OpenURL

virtual wxFSFile* OpenURL(wxHtmlURLType type, const wxString& url)

Opens given URL and returns `wxFSFile` object that can be used to read data from it. This method may return NULL in one of two cases: either the URL doesn't point to any valid resource or the URL is blocked by overridden implementation of *OpenURL* in derived class.

Parameters

type

Indicates type of the resource. Is one of:

wxHTML_URL_PAGE	Opening a HTML page.
wxHTML_URL_IMAGE	Opening an image.
wxHTML_URL_OTHER	Opening a resource that doesn't fall into any other category.

url

URL being opened.

Notes

Always use this method in tag handlers instead of `GetFS()->OpenFile()` because it can block the URL and is thus more secure.

Default behaviour is to call `wxHtmlWindow::OnOpeningURL` (p. 756) of the associated `wxHtmlWindow` object (which may decide to block the URL or redirect it to another one), if there's any, and always open the URL if the parser is not used with `wxHtmlWindow`.

Returned `wxFsFile` object is not guaranteed to point to *url*, it might have been redirected!

wxHtmlParser::Parse

wxObject* Parse(const wxString& source)

Proceeds parsing of the document. This is end-user method. You can simply call it when you need to obtain parsed output (which is parser-specific)

The method does these things:

1. calls *InitParser(source)* (p. 740)
2. calls *DoParsing* (p. 739)
3. calls *GetProduct* (p. 740)
4. calls *DoneParser* (p. 740)
5. returns value returned by *GetProduct*

You shouldn't use *InitParser*, *DoParsing*, *GetProduct* or *DoneParser* directly.

wxHtmlParser::PushTagHandler

void PushTagHandler(wxHtmlTagHandler* handler, const wxString& tags)

Forces the handler to handle additional tags (not returned by *GetSupportedTags* (p. 749)). 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. 742)).

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. 741).

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.)

wxHtmlParser::StopParsing

void StopParsing()

Call this function to interrupt parsing from a tag handler. No more tags will be parsed afterward. This function may only be called from *wxHtmlParser::Parse* (p. 741) or any function called by it (i.e. from tag handlers).

wxHtmlPrintout

This class serves as printout class for HTML documents.

Derived from

wxPrintout (p. **Error! Bookmark not defined.**)

Include files

`<wx/html/htmprint.h>`

wxHtmlPrintout::wxHtmlPrintout**wxHtmlPrintout**(const wxString& *title* = "Printout")

Constructor.

wxHtmlPrintout::AddFilter**static void AddFilter**(wxHtmlFilter* *filter*)

Adds a filter to the static list of filters for wxHtmlPrintout. See *wxHtmlFilter* (p. 720) for further information.

wxHtmlPrintout::SetFont**void SetFont**(const wxString& *normal_face*, const wxString& *fixed_face*, const int **sizes* = NULL)

Sets fonts. See *wxHtmlWindow::SetFont* (p. 758) for detailed description.

wxHtmlPrintout::SetFooter**void SetFooter**(const wxString& *footer*, int *pg* = wxPAGE_ALL)

Sets page footer.

Parameters

footer

HTML text to be used as footer. You can use macros in it:

- @PAGENUM@ is replaced by page number
- @PAGECNT@ 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. 542) 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. **Error! Bookmark not defined.**).

Derived from

wxObject

Include files

<wx/html/htmltag.h>

wxHtmlTag::wxHtmlTag

wxHtmlTag(wxHtmlTag *parent, const wxString& source, int pos, int end_pos, wxHtmlTagsCache* cache, wxHtmlEntitiesParser *entParser)

Constructor. You will probably never have to construct a wxHtmlTag object yourself. Feel free to ignore the constructor parameters. Have a look at src/html/htmlpars.cpp if you're interested in creating it.

wxHtmlTag::GetAllParams

const wxString& GetAllParams() const

Returns a string containing all parameters.

Example : tag contains . Call to tag.GetAllParams() would return SIZE=+2 COLOR="#000000".

wxHtmlTag::GetBeginPos

int GetBeginPos() const

Returns beginning position of the text *between* this tag and paired ending tag. See explanation (returned position is marked with '|'):

```
bla bla bla <MYTAG> | bla bla internal text</MYTAG> bla bla
```

wxHtmlTag::GetEndPos1

int GetEndPos1() const

Returns ending position of the text *between* this tag and paired ending tag. See explanation (returned position is marked with '|'):

```
bla bla bla <MYTAG> bla bla internal text</MYTAG> | bla bla
```

wxHtmlTag::GetEndPos2

int GetEndPos2() const

Returns ending position 2 of the text *between* this tag and paired ending tag. See explanation (returned position is marked with '|'):

```
bla bla bla <MYTAG> bla bla internal text</MYTAG> bla bla
```

|

wxHtmlTag::GetName**wxString GetName() const**

Returns tag's name. The name is always in uppercase and it doesn't contain '<' or '/' characters. (So the name of tag is "FONT" and name of </table> is "TABLE")

wxHtmlTag::GetParam**wxString GetParam(const wxString& par, bool with_commas = false) const**

Returns the value of the parameter. You should check whether the parameter exists or not (use *HasParam* (p. 747)) first.

Parameters

par

The parameter's name.

with_commas

true if you want to get commas as well. See example.

Example

```
...
/* you have wxHtmlTag variable tag which is equal to
   HTML tag <FONT SIZE=+2 COLOR="#0000FF"> */
dummy = tag.GetParam("SIZE");
// dummy == "+2"
dummy = tag.GetParam("COLOR");
// dummy == "#0000FF"
dummy = tag.GetParam("COLOR", true);
// dummy == "\"#0000FF\"" -- see the difference!!
```

wxHtmlTag::GetParamAsColour**bool GetParamAsColour(const wxString& par, wxColour *clr) const**

Interprets tag parameter *par* as colour specification and saves its value into *wxColour* variable pointed by *clr*.

Returns true on success and false if *par* is not colour specification or if the tag has no such parameter.

wxHtmlTag::GetParamAsInt**bool GetParamAsInt(const wxString& par, int *value) const**

Interprets tag parameter *par* as an integer and saves its value into *int* variable pointed by

value.

Returns true on success and false if *par* is not an integer or if the tag has no such parameter.

wxHtmlTag::HasEnding

bool HasEnding() const

Returns true if this tag is paired with ending tag, false otherwise.

See the example of HTML document:

```
<html><body>
Hello<p>
How are you?
<p align=center>This is centered...</p>
Oops<br>Oooops!
</body></html>
```

In this example tags HTML and BODY have ending tags, first P and BR doesn't have ending tag while the second P has. The third P tag (which is ending itself) of course doesn't have ending tag.

wxHtmlTag::HasParam

bool HasParam(const wxString& par) const

Returns true if the tag has a parameter of the given name. Example : has two parameters named "SIZE" and "COLOR".

Parameters

par

the parameter you're looking for.

wxHtmlTag::ScanParam

wxString ScanParam(const wxString& par, const wxChar *format, void *value) const

This method scans the given parameter. Usage is exactly the same as sscanf's usage except that you don't pass a string but a parameter name as the first argument and you can only retrieve one value (i.e. you can use only one "%" element in *format*).

Parameters

par

The name of the tag you want to query

format

scanf()-like format string.

value

pointer to a variable to store the value in

wxHtmlTagHandler

Derived from

wxObject (p. **Error! Bookmark not defined.**)

Include files

<wx/html/htmlpars.h>

See Also

Overview (p. **Error! Bookmark not defined.**), *wxHtmlTag* (p. 745)

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. 745) for details).

Return value

true if *ParseInner* (p. 749) was called, false otherwise.

Example

```
bool MyHandler::HandleTag(const wxHtmlTag& tag)
{
    ...
    // change state of parser (e.g. set bold face)
    ParseInner(tag);
    ...
    // restore original state of parser
}
```

You shouldn't call *ParseInner* if the tag is not paired with an ending one.

wxHtmlTagHandler::ParseInner

void ParseInner(const wxHtmlTag& tag)

This method calls parser's *DoParsing* (p. 739) method for the string between this tag and the paired ending tag:

```
...<A HREF="x.htm">Hello, world!</A>...
```

In this example, a call to *ParseInner* (with *tag* pointing to A tag) will parse 'Hello, world!'.

wxHtmlTagHandler::SetParser

virtual void SetParser(wxHtmlParser *parser)

Assigns *parser* to this handler. Each **instance** of handler is guaranteed to be called only from the parser.

wxHtmlTagsModule

This class provides easy way of filling *wxHtmlWinParser*'s table of tag handlers. It is used almost exclusively together with the set of *TAGS_MODULE_** macros (p. **Error! Bookmark not defined.**)

Derived from

wxModule (p. **Error! Bookmark not defined.**)

Include files

```
<wx/html/winpars.h>
```

See Also

Tag Handlers (p. **Error! Bookmark not defined.**), *wxHtmlTagHandler* (p. 748), *wxHtmlWinTagHandler* (p. 766),

wxHtmlTagsModule::FillHandlersTable

virtual void FillHandlersTable(wxHtmlWinParser *parser)

You must override this method. In most common case its body consists only of lines of the following type:

```
parser -> AddTagHandler(new MyHandler);
```

I recommend using the **TAGS_MODULE_*** macros.

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. 704)

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. 759), *LoadPage(filename)* (p. 754) or *LoadFile* (p. 754).

Note

wxHtmlWindow uses the *wxImage* (p. 790) class for displaying images. Don't forget to initialize all image formats you need before loading any page! (See *wxInitAllImageHandlers* (p. **Error! Bookmark not defined.**) and *wxImage::AddHandler* (p. 795).)

Derived from

wxScrolledWindow (p. **Error! Bookmark not defined.**)

Include files

<wx/html/htmlwin.h>

Window styles

wxHW_SCROLLBAR_NEVER Never display scrollbars, not even when the page is larger than the window.

wxHW_SCROLLBAR_AUTO Display scrollbars only if page's size exceeds window's size.

wxHW_NO_SELECTION Don't allow the user to select text.

wxHtmlWindow::wxHtmlWindow

wxHtmlWindow()

Default constructor.

wxHtmlWindow(wxWindow *parent, wxWindowID id = -1, const wxPoint& pos = wxDefaultPosition, const wxSize& size = wxDefaultSize, long style = wxHW_DEFAULT_STYLE, const wxString& name = "htmlWindow")

Constructor. The parameters are the same as for the *wxScrolledWindow* (p. **Error! Bookmark not defined.**) constructor.

Parameters

style

Window style. See *wxHtmlWindow* (p. 751).

wxHtmlWindow::AddFilter**static void AddFilter**(wxHtmlFilter **filter*)

Adds *input filter* (p. **Error! Bookmark not defined.**) to the static list of available filters. These filters are present by default:

- text/html MIME type
- image/* MIME types
- Plain Text filter (this filter is used if no other filter matches)

wxHtmlWindow::AppendToPage**bool AppendToPage**(const wxString& *source*)

Appends HTML fragment to currently displayed text and refreshes the window.

Parameters*source*

HTML code fragment

Return value

false if an error occurred, true otherwise.

wxHtmlWindow::GetInternalRepresentation**wxHtmlContainerCell* GetInternalRepresentation()** const

Returns pointer to the top-level container.

See also: *Cells Overview* (p. **Error! Bookmark not defined.**), *Printing Overview* (p. **Error! Bookmark not defined.**)

wxHtmlWindow::GetOpenedAnchor**wxString GetOpenedAnchor()**

Returns anchor within currently opened page (see *GetOpenedPage* (p. 753)). 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. 754) is stored in history list.)

wxHtmlWindow::HistoryCanBack**bool HistoryCanBack()**

Returns true if it is possible to go back in the history (i.e. HistoryBack() won't fail).

wxHtmlWindow::HistoryCanForward**bool HistoryCanForward()**

Returns true if it is possible to go forward in the history (i.e. HistoryBack() won't fail).

wxHtmlWindow::HistoryClear**void HistoryClear()**

Clears history.

wxHtmlWindow::HistoryForward**bool HistoryForward()**

Moves to next page in history.

wxHtmlWindow::LoadFile**virtual bool LoadFile(const wxFileName& filename)**

Loads HTML page from file and displays it.

Return value

false if an error occurred, true otherwise

See also

LoadPage (p. 754)

wxHtmlWindow::LoadPage

virtual bool LoadPage(const wxString& location)

Unlike SetPage this function first loads HTML page from *location* and then displays it. See example:

```
htmlwin->LoadPage( "help/myproject/index.htm" );
```

Parameters

location

The address of document. See *wxFileSystem* (p. 542) for details on address format and behaviour of "opener".

Return value

false if an error occurred, true otherwise

See also

LoadFile (p. 754)

wxHtmlWindow::OnCellClicked

virtual void OnCellClicked(wxHtmlCell *cell, wxCoord x, wxCoord y, const wxMouseEvent& event)

This method is called when a mouse button is clicked inside wxHtmlWindow. The default behaviour is to call *OnLinkClicked* (p. 756) if the cell contains a hypertext link.

Parameters

cell

The cell inside which the mouse was clicked, always a simple (i.e. non container) cell

x, y

The logical coordinates of the click point

event

The mouse event containing other information about the click

wxHtmlWindow::OnCellMouseHover**virtual void OnCellMouseHover(wxHtmlCell *cell, wxCoord x, wxCoord y)**

This method is called when a mouse moves over an HTML cell.

Parameters*cell*

The cell inside which the mouse is currently, always a simple (i.e. non container) cell

x, y

The logical coordinates of the click point

wxHtmlWindow::OnLinkClicked**virtual void OnLinkClicked(const wxHtmlLinkInfo& link)**

Called when user clicks on hypertext link. Default behaviour is to call *LoadPage* (p. 754) and do nothing else.

Also see *wxHtmlLinkInfo* (p. 735).

wxHtmlWindow::OnOpeningURL**virtual wxHtmlOpeningStatus OnOpeningURL(wxHtmlURLType type, const wxString& url, wxString *redirect)**

Called when an URL is being opened (either when the user clicks on a link or an image is loaded). The URL will be opened only if *OnOpeningURL* returns `wxHTML_OPEN`. This method is called by *wxHtmlParser::OpenURL* (p. 740). You can override *OnOpeningURL* to selectively block some URLs (e.g. for security reasons) or to redirect them elsewhere. Default behaviour is to always return `wxHTML_OPEN`.

Parameters*type*

Indicates type of the resource. Is one of **wxHTML_URL_PAGE** Opening a HTML page.

wxHTML_URL_IMAGE Opening an image.

wxHTML_URL_OTHER Opening a resource that doesn't fall into any other category.

url

URL being opened.

redirect

Pointer to `wxString` variable that must be filled with an URL if `OnOpeningURL` returns `wxHTML_REDIRECT`.

Return value`wxHTML_OPEN` Open the URL.

`wxHTML_BLOCK` Deny access to the URL, `wxHtmlParser::OpenURL` (p. 740) will return `NULL`.

`wxHTML_REDIRECT` Don't open *url*, redirect to another URL. `OnOpeningURL` must fill **redirect* with the new URL. `OnOpeningURL` will be called again on returned URL.

`wxHtmlWindow::SetTitle`

`virtual void SetTitle(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 `SetFont`s, `SetBorder`s.

Parameters

cfg

`wxConfig` from which you want to read the configuration.

path

Optional path in config tree. If not given current path is used.

`wxHtmlWindow::SelectAll`

`void SelectAll()`

Selects all text in the window.

See also

SelectLine (p. 757), *SelectWord* (p. 758)

`wxHtmlWindow::SelectionToText`

wxString SelectionToText()

Returns current selection as plain text. Returns empty string if no text is currently selected.

wxHtmlWindow::SelectLine**void SelectLine(const wxPoint& pos)**

Selects the line of text that *pos* points at. Note that *pos* is relative to the top of displayed page, not to window's origin, use *CalcUnscrolledPosition* (p. **Error! Bookmark not defined.**) to convert physical coordinate.

See also

SelectAll (p. 757), *SelectWord* (p. 758)

wxHtmlWindow::SelectWord**void SelectWord(const wxPoint& pos)**

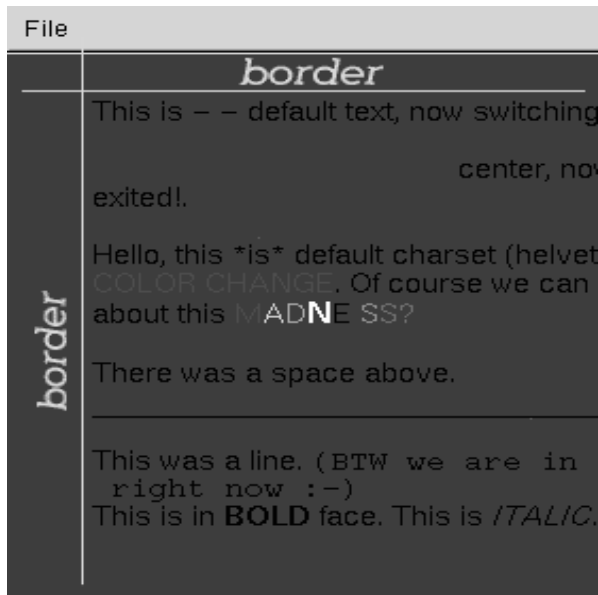
Selects the word at position *pos*. Note that *pos* is relative to the top of displayed page, not to window's origin, use *CalcUnscrolledPosition* (p. **Error! Bookmark not defined.**) to convert physical coordinate.

See also

SelectAll (p. 757), *SelectLine* (p. 757)

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(const wxString& normal_face, const wxString& fixed_face, const int *sizes = NULL)

This function sets font sizes and faces.

Parameters

normal_face

This is face name for normal (i.e. non-fixed) font. It can be either empty string (then the default face is chosen) or platform-specific face name. Examples are "helvetica" under Unix or "Times New Roman" under Windows.

fixed_face

The same thing for fixed face (<TT>..<</TT>)

sizes

This is an array of 7 items of *int* type. The values represent size of font with HTML size from -2 to +4 (to). Default sizes are used if *sizes* is NULL.

Defaults

Default font sizes are defined by constants wxHTML_FONT_SIZE_1,

wxHTML_FONT_SIZE_2, ..., wxHTML_FONT_SIZE_7. Note that they differ among platforms. Default face names are empty strings.

wxHtmlWindow::SetPage

bool SetPage(const wxString& source)

Sets HTML page and display it. This won't **load** the page!! It will display the *source*. See example:

```
htmlwin -> SetPage("<html><body>Hello, world!</body></html>");
```

If you want to load a document from some location use *LoadPage* (p. 754) 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. 759), this sets statusbar slot where messages will be displayed. (Default is -1 = no messages.)

Parameters

bar

statusbar slot number (0..n)

wxHtmlWindow::ToText

wxString ToText()

Returns content of currently displayed page as plain text.

wxHtmlWindow::WriteCustomization

virtual void WriteCustomization(wxConfigBase *cfg, wxString path = wxEmptyString)

Saves custom settings into wxConfig. It uses the path 'path' if given, otherwise it saves info into currently selected path. Regardless of whether the path is given or not, the function creates sub-path wxHtmlWindow.

Saved values: all things set by SetFonts, SetBorders.

Parameters

cfg

wxConfig to which you want to save the configuration.

path

Optional path in config tree. If not given, the current path is used.

wxHtmlWinParser

This class is derived from *wxHtmlParser* (p. 738) and its main goal is to parse HTML input so that it can be displayed in *wxHtmlWindow* (p. 751). It uses a special *wxHtmlWinTagHandler* (p. 766).

Notes

The product of parsing is a wxHtmlCell (resp. wxHtmlContainer) object.

Derived from

wxHtmlParser (p. 738)

Include files

<wx/html/winpars.h>

See Also

Handlers overview (p. **Error! Bookmark not defined.**)

wxHtmlWinParser::wxHtmlWinParser

wxHtmlWinParser()

wxHtmlWinParser(wxHtmlWindow *wnd)

Constructor. Don't use the default one, use constructor with *wnd* parameter (*wnd* is pointer to associated *wxHtmlWindow* (p. 751))

wxHtmlWinParser::AddModule

static void AddModule(wxHtmlTagsModule *module)

Adds *module* (p. **Error! Bookmark not defined.**) 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. **Error! Bookmark not defined.**)).

wxHtmlWinParser::CreateCurrentFont

virtual wxFont* CreateCurrentFont()

Creates font based on current setting (see *SetFontSize* (p. 765), *SetFontBold* (p. 765), *SetFontItalic* (p. 765), *SetFontFixed* (p. 765), *SetFontUnderlined* (p. 765)) and returns pointer to it. If the font was already created only a pointer is returned.

wxHtmlWinParser::GetActualColor

const wxColour& GetActualColor() const

Returns actual text colour.

wxHtmlWinParser::GetAlign

int GetAlign() const

Returns default horizontal alignment.

wxHtmlWinParser::GetCharHeight

int GetCharHeight() const

Returns (average) char height in standard font. It is used as DC-independent metrics.

Note: This function doesn't return the *actual* height. If you want to know the height of the current font, call `GetDC -> GetCharHeight()`.

wxHtmlWinParser::GetCharWidth

int GetCharWidth() const

Returns average char width in standard font. It is used as DC-independent metrics.

Note: This function doesn't return the *actual* width. If you want to know the height of the current font, call `GetDC -> GetCharWidth()`

wxHtmlWinParser::GetContainer**wxHtmlContainerCell* GetContainer() const**

Returns pointer to the currently opened container (see *Overview* (p. **Error! Bookmark not defined.**)). 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. 482) class used to do conversion between *input encoding* (p. 763) and *output encoding* (p. 764).

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. 735) 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

wxHtmlWindow* GetWindow()

Returns associated window (`wxHtmlWindow`). This may be `NULL`! (You should always test if it is non-`NULL`. For example `TITLE` handler sets window title only if some window is associated, otherwise it does nothing)

wxHtmlWinParser::OpenContainer

wxHtmlContainerCell* OpenContainer()

Opens new container and returns pointer to it (see *Overview* (p. **Error! Bookmark not defined.**)).

wxHtmlWinParser::SetActualColor

void SetActualColor(const wxColour& clr)

Sets actual text colour. Note: this DOESN'T change the colour! You must create *wxHtmlColourCell* (p. 709) yourself.

wxHtmlWinParser::SetAlign**void SetAlign(int a)**

Sets default horizontal alignment (see *wxHtmlContainerCell::SetAlignHor* (p. 711).) 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. 741)! *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 or 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::SetFonts

void SetFonts(const wxString& *normal_face*, const wxString& *fixed_face*, const int **sizes* = NULL)

Sets fonts. See `wxHtmlWindow::SetFonts` (p. 758) for detailed description.

wxHtmlWinParser::SetInputEncoding

void SetInputEncoding(wxFontEncoding *enc*)

Sets input encoding. The parser uses this information to build conversion tables from document's encoding to some encoding supported by operating system.

wxHtmlWinParser::SetLink

void SetLink(const wxHtmlLinkInfo& *link*)

Sets actual hypertext link. Empty link is represented by `wxHtmlLinkInfo` (p. 735) 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. 748)

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. **Error! Bookmark not defined.**)

Include files

<wx/protocol/http.h>

See also

wxSocketBase (p. **Error! Bookmark not defined.**), *wxURL* (p. **Error! Bookmark not defined.**)

wxHTTP::GetResponse

int GetResponse() const

Returns the HTTP response code returned by the server. Please refer to RFC 2616 (<http://www.faqs.org/rfcs/rfc2616.html>) for the list of responses.

wxHTTP::GetInputStream

wxInputStream * GetInputStream(const wxString& path)

Creates a new input stream on the specified path. You can use all except the seek functionality of wxStream. Seek isn't available on all streams. For example, http or ftp streams doesn't deal with it. Other functions like Tell and 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. **Error! Bookmark not defined.**). 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. 827):

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 since wxHTTP reestablishes it automatically.

See also

wxInputStream (p. 826)

wxHTTP::SetHeader

void SetHeader(const wxString& header, const wxString& h_data)

It sets data of a field to be sent during the next request to the HTTP server. The field name is specified by *header* and the content by *h_data*. This is a low level function and it assumes that you know what you are doing.

wxHTTP::GetHeader

wxString GetHeader(const wxString& header)

Returns the data attached with a field whose name is specified by *header*. If the field doesn't exist, it will return an empty string and not a NULL string.

Note

The header is not case-sensitive, i.e. "CONTENT-TYPE" and "content-type" represent the same header.

wxHVScrolledWindow

This class is strongly influenced by *wxVScrolledWindow* (p. **Error! Bookmark not defined.**). Like *wxVScrolledWindow*, this class is here to provide an easy way to implement variable line sizes. The difference is that *wxVScrolledWindow* only works with vertical scrolling. This class extends the behavior of *wxVScrolledWindow* to the horizontal axis in addition to the vertical axis.

The scrolling is also "virtual" in the sense that row widths and column heights only need to be known for the rows and columns that are currently visible.

Like *wxVScrolledWindow* (p. **Error! Bookmark not defined.**), this is a generalization of the *wxScrolledWindow* (p. **Error! Bookmark not defined.**) class which can be only used when all rows have a constant height and columns have a constant width. Like *wxVScrolledWindow* it lacks some of *wxScrolledWindow* features such as scrolling another window or only scrolling a rectangle of the window and not its entire client area.

If only vertical scrolling is needed, *wxVScrolledWindow* is recommended because it is

simpler to use. There is no `wxHScrolledWindow` but horizontal-only scrolling is implemented easily enough with this class.

To use this class, you need to derive from it and implement both the `OnGetRowHeight()` (p. 774) and the `OnGetColumnWidth()` (p. 773) pure virtual methods. You also must call `SetRowColumnCounts` (p. 778) to let the base class know how many rows and columns it should display. After these requirements are met scrolling is handled entirely by `wxHVScrolledWindow`. You only need to draw the visible part of contents in your `OnPaint()` method as usual. You should use `GetVisibleRowsBegin()` (p. 772), `GetVisibleColumnsBegin()` (p. 771), `GetVisibleRowsEnd()` (p. 772), and `GetVisibleColumnsEnd()` (p. 772) to determine which lines to display. If physical scrolling is enabled the device context origin is shifted by the scroll position (through `PrepareDC()`), child windows are moved as the window scrolls, and the pixels on the screen are moved to minimize the region that requires painting. Physical scrolling is enabled by default.

Derived from

`wxPanel` (p. **Error! Bookmark not defined.**)

Include files

<wx/vscroll.h>

wxHVScrolledWindow::wxHVScrolledWindow

wxHVScrolledWindow(*wxWindow** parent, **wxWindowID** id = `wxID_ANY`, **const wxPoint&** pos = `wxDefaultPosition`, **const wxSize&** size = `wxDefaultSize`, **long** style = 0, **const wxString&** name = `wxPanelNameStr`)

This is the normal constructor, no need to call `Create()` after using this one.

Note that `wxVSCROLL` and `wxHSCROLL` are always automatically added to our style, there is no need to specify them explicitly.

wxHVScrolledWindow()

Default constructor, you must call `Create()` (p. 769) later.

Parameters

parent

The parent window, must not be `NULL`

id

The identifier of this window, `wxID_ANY` by default

pos

The initial window position

size

The initial window size

style

The window style. There are no special style bits defined for this class.

name

The name for this window; usually not used

wxHVScrolledWindow::Create

bool Create(wxWindow* *parent*, wxWindowID *id* = wxID_ANY, const wxPoint& *pos* = wxDefaultPosition, const wxSize& *size* = wxDefaultSize, long *style* = 0, const wxString& *name* = wxPanelNameStr)

Same as the *non default ctor* (p. 769) but returns status code: `true` if ok, `false` if the window couldn't have been created.

Just as with the ctor above, both the `wxVSCROLL` and the `wxHSCROLL` styles are always used. There is no need to specify either explicitly.

wxHVScrolledWindow::EnablePhysicalScrolling

EnablePhysicalScrolling(bool *scrolling* = `true`)

With physical scrolling enabled the device origin is changed properly when a wxDC is prepared using `PrepareDC()`, children are actually moved and layed out according to the current scroll position, and the contents of the window (pixels) are actually moved to reduce the amount of redraw needed.

Physical scrolling is enabled by default but can be disabled or re-enabled at any time. An example of when you'd want to disable it would be if you have statically positioned graphic elements or children you do not want to move while the window is being scrolled. If you disable physical scrolling you must manually adjust positioning for items within the scrolled window yourself. Also note that an unprepared wxDC requires you to do the same, regardless of the physical scrolling state.

wxHVScrolledWindow::EstimateTotalHeight

virtual wxCoord EstimateTotalHeight() const

This protected function is used internally by wxHVScrolledWindow to estimate the total height of the window when `SetRowColumnCounts` (p. 778) is called. The default implementation uses the brute force approach if the number of the items in the control is small enough. Otherwise, it tries to find the average row height using some rows in the beginning, middle and the end.

If it is undesirable to query all these rows (some of which might be never shown) just for the total height calculation, you may override the function and provide your own guess

using a better and/or faster method.

Note that although returning a totally wrong value would still work, it risks causing some very strange scrollbar behaviour so this function should really try to make the best guess possible.

wxHVScrolledWindow::EstimateTotalWidth

virtual wxCoord EstimateTotalWidth() const

This protected function is used internally by `wxHVScrolledWindow` to estimate the total width of the window when `SetRowColumnCounts` (p. 778) is called. The default implementation uses the brute force approach if the number of the items in the control is small enough. Otherwise, it tries to find the average column width using some columns in the beginning, middle and the end.

If it is undesirable to query all these columns (some of which might be never shown) just for the total width calculation, you may override the function and provide your own guess using a better and/or faster method.

Note that although returning a totally wrong value would still work, it risks causing some very strange scrollbar behaviour so this function should really try to make the best guess possible.

wxHVScrolledWindow::GetColumnCount

wxSize GetColumnCount() const

Get the number of columns this window contains (previously set by `SetRowColumnCounts()` (p. 778))

wxHVScrolledWindow::GetRowCount

wxSize GetRowCount() const

Get the number of rows this window contains (previously set by `SetRowColumnCounts()` (p. 778))

wxHVScrolledWindow::GetRowColumnCounts

wxSize GetRowColumnCounts() const

Get the number of rows (X or width) and columns (Y or height) this window contains (previously set by `SetRowColumnCounts()` (p. 778))

wxHVScrolledWindow::GetVisibleBegin

wxPoint GetVisibleBegin() const

Returns the indices of the first visible row (Y) and column (X).

See also

GetVisibleRowsBegin (p. 772), *GetVisibleColumnsBegin* (p. 771)

wxHVSrolledWindow::GetVisibleColumnsBegin

size_t GetVisibleColumnsBegin() const

Returns the index of the first currently visible column.

See also

GetVisibleColumnsEnd (p. 772)

wxHVSrolledWindow::GetVisibleColumnsEnd

size_t GetVisibleColumnsEnd() const

Returns the index of the first column after the currently visible page. If the return value is 0 it means that no columns are currently shown (which only happens if the control is empty). Note that the index returned by this method is not always a valid index as it may be equal to *GetColumnCount* (p. 771).

See also

GetVisibleColumnsBegin (p. 771)

wxHVSrolledWindow::GetVisibleEnd

wxPoint GetVisibleEnd() const

Returns the indices of the row and column after the last visible row (Y) and last visible column (X), respectively.

See also

GetVisibleRowsEnd (p. 772), *GetVisibleColumnsEnd* (p. 772)

wxHVSrolledWindow::GetVisibleRowsBegin

size_t GetVisibleRowsBegin() const

Returns the index of the first currently visible row.

See also

GetVisibleRowsEnd (p. 772)

wxHVSrolledWindow::GetVisibleRowsEnd

size_t GetVisibleRowsEnd() const

Returns the index of the first row after the currently visible page. If the return value is 0 it means that no rows are currently shown (which only happens if the control is empty). Note that the index returned by this method is not always a valid index as it may be equal to *GetRowCount* (p. 771).

See also

GetVisibleRowsBegin (p. 772)

wxHVScrolledWindow::HitTest

wxPoint HitTest(wxCoord x, wxCoord y) const

wxPoint HitTest(const wxPoint& pt) const

Return the position (X as column, Y as row) of the cell occupying the specified position (in physical coordinates). A value of `wxNOT_FOUND` in either X, Y, or X and Y means it is outside the range available rows and/or columns.

wxHVScrolledWindow::IsColumnVisible

bool IsColumnVisible(size_t column) const

Returns `true` if the given column is at least partially visible or `false` otherwise.

wxHVScrolledWindow::IsRowVisible

bool IsRowVisible(size_t row) const

Returns `true` if the given row is at least partially visible or `false` otherwise.

wxHVScrolledWindow::IsVisible

bool IsVisible(size_t row, size_t column) const

Returns `true` if the given row and column are both at least partially visible or `false` otherwise.

wxHVScrolledWindow::OnGetColumnWidth

wxCoord OnGetColumnWidth(size_t n) const

This protected pure virtual function must be overridden in the derived class and should return the width of the given column in pixels.

See also

OnGetColumnsWidthHint (p. 773)

wxHVScrolledWindow::OnGetColumnsWidthHint

void OnGetColumnsWidthHint(size_t columnMin, size_t columnMax) const

This function doesn't have to be overridden but it may be useful to do it if calculating the columns' heights is a relatively expensive operation as it gives the user code a possibility to calculate several of them at once.

`OnGetColumnsWidthHint()` is normally called just before `OnGetColumnWidth()` (p. 773) but you shouldn't rely on the latter being called for all columns in the interval specified here. It is also possible that `OnGetColumnWidth()` will be called for the columns outside of this interval, so this is really just a hint, not a promise.

Finally note that *columnMin* is inclusive, while *columnMax* is exclusive, as usual.

wxHVSrolledWindow::OnGetRowHeight**wxCoord OnGetRowHeight(size_t n) const**

This protected pure virtual function must be overridden in the derived class and should return the height of the given row in pixels.

See also

OnGetRowsHeightHint (p. 774)

wxHVSrolledWindow::OnGetRowsHeightHint**void OnGetRowsHeightHint(size_t rowMin, size_t rowMax) const**

This function doesn't have to be overridden but it may be useful to do it if calculating the row's heights is a relatively expensive operation as it gives the user code a possibility to calculate several of them at once.

`OnGetRowsHeightHint()` is normally called just before `OnGetRowHeight()` (p. 774) but you shouldn't rely on the latter being called for all rows in the interval specified here. It is also possible that `OnGetRowHeight()` will be called for the rows outside of this interval, so this is really just a hint, not a promise.

Finally note that *rowMin* is inclusive, while *rowMax* is exclusive, as usual.

wxHVSrolledWindow::RefreshColumn**void RefreshColumn(size_t column)**

Refreshes the specified column -- it will be redrawn during the next main loop iteration.

wxHVSrolledWindow::RefreshRow**void RefreshRow(size_t row)**

Refreshes the specified row -- it will be redrawn during the next main loop iteration.

wxHVScrolledWindow::RefreshRowColumn**void RefreshRowColumn(size_t row, size_t column)**

Refreshes the specified cell -- it will be redrawn during the next main loop iteration.

See also

RefreshRowsColumns (p. 775)

wxHVScrolledWindow::RefreshColumns**void RefreshColumns(size_t fromColumn, size_t toColumn)**

Refreshes the columns between *fromColumn* and *toColumn* (inclusive). *fromColumn* should be less than or equal to *toColumn*.

See also

RefreshColumn (p. 774)

wxHVScrolledWindow::RefreshRows**void RefreshRows(size_t fromRow, size_t toRow)**

Refreshes the rows between *fromRow* and *toRow* (inclusive). *fromRow* should be less than or equal to *toRow*.

See also

RefreshRow (p. 774)

wxHVScrolledWindow::RefreshRowsColumns**void RefreshRowsColumns(size_t fromRow, size_t toRow, size_t fromColumn, size_t toColumn)**

Refreshes the region of cells between *fromRow*, *fromColumn* and *toRow*, *toColumn* (inclusive). *fromRow* and *fromColumn* should be less than or equal to *toRow* and *toColumn*, respectively.

See also

RefreshRowColumn (p. 775)

wxHVScrolledWindow::RefreshAll**void RefreshAll()**

This function completely refreshes the control, recalculating the number of items shown on screen and repainting them. It should be called when the values returned by either *OnGetRowHeight* (p. 774) or *OnGetColumnWidth* (p. 773) change for some reason and

the window must be updated to reflect this.

wxHVSrolledWindow::ScrollColumns

bool ScrollColumns(int columns)

Scroll by the specified number of columns which may be positive (to scroll right) or negative (to scroll left).

Returns `true` if the window was scrolled, `false` otherwise (for example if we're trying to scroll right but we are already showing the last column).

wxHVSrolledWindow::ScrollRows

bool ScrollRows(int rows)

Scroll by the specified number of rows which may be positive (to scroll down) or negative (to scroll up).

Returns `true` if the window was scrolled, `false` otherwise (for example if we're trying to scroll down but we are already showing the last row).

See also

LineUp (p. **Error! Bookmark not defined.**), *LineDown* (p. **Error! Bookmark not defined.**)

wxHVSrolledWindow::ScrollRowsColumns

bool ScrollRowsColumns(int rows, int columns)

Scroll by the specified number of rows and columns which may be positive (to scroll down or right) or negative (to scroll up or left).

Returns `true` if the window was scrolled, `false` otherwise (for example if we're trying to scroll down but we are already showing the last row).

See also

LineUp (p. **Error! Bookmark not defined.**), *LineDown* (p. **Error! Bookmark not defined.**)

wxHVSrolledWindow::ScrollColumnPages

bool ScrollColumnPages(int columnPages)

Scroll by the specified number of column pages, which may be positive (to scroll right) or negative (to scroll left).

wxHVSrolledWindow::ScrollPages

bool ScrollPages(int rowPages, int columnPages)

Scroll by the specified number of row pages and column pages, both of which may be positive (to scroll down or right) or negative (to scroll up or left).

See also

ScrollRowsColumns (p. 776),
PageUp (p. **Error! Bookmark not defined.**), *PageDown* (p. **Error! Bookmark not defined.**)

wxHVScrolledWindow::ScrollRowPages

bool ScrollRowPages(int *rowPages*)

Scroll by the specified number of row pages, which may be positive (to scroll down) or negative (to scroll up).

See also

PageUp (p. **Error! Bookmark not defined.**), *PageDown* (p. **Error! Bookmark not defined.**)

wxHVScrolledWindow::ScrollToColumn

bool ScrollToColumn(size_t *column*)

Scroll to the specified column. The specified column will be the first visible column on the left side afterwards.

Return `true` if we scrolled the window, `false` if nothing was done.

wxHVScrolledWindow::ScrollToRow

bool ScrollToRow(size_t *row*)

Scroll to the specified row. The specified column will be the first visible row on the top afterwards.

Return `true` if we scrolled the window, `false` if nothing was done.

wxHVScrolledWindow::ScrollToRowColumn

bool ScrollToRowColumn(size_t *row*, size_t *column*)

Scroll to the specified row and column. The cell described will be the top left visible cell afterwards.

Return `true` if we scrolled the window, `false` if nothing was done.

wxHVScrolledWindow::SetRowColumnCounts

void SetLineCount(size_t *row*, size_t *column*)

Set the number of rows and columns the window contains. The derived class must provide the heights for all rows and the widths for all columns with indices up to the respective values given here in its *OnGetRowHeight()* (p. 774) and *OnGetColumnWidth()* (p. 773) implementations. **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. 84)
wxGDIObject (p. 609)
wxObject (p. **Error! Bookmark not defined.**)

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. **Error! Bookmark not defined.**) and *wxListCtrl* (p. 864).

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. **Error! Bookmark not defined.**).

See also

Bitmap and icon overview (p. **Error! Bookmark not defined.**), *supported bitmap file formats* (p. **Error! Bookmark not defined.**), *wxDC::DrawIcon* (p. 377), *wxCursor* (p. 230)

wxIcon::wxIcon

wxIcon()

Default constructor.

wxIcon(const wxIcon& icon)

Copy constructor.

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

Creates an icon from the given data, which can be of arbitrary type.

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

Creates an icon from an array of bits.

wxIcon(int width, int height, int depth = -1)

Creates a new icon.

wxIcon(char bits)**

wxIcon(const char bits)**

Creates an icon from XPM data.

wxIcon(const wxString& name, wxBitmapType type, int desiredWidth = -1, int desiredHeight = -1)

Loads an icon from a file or resource.

wxIcon(const wxIconLocation& loc)

Loads an icon from the specified *location* (p. 786).

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.

loc

The object describing the location of the native icon, see *wxIconLocation* (p. 786).

type

May be one of the following:

`wxBITMAP_TYPE_ICO` Load a Windows icon file.

`wxBITMAP_TYPE_ICO_RESOURCE` Load a Windows icon from the resource database.

`wxBITMAP_TYPE_GIF` Load a GIF bitmap file.

`wxBITMAP_TYPE_XBM` Load an X bitmap file.

`wxBITMAP_TYPE_XPM` Load an XPM bitmap file.

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

Remarks

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

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

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

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

The sixth form constructs a new icon.

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

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

```
#include "mybitmap.xpm"

...

wxIcon *icon = new wxIcon(mybitmap);
```

A macro, `wxICON`, is available which creates an icon using an XPM on the appropriate platform, or an icon resource on Windows.

```
wxIcon icon(wxICON(mondrian));

// Equivalent to:

#if defined(__WXGTK__) || defined(__WXMOTIF__)
wxIcon icon(mondrian_xpm);
#endif

#if defined(__WXMSW__)
wxIcon icon("mondrian");
#endif
```

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

Under Windows, *type* defaults to `wxBITMAP_TYPE_ICO_RESOURCE`. Under X, *type* defaults to `wxBITMAP_TYPE_XPM`.

See also

wxIcon::CopyFromBitmap

void CopyFromBitmap(const wxBitmap& bmp)

Copies *bmp* bitmap to this icon. Under MS Windows the bitmap must have mask colour set.

wxIcon::LoadFile (p. 783)

wxPerl note: Constructors supported by wxPerl are:

- `Icon->new(width, height, depth = -1)`

- `::Icon->new(name, type, desiredWidth = -1, desiredHeight = -1)`
- `::Icon->newFromBits(bits, width, height, depth = 1)`
- `::Icon->newFromXPM(data)`

wxIcon::~wxIcon

~wxIcon()

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

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

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

wxIcon::GetDepth

int GetDepth() const

Gets the colour depth of the icon. A value of 1 indicates a monochrome icon.

wxIcon::GetHeight

int GetHeight() const

Gets the height of the icon in pixels.

wxIcon::GetWidth

int GetWidth() const

Gets the width of the icon in pixels.

See also

wxIcon::GetHeight (p. 782)

wxIcon::LoadFile

bool LoadFile(const wxString& name, wxBitmapType type)

Loads an icon from a file or resource.

Parameters

name

Either a filename or a Windows resource name. The meaning of *name* is

determined by the *type* parameter.

type

One of the following values:

wxBITMAP_TYPE_ICO Load a Windows icon file.

wxBITMAP_TYPE_ICO_RESOURCE Load a Windows icon from the resource database.

wxBITMAP_TYPE_GIF Load a GIF bitmap file.

wxBITMAP_TYPE_XBM Load an X bitmap file.

wxBITMAP_TYPE_XPM Load an XPM bitmap file.

The validity of these flags depends on the platform and wxWidgets configuration.

Return value

true if the operation succeeded, false otherwise.

See also

wxIcon::wxIcon (p. 779)

wxIcon::Ok

bool Ok() const

Returns true if icon data is present.

wxIcon::SetDepth

void SetDepth(int *depth*)

Sets the depth member (does not affect the icon data).

Parameters

depth

Icon depth.

wxIcon::SetHeight

void SetHeight(int *height*)

Sets the height member (does not affect the icon data).

Parameters

height

Icon height in pixels.

wxIcon::SetWidth

void SetWidth(int width)

Sets the width member (does not affect the icon data).

Parameters

width

Icon width in pixels.

wxIcon::operator =

wxIcon& operator =(const wxIcon& icon)

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

Parameters

icon

Icon to assign.

Return value

Returns 'this' object.

wxIcon::operator ==

bool operator ==(const wxIcon& icon)

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

Parameters

icon

Icon to compare with 'this'

Return value

Returns true if the icons were effectively equal, false otherwise.

wxIcon::operator !=

bool operator !=(const wxIcon& icon)

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

fast test).

Parameters

icon

Icon to compare with 'this'

Return value

Returns true if the icons were unequal, false otherwise.

wxIconBundle

This class contains multiple copies of an icon in different sizes, see also *wxDialog::SetIcons* (p. 420) and *wxTopLevelWindow::SetIcons* (p. **Error! Bookmark not defined.**).

Derived from

No base class

wxIconBundle::wxIconBundle

wxIconBundle()

Default constructor.

wxIconBundle(const wxString& file, long type)

Initializes the bundle with the icon(s) found in the file.

wxIconBundle(const wxIcon& icon)

Initializes the bundle with a single icon.

wxIconBundle(const wxIconBundle& ic)

Copy constructor.

wxIconBundle::~wxIconBundle

~wxIconBundle()

Destructor.

wxIconBundle::AddIcon

void AddIcon(const wxString& file, long type)

Adds all the icons contained in the file to the bundle; if the collection already contains icons with the same width and height, they are replaced by the new ones.

void AddIcon(const wxIcon& icon)

Adds the icon to the collection; if the collection already contains an icon with the same width and height, it is replaced by the new one.

wxIconBundle::GetIcon

const wxIcon& GetIcon(const wxSize& size) const

Returns the icon with the given size; if no such icon exists, returns the icon with size `wxSYS_ICON_X/wxSYS_ICON_Y`; if no such icon exists, returns the first icon in the bundle. If `size = wxSize(-1, -1)`, returns the icon with size `wxSYS_ICON_X/wxSYS_ICON_Y`.

const wxIcon& GetIcon(wxCoord size = -1) const

Same as `GetIcon(wxSize(size, size))`.

wxIconBundle::operator=

const wxIconBundle& operator=(const wxIconBundle& ic)

Assignment operator.

wxIconLocation

`wxIconLocation` is a tiny class describing the location of an (external, i.e. not embedded into the application resources) icon. For most platforms it simply contains the file name but under some others (notably Windows) the same file may contain multiple icons and so this class also stores the index of the icon inside the file.

In any case, its details should be of no interest to the application code and most of them are not even documented here (on purpose) as it is only meant to be used as an opaque class: the application may get the object of this class from somewhere and the only reasonable thing to do with it later is to create a `wxIcon` (p. 778) from it.

Derived from

None.

Include files

<wx/iconloc.h>

See also

`wxIcon` (p. 778), `wxFileType::GetIcon` (p. 550)

wxIconLocation::IsOk**bool IsOk() const**

Returns `true` if the object is valid, i.e. was properly initialized, and `false` otherwise.

wxIconizeEvent

An event being sent when the frame is iconized (minimized) or restored.

Currently only `wxMSW` and `wxGTK` generate such events.

Derived from

wxEvent (p. 487)

wxObject (p. **Error! Bookmark not defined.**)

Include files

<wx/event.h>

Event table macros

To process an iconize event, use this event handler macro to direct input to a member function that takes a `wxIconizeEvent` argument.

EVT_ICONIZE(func) Process a `wxEVT_ICONIZE` event.

See also

Event handling overview (p. **Error! Bookmark not defined.**),

wxTopLevelWindow::Iconize (p. **Error! Bookmark not defined.**),

wxTopLevelWindow::IsIconized (p. **Error! Bookmark not defined.**)

wxIconizeEvent::wxIconizeEvent

wxIconizeEvent(int id = 0, bool iconized = true)

Constructor.

wxIconizeEvent::IsIconized**bool IsIconized() const**

Returns `true` if the frame has been iconized, `false` if it has been restored.

wxIdleEvent

This class is used for idle events, which are generated when the system becomes idle.

Note that, unless you do something specifically, the idle events are not sent if the system

remains idle once it has become it, e.g. only a single idle event will be generated until something else resulting in more normal events happens and only then is the next idle event sent again. If you need to ensure a continuous stream of idle events, you can either use *RequestMore* (p. 789) method in your handler or call *wxWakeUpIdle* (p. **Error! Bookmark not defined.**) periodically (for example from timer event), but note that both of these approaches (and especially the first one) increase the system load and so should be avoided if possible.

By default, idle events are sent to all windows (and also *wxApp* (p. 36), as usual). If this is causing a significant overhead in your application, you can call *wxIdleEvent::SetMode* (p. 790) with the value *wxIDLE_PROCESS_SPECIFIED*, and set the *wxWS_EX_PROCESS_IDLE* extra window style for every window which should receive idle events.

Derived from

wxEvent (p. 487)

wxObject (p. **Error! Bookmark not defined.**)

Include files

<wx/event.h>

Event table macros

To process an idle event, use this event handler macro to direct input to a member function that takes a *wxIdleEvent* argument.

EVT_IDLE(func) Process a *wxEVT_IDLE* event.

See also

Event handling overview (p. **Error! Bookmark not defined.**), *wxUpdateUIEvent* (p. **Error! Bookmark not defined.**), *wxWindow::OnInternalIdle* (p. **Error! Bookmark not defined.**)

wxIdleEvent::wxIdleEvent

wxIdleEvent()

Constructor.

wxIdleEvent::CanSend

static bool CanSend(wxWindow* window)

Returns *true* if it is appropriate to send idle events to this window.

This function looks at the mode used (see *wxIdleEvent::SetMode* (p. 790)), and the *wxWS_EX_PROCESS_IDLE* style in *window* to determine whether idle events should be sent to this window now. By default this will always return *true* because the update mode is initially *wxIDLE_PROCESS_ALL*. You can change the mode to only send idle

events to windows with the `wxWS_EX_PROCESS_IDLE` extra window style set.

See also

wxIdleEvent::SetMode (p. 790)

wxIdleEvent::GetMode

static wxIdleMode GetMode()

Static function returning a value specifying how wxWidgets will send idle events: to all windows, or only to those which specify that they will process the events.

See *wxIdleEvent::SetMode* (p. 790).

wxIdleEvent::RequestMore

void RequestMore(bool needMore = true)

Tells wxWidgets that more processing is required. This function can be called by an OnIdle handler for a window or window event handler to indicate that wxApp::OnIdle should forward the OnIdle event once more to the application windows. If no window calls this function during OnIdle, then the application will remain in a passive event loop (not calling OnIdle) until a new event is posted to the application by the windowing system.

See also

wxIdleEvent::MoreRequested (p. 789)

wxIdleEvent::MoreRequested

bool MoreRequested() const

Returns true if the OnIdle function processing this event requested more processing time.

See also

wxIdleEvent::RequestMore (p. 789)

wxIdleEvent::SetMode

static void SetMode(wxIdleMode mode)

Static function for specifying how wxWidgets will send idle events: to all windows, or only to those which specify that they will process the events.

mode can be one of the following values. The default is `wxIDLE_PROCESS_ALL`.

```
enum wxIdleMode
{
    // Send idle events to all windows
```

```
wxIDLE_PROCESS_ALL,  
  
    // Send idle events to windows that have  
    // the wxWS_EX_PROCESS_IDLE flag specified  
wxIDLE_PROCESS_SPECIFIED  
};
```

wxImage

This class encapsulates a platform-independent image. An image can be created from data, or using *wxBitmap::ConvertToImage* (p. 89). 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. 372). Instead, a platform-specific *wxBitmap* (p. 84) object must be created from it using the *wxBitmap::wxBitmap(wxImage, int depth)* (p. 84) constructor. This bitmap can then be drawn in a device context, using *wxDC::DrawBitmap* (p. 376).

One colour value of the image may be used as a mask colour which will lead to the automatic creation of a *wxMask* (p. 920) object associated to the bitmap object.

Alpha channel support

Starting from *wxWidgets* 2.5.0 *wxImage* supports alpha channel data, that is in addition to a byte for the red, green and blue colour components for each pixel it also stores a byte representing the pixel opacity. An alpha value of 0 corresponds to a transparent pixel (null opacity) while a value of 255 means that the pixel is 100% opaque.

Unlike RGB data, not all images have an alpha channel and before using *GetAlpha* (p. 799) you should check if this image contains an alpha channel with *HasAlpha* (p. 802). Note that currently only images loaded from PNG files with transparency information will have an alpha channel but alpha support will be added to the other formats as well (as well as support for saving images with alpha channel which also isn't implemented).

Available image handlers

The following image handlers are available. **wxBMPHandler** is always installed by default. To use other image formats, install the appropriate handler with *wxImage::AddHandler* (p. 795) or *wxInitAllImageHandlers* (p. **Error! Bookmark not defined.**).

<i>wxBMPHandler</i>	For loading and saving, always installed.
<i>wxPNGHandler</i>	For loading (including alpha support) and saving.
<i>wxJPEGHandler</i>	For loading and saving.
<i>wxGIFHandler</i>	Only for loading, due to legal issues.
<i>wxPCXHandler</i>	For loading and saving (see below).

<code>wxPNMHandler</code>	For loading and saving (see below).
<code>wxTIFFHandler</code>	For loading and saving.
<code>wxIFFHandler</code>	For loading only.
<code>wxXPMHandler</code>	For loading and saving.
<code>wxICOHandler</code>	For loading and saving.
<code>wxCURHandler</code>	For loading and saving.
<code>wxANIHandler</code>	For loading only.

When saving in PCX format, **wxPCXHandler** will count the number of different colours in the image; if there are 256 or less colours, it will save as 8 bit, else it will save as 24 bit.

Loading PNMs only works for ASCII or raw RGB images. When saving in PNM format, **wxPNMHandler** will always save as raw RGB.

Derived from

`wxObject` (p. **Error! Bookmark not defined.**)

Include files

`<wx/image.h>`

See also

`wxBitmap` (p. 84), `wxInitAllImageHandlers` (p. **Error! Bookmark not defined.**)

wxImage::wxImage

wxImage()

Default constructor.

wxImage(const wxImage& image)

Copy constructor.

wxImage(const wxBitmap& bitmap)

(Deprecated form, use `wxBitmap::ConvertToImage` (p. 89) instead.) Constructs an image from a platform-dependent bitmap. This preserves mask information so that bitmaps and images can be converted back and forth without loss in that respect.

wxImage(int width, int height, bool clear=true)

Creates an image with the given width and height. If *clear* is true, the new image will be initialized to black. Otherwise, the image data will be uninitialized.

wxImage(int width, int height, unsigned char* data, bool static_data = false)

Creates an image from given data with the given width and height. If *static_data* is true, then *wxImage* will not delete the actual image data in its destructor, otherwise it will free it by calling *free()*.

wxImage(const wxString& name, long type = wxBITMAP_TYPE_ANY, int index = -1)

wxImage(const wxString& name, const wxString& mimetype, int index = -1)

Loads an image from a file.

wxImage(wxInputStream& stream, long type = wxBITMAP_TYPE_ANY, int index = -1)

wxImage(wxInputStream& stream, const wxString& mimetype, int index = -1)

Loads an image from an input stream.

wxImage(const char xpmData)**

Creates an image from XPM data.

Parameters

width

Specifies the width of the image.

height

Specifies the height of the image.

name

Name of the file from which to load the image.

stream

Opened input stream from which to load the image. Currently, the stream must support seeking.

type

May be one of the following:

<code>wxBITMAP_TYPE_BMP</code>	Load a Windows bitmap file.
<code>wxBITMAP_TYPE_GIF</code>	Load a GIF bitmap file.
<code>wxBITMAP_TYPE_JPEG</code>	Load a JPEG bitmap file.
<code>wxBITMAP_TYPE_PNG</code>	Load a PNG bitmap file.
<code>wxBITMAP_TYPE_PCX</code>	Load a PCX bitmap file.

<code>wxBITMAP_TYPE_PNM</code>	Load a PNM bitmap file.
<code>wxBITMAP_TYPE_TIF</code>	Load a TIFF bitmap file.
<code>wxBITMAP_TYPE_XPM</code>	Load a XPM bitmap file.
<code>wxBITMAP_TYPE_ICO</code>	Load a Windows icon file (ICO).
<code>wxBITMAP_TYPE_CUR</code>	Load a Windows cursor file (CUR).
<code>wxBITMAP_TYPE_ANI</code>	Load a Windows animated cursor file (ANI).
<code>wxBITMAP_TYPE_ANY</code>	Will try to autodetect the format.

mimetype

MIME type string (for example 'image/jpeg')

index

Index of the image to load in the case that the image file contains multiple images. This is only used by GIF, ICO and TIFF handlers. The default value (-1) means "choose the default image" and is interpreted as the first image (index=0) by the GIF and TIFF handler and as the largest and most colourful one by the ICO handler.

xpmData

A pointer to XPM image data.

Remarks

Depending on how wxWidgets has been configured, not all formats may be available.

Note: any handler other than BMP must be previously initialized with `wxImage::AddHandler` (p. 795) or `wxInitAllImageHandlers` (p. **Error! Bookmark not defined.**).

Note: you can use `GetOptionInt` (p. 803) to get the hotspot for loaded cursor file: int
hotspot_x = image.GetOptionInt(wxIMAGE_OPTION_CUR_HOTSPOT_X);
int hotspot_y =
image.GetOptionInt(wxIMAGE_OPTION_CUR_HOTSPOT_Y);

See also

`wxImage::LoadFile` (p. 804)

wxPython note: Constructors supported by wxPython are:

<code>wxImage(name, flag)</code>	Loads an image from a file
<code>wxNullImage()</code>	Create a null image (has no size or image data)
<code>wxEmptyImage(width, height)</code>	Creates an empty image of the given size

wxImageFromMime(name, mimetype) Creates an image from the given file of the given mimetype

wxImageFromBitmap(bitmap) Creates an image from a platform-dependent bitmap

wxPerl note: Constructors supported by wxPerl are:

- `Image->new(bitmap)`
- `Image->new(icon)`
- `Image->new(width, height)`
- `Image->new(width, height, data)`
- `Image->new(file, type, index)`
- `Image->new(file, mimetype, index)`
- `Image->new(stream, type, index)`
- `Image->new(stream, mimetype, index)`

wxImage::~wxImage

~wxImage()

Destructor.

wxImage::AddHandler

static void AddHandler(wxImageHandler* handler)

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

handler

A new image format handler object. There is usually only one instance of a given handler class in an application session.

See also

wxImageHandler (p. 814)

bool CanRead(const wxString& filename)

returns true if the current image handlers can read this file

wxPython note: In wxPython this static method is named `wxImage_AddHandler`.

wxImage::CleanUpHandlers

static void CleanUpHandlers()

Deletes all image handlers.

This function is called by wxWidgets on exit.

wxImage::ComputeHistogram**unsigned long ComputeHistogram(wxImageHistogram& *histogram*) const**

Computes the histogram of the image. *histogram* is a reference to wxImageHistogram object. wxImageHistogram is a specialization of *wxHashMap* (p. 684) "template" and is defined as follows:

```
class WXDLLEXPORT wxImageHistogramEntry
{
public:
    wxImageHistogramEntry() : index(0), value(0) {}
    unsigned long index;
    unsigned long value;
};

WX_DECLARE_EXPORTED_HASH_MAP(unsigned long, wxImageHistogramEntry,
                             wxIntegerHash, wxIntegerEqual,
                             wxImageHistogram);
```

Return value

Returns number of colours in the histogram.

wxImage::ConvertAlphaToMask**bool ConvertAlphaToMask(unsigned char *threshold* = 128)**

If the image has alpha channel, this method converts it to mask. All pixels with alpha value less than *threshold* are replaced with mask colour and the alpha channel is removed. Mask colour is chosen automatically using *FindFirstUnusedColour* (p. 797).

If the image image doesn't have alpha channel, ConvertAlphaToMask does nothing.

Return value

false if FindFirstUnusedColour returns false, true otherwise.

wxImage::ConvertToBitmap**wxBitmap ConvertToBitmap() const**

Deprecated, use equivalent *wxBitmap* constructor (p. 84)(which takes wxImage and depth as its arguments) instead.

wxImage::ConvertToGreyscale**wxImage ConvertToGreyscale(double *lr* = 0.299, double *lg* = 0.587, double *lb* =**

0.114) const

Returns a greyscale version of the image. The returned image uses the luminance component of the original to calculate the greyscale. Defaults to using ITU-T BT.601 when converting to YUV, where every pixel equals $(R * Ir) + (G * Ig) + (B * Ib)$.

wxImage::ConvertToMono

wxImage ConvertToMono(unsigned char *r*, unsigned char *g*, unsigned char *b*) const

Returns monochromatic version of the image. The returned image has white colour where the original has (r,g,b) colour and black colour everywhere else.

wxImage::Copy

wxImage Copy() const

Returns an identical copy of the image.

wxImage::Create

bool Create(int *width*, int *height*, bool *clear=true*)

Creates a fresh image. If *clear* is true, the new image will be initialized to black. Otherwise, the image data will be uninitialized.

Parameters

width

The width of the image in pixels.

height

The height of the image in pixels.

Return value

true if the call succeeded, false otherwise.

wxImage::Destroy

void Destroy()

Destroys the image data.

wxImage::FindFirstUnusedColour

bool FindFirstUnusedColour(unsigned char * *r*, unsigned char * *g*, unsigned char * *b*, unsigned char *startR* = 1, unsigned char *startG* = 0, unsigned char *startB* = 0)

Parameters*r,g,b*

Pointers to variables to save the colour.

startR,startG,startB

Initial values of the colour. Returned colour will have RGB values equal to or greater than these.

Finds the first colour that is never used in the image. The search begins at given initial colour and continues by increasing R, G and B components (in this order) by 1 until an unused colour is found or the colour space exhausted.

Return value

Returns false if there is no unused colour left, true on success.

Notes

Note that this method involves computing the histogram, which is computationally intensive operation.

wxImage::FindHandler**static wxImageHandler* FindHandler(const wxString& name)**

Finds the handler with the given name.

static wxImageHandler* FindHandler(const wxString& extension, long imageType)

Finds the handler associated with the given extension and type.

static wxImageHandler* FindHandler(long imageType)

Finds the handler associated with the given image type.

static wxImageHandler* FindHandlerMime(const wxString& mimetype)

Finds the handler associated with the given MIME type.

name

The handler name.

extension

The file extension, such as "bmp".

imageType

The image type, such as wxBITMAP_TYPE_BMP.

mimetype

MIME type.

Return value

A pointer to the handler if found, NULL otherwise.

See also

wxImageHandler (p. 814)

wxImage::GetImageExtWildcard**static wxString GetImageExtWildcard()**

Iterates all registered *wxImageHandler* objects, and returns a string containing file extension masks suitable for passing to file open/save dialog boxes.

Return value

The format of the returned string is `"(*.ext1;*.ext2)|*.ext1;*.ext2"`.

It is usually a good idea to prepend a description before passing the result to the dialog.

Example:

```
wxFileDialog FileDlg( this, "Choose Image", ::wxGetCwd(), "",
_("Image Files ") + wxImage::GetImageExtWildcard(), wxOPEN );
```

See also

wxImageHandler (p. 814)

wxImage::GetAlpha**unsigned char GetAlpha(int x, int y) const**

Returns the alpha value for the given pixel. This function may only be called for the images with alpha channel, use *HasAlpha* (p. 802) to check for this.

The returned value is the *opacity* of the image, i.e. the value of 0 corresponds to the transparent pixels while the value of 255 -- to the opaque ones.

unsigned char * GetAlpha() const

Returns pointer to the array storing the alpha values for this image. This pointer is NULL for the images without the alpha channel. If the image does have it, this pointer may be used to directly manipulate the alpha values which are stored as the *RGB* (p. 799) ones.

wxImage::GetBlue**unsigned char GetBlue(int x, int y) const**

Returns the blue intensity at the given coordinate.

wxImage::GetData**unsigned char* GetData() const**

Returns the image data as an array. This is most often used when doing direct image manipulation. The return value points to an array of characters in RGBRGBRGB... format in the top-to-bottom, left-to-right order, that is the first RGB triplet corresponds to the pixel first pixel of the first row, the second one --- to the second pixel of the first row and so on until the end of the first row, with second row following after it and so on.

You should not delete the returned pointer nor pass it to *wxImage::SetData* (p. 811).

wxImage::GetGreen**unsigned char GetGreen(int x, int y) const**

Returns the green intensity at the given coordinate.

wxImage::GetImageCount**static int GetImageCount(const wxString& filename, long type = wxBITMAP_TYPE_ANY)****static int GetImageCount(wxInputStream& stream, long type = wxBITMAP_TYPE_ANY)**

If the image file contains more than one image and the image handler is capable of retrieving these individually, this function will return the number of available images.

name

Name of the file to query.

stream

Opened input stream with image data. Currently, the stream must support seeking.

type

May be one of the following:

wxBITMAP_TYPE_BMP	Load a Windows bitmap file.
wxBITMAP_TYPE_GIF	Load a GIF bitmap file.
wxBITMAP_TYPE_JPEG	Load a JPEG bitmap file.
wxBITMAP_TYPE_PNG	Load a PNG bitmap file.
wxBITMAP_TYPE_PCX	Load a PCX bitmap file.
wxBITMAP_TYPE_PNM	Load a PNM bitmap file.
wxBITMAP_TYPE_TIF	Load a TIFF bitmap file.

<code>wxBITMAP_TYPE_XPM</code>	Load a XPM bitmap file.
<code>wxBITMAP_TYPE_ICO</code>	Load a Windows icon file (ICO).
<code>wxBITMAP_TYPE_CUR</code>	Load a Windows cursor file (CUR).
<code>wxBITMAP_TYPE_ANI</code>	Load a Windows animated cursor file (ANI).
<code>wxBITMAP_TYPE_ANY</code>	Will try to autodetect the format.

Return value

Number of available images. For most image handlers, this is 1 (exceptions are TIFF and ICO formats).

`wxImage::GetHandlers`**`static wxList& GetHandlers()`**

Returns the static list of image format handlers.

See also

wxImageHandler (p. 814)

`wxImage::GetHeight`**`int GetHeight() const`**

Gets the height of the image in pixels.

`wxImage::GetMaskBlue`**`unsigned char GetMaskBlue() const`**

Gets the blue value of the mask colour.

`wxImage::GetMaskGreen`**`unsigned char GetMaskGreen() const`**

Gets the green value of the mask colour.

`wxImage::GetMaskRed`**`unsigned char GetMaskRed() const`**

Gets the red value of the mask colour.

`wxImage::GetOrFindMaskColour`

**bool GetOrFindMaskColour(unsigned char *r, unsigned char *g, unsigned char *b)
const**

Get the current mask colour or find a suitable unused colour that could be used as a mask colour. Returns `true` if the image currently has a mask.

wxImage::GetPalette

const wxPalette& GetPalette() const

Returns the palette associated with the image. Currently the palette is only used when converting to `wxBitmap` under Windows.

Eventually `wxImage` handlers will set the palette if one exists in the image file.

wxImage::GetRed

unsigned char GetRed(int x, int y) const

Returns the red intensity at the given coordinate.

wxImage::GetSubImage

wxImage GetSubImage(const wxRect& rect) const

Returns a sub image of the current one as long as the rect belongs entirely to the image.

wxImage::GetWidth

int GetWidth() const

Gets the width of the image in pixels.

See also

wxImage::GetHeight (p. 801)

HSVValue::HSVValue

HSVValue(double h = 0.0, double s = 0.0, double v = 0.0)

Constructor for `HSVValue`, an object that contains values for hue, saturation and value which represent the value of a color. It is used by *wxImage::HSVtoRGB* (p. 802) and *wxImage::RGBtoHSV* (p. 807), which converts between HSV color space and RGB color space.

wxPython note: use `wxImage_HSVValue` in wxPython

wxImage::HSVtoRGB

wxImage::RGBValue HSVtoRGB(const HSVValue & hsv)

Converts a color in HSV color space to RGB color space.

wxImage::HasAlpha

bool HasAlpha() const

Returns true if this image has alpha channel, false otherwise.

See also

GetAlpha (p. 799), *SetAlpha* (p. 811)

wxImage::HasMask

bool HasMask() const

Returns true if there is a mask active, false otherwise.

wxImage::GetOption

wxString GetOption(const wxString& name) const

Gets a user-defined option. The function is case-insensitive to *name*.

For example, when saving as a JPEG file, the option **quality** is used, which is a number between 0 and 100 (0 is terrible, 100 is very good).

See also

wxImage::SetOption (p. 812), *wxImage::GetOptionInt* (p. 803), *wxImage::HasOption* (p. 803)

wxImage::GetOptionInt

int GetOptionInt(const wxString& name) const

Gets a user-defined option as an integer. The function is case-insensitive to *name*.

If the given option is not present, the function returns 0. Use *wxImage::HasOption* (p. 803) to check if 0 is a possibly valid value for the option.

Options for `wxPNGHandler`: `wxIMAGE_OPTION_PNG_FORMAT` Format for saving a PNG file.

`wxIMAGE_OPTION_PNG_BITDEPTH` Bit depth for every channel (R/G/B/A).

Supported values for `wxIMAGE_OPTION_PNG_FORMAT`: `wxPNG_TYPE_COLOUR` Stores RGB image.

`wxPNG_TYPE_GREY` Stores grey image, converts from RGB.

`wxPNG_TYPE_GREY_RED` Stores grey image, uses red value as grey.

See also

wxImage::SetOption (p. 812), *wxImage::GetOption* (p. 803)

wxImage::HasOption

bool HasOption(const wxString& name) const

Returns true if the given option is present. The function is case-insensitive to *name*.

See also

wxImage::SetOption (p. 812), *wxImage::GetOption* (p. 803), *wxImage::GetOptionInt* (p. 803)

wxImage::InitAlpha

void InitAlpha()

Initializes the image alpha channel data. It is an error to call it if the image already has alpha data. If it doesn't, alpha data will be by default initialized to all pixels being fully opaque. But if the image has a mask colour, all mask pixels will be completely transparent.

wxImage::InitStandardHandlers

static void InitStandardHandlers()

Internal use only. Adds standard image format handlers. It only install BMP for the time being, which is used by wxBitmap.

This function is called by wxWidgets on startup, and shouldn't be called by the user.

See also

wxImageHandler (p. 814), *wxInitAllImageHandlers* (p. **Error! Bookmark not defined.**)

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. 814)

wxImage::IsTransparent**bool IsTransparent(int x, int y, unsigned char threshold = 128) const**

Returns `true` if the given pixel is transparent, i.e. either has the mask colour if this image has a mask or if this image has alpha channel and alpha value of this pixel is strictly less than *threshold*.

wxImage::LoadFile**bool LoadFile(const wxString& name, long type = wxBITMAP_TYPE_ANY, int index = -1)****bool LoadFile(const wxString& name, const wxString& mimetype, int index = -1)**

Loads an image from a file. If no handler type is provided, the library will try to autodetect the format.

bool LoadFile(wxInputStream& stream, long type, int index = -1)**bool LoadFile(wxInputStream& stream, const wxString& mimetype, int index = -1)**

Loads an image from an input stream.

Parameters*name*

Name of the file from which to load the image.

stream

Opened input stream from which to load the image. Currently, the stream must support seeking.

type

One of the following values:

wxBITMAP_TYPE_BMP	Load a Windows image file.
wxBITMAP_TYPE_GIF	Load a GIF image file.
wxBITMAP_TYPE_JPEG	Load a JPEG image file.
wxBITMAP_TYPE_PCX	Load a PCX image file.
wxBITMAP_TYPE_PNG	Load a PNG image file.
wxBITMAP_TYPE_PNM	Load a PNM image file.
wxBITMAP_TYPE_TIF	Load a TIFF image file.
wxBITMAP_TYPE_XPM	Load a XPM image file.

wxBITMAP_TYPE_ICO	Load a Windows icon file (ICO).
wxBITMAP_TYPE_CUR	Load a Windows cursor file (CUR).
wxBITMAP_TYPE_ANI	Load a Windows animated cursor file (ANI).
wxBITMAP_TYPE_ANY	Will try to autodetect the format.

mimetype

MIME type string (for example 'image/jpeg')

index

Index of the image to load in the case that the image file contains multiple images. This is only used by GIF, ICO and TIFF handlers. The default value (-1) means "choose the default image" and is interpreted as the first image (index=0) by the GIF and TIFF handler and as the largest and most colourful one by the ICO handler.

Remarks

Depending on how wxWidgets has been configured, not all formats may be available.

```
Note: you can use GetOptionInt (p. 803) to get the hotspot for loaded cursor file:      int
hotspot_x = image.GetOptionInt(wxIMAGE_OPTION_CUR_HOTSPOT_X);
int hotspot_y =
image.GetOptionInt(wxIMAGE_OPTION_CUR_HOTSPOT_Y);
```

Return value

true if the operation succeeded, false otherwise. If the optional index parameter is out of range, false is returned and a call to `wxLogError()` takes place.

See also

wxImage::SaveFile (p. 808)

wxPython note: In place of a single overloaded method name, wxPython implements the following methods:

LoadFile(filename, type) Loads an image of the given type from a file
LoadMimeFile(filename, mimetype) Loads an image of the given
mimetype from a file

wxPerl note: Methods supported by wxPerl are:

- >LoadFile(name, type)
- >LoadFile(name, mimetype)

wxImage::Ok**bool Ok() const**

Returns true if image data is present.

RGBValue::RGBValue**RGBValue(unsigned char *r* = 0, unsigned char *g* = 0, unsigned char *b* = 0)**

Constructor for RGBValue, an object that contains values for red, green and blue which represent the value of a color. It is used by *wxImage::HSVtoRGB* (p. 802) and *wxImage::RGBtoHSV* (p. 807), which converts between HSV color space and RGB color space.

wxPython note: use *wxImage_RGBValue* in wxPython

wxImage::RGBtoHSV**wxImage::HSVValue RGBtoHSV(const RGBValue& *rgb*)**

Converts a color in RGB color space to HSV color space.

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. 814)

wxImage::Mirror**wxImage Mirror(bool *horizontally* = true) const**

Returns a mirrored copy of the image. The parameter *horizontally* indicates the orientation.

wxImage::Replace**void Replace(unsigned char *r1*, unsigned char *g1*, unsigned char *b1*, unsigned char *r2*, unsigned char *g2*, unsigned char *b2*)**

Replaces the colour specified by *r1,g1,b1* by the colour *r2,g2,b2*.

wxImage::Rescale

wxImage & Rescale(int *width*, int *height*)

Changes the size of the image in-place by scaling it: after a call to this function, the image will have the given width and height.

Returns the (modified) image itself.

See also

Scale (p. 810)

wxImage::Resize

wxImage & Resize(const wxSize& *size*, const wxPoint& *pos*, int *red* = -1, int *green* = -1, int *blue* = -1)

Changes the size of the image in-place without scaling it by adding either a border with the given colour or cropping as necessary. The image is pasted into a new image with the given *size* and background colour at the position *pos* relative to the upper left of the new image. If *red* = *green* = *blue* = -1 then use either the current mask colour if set or find, use, and set a suitable mask colour for any newly exposed areas.

Returns the (modified) image itself.

See also

Size (p. 811)

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::RotateHue

void RotateHue(double *angle*)

Rotates the hue of each pixel in the image by *angle*, which is a double in the range of -1.0 to +1.0, where -1.0 corresponds to -360 degrees and +1.0 corresponds to +360 degrees.

wxImage::Rotate90**wxImage Rotate90**(*bool clockwise = true*) **const**

Returns a copy of the image rotated 90 degrees in the direction indicated by *clockwise*.

wxImage::SaveFile**bool SaveFile**(**const wxString& name**, **int type**) **const****bool SaveFile**(**const wxString& name**, **const wxString& mimetype**) **const**

Saves an image in the named file.

bool SaveFile(**const wxString& name**) **const**

Saves an image in the named file. File type is determined from the extension of the file name. Note that this function may fail if the extension is not recognized! You can use one of the forms above to save images to files with non-standard extensions.

bool SaveFile(**wxOutputStream& stream**, **int type**) **const****bool SaveFile**(**wxOutputStream& stream**, **const wxString& mimetype**) **const**

Saves an image in the given stream.

Parameters*name*

Name of the file to save the image to.

stream

Opened output stream to save the image to.

type

Currently these types can be used:

wxBITMAP_TYPE_BMP Save a BMP image file.

wxBITMAP_TYPE_JPEG Save a JPEG image file.

wxBITMAP_TYPE_PNG Save a PNG image file.

wxBITMAP_TYPE_PCX Save a PCX image file (tries to save as 8-bit if possible, falls back to 24-bit otherwise).

wxBITMAP_TYPE_PNM Save a PNM image file (as raw RGB always).

wxBITMAP_TYPE_TIFF Save a TIFF image file.

wxBITMAP_TYPE_XPM Save a XPM image file.

wxBITMAP_TYPE_ICO	Save a Windows icon file (ICO) (the size may be up to 255 wide by 127 high. A single image is saved in 8 colors at the size supplied).
wxBITMAP_TYPE_CUR	Save a Windows cursor file (CUR).

mimetype

MIME type.

Return value

true if the operation succeeded, false otherwise.

Remarks

Depending on how wxWidgets has been configured, not all formats may be available.

Note: you can use *GetOptionInt* (p. 803) to set the hotspot before saving an image into a cursor file (default hotspot is in the centre of the image):

```
image.SetOption(wxIMAGE_OPTION_CUR_HOTSPOT_X, hotspotX);  
image.SetOption(wxIMAGE_OPTION_CUR_HOTSPOT_Y, hotspotY);
```

See also

wxImage::LoadFile (p. 804)

wxPython note: In place of a single overloaded method name, wxPython implements the following methods:

SaveFile(filename, type) Saves the image using the given type to the named file

SaveMimeFile(filename, mimetype) Saves the image using the given mimetype to the named file

wxPerl note: Methods supported by wxPerl are:

- >SaveFile(name, type)
- >SaveFile(name, mimetype)

wxImage::Scale

wxImage Scale(int width, int height) const

Returns a scaled version of the image. This is also useful for scaling bitmaps in general as the only other way to scale bitmaps is to blit a wxMemoryDC into another wxMemoryDC.

It may be mentioned that the GTK port uses this function internally to scale bitmaps when using mapping modes in wxDC.

Example:

```
// get the bitmap from somewhere
wxBitmap bmp = ...;

// rescale it to have size of 32*32
if ( bmp.GetWidth() != 32 || bmp.GetHeight() != 32 )
{
    wxImage image = bmp.ConvertToImage();
    bmp = wxBitmap(image.Scale(32, 32));

    // another possibility:
    image.Rescale(32, 32);
    bmp = image;
}
```

See also

Rescale (p. 807)

wxImage::Size

wxImage Size(const wxSize& size, const wxPoint& pos, int red = -1, int green = -1, int blue = -1) const

Returns a resized version of this image without scaling it by adding either a border with the given colour or cropping as necessary. The image is pasted into a new image with the given *size* and background colour at the position *pos* relative to the upper left of the new image. If *red* = *green* = *blue* = -1 then use either the current mask colour if set or find, use, and set a suitable mask colour for any newly exposed areas.

See also

Resize (p. 808)

wxImage::SetAlpha

void SetAlpha(unsigned char *alpha = NULL, bool static_data = false)

This function is similar to *SetData* (p. 811) and has similar restrictions. The pointer passed to it may however be `NULL` in which case the function will allocate the alpha array internally -- this is useful to add alpha channel data to an image which doesn't have any. If the pointer is not `NULL`, it must have one byte for each image pixel and be allocated with `malloc()`. `wxImage` takes ownership of the pointer and will free it unless *static_data* parameter is set to `true` -- in this case the caller should do it.

void SetAlpha(int x, int y, unsigned char alpha)

Sets the alpha value for the given pixel. This function should only be called if the image has alpha channel data, use *HasAlpha* (p. 802) to check for this.

wxImage::SetData

void SetData(unsigned char*data)

Sets the image data without performing checks. The data given must have the size (width*height*3) or results will be unexpected. Don't use this method if you aren't sure you know what you are doing.

The data must have been allocated with `malloc()`, **NOT** with `operator new`.

After this call the pointer to the data is owned by the `wxImage` object, that will be responsible for deleting it. Do not pass to this function a pointer obtained through `wxImage::GetData` (p. 799).

wxImage::SetMask**void SetMask(bool hasMask = true)**

Specifies whether there is a mask or not. The area of the mask is determined by the current mask colour.

wxImage::SetMaskColour**void SetMaskColour(unsigned char red, unsigned char green, unsigned char blue)**

Sets the mask colour for this image (and tells the image to use the mask).

wxImage::SetMaskFromImage**bool SetMaskFromImage(const wxImage& mask, unsigned char mr, unsigned char mg, unsigned char mb)****Parameters**

mask

The mask image to extract mask shape from. Must have same dimensions as the image.

mr,mg,mb

RGB value of pixels in *mask* that will be used to create the mask.

Sets image's mask so that the pixels that have RGB value of *mr,mg,mb* in *mask* will be masked in the image. This is done by first finding an unused colour in the image, setting this colour as the mask colour and then using this colour to draw all pixels in the image who corresponding pixel in *mask* has given RGB value.

Return value

Returns false if *mask* does not have same dimensions as the image or if there is no unused colour left. Returns true if the mask was successfully applied.

Notes

Note that this method involves computing the histogram, which is computationally intensive operation.

wxImage::SetOption

void SetOption(const wxString& name, const wxString& value)

void SetOption(const wxString& name, int value)

Sets a user-defined option. The function is case-insensitive to *name*.

For example, when saving as a JPEG file, the option **quality** is used, which is a number between 0 and 100 (0 is terrible, 100 is very good).

See also

wxImage::GetOption (p. 803), *wxImage::GetOptionInt* (p. 803), *wxImage::HasOption* (p. 803)

wxImage::SetPalette

void SetPalette(const wxPalette& palette)

Associates a palette with the image. The palette may be used when converting *wxImage* to *wxBitmap* (MSW only at present) or in file save operations (none as yet).

wxImage::SetRGB

void SetRGB(int x, int y, unsigned char red, unsigned char green, unsigned char blue)

Sets the pixel at the given coordinate. This routine performs bounds-checks for the coordinate so it can be considered a safe way to manipulate the data, but in some cases this might be too slow so that the data will have to be set directly. In that case you will have to get access to the image data using the *GetData* (p. 799) method.

wxImage::SetRGB

void SetRGB(wxRect & rect, unsigned char red, unsigned char green, unsigned char blue)

Sets the colour of the pixels within the given rectangle. This routine performs bounds-checks for the coordinate so it can be considered a safe way to manipulate the data.

wxImage::operator =

wxImage& operator =(const wxImage& image)

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

Parameters*image*

Image to assign.

Return value

Returns 'this' object.

wxImage::operator ==**bool operator ==(const wxImage& *image*) const**

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

Parameters*image*

Image to compare with 'this'

Return value

Returns true if the images were effectively equal, false otherwise.

wxImage::operator !=**bool operator !=(const wxImage& *image*) const**

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

Parameters*image*

Image to compare with 'this'

Return value

Returns true if the images were unequal, false otherwise.

wxImageHandler

This is the base class for implementing image file loading/saving, and image creation from data. It is used within wxImage and is not normally seen by the application.

If you wish to extend the capabilities of wxImage, derive a class from wxImageHandler and add the handler using *wxImage::AddHandler* (p. 795) in your application initialisation.

Note (Legal Issue)

This software is based in part on the work of the Independent JPEG Group.

(Applies when wxWidgets is linked with JPEG support. wxJPEGHandler uses libjpeg created by IJG.)

Derived from

wxObject (p. **Error! Bookmark not defined.**)

Include files

<wx/image.h>

See also

wxImage (p. 790), *wxInitAllImageHandlers* (p. **Error! Bookmark not defined.**)

wxImageHandler::wxImageHandler

wxImageHandler()

Default constructor. In your own default constructor, initialise the members `m_name`, `m_extension` and `m_type`.

wxImageHandler::~~wxImageHandler

~wxImageHandler()

Destroys the `wxImageHandler` object.

wxImageHandler::GetName

wxString GetName() const

Gets the name of this handler.

wxImageHandler::GetExtension

wxString GetExtension() const

Gets the file extension associated with this handler.

wxImageHandler::GetImageCount

int GetImageCount(wxInputStream& stream)

If the image file contains more than one image and the image handler is capable of retrieving these individually, this function will return the number of available images.

stream

Opened input stream for reading image data. Currently, the stream must support seeking.

Return value

Number of available images. For most image handlers, this is 1 (exceptions are TIFF and ICO formats).

wxImageHandler::GetType

long GetType() const

Gets the image type associated with this handler.

wxImageHandler::GetMimeType

wxString GetMimeType() const

Gets the MIME type associated with this handler.

wxImageHandler::LoadFile

bool LoadFile(wxImage* image, wxInputStream& stream, bool verbose=true, int index=0)

Loads a image from a stream, putting the resulting data into *image*. If the image file contains more than one image and the image handler is capable of retrieving these individually, *index* indicates which image to read from the stream.

Parameters

image

The image object which is to be affected by this operation.

stream

Opened input stream for reading image data.

verbose

If set to true, errors reported by the image handler will produce wxLogMessages.

index

The index of the image in the file (starting from zero).

Return value

true if the operation succeeded, false otherwise.

See also

wxImage::LoadFile (p. 804), *wxImage::SaveFile* (p. 808), *wxImageHandler::SaveFile* (p. 817)

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. 804), *wxImage::SaveFile* (p. 808), *wxImageHandler::LoadFile* (p. 816)

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. **Error! Bookmark not defined.**) and `wxListCtrl` (p. 864) classes.

Derived from

`wxObject` (p. **Error! Bookmark not defined.**)

Include files

<wx/imaglist.h>

See also

`wxTreeCtrl` (p. **Error! Bookmark not defined.**), `wxListCtrl` (p. 864)

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. 820)

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. 818) for details.

wxImageList::Draw

bool Draw(int index, wxDC& dc, int x, int y, int flags = wxIMAGELIST_DRAW_NORMAL, const bool solidBackground = false)

Draws a specified image onto a device context.

Parameters

index

Image index, starting from zero.

dc

Device context to draw on.

x

X position on the device context.

y

Y position on the device context.

flags

How to draw the image. A bitlist of a selection of the following:

wxIMAGELIST_DRAW_NORMAL Draw the image normally.

wxIMAGELIST_DRAW_TRANSPARENT Draw the image with transparency.

wxIMAGELIST_DRAW_SELECTED Draw the image in selected state.

wxIMAGELIST_DRAW_FOCUSED Draw the image in a focused state.

solidBackground

For optimisation - drawing can be faster if the function is told that the background is solid.

wxImageList::GetBitmap

wxBitmap GetBitmap(int *index*) const

Returns the bitmap corresponding to the given index.

wxImageList::GetIcon

wxIcon GetIcon(int *index*) const

Returns the icon corresponding to the given index.

wxImageList::GetImageCount

int GetImageCount() const

Returns the number of images in the list.

wxImageList::GetSize

bool GetSize(int *index*, int& *width*, int &*height*) const

Retrieves the size of the images in the list. Currently, the *index* parameter is ignored as all images in the list have the same size.

Parameters

index

currently unused, should be 0

width

receives the width of the images in the list

height

receives the height of the images in the list

Return value

true if the function succeeded, false if it failed (for example, if the image list was not yet initialized).

wxImageList::Remove

bool Remove(int *index*)

Removes the image at the given position.

wxImageList::RemoveAll

bool RemoveAll()

Removes all the images in the list.

wxImageList::Replace

bool Replace(int *index*, const wxBitmap& *bitmap*, const wxBitmap& *mask* = wxNullBitmap)

Replaces the existing image with the new image.

Windows only.

bool Replace(int *index*, const wxIcon& *icon*)

Replaces the existing image with the new image.

Parameters

bitmap

Bitmap representing the opaque areas of the image.

mask

Monochrome mask bitmap, representing the transparent areas of the image.

icon

Icon to use as the image.

Return value

true if the replacement was successful, false otherwise.

Remarks

The original bitmap or icon is not affected by the **Replace** operation, and can be deleted afterwards.

wxPython note: The second form is called `ReplaceIcon` in wxPython.

wxIndividualLayoutConstraint

Objects of this class are stored in the `wxLayoutConstraint` class as one of eight possible constraints that a window can be involved in.

Constraints are initially set to have the relationship `wxUnconstrained`, which means that their values should be calculated by looking at known constraints.

Derived from

wxObject (p. **Error! Bookmark not defined.**)

Include files

<wx/layout.h>

See also

Overview and examples (p. **Error! Bookmark not defined.**), *wxLayoutConstraints* (p. 849), *wxWindow::SetConstraints* (p. **Error! Bookmark not defined.**).

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. The default handler calls `wxWindow::TransferDataToWindow` (p. **Error! Bookmark not defined.**).

Derived from

`wxEvent` (p. 487)

`wxObject` (p. **Error! Bookmark not defined.**)

Include files

<wx/event.h>

Event table macros

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

EVT_INIT_DIALOG(func) Process a `wxEVT_INIT_DIALOG` event.

See also

Event handling overview (p. **Error! Bookmark not defined.**)

wxInitDialogEvent::wxInitDialogEvent

wxInitDialogEvent(int id = 0)

Constructor.

wxInputStream

`wxInputStream` is an abstract base class which may not be used directly.

Derived from

wxStreamBase (p. **Error! Bookmark not defined.**)

Include files

<wx/stream.h>

wxInputStream::wxInputStream

wxInputStream()

Creates a dummy input stream.

wxInputStream::~~wxInputStream

~wxInputStream()

Destructor.

wxInputStream::CanRead

bool CanRead() const

Returns true if some data is available in the stream right now, so that calling *Read()* (p. 827) wouldn't block.

wxInputStream::GetC

char GetC()

Returns the first character in the input queue and removes it, blocking until it appears if necessary.

Note

If EOF, return value is undefined and *LastRead()* will return 0 and not 1.

wxInputStream::Eof

bool Eof() const

Returns true after an attempt has been made to read past the end of the stream.

Note

In wxWidgets 2.6.x and below some streams returned *Eof()* when the last byte had been read rather than when an attempt had been made to read past the last byte. If you want to avoid depending on one behaviour or the other then call *LastRead()* (p. 827) to check the number of bytes actually read.

wxInputStream::LastRead

size_t LastRead() const

Returns the last number of bytes read.

wxInputStream::Peek

char Peek()

Returns the first character in the input queue without removing it.

Note

Blocks until something appears in the stream if necessary, if nothing ever does (i.e. EOF) *LastRead()* will return 0 (and the return value is undefined), otherwise *LastRead()* returns 1.

wxInputStream::Read

wxInputStream& Read(void *buffer, size_t size)

Reads the specified amount of bytes and stores the data in *buffer*.

Warning

The buffer absolutely needs to have at least the specified size.

Return value

This function returns a reference on the current object, so the user can test any states of the stream right away.

wxInputStream& Read(wxOutputStream& *stream_out*)

Reads data from the input queue and stores it in the specified output stream. The data is read until an error is raised by one of the two streams.

Return value

This function returns a reference on the current object, so the user can test any states of the stream right away.

wxInputStream::Seekl

off_t Seekl(off_t *pos*, wxSeekMode *mode* = wxFromStart)

Changes the stream current position.

Parameters

pos

Offset to seek to.

mode

One of **wxFromStart**, **wxFromEnd**, **wxFromCurrent**.

Return value

The new stream position or wxInvalidOffset on error.

wxInputStream::Telll

off_t Telll() 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.

wxIPAddress

`wxIPAddress` is an abstract base class for all internet protocol address objects. Currently, only `wxIPv4address` (p. 831) is implemented. An experimental implementation for IPV6, `wxIPv6address`, is being developed.

Derived from

`wxSocketAddress` (p. **Error! Bookmark not defined.**)

Include files

<wx/socket.h>

wxIPAddress::Hostname**virtual bool Hostname(const wxString& hostname)**

Set the address to *hostname*, which can be a host name or an IP-style address in a format dependent on implementation.

Return value

Returns true on success, false if something goes wrong (invalid hostname or invalid IP address).

virtual wxString Hostname()

Returns the hostname which matches the IP address.

wxIPAddress::IPAddress**virtual wxString IPAddress()**

Returns a `wxString` containing the IP address.

wxIPAddress::Service

virtual bool Service(const wxString& service)

Set the port to that corresponding to the specified *service*.

Return value

Returns true on success, false if something goes wrong (invalid service).

virtual bool Service(unsigned short service)

Set the port to that corresponding to the specified *service*.

Return value

Returns true on success, false if something goes wrong (invalid service).

virtual unsigned short Service()

Returns the current service.

wxIPAddress::AnyAddress

virtual bool AnyAddress()

Internally, this is the same as setting the IP address to **INADDR_ANY**.

On IPV4 implementations, 0.0.0.0

On IPV6 implementations, ::

Return value

Returns true on success, false if something went wrong.

wxIPAddress::LocalHost

virtual bool LocalHost()

Set address to localhost.

On IPV4 implementations, 127.0.0.1

On IPV6 implementations, ::1

Return value

Returns true on success, false if something went wrong.

wxIPAddress::IsLocalHost

virtual bool IsLocalHost()

Determines if current address is set to localhost.

Return value

Returns true if address is localhost, false if internet address.

wxIPv4address**Derived from**

wxIPAddress (p. 829)

Include files

<wx/socket.h>

wxIPv4address::Hostname

bool Hostname(const wxString& *hostname*)

Set the address to *hostname*, which can be a host name or an IP-style address in dot notation (a.b.c.d)

Return value

Returns true on success, false if something goes wrong (invalid hostname or invalid IP address).

wxString Hostname()

Returns the hostname which matches the IP address.

wxIPv4address::IPAddress

wxString IPAddress()

Returns a wxString containing the IP address in dot quad (127.0.0.1) format.

wxIPv4address::Service

bool Service(const wxString& *service*)

Set the port to that corresponding to the specified *service*.

Return value

Returns true on success, false if something goes wrong (invalid service).

bool Service(unsigned short *service*)

Set the port to that corresponding to the specified *service*.

Return value

Returns true on success, false if something goes wrong (invalid service).

unsigned short Service()

Returns the current service.

wxIPV4address::AnyAddress**bool AnyAddress()**

Set address to any of the addresses of the current machine. Whenever possible, use this function instead of *wxIPV4address::LocalHost* (p. 832), 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. 832), 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. **Error! Bookmark not defined.**)

Include files

<wx/joystick.h>

See also

wxJoystickEvent (p. 838)

wxJoystick::wxJoystick

wxJoystick(int joystick = wxJOYSTICK1)

Constructor. *joystick* may be one of wxJOYSTICK1, wxJOYSTICK2, indicating the joystick controller of interest.

wxJoystick::~~wxJoystick

~wxJoystick()

Destroys the wxJoystick object.

wxJoystick::GetButtonState

int GetButtonState() const

Returns the state of the joystick buttons. Every button is mapped to a single bit in the returned integer, with the first button being mapped to the least significant bit, and so on. A bitlist of wxJOY_BUTTONn identifiers, where n is 1, 2, 3 or 4 is available for historical reasons.

wxJoystick::GetManufacturerId

int GetManufacturerId() const

Returns the manufacturer id.

wxJoystick::GetMovementThreshold

int GetMovementThreshold() const

Returns the movement threshold, the number of steps outside which the joystick is deemed to have moved.

wxJoystick::GetNumberAxes

int GetNumberAxes() const

Returns the number of axes for this joystick.

wxJoystick::GetNumberButtons

int GetNumberButtons() const

Returns the number of buttons for this joystick.

wxJoystick::GetNumberJoysticks

static int GetNumberJoysticks()

Returns the number of joysticks currently attached to the computer.

wxJoystick::GetPollingMax**int GetPollingMax() const**

Returns the maximum polling frequency.

wxJoystick::GetPollingMin**int GetPollingMin() const**

Returns the minimum polling frequency.

wxJoystick::GetProductId**int GetProductId() const**

Returns the product id for the joystick.

wxJoystick::GetProductName**wxString GetProductName() const**

Returns the product name for the joystick.

wxJoystick::GetPosition**wxPoint GetPosition() const**

Returns the x, y position of the joystick.

wxJoystick::GetPOVPosition**int GetPOVPosition() const**

Returns the point-of-view position, expressed in continuous, one-hundredth of a degree units, but limited to return 0, 9000, 18000 or 27000. Returns -1 on error.

wxJoystick::GetPOVCTSPosition**int GetPOVCTSPosition() const**

Returns the point-of-view position, expressed in continuous, one-hundredth of a degree units. Returns -1 on error.

wxJoystick::GetRudderMax**int GetRudderMax() const**

Returns the maximum rudder position.

wxJoystick::GetRudderMin**int GetRudderMin() const**

Returns the minimum rudder position.

wxJoystick::GetRudderPosition**int GetRudderPosition() const**

Returns the rudder position.

wxJoystick::GetUMax**int GetUMax() const**

Returns the maximum U position.

wxJoystick::GetUMin**int GetUMin() const**

Returns the minimum U position.

wxJoystick::GetUPosition**int GetUPosition() const**

Gets the position of the fifth axis of the joystick, if it exists.

wxJoystick::GetVMax**int GetVMax() const**

Returns the maximum V position.

wxJoystick::GetVMin**int GetVMin() const**

Returns the minimum V position.

wxJoystick::GetVPosition**int GetVPosition() const**

Gets the position of the sixth axis of the joystick, if it exists.

wxJoystick::GetXMax

int GetXMax() const

Returns the maximum x position.

wxJoystick::GetXMin

int GetXMin() const

Returns the minimum x position.

wxJoystick::GetYMax

int GetYMax() const

Returns the maximum y position.

wxJoystick::GetYMin

int GetYMin() const

Returns the minimum y position.

wxJoystick::GetZMax

int GetZMax() const

Returns the maximum z position.

wxJoystick::GetZMin

int GetZMin() const

Returns the minimum z position.

wxJoystick::GetZPosition

int GetZPosition() const

Returns the z position of the joystick.

wxJoystick::HasPOV

bool HasPOV() const

Returns true if the joystick has a point of view control.

wxJoystick::HasPOV4Dir

bool HasPOV4Dir() const

Returns true if the joystick point-of-view supports discrete values (centered, forward, backward, left, and right).

wxJoystick::HasPOVCTS

bool HasPOVCTS() const

Returns true if the joystick point-of-view supports continuous degree bearings.

wxJoystick::HasRudder

bool HasRudder() const

Returns true if there is a rudder attached to the computer.

wxJoystick::HasU

bool HasU() const

Returns true if the joystick has a U axis.

wxJoystick::HasV

bool HasV() const

Returns true if the joystick has a V axis.

wxJoystick::HasZ

bool HasZ() const

Returns true if the joystick has a Z axis.

wxJoystick::IsOk

bool IsOk() const

Returns true if the joystick is functioning.

wxJoystick::ReleaseCapture

bool ReleaseCapture()

Releases the capture set by **SetCapture**.

Return value

true if the capture release succeeded.

See also

wxJoystick::SetCapture (p. 838), *wxJoystickEvent* (p. 838)

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. 837), *wxJoystickEvent* (p. 838)

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. 487)

Include files

<wx/event.h>

Event table macros

To process a mouse event, use these event handler macros to direct input to member functions that take a *wxJoystickEvent* argument.

EVT_JOY_BUTTON_DOWN(func) Process a *wxEVT_JOY_BUTTON_DOWN*

event.

EVT_JOY_BUTTON_UP(func)	Process a wxEVT_JOY_BUTTON_UP event.
EVT_JOY_MOVE(func)	Process a wxEVT_JOY_MOVE event.
EVT_JOY_ZMOVE(func)	Process a wxEVT_JOY_ZMOVE event.
EVT_JOYSTICK_EVENTS(func)	Processes all joystick events.

See also

wxJoystick (p. 832)

wxJoystickEvent::wxJoystickEvent

wxJoystickEvent(WXTYPE eventType = 0, int state = 0, int joystick = wxJOYSTICK1, int change = 0)

Constructor.

wxJoystickEvent::ButtonDown

bool ButtonDown(int button = wxJOY_BUTTON_ANY) const

Returns true if the event was a down event from the specified button (or any button).

Parameters

button

Can be wxJOY_BUTTONn where n is 1, 2, 3 or 4; or wxJOY_BUTTON_ANY to indicate any button down event.

wxJoystickEvent::ButtonIsDown

bool ButtonIsDown(int button = wxJOY_BUTTON_ANY) const

Returns true if the specified button (or any button) was in a down state.

Parameters

button

Can be wxJOY_BUTTONn where n is 1, 2, 3 or 4; or wxJOY_BUTTON_ANY to indicate any button down event.

wxJoystickEvent::ButtonUp

bool ButtonUp(int button = wxJOY_BUTTON_ANY) const

Returns true if the event was an up event from the specified button (or any button).

Parameters*button*

Can be wxJOY_BUTTONn where n is 1, 2, 3 or 4; or wxJOY_BUTTON_ANY to indicate any button down event.

wxJoystickEvent::GetButtonChange**int GetButtonChange() const**

Returns the identifier of the button changing state. This is a wxJOY_BUTTONn identifier, where n is one of 1, 2, 3, 4.

wxJoystickEvent::GetButtonState**int GetButtonState() const**

Returns the down state of the buttons. This is a bitlist of wxJOY_BUTTONn identifiers, where n is one of 1, 2, 3, 4.

wxJoystickEvent::GetJoystick**int GetJoystick() const**

Returns the identifier of the joystick generating the event - one of wxJOYSTICK1 and wxJOYSTICK2.

wxJoystickEvent::GetPosition**wxPoint GetPosition() const**

Returns the x, y position of the joystick event.

wxJoystickEvent::GetZPosition**int GetZPosition() const**

Returns the z position of the joystick event.

wxJoystickEvent::IsButton**bool IsButton() const**

Returns true if this was a button up or down event (*not* 'is any button down?').

wxJoystickEvent::IsMove**bool IsMove() const**

Returns true if this was an x, y move event.

wxJoystickEvent::IsZMove

bool IsZMove() const

Returns true if this was a z move event.

wxKeyEvent

This event class contains information about keypress (character) events.

Notice that there are three different kinds of keyboard events in wxWidgets: key down and up events and char events. The difference between the first two is clear - the first corresponds to a key press and the second to a key release - otherwise they are identical. Just note that if the key is maintained in a pressed state you will typically get a lot of (automatically generated) down events but only one up so it is wrong to assume that there is one up event corresponding to each down one.

Both key events provide untranslated key codes while the char event carries the translated one. The untranslated code for alphanumeric keys is always an upper case value. For the other keys it is one of `WXK_XXX` values from the *keycodes table* (p. **Error! Bookmark not defined.**). The translated key is, in general, the character the user expects to appear as the result of the key combination when typing the text into a text entry zone, for example.

A few examples to clarify this (all assume that CAPS LOCK is unpressed and the standard US keyboard): when the 'A' key is pressed, the key down event key code is equal to `ASCII_A == 65`. But the char event key code is `ASCII_a == 97`. On the other hand, if you press both SHIFT and 'A' keys simultaneously, the key code in key down event will still be just 'A' while the char event key code parameter will now be 'A' as well.

Although in this simple case it is clear that the correct key code could be found in the key down event handler by checking the value returned by *ShiftDown()* (p. 846), in general you should use `EVT_CHAR` for this as for non alphanumeric keys the translation is keyboard-layout dependent and can only be done properly by the system itself.

Another kind of translation is done when the control key is pressed: for example, for CTRL-A key press the key down event still carries the same key code 'a' as usual but the char event will have key code of 1, the ASCII value of this key combination.

You may discover how the other keys on your system behave interactively by running the *text* (p. **Error! Bookmark not defined.**) wxWidgets sample and pressing some keys in any of the text controls shown in it.

Note: If a key down (`EVT_KEY_DOWN`) event is caught and the event handler does not call `event.Skip()` then the corresponding char event (`EVT_CHAR`) will not happen. This is by design and enables the programs that handle both types of events to be a bit simpler.

Note for Windows programmers: The key and char events in wxWidgets are similar to but slightly different from Windows `WM_KEYDOWN` and `WM_CHAR` events. In particular, Alt-x combination will generate a char event in wxWidgets (unless it is used as an accelerator).

Tip: be sure to call `event.Skip()` for events that you don't process in key event function, otherwise menu shortcuts may cease to work under Windows.

Derived from

`wxEvent` (p. 487)

Include files

`<wx/event.h>`

Event table macros

To process a key event, use these event handler macros to direct input to member functions that take a `wxKeyEvent` argument.

EVT_KEY_DOWN(func)	Process a <code>wxEVT_KEY_DOWN</code> event (any key has been pressed).
EVT_KEY_UP(func)	Process a <code>wxEVT_KEY_UP</code> event (any key has been released).
EVT_CHAR(func)	Process a <code>wxEVT_CHAR</code> event.

`wxKeyEvent::m_altDown`

bool m_altDown

Deprecated: Please use *GetModifiers* (p. 844) instead!

true if the Alt key is pressed down.

`wxKeyEvent::m_controlDown`

bool m_controlDown

Deprecated: Please use *GetModifiers* (p. 844) instead!

true if control is pressed down.

`wxKeyEvent::m_keyCode`

long m_keyCode

Deprecated: Please use *GetKeyCode* (p. 844) instead!

Virtual keycode. See *Keycodes* (p. **Error! Bookmark not defined.**) for a list of

identifiers.

wxKeyEvent::m_metaDown

bool m_metaDown

Deprecated: Please use *GetModifiers* (p. 844) instead!

true if the Meta key is pressed down.

wxKeyEvent::m_shiftDown

bool m_shiftDown

Deprecated: Please use *GetModifiers* (p. 844) instead!

true if shift is pressed down.

wxKeyEvent::m_x

int m_x

Deprecated: Please use *GetX* (p. 846) instead!

X position of the event.

wxKeyEvent::m_y

int m_y

Deprecated: Please use *GetY* (p. 846) instead!

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.

Notice that *GetModifiers* (p. 844) is easier to use correctly than this function so you should consider using it in new code.

wxKeyEvent::CmdDown

bool CmdDown() const

CMD is a pseudo key which is the same as Control for PC and Unix platforms but the special APPLE (a.k.a as COMMAND) key under Macs: it makes often sense to use it instead of, say, `ControlDown()` because Cmd key is used for the same thing under Mac as Ctrl elsewhere (but Ctrl still exists, just not used for this purpose under Mac). So for non-Mac platforms this is the same as *ControlDown()* (p. 844) and under Mac this is the same as *MetaDown()* (p. 846).

wxKeyEvent::ControlDown**bool ControlDown() const**

Returns true if the control key was down at the time of the key event.

Notice that *GetModifiers* (p. 844) is easier to use correctly than this function so you should consider using it in new code.

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. **Error! Bookmark not defined.**) for a full list of the virtual key codes.

Note that in Unicode build, the returned value is meaningful only if the user entered a character that can be represented in current locale's default charset. You can obtain the corresponding Unicode character using *GetUnicodeKey* (p. 845).

wxKeyEvent::GetModifiers**int GetModifiers() const**

Return the bitmask of modifier keys which were pressed when this event happened. See *key modifier constants* (p. **Error! Bookmark not defined.**) for the full list of modifiers.

Notice that this function is easier to use correctly than, for example, *ControlDown* (p. 844) because when using the latter you also have to remember to test that none of the other modifiers is pressed:

```
if ( ControlDown() && !AltDown() && !ShiftDown() &&
    !MetaDown() )
    ... handle Ctrl-XXX ...
```

and forgetting to do it can result in serious program bugs (e.g. program not working with European keyboard layout where ALTGR key which is seen by the program as combination of CTRL and ALT is used). On the other hand, you can simply write

```
if ( GetModifiers() == wxMOD_CONTROL )
    ... handle Ctrl-XXX ...
```

with this function.

wxKeyEvent::GetPosition**wxPoint GetPosition() const****void GetPosition(long *x, long *y) const**

Obtains the position (in client coordinates) at which the key was pressed.

wxKeyEvent::GetRawKeyCode**wxUint32 GetRawKeyCode() const**

Returns the raw key code for this event. This is a platform-dependent scan code which should only be used in advanced applications.

NB: Currently the raw key codes are not supported by all ports, use `#ifdef wxHAS_RAW_KEY_CODES` to determine if this feature is available.

wxKeyEvent::GetRawKeyFlags**wxUint32 GetRawKeyFlags() const**

Returns the low level key flags for this event. The flags are platform-dependent and should only be used in advanced applications.

NB: Currently the raw key flags are not supported by all ports, use `#ifdef wxHAS_RAW_KEY_CODES` to determine if this feature is available.

wxKeyEvent::GetUnicodeKey**wxChar GetUnicodeKey() const**

Returns the Unicode character corresponding to this key event.

This function is only available in Unicode build, i.e. when `wxUSE_UNICODE` is 1.

wxKeyEvent::GetX**long GetX() const**

Returns the X position (in client coordinates) of the event.

wxKeyEvent::GetY**long GetY() const**

Returns the Y (in client coordinates) position of the event.

wxKeyEvent::HasModifiers**bool HasModifiers() const**

Returns true if either CTRL or ALT keys was down at the time of the key event. Note that this function does not take into account neither SHIFT nor META key states (the reason for ignoring the latter is that it is common for NUMLOCK key to be configured as META under X but the key presses even while NUMLOCK is on should be still processed normally).

wxKeyEvent::MetaDown

bool MetaDown() const

Returns true if the Meta key was down at the time of the key event.

Notice that *GetModifiers* (p. 844) is easier to use correctly than this function so you should consider using it in new code.

wxKeyEvent::ShiftDown

bool ShiftDown() const

Returns true if the shift key was down at the time of the key event.

Notice that *GetModifiers* (p. 844) is easier to use correctly than this function so you should consider using it in new code.

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. **Error! Bookmark not defined.**) for your subwindows since this class provides handlers for the required events, and accessors to specify the desired size of the window. The sash behaviour in the base class can be used, optionally, to make the windows user-resizable.

wxLayoutAlgorithm is typically used in IDE (integrated development environment) applications, where there are several resizable windows in addition to the MDI client window, or other primary editing window. Resizable windows might include toolbars, a project window, and a window for displaying error and warning messages.

When a window receives an OnCalculateLayout event, it should call SetRect in the given event object, to be the old supplied rectangle minus whatever space the window takes up. It should also set its own size accordingly.

wxSashLayoutWindow::OnCalculateLayout generates an OnQueryLayoutInfo event which it sends to itself to determine the orientation, alignment and size of the window, which it gets from internal member variables set by the application.

The algorithm works by starting off with a rectangle equal to the whole frame client area. It iterates through the frame children, generating `OnCalculateLayout` events which subtract the window size and return the remaining rectangle for the next window to process. It is assumed (by `wxSashLayoutWindow::OnCalculateLayout`) that a window stretches the full dimension of the frame client, according to the orientation it specifies. For example, a horizontal window will stretch the full width of the remaining portion of the frame client area. In the other orientation, the window will be fixed to whatever size was specified by `OnQueryLayoutInfo`. An alignment setting will make the window 'stick' to the left, top, right or bottom of the remaining client area. This scheme implies that order of window creation is important. Say you wish to have an extra toolbar at the top of the frame, a project window to the left of the MDI client window, and an output window above the status bar. You should therefore create the windows in this order: toolbar, output window, project window. This ensures that the toolbar and output window take up space at the top and bottom, and then the remaining height in-between is used for the project window.

`wxLayoutAlgorithm` is quite independent of the way in which `OnCalculateLayout` chooses to interpret a window's size and alignment. Therefore you could implement a different window class with a new `OnCalculateLayout` event handler, that has a more sophisticated way of laying out the windows. It might allow specification of whether stretching occurs in the specified orientation, for example, rather than always assuming stretching. (This could, and probably should, be added to the existing implementation).

Note: `wxLayoutAlgorithm` has nothing to do with `wxLayoutConstraints`. It is an alternative way of specifying layouts for which the normal constraint system is unsuitable.

Derived from

`wxObject` (p. **Error! Bookmark not defined.**)

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 <code>wxQueryLayoutInfoEvent</code> (p. Error! Bookmark not defined.). |
| 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 <code>wxCalculateLayoutEvent</code> (p. 125). |

Data types

```
enum wxLayoutOrientation {  
    wxLAYOUT_HORIZONTAL,  
    wxLAYOUT_VERTICAL
```

```
};  
  
enum wxLayoutAlignment {  
    wxLAYOUT_NONE,  
    wxLAYOUT_TOP,  
    wxLAYOUT_LEFT,  
    wxLAYOUT_RIGHT,  
    wxLAYOUT_BOTTOM,  
};
```

See also

wxSashEvent (p. **Error! Bookmark not defined.**), *wxSashLayoutWindow* (p. **Error! Bookmark not defined.**), *Event handling overview* (p. **Error! Bookmark not defined.**)

wxCalculateLayoutEvent (p. 125), *wxQueryLayoutInfoEvent* (p. **Error! Bookmark not defined.**), *wxSashLayoutWindow* (p. **Error! Bookmark not defined.**), *wxSashWindow* (p. **Error! Bookmark not defined.**)

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. 849).

wxLayoutAlgorithm::LayoutMDIFrame

bool LayoutMDIFrame(wxMDIParentFrame* frame, wxRect* rect = NULL) const

Lays out the children of an MDI parent frame. If *rect* is non-NULL, the given rectangle will be used as a starting point instead of the frame's client area.

The MDI client window is set to occupy the remaining space.

wxLayoutAlgorithm::LayoutWindow

bool LayoutWindow(wxWindow* parent, wxWindow* mainWindow = NULL) const

Lays out the children of a normal frame or other window.

mainWindow is set to occupy the remaining space. If this is not specified, then the last window that responds to a calculate layout event in query mode will get the remaining space (that is, a non-query OnCalculateLayout event will not be sent to this window and the window will be set to the remaining size).

wxLayoutConstraints

Note: constraints are now deprecated and you should use *sizers* (p. **Error! Bookmark not defined.**) instead.

Objects of this class can be associated with a window to define its layout constraints, with respect to siblings or its parent.

The class consists of the following eight constraints of class `wxIndividualLayoutConstraint`, some or all of which should be accessed directly to set the appropriate constraints.

- **left:** represents the left hand edge of the window
- **right:** represents the right hand edge of the window
- **top:** represents the top edge of the window
- **bottom:** represents the bottom edge of the window
- **width:** represents the width of the window
- **height:** represents the height of the window
- **centreX:** represents the horizontal centre point of the window
- **centreY:** represents the vertical centre point of the window

Most constraints are initially set to have the relationship `wxUnconstrained`, which means that their values should be calculated by looking at known constraints. The exceptions are *width* and *height*, which are set to `wxAsIs` to ensure that if the user does not specify a constraint, the existing width and height will be used, to be compatible with panel items which often have take a default size. If the constraint is `wxAsIs`, the dimension will not be changed.

wxPerl note: In wxPerl the constraints are accessed as `constraint = Wx::LayoutConstraints->new(); constraint->centreX->AsIs(); constraint->centreY->Unconstrained();`

Derived from

wxObject (p. **Error! Bookmark not defined.**)

Include files

<wx/layout.h>

See also

Overview and examples (p. **Error! Bookmark not defined.**),
wxIndividualLayoutConstraint (p. 823), *wxWindow::SetConstraints* (p. **Error! Bookmark not defined.**)

wxLayoutConstraints::wxLayoutConstraints

wxLayoutConstraints()

Constructor.

wxLayoutConstraints::bottom

wxIndividualLayoutConstraint bottom

Constraint for the bottom edge.

wxLayoutConstraints::centreX

wxIndividualLayoutConstraint centreX

Constraint for the horizontal centre point.

wxLayoutConstraints::centreY

wxIndividualLayoutConstraint centreY

Constraint for the vertical centre point.

wxLayoutConstraints::height

wxIndividualLayoutConstraint height

Constraint for the height.

wxLayoutConstraints::left

wxIndividualLayoutConstraint left

Constraint for the left-hand edge.

wxLayoutConstraints::right**wxIndividualLayoutConstraint right**

Constraint for the right-hand edge.

wxLayoutConstraints::top**wxIndividualLayoutConstraint top**

Constraint for the top edge.

wxLayoutConstraints::width**wxIndividualLayoutConstraint width**

Constraint for the width.

wxList

wxList classes provide linked list functionality for wxWidgets, and for an application if it wishes. Depending on the form of constructor used, a list can be keyed on integer or string keys to provide a primitive look-up ability, but please note that this feature is **deprecated**. See *wxHashMap* (p. 684) for a faster method of storage when random access is required.

While wxList class in the previous versions of wxWidgets only could contain elements of type wxObject and had essentially untyped interface (thus allowing you to put apples in the list and read back oranges from it), the new wxList classes family may contain elements of any type and has much more strict type checking. Unfortunately, it also requires an additional line to be inserted in your program for each list class you use (which is the only solution short of using templates which is not done in wxWidgets because of portability issues).

The general idea is to have the base class wxListBase working with *void *data* but make all of its dangerous (because untyped) functions protected, so that they can only be used from derived classes which, in turn, expose a type safe interface. With this approach a new wxList-like class must be defined for each list type (i.e. list of ints, of wxString or of MyObjects). This is done with *WX_DECLARE_LIST* and *WX_DEFINE_LIST* macros like this (notice the similarity with *WX_DECLARE_OBJARRAY* and *WX_IMPLEMENT_OBJARRAY* macros):

Example

```
// this part might be in a header or source (.cpp) file
class MyListElement
{
    ... // whatever
};

// declare our list class: this macro declares and partly
implements MyList
```

```

// class (which derives from wxListBase)
WX_DECLARE_LIST(MyListElement, MyList);

...

// the only requirement for the rest is to be AFTER the full
// declaration of
// MyListElement (for WX_DECLARE_LIST forward declaration is
// enough), but
// usually it will be found in the source file and not in the
// header

#include <wx/listimpl.cpp>
WX_DEFINE_LIST(MyList);

// now MyList class may be used as a usual wxList, but all of
// its methods
// will take/return the objects of the right (i.e.
// MyListElement) type. You
// also have MyList::Node type which is the type-safe version
// of wxNode.
MyList list;
MyListElement element;
list.Append(&element);           // ok
list.Append(17);                 // error: incorrect type

// let's iterate over the list
for ( MyList::Node *node = list.GetFirst(); node; node = node-
>GetNext() )
{
    MyListElement *current = node->GetData();

    ...process the current element...
}

```

For compatibility with previous versions `wxList` and `wxStringList` classes are still defined, but their usage is deprecated and they will disappear in the future versions completely. The use of the latter is especially discouraged as it is not only unsafe but is also much less efficient than `wxArrayString` (p. 70) class.

In the documentation of the list classes below, the template notations are used even though these classes are not really templates at all -- but it helps to think about them as if they were. You should replace `wxNode<T>` with `wxListName::Node` and `T` itself with the list element type (i.e. the first parameter of `WX_DECLARE_LIST`).

Derived from

`wxObject` (p. **Error! Bookmark not defined.**)

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. **Error! Bookmark not defined.**) for members that retrieve the data associated with a node, and members for getting to the next or previous node.

See also

wxNode (p. **Error! Bookmark not defined.**), *wxArray* (p. 57)

wxList::wxList

wxList()

wxList(int *n*, T **objects*[])

wxList(T **object*, ...)

Note: keyed lists are deprecated and should not be used in new code.

wxList(unsigned int *key_type*)

Constructors. *key_type* is one of `wxKEY_NONE`, `wxKEY_INTEGER`, or `wxKEY_STRING`, and indicates what sort of keying is required (if any).

objects is an array of *n* objects with which to initialize the list.

The variable-length argument list constructor must be supplied with a terminating `NULL`.

wxList::~~wxList**~wxList()**

Destroys the list. Also destroys any remaining nodes, but does not destroy client data held in the nodes.

wxList::Append**wxNode<T> * Append(T *object)**

Note: keyed lists are deprecated and should not be used in new code.

wxNode<T> * Append(long key, T *object)**wxNode<T> * Append(const wxString& key, T *object)**

Appends a new *wxNode* (p. **Error! Bookmark not defined.**) to the end of the list and puts a pointer to the *object* in the node. The last two forms store a key with the object for later retrieval using the key. The new node is returned in each case.

The key string is copied and stored by the list implementation.

wxList::Clear**void Clear()**

Clears the list (but does not delete the client data stored with each node unless you called `DeleteContents(true)`, in which case it deletes data).

wxList::DeleteContents**void DeleteContents(bool destroy)**

If *destroy* is `true`, instructs the list to call *delete* on the client contents of a node whenever the node is destroyed. The default is `false`.

wxList::DeleteNode**bool DeleteNode(wxNode<T> *node)**

Deletes the given node from the list, returning `true` if successful.

wxList::DeleteObject**bool DeleteObject(T *object)**

Finds the given client *object* and deletes the appropriate node from the list, returning `true` if successful. The application must delete the actual object separately.

wxList::Erase

void Erase(wxNode<T> *node)

Removes element at given position.

wxList::Find

wxNode<T> * Find(T * object)

Returns the node whose client data is *object* or NULL if none found.

Note: keyed lists are deprecated and should not be used in new code.

wxNode<T> * Find(long key)

wxNode<T> * Find(const wxString& key)

Returns the node whose stored key matches *key*. Use on a keyed list only.

wxList::GetCount

size_t GetCount() const

Returns the number of elements in the list.

wxList::GetFirst

wxNode<T> * GetFirst()

Returns the first node in the list (NULL if the list is empty).

wxList::GetLast

wxNode<T> * GetLast()

Returns the last node in the list (NULL if the list is empty).

wxList::IndexOf

int IndexOf(T* obj)

Returns the index of *obj* within the list or `wxNOT_FOUND` if *obj* is not found in the list.

wxList::Insert

wxNode<T> * Insert(T *object)

Insert object at front of list.

wxNode<T> * Insert(size_t position, T *object)

Insert object before *position*, i.e. the index of the new item in the list will be equal to *position*. *position* should be less than or equal to *GetCount* (p. 855); if it is equal to it, this

is the same as calling *Append* (p. 854).

wxNode<T> * Insert(wxNode<T> *node, T *object)

Inserts the object before the given *node*.

wxList::IsEmpty

bool IsEmpty() const

Returns *true* if the list is empty, *false* otherwise.

wxList::Item

wxNode<T> * Item(size_t index) const

Returns the node at given position in the list.

wxList::Member

wxNode<T> * Member(T *object)

NB: This function is deprecated, use *Find* (p. 855) instead.

Returns the node associated with *object* if it is in the list, *NULL* otherwise.

wxList::Nth

wxNode<T> * Nth(int n)

NB: This function is deprecated, use *Item* (p. 856) 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. 855) instead.

Returns the number of elements in the list.

wxList::Sort

void Sort(wxSortCompareFunction compfunc)

```
// Type of compare function for list sort operation (as in
'qsort')
typedef int (*wxSortCompareFunction)(const void *elem1, const
void *elem2);
```

Allows the sorting of arbitrary lists by giving a function to compare two list elements. We use the system **qsort** function for the actual sorting process.

If you use untyped `wxList` the sort function receives pointers to `wxObject` pointers (`wxObject **`), so be careful to dereference appropriately - but, of course, a better solution is to use list of appropriate type defined with `WX_DECLARE_LIST`.

Example:

```
int listcompare(const void *arg1, const void *arg2)
{
    return(compare(**(wxString **)arg1,      // use the wxString
'compare'          **(wxString **)arg2)); // function
}

void main()
{
    wxList list;

    list.Append(new wxString("DEF"));
    list.Append(new wxString("GHI"));
    list.Append(new wxString("ABC"));
    list.Sort(listcompare);
}
```

wxListbook

`wxListbook` is a class similar to `wxNotebook` (p. **Error! Bookmark not defined.**) but which uses a `wxListCtrl` (p. 864) to show the labels instead of the tabs.

There is no documentation for this class yet but its usage is identical to `wxNotebook` (except for the features clearly related to tabs only), so please refer to that class documentation for now. You can also use the *notebook sample* (p. **Error! Bookmark not defined.**) to see `wxListbook` in action.

Derived from

`wxControl` (p. 218)
`wxWindow` (p. **Error! Bookmark not defined.**)
`wxEvtHandler` (p. 490)
`wxObject` (p. **Error! Bookmark not defined.**)

Include files

<wx/listbook.h>

Window styles

wxLB_DEFAULT	Choose the default location for the labels depending on the current platform (left everywhere except Mac where it is top).
wxLB_TOP	Place labels above the page area.

wxLB_LEFT	Place labels on the left side.
wxLB_RIGHT	Place labels on the right side.
wxLB_BOTTOM	Place labels below the page area.

See also

wxBookCtrl (p. **Error! Bookmark not defined.**), *wxNotebook* (p. **Error! Bookmark not defined.**), *notebook sample* (p. **Error! Bookmark not defined.**)

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_SELECTED` for single clicks, and `wxEVT_COMMAND_LISTBOX_DOUBLE_CLICKED` for double clicks.

Derived from

wxControlWithItems (p. 219)
wxControl (p. 218)
wxWindow (p. **Error! Bookmark not defined.**)
wxEvtHandler (p. 490)
wxObject (p. **Error! Bookmark not defined.**)

Include files

<wx/listbox.h>

Window styles

wxLB_SINGLE	Single-selection list.
wxLB_MULTIPLE	Multiple-selection list: the user can toggle multiple items on and off.
wxLB_EXTENDED	Extended-selection list: the user can select multiple items using the SHIFT key and the mouse or special key combinations.
wxLB_HSCROLL	Create horizontal scrollbar if contents are too wide (Windows only).
wxLB_ALWAYS_SB	Always show a vertical scrollbar.
wxLB_NEEDED_SB	Only create a vertical scrollbar if needed.

wxLB_SORT The listbox contents are sorted in alphabetical order.

Note that `wxLB_SINGLE`, `wxLB_MULTIPLE` and `wxLB_EXTENDED` styles are mutually exclusive and you can specify at most one of them (single selection is the default).

See also *window styles overview* (p. **Error! Bookmark not defined.**).

Event handling

EVT_LISTBOX(id, func) Process a `wxEVT_COMMAND_LISTBOX_SELECTED` event, when an item on the list is selected.

EVT_LISTBOX_DCLICK(id, func) Process a `wxEVT_COMMAND_LISTBOX_DOUBLECLICKED` event, when the listbox is double-clicked.

See also

wxChoice (p. 145), *wxComboBox* (p. 176), *wxListCtrl* (p. 864), *wxCommandEvent* (p. 184)

wxListBox::wxListBox

wxListBox()

Default constructor.

wxListBox(wxWindow* parent, wxWindowID id, const wxPoint& pos = wxDefaultPosition, const wxSize& size = wxDefaultSize, int n, const wxString choices[] = NULL, long style = 0, const wxValidator& validator = wxDefaultValidator, const wxString& name = "listBox")

wxListBox(wxWindow* parent, wxWindowID id, const wxPoint& pos, const wxSize& size, const wxString& choices, long style = 0, const wxValidator& validator = wxDefaultValidator, const wxString& name = "listBox")

Constructor, creating and showing a list box.

Parameters

parent

Parent window. Must not be NULL.

id

Window identifier. A value of -1 indicates a default value.

pos

Window position.

size

Window size. If the default size (-1, -1) is specified then the window is sized appropriately.

n

Number of strings with which to initialise the control.

choices

An array of strings with which to initialise the control.

style

Window style. See *wxListBox* (p. 858).

validator

Window validator.

name

Window name.

See also

wxListBox::Create (p. 861), *wxValidator* (p. **Error! Bookmark not defined.**)

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

wxPerl note: In wxPerl there is just an array reference in place of *n* and *choices*.

wxListBox::~~wxListBox

void ~wxListBox()

Destructor, destroying the list box.

wxListBox::Create

bool Create(wxWindow* parent, wxWindowID id, const wxPoint& pos = wxDefaultPosition, const wxSize& size = wxDefaultSize, int n, const wxString choices[] = NULL, long style = 0, const wxValidator& validator = wxDefaultValidator, const wxString& name = "listBox")

bool Create(wxWindow* parent, wxWindowID id, const wxPoint& pos, const wxSize& size, const wxString& choices, long style = 0, const wxValidator& validator = wxDefaultValidator, const wxString& name = "listBox")

Creates the listbox for two-step construction. See *wxListBox::wxListBox* (p. 860) for further details.

wxListBox::Deselect**void Deselect(int *n*)**

Deselects an item in the list box.

Parameters*n*

The zero-based item to deselect.

Remarks

This applies to multiple selection listboxes only.

wxListBox::GetSelections**int GetSelections(wxArrayInt& *selections*) const**

Fill an array of ints with the positions of the currently selected items.

Parameters*selections*

A reference to an wxArrayInt instance that is used to store the result of the query.

Return value

The number of selections.

Remarks

Use this with a multiple selection listbox.

See also

wxControlWithItems::GetSelection (p. 222), *wxControlWithItems::GetStringSelection* (p. 223), *wxControlWithItems::SetSelection* (p. 225)

wxPython note: The wxPython version of this method takes no parameters and returns a tuple of the selected items.

wxPerl note: In wxPerl this method takes no parameters and return the selected items as a list.

wxListBox::InsertItems**void InsertItems(int *nItems*, const wxString **items*, int *pos*)****void InsertItems(const wxArrayString& *nItems*, int *pos*)**

Insert the given number of strings before the specified position.

Parameters*nItems*Number of items in the array *items**items*

Labels of items to be inserted

*pos*Position before which to insert the items: for example, if *pos* is 0 the items will be inserted in the beginning of the listbox**wxPython note:** The first two parameters are collapsed into a single parameter for wxPython, which is a list of strings.**wxPerl note:** In wxPerl there is just an array reference in place of *nItems* and *items*.**wxListBox::HitTest****int HitTest(const wxPoint& *point*) const**Returns the item located at *point*, or `wxNOT_FOUND` if there is no item located at *point*.

This function is new since wxWidgets version 2.7.0. It is currently implemented for wxMSW, wxMac and wxGTK2 ports.

Parameters*point*

Point of item (in client coordinates) to obtain

Return valueItem located at *point*, or `wxNOT_FOUND` if unimplemented or the item does not exist.**wxListBox::IsSelected****bool IsSelected(int *n*) const**

Determines whether an item is selected.

Parameters*n*

The zero-based item index.

Return value

true if the given item is selected, false otherwise.

wxListBox::Set**void Set(int *n*, const wxString* *choices*, void ***clientData* = NULL)****void Set(const wxArrayString& *choices*, void ***clientData* = NULL)**

Clears the list box and adds the given strings to it.

Parameters*n*

The number of strings to set.

choices

An array of strings to set.

clientData

Options array of client data pointers

Remarks

You may free the array from the calling program after this function has been called.

wxListBox::SetFirstItem**void SetFirstItem(int *n*)****void SetFirstItem(const wxString& *string*)**

Set the specified item to be the first visible item. Windows only.

Parameters*n*

The zero-based item index.

string

The string that should be visible.

wxListCtrl

A list control presents lists in a number of formats: list view, report view, icon view and small icon view. In any case, elements are numbered from zero. For all these modes, the items are stored in the control and must be added to it using *InsertItem* (p. 876) method.

A special case of report view quite different from the other modes of the list control is a virtual control in which the items data (including text, images and attributes) is managed

by the main program and is requested by the control itself only when needed which allows to have controls with millions of items without consuming much memory. To use virtual list control you must use *SetItemCount* (p. 881) first and overload at least *OnGetItemText* (p. 878) (and optionally *OnGetItemImage* (p. 877) or *OnGetItemColumnImage* (p. 877) and *OnGetItemAttr* (p. 877)) to return the information about the items when the control requests it. Virtual list control can be used as a normal one except that no operations which can take time proportional to the number of items in the control happen -- this is required to allow having a practically infinite number of items. For example, in a multiple selection virtual list control, the selections won't be sent when many items are selected at once because this could mean iterating over all the items.

Using many of *wxListCtrl* features is shown in the *corresponding sample* (p. **Error! Bookmark not defined.**).

To intercept events from a list control, use the event table macros described in *wxListEvent* (p. 884).

Derived from

wxControl (p. 218)
wxWindow (p. **Error! Bookmark not defined.**)
wxEvtHandler (p. 490)
wxObject (p. **Error! Bookmark not defined.**)

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 <i>wxLC_REPORT</i> . In other words, the list wraps, unlike a <i>wxListBox</i> .
wxLC_REPORT	Single or multicolumn report view, with optional header.
wxLC_VIRTUAL	The application provides items text on demand. May only be used with <i>wxLC_REPORT</i> .
wxLC_ICON	Large icon view, with optional labels.
wxLC_SMALL_ICON	Small icon view, with optional labels.
wxLC_ALIGN_TOP	Icons align to the top. Win32 default, Win32 only.
wxLC_ALIGN_LEFT	Icons align to the left.
wxLC_AUTOARRANGE	Icons arrange themselves. Win32 only.
wxLC_EDIT_LABELS	Labels are editable: the application will be

	notified when editing starts.
wxLC_NO_HEADER	No header in report mode.
wxLC_SINGLE_SEL	Single selection (default is multiple).
wxLC_SORT_ASCENDING	Sort in ascending order (must still supply a comparison callback in <code>SortItems</code>).
wxLC_SORT_DESCENDING	Sort in descending order (must still supply a comparison callback in <code>SortItems</code>).
wxLC_HRULES	Draws light horizontal rules between rows in report mode.
wxLC_VRULES	Draws light vertical rules between columns in report mode.

See also *window styles overview* (p. **Error! Bookmark not defined.**).

Event handling

To process input from a list control, use these event handler macros to direct input to member functions that take a *wxListEvent* (p. 884) argument.

EVT_LIST_BEGIN_DRAG(id, func) Begin dragging with the left mouse button.

EVT_LIST_BEGIN_RDRAG(id, func) Begin dragging with the right mouse button.

EVT_LIST_BEGIN_LABEL_EDIT(id, func) Begin editing a label. This can be prevented by calling *Veto()* (p. **Error! Bookmark not defined.**).

EVT_LIST_END_LABEL_EDIT(id, func) Finish editing a label. This can be prevented by calling *Veto()* (p. **Error! Bookmark not defined.**).

EVT_LIST_DELETE_ITEM(id, func) Delete an item.

EVT_LIST_DELETE_ALL_ITEMS(id, func) Delete all items.

EVT_LIST_ITEM_SELECTED(id, func) The item has been selected.

EVT_LIST_ITEM_DESELECTED(id, func) The item has been deselected.

EVT_LIST_ITEM_ACTIVATED(id, func) The item has been activated (ENTER or double click).

EVT_LIST_ITEM_FOCUSED(id, func) The currently focused item has changed.

EVT_LIST_ITEM_MIDDLE_CLICK(id, func) The middle mouse button has been clicked on an item.

EVT_LIST_ITEM_RIGHT_CLICK(id, func) The right mouse button has been clicked on an item.

EVT_LIST_KEY_DOWN(id, func)	A key has been pressed.
EVT_LIST_INSERT_ITEM(id, func)	An item has been inserted.
EVT_LIST_COL_CLICK(id, func)	A column (m_col) has been left-clicked.
EVT_LIST_COL_RIGHT_CLICK(id, func)	A column (m_col) has been right-clicked.
EVT_LIST_COL_BEGIN_DRAG(id, func)	The user started resizing a column - can be vetoed.
EVT_LIST_COL_DRAGGING(id, func)	The divider between columns is being dragged.
EVT_LIST_COL_END_DRAG(id, func)	A column has been resized by the user.
EVT_LIST_CACHE_HINT(id, func)	Prepare cache for a virtual list control

See also

wxListCtrl overview (p. **Error! Bookmark not defined.**), *wxListView* (p. 893), *wxListBox* (p. 858), *wxTreeCtrl* (p. **Error! Bookmark not defined.**), *wxImageList* (p. 818), *wxListEvent* (p. 884), *wxListItem* (p. 887)

wxListCtrl::wxListCtrl**wxListCtrl()**

Default constructor.

wxListCtrl(wxWindow* parent, wxWindowID id, const wxPoint& pos = wxDefaultPosition, const wxSize& size = wxDefaultSize, long style = wxLC_ICON, const wxValidator& validator = wxDefaultValidator, const wxString& name = wxListCtrlNameStr)

Constructor, creating and showing a list control.

Parameters

parent

Parent window. Must not be NULL.

id

Window identifier. A value of -1 indicates a default value.

pos

Window position.

size

Window size. If the default size (-1, -1) is specified then the window is sized appropriately.

style

Window style. See *wxListCtrl* (p. 864).

validator

Window validator.

name

Window name.

See also

wxListCtrl::Create (p. 868), *wxValidator* (p. **Error! Bookmark not defined.**)

wxListCtrl::~~wxListCtrl

void ~wxListCtrl()

Destructor, destroying the list control.

wxListCtrl::Arrange

bool Arrange(int flag = wxLIST_ALIGN_DEFAULT)

Arranges the items in icon or small icon view. This only has effect on Win32. *flag* is one of:

wxLIST_ALIGN_DEFAULT Default alignment.

wxLIST_ALIGN_LEFT Align to the left side of the control.

wxLIST_ALIGN_TOP Align to the top side of the control.

wxLIST_ALIGN_SNAP_TO_GRID Snap to grid.

wxListCtrl::AssignImageList

void AssignImageList(wxImageList* imageList, int which)

Sets the image list associated with the control and takes ownership of it (i.e. the control will, unlike when using *SetImageList*, delete the list when destroyed). *which* is one of *wxIMAGE_LIST_NORMAL*, *wxIMAGE_LIST_SMALL*, *wxIMAGE_LIST_STATE* (the last is unimplemented).

See also

wxListCtrl::SetImageList (p. 879)

wxListCtrl::ClearAll

void ClearAll()

Deletes all items and all columns.

wxListCtrl::Create

bool Create(wxWindow* parent, wxWindowID id, const wxPoint& pos = wxDefaultPosition, const wxSize& size = wxDefaultSize, long style = wxLC_ICON, const wxValidator& validator = wxDefaultValidator, const wxString& name = wxListCtrlNameStr)

Creates the list control. See *wxListCtrl::wxListCtrl* (p. 867) for further details.

wxListCtrl::DeleteAllItems

bool DeleteAllItems()

Deletes all 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. 869)).

wxListCtrl::DeleteColumn

bool DeleteColumn(int col)

Deletes a column.

wxListCtrl::DeleteItem

bool DeleteItem(long item)

Deletes the specified item. This function sends the `wxEVT_COMMAND_LIST_DELETE_ITEM` event for the item being deleted.

See also: *DeleteAllItems* (p. 869)

wxListCtrl::EditLabel

void EditLabel(long item)

Starts editing the label of the given item. This function generates a `EVT_LIST_BEGIN_LABEL_EDIT` event which can be vetoed so that no text control will appear for in-place editing.

If the user changed the label (i.e. s/he does not press ESC or leave the text control without changes, a `EVT_LIST_END_LABEL_EDIT` event will be sent which can be vetoed as well.

wxListCtrl::EnsureVisible

bool EnsureVisible(long item)

Ensures this item is visible.

wxListCtrl::FindItem

long FindItem(long start, const wxString& str, const bool partial = false)

Find an item whose label matches this string, starting from *start* or the beginning if *start* is -1.

long FindItem(long start, long data)

Find an item whose data matches this data, starting from *start* or the beginning if 'start' is -1.

long FindItem(long start, const wxPoint& pt, int direction)

Find an item nearest this position in the specified direction, starting from *start* or the beginning if *start* is -1.

wxPython note: In place of a single overloaded method name, wxPython implements the following methods:

FindItem(start, str, partial=false)

FindItemData(start, data)

FindItemAtPos(start, point, direction)

wxPerl note: In wxPerl there are three methods instead of a single overloaded method:

FindItem(start, str, partial = false)

FindItemData(start, data)

FindItemAtPos(start, point, direction)

wxListCtrl::GetColumn

bool GetColumn(int col, wxListItem& item) const

Gets information about this column. See *wxListCtrl::SetItem* (p. 879) for more information.

wxPerl note: In wxPerl this method takes only the **col** parameter and returns a *Wx::ListItem* (or undef).

wxListCtrl::GetColumnCount

int GetColumnCount() const

Returns the number of columns.

wxListCtrl::GetColumnWidth**int GetColumnWidth(int col) const**

Gets the column width (report view only).

wxListCtrl::GetCountPerPage**int GetCountPerPage() const**

Gets the number of items that can fit vertically in the visible area of the list control (list or report view) or the total number of items in the list control (icon or small icon view).

wxListCtrl::GetEditControl**wxTextCtrl * GetEditControl() const**

Returns the edit control being currently used to edit a label. Returns `NULL` if no label is being edited.

NB: It is currently only implemented for wxMSW.

wxListCtrl::GetImageList**wxImageList* GetImageList(int which) const**

Returns the specified image list. *which* may be one of:

wxIMAGE_LIST_NORMAL The normal (large icon) image list.

wxIMAGE_LIST_SMALL The small icon image list.

wxIMAGE_LIST_STATE The user-defined state image list (unimplemented).

wxListCtrl::GetItem**bool GetItem(wxListItem& info) const**

Gets information about the item. See *wxListCtrl::SetItem* (p. 879) for more information.

You must call *info.SetId()* to the ID of item you're interested in before calling this method.

wxPython note: The wxPython version of this method takes an integer parameter for the item ID, an optional integer for the column number, and returns the `wxListItem` object.

wxPerl note: In wxPerl this method takes as parameter the **ID** of the item and (optionally) the column, and returns a `Wx::ListItem` object.

wxListCtrl::GetItemBackgroundColour**wxColour GetItemBackgroundColour(long item) const**

Returns the colour for this item. If the item has no specific colour, returns an invalid colour (and not the default background control of the control itself).

See also

GetItemTextColour (p. 873)

wxListCtrl::GetItemCount

int GetItemCount() const

Returns the number of items in the list control.

wxListCtrl::GetItemData

long GetItemData(long *item*) const

Gets the application-defined data associated with this item.

wxListCtrl::GetItemFont

wxFont GetItemFont(long *item*) const

Returns the item's font.

wxListCtrl::GetItemPosition

bool GetItemPosition(long *item*, wxPoint& *pos*) const

Returns the position of the item, in icon or small icon view.

wxPython note: The wxPython version of this method accepts only the item ID and returns the wxPoint.

wxPerl note: In wxPerl this method takes only the **item** parameter and returns a Wx::Point (or undef).

wxListCtrl::GetItemRect

bool GetItemRect(long *item*, wxRect& *rect*, int *code* = wxLIST_RECT_BOUNDS) const

Returns the rectangle representing the item's size and position, in physical coordinates.

code is one of wxLIST_RECT_BOUNDS, wxLIST_RECT_ICON, wxLIST_RECT_LABEL.

wxPython note: The wxPython version of this method accepts only the item ID and *code* and returns the wxRect.

wxPerl note: In wxPerl this method takes only the **item** parameter and returns a Wx::Rect (or undef).

wxListCtrl::GetItemSpacing**wxSize GetItemSpacing() const**

Retrieves the spacing between icons in pixels: horizontal spacing is returned as *x* component of the *wxSize* (p. **Error! Bookmark not defined.**) object and the vertical spacing as its *y* component.

wxListCtrl::GetItemState**int GetItemState(long item, long stateMask) const**

Gets the item state. For a list of state flags, see *wxListCtrl::SetItem* (p. 879).

The **stateMask** indicates which state flags are of interest.

wxListCtrl::GetItemText**wxString GetItemText(long item) const**

Gets the item text for this item.

wxListCtrl::GetItemTextColour**wxColour GetItemTextColour(long item) const**

Returns the colour for this item. If the item has no specific colour, returns an invalid colour (and not the default foreground control of the control itself as this wouldn't allow distinguishing between items having the same colour as the current control foreground and items with default colour which, hence, have always the same colour as the control).

wxListCtrl::GetNextItem**long GetNextItem(long item, int geometry = wxLIST_NEXT_ALL, int state = wxLIST_STATE_DONTCARE) const**

Searches for an item with the given geometry or state, starting from *item* but excluding the *item* itself. If *item* is -1, the first item that matches the specified flags will be returned.

Returns the first item with given state following *item* or -1 if no such item found.

This function may be used to find all selected items in the control like this:

```
long item = -1;
for ( ;; )
{
    item = listctrl->GetNextItem(item,
                                wxLIST_NEXT_ALL,
                                wxLIST_STATE_SELECTED);

    if ( item == -1 )
        break;

    // this item is selected - do whatever is needed with it
    wxLogMessage("Item %ld is selected.", item);
}
```

```
}
```

geometry can be one of:

- | | |
|--------------------------------|--|
| <code>wxLIST_NEXT_ABOVE</code> | Searches for an item above the specified item. |
| <code>wxLIST_NEXT_ALL</code> | Searches for subsequent item by index. |
| <code>wxLIST_NEXT_BELOW</code> | 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 parameter 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::GetViewRect

wxRect GetViewRect() const

Returns the rectangle taken by all items in the control. In other words, if the controls client size were equal to the size of this rectangle, no scrollbars would be needed and no

free space would be left.

Note that this function only works in the icon and small icon views, not in list or report views (this is a limitation of the native Win32 control).

wxListCtrl::HitTest

long HitTest(const wxPoint& point, int& flags)

Determines which item (if any) is at the specified point, giving details in *flags*. Returns index of the item or `wxNOT_FOUND` if no item is at the specified point. *flags* will be a combination of the following flags:

`wxLIST_HITTEST_ABOVE` Above the client area.

`wxLIST_HITTEST_BELOW` Below the client area.

`wxLIST_HITTEST_NOWHERE` In the client area but below the last item.

`wxLIST_HITTEST_ONITEMICON` On the bitmap associated with an item.

`wxLIST_HITTEST_ONITEMLABEL` On the label (string) associated with an item.

`wxLIST_HITTEST_ONITEMRIGHT` In the area to the right of an item.

`wxLIST_HITTEST_ONITEMSTATEICON` On the state icon for a tree view item that is in a user-defined state.

`wxLIST_HITTEST_TOLEFT` To the right of the client area.

`wxLIST_HITTEST_TORIGHT` To the left of the client area.

`wxLIST_HITTEST_ONITEM` Combination of `wxLIST_HITTEST_ONITEMICON`, `wxLIST_HITTEST_ONITEMLABEL`, `wxLIST_HITTEST_ONITEMSTATEICON`.

wxPython note: A tuple of values is returned in the wxPython version of this method. The first value is the item id and the second is the flags value mentioned above.

wxPerl note: In wxPerl this method only takes the **point** parameter and returns a 2-element list (*item*, *flags*).

wxListCtrl::InsertColumn

long InsertColumn(long col, wxListItem& info)

long InsertColumn(long col, const wxString& heading, int format = `wxLIST_FORMAT_LEFT`, int width = -1)

For report view mode (only), inserts a column. For more details, see `wxListCtrl::SetItem` (p. 879).

wxPython note: In place of a single overloaded method name, wxPython implements the following methods:

InsertColumn(col, heading, format=wxLIST_FORMAT_LEFT, width=-1)
Creates a column using a header string only.

InsertColumnItem(col, item) Creates a column using a wxListItem.

wxListCtrl::InsertItem

long InsertItem(wxListItem& info)

Inserts an item, returning the index of the new item if successful, -1 otherwise.

long InsertItem(long index, const wxString& label)

Inserts a string item.

long InsertItem(long index, int imageIndex)

Inserts an image item.

long InsertItem(long index, const wxString& label, int imageIndex)

Insert an image/string item.

Parameters

info

wxListItem object

index

Index of the new item, supplied by the application

label

String label

imageIndex

index into the image list associated with this control and view style

wxPython note: In place of a single overloaded method name, wxPython implements the following methods:

InsertItem(item) Inserts an item using a wxListItem.

InsertStringItem(index, label) Inserts a string item.

InsertImageItem(index, imageIndex) Inserts an image item.

InsertImageStringItem(index, label, imageIndex) Insert an image/string item.

wxPerl note: In wxPerl there are four methods instead of a single overloaded method:

InsertItem(item) Inserts a `Wx::ListItem`
InsertStringItem(index, label) Inserts a string item
InsertImageItem(index, imageIndex) Inserts an image item
InsertImageStringItem(index, label, imageIndex) Inserts an item with a string and an image

wxListCtrl::OnGetItemAttr

virtual wxListItemAttr * OnGetItemAttr(long item) const

This function may be overloaded in the derived class for a control with `wxLC_VIRTUAL` style. It should return the attribute for the specified `item` or `NULL` to use the default appearance parameters.

`wxListCtrl` will not delete the pointer or keep a reference of it. You can return the same `wxListItemAttr` pointer for every `OnGetItemAttr` call.

The base class version always returns `NULL`.

See also

OnGetItemImage (p. 877),
OnGetItemColumnImage (p. 877),
OnGetItemText (p. 878)

wxListCtrl::OnGetItemImage

virtual int OnGetItemImage(long item) const

This function must be overloaded in the derived class for a control with `wxLC_VIRTUAL` style having an *image list* (p. 879) (if the control doesn't have an image list, it is not necessary to overload it). It should return the index of the item's image in the control's image list or -1 for no image. In a control with `wxLC_REPORT` style, `OnGetItemImage` only gets called for the first column of each line.

The base class version always returns -1.

See also

OnGetItemText (p. 878),
OnGetItemColumnImage (p. 877),
OnGetItemAttr (p. 877)

wxListCtrl::OnGetItemColumnImage

virtual int OnGetItemColumnImage(long item, long column) const

Overload this function in the derived class for a control with `wxLC_VIRTUAL` and `wxLC_REPORT` styles in order to specify the image index for the given line and column.

The base class version always calls `OnGetItemImage` for the first column, else it returns -1.

See also

OnGetItemText (p. 878),
OnGetItemImage (p. 877),
OnGetItemAttr (p. 877)

wxListCtrl::OnGetItemText

virtual wxString OnGetItemText(long item, long column) const

This function **must** be overloaded in the derived class for a control with `wxLC_VIRTUAL` style. It should return the string containing the text of the given *column* for the specified *item*.

See also

SetItemCount (p. 881),
OnGetItemImage (p. 877),
OnGetItemColumnImage (p. 877),
OnGetItemAttr (p. 877)

wxListCtrl::RefreshItem

void RefreshItem(long item)

Redraws the given *item*. This is only useful for the virtual list controls as without calling this function the displayed value of the item doesn't change even when the underlying data does change.

See also

RefreshItems (p. 878)

wxListCtrl::RefreshItems

void RefreshItems(long itemFrom, long itemTo)

Redraws the items between *itemFrom* and *itemTo*. The starting item must be less than or equal to the ending one.

Just as *RefreshItem* (p. 878) this is only useful for virtual list controls.

wxListCtrl::ScrollList

bool ScrollList(int dx, int dy)

Scrolls the list control. If in icon, small icon or report view mode, *dx* specifies the number of pixels to scroll. If in list view mode, *dx* specifies the number of columns to scroll. *dy* always specifies the number of pixels to scroll vertically.

NB: This method is currently only implemented in the Windows version.

wxListCtrl::SetBackgroundColour

void SetBackgroundColour(const wxColour& col)

Sets the background colour (GetBackgroundColour already implicit in wxWindow class).

wxListCtrl::SetColumn

bool SetColumn(int col, wxListItem& item)

Sets information about this column. See *wxListCtrl::SetItem* (p. 879) for more information.

wxListCtrl::SetColumnWidth

bool SetColumnWidth(int col, int width)

Sets the column width.

width can be a width in pixels or wxLIST_AUTOSIZE (-1) or wxLIST_AUTOSIZE_USEHEADER (-2). wxLIST_AUTOSIZE will resize the column to the length of its longest item. wxLIST_AUTOSIZE_USEHEADER will resize the column to the length of the header (Win32) or 80 pixels (other platforms).

In small or normal icon view, *col* must be -1, and the column width is set for all columns.

wxListCtrl::SetImageList

void SetImageList(wxImageList* imageList, int which)

Sets the image list associated with the control. *which* is one of wxIMAGE_LIST_NORMAL, wxIMAGE_LIST_SMALL, wxIMAGE_LIST_STATE (the last is unimplemented).

This method does not take ownership of the image list, you have to delete it yourself.

See also

wxListCtrl::AssignImageList (p. 868)

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 <code>wxLIST_FORMAT_LEFT</code> , <code>wxLIST_FORMAT_RIGHT</code> or <code>wxLIST_FORMAT_CENTRE</code> .
<code>int m_width</code>	For columns only: the column width.

The **`m_mask`** member contains a bitlist specifying which of the other fields are valid. The flags are:

<code>wxLIST_MASK_STATE</code>	The <code>m_state</code> field is valid.
<code>wxLIST_MASK_TEXT</code>	The <code>m_text</code> field is valid.
<code>wxLIST_MASK_IMAGE</code>	The <code>m_image</code> field is valid.
<code>wxLIST_MASK_DATA</code>	The <code>m_data</code> field is valid.
<code>wxLIST_MASK_WIDTH</code>	The <code>m_width</code> field is valid.
<code>wxLIST_MASK_FORMAT</code>	The <code>m_format</code> field is valid.

The **`m_stateMask`** and **`m_state`** members take flags from the following:

<code>wxLIST_STATE_DONTCARE</code>	Don't care what the state is. Win32 only.
<code>wxLIST_STATE_DROPHILITED</code>	The item is highlighted to receive a drop event. Win32 only.
<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 in the cut state. Win32 only.

The `wxListItem` object can also contain item-specific colour and font information: for this you need to call one of `SetTextColour()`, `SetBackgroundColour()` or `SetFont()` functions on it passing it the colour/font to use. If the colour/font is not specified, the default list control colour/font is used.

`long SetItem(long index, int col, const wxString& label, int imageId = -1)`

Sets a string field at a particular column.

wxPython note: In place of a single overloaded method name, wxPython implements the following methods:

`SetItem(item)` Sets information about the given `wxListItem`.

`SetStringItem(index, col, label, imageId)` Sets a string or image at a given location.

`wxListCtrl::SetItemBackgroundColour`

`void SetItemBackgroundColour(long item, const wxColour& col)`

Sets the background colour for this item. This function only works in report view.

The colour can be retrieved using `GetItemBackgroundColour` (p. 871).

`wxListCtrl::SetItemCount`

`void SetItemCount(long count)`

This method can only be used with virtual list controls. It is used to indicate to the control the number of items it contains. After calling it, the main program should be ready to handle calls to various item callbacks (such as `OnGetItemText` (p. 878)) for all items in the range from 0 to *count*.

`wxListCtrl::SetItemData`

`bool SetItemData(long item, long data)`

Associates application-defined data with this item.

`wxListCtrl::SetItemFont`

`void SetItemFont(long item, const wxFont& font)`

Sets the item's font.

`wxListCtrl::SetItemImage`

`bool SetItemImage(long item, int image)`

Sets the image associated with the item. The image is an index into the image list

associated with the list control. In report view, this only sets the image for the first column.

bool SetItemImage(long item, int image, int selImage)

Sets the unselected and selected images associated with the item. The images are indices into the image list associated with the list control. This form is deprecated: *selImage* is not used.

wxListCtrl::SetItemColumnImage

bool SetItemImage(long item, long column, int image)

Sets the image associated with the item. In report view, you can specify the column. The image is an index 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. Windows only.

wxListCtrl::SetItemState

bool SetItemState(long item, long state, long stateMask)

Sets the item state. For a list of state flags, see *wxListCtrl::SetItem* (p. 879).

The **stateMask** indicates which state flags are valid.

wxListCtrl::SetItemText

void SetItemText(long item, const wxString& text)

Sets the item text for this item.

wxListCtrl::SetItemTextColour

void SetItemTextColour(long item, const wxColour& col)

Sets the colour for this item. This function only works in report view.

The colour can be retrieved using *GetItemTextColour* (p. 873).

wxListCtrl::SetSingleStyle

void SetSingleStyle(long style, const bool add = true)

Adds or removes a single window style.

wxListCtrl::SetTextColour

void SetTextColour(const wxColour& col)

Sets the text colour of the list control.

wxListCtrl::SetWindowStyleFlag

void SetWindowStyleFlag(long style)

Sets the whole window style, deleting all items.

wxListCtrl::SortItems

bool SortItems(wxListCtrlCompare fnSortCallBack, long data)

Call this function to sort the items in the list control. Sorting is done using the specified *fnSortCallBack* function. This function must have the following prototype:

```
int wxCALLBACK wxListCompareFunction(long item1, long item2, long
sortData)
```

It is called each time when the two items must be compared and should return 0 if the items are equal, negative value if the first item is less than the second one and positive value if the first one is greater than the second one (the same convention as used by `qsort(3)`).

Parameters

item1

client data associated with the first item (**NOT** the index).

item2

client data associated with the second item (**NOT** the index).

data

the value passed to `SortItems()` itself.

Notice that the control may only be sorted on client data associated with the items, so you **must** use *SetItemData* (p. 881) if you want to be able to sort the items in the control.

Please see the *listctrl sample* (p. **Error! Bookmark not defined.**) for an example of using this function.

wxPython note: wxPython uses the `sortData` parameter to pass the Python function to call, so it is not available for programmer use. Call `SortItems` with a reference to a callable object that expects two parameters.

wxPerl note: In wxPerl the comparison function must take just two parameters; however, you may use a closure to achieve an effect similar to the `SortItems` third parameter.

wxListEvent

A list event holds information about events associated with wxListCtrl objects.

Derived from

wxNotifyEvent (p. **Error! Bookmark not defined.**)

wxCommandEvent (p. 184)

wxEvent (p. 487)

wxObject (p. **Error! Bookmark not defined.**)

Include files

<wx/listctrl.h>

Event table macros

To process input from a list control, use these event handler macros to direct input to member functions that take a wxListEvent argument.

EVT_LIST_BEGIN_DRAG(id, func) Begin dragging with the left mouse button.

EVT_LIST_BEGIN_RDRAG(id, func) Begin dragging with the right mouse button.

EVT_LIST_BEGIN_LABEL_EDIT(id, func) Begin editing a label. This can be prevented by calling *Veto()* (p. **Error! Bookmark not defined.**).

EVT_LIST_END_LABEL_EDIT(id, func) Finish editing a label. This can be prevented by calling *Veto()* (p. **Error! Bookmark not defined.**).

EVT_LIST_DELETE_ITEM(id, func) Delete an item.

EVT_LIST_DELETE_ALL_ITEMS(id, func) Delete all items.

EVT_LIST_ITEM_SELECTED(id, func) The item has been selected.

EVT_LIST_ITEM_DESELECTED(id, func) The item has been deselected.

EVT_LIST_ITEM_ACTIVATED(id, func) The item has been activated (ENTER or double click).

EVT_LIST_ITEM_FOCUSED(id, func) The currently focused item has changed.

EVT_LIST_ITEM_MIDDLE_CLICK(id, func) The middle mouse button has been clicked on an item.

EVT_LIST_ITEM_RIGHT_CLICK(id, func) The right mouse button has been clicked on an item.

EVT_LIST_KEY_DOWN(id, func) A key has been pressed.

EVT_LIST_INSERT_ITEM(id, func) An item has been inserted.

- EVT_LIST_COL_CLICK(id, func)** A column (**m_col**) has been left-clicked.
- EVT_LIST_COL_RIGHT_CLICK(id, func)** A column (**m_col**) (which can be -1 if the click occurred outside any column) has been right-clicked.
- EVT_LIST_COL_BEGIN_DRAG(id, func)** The user started resizing a column - can be vetoed.
- EVT_LIST_COL_DRAGGING(id, func)** The divider between columns is being dragged.
- EVT_LIST_COL_END_DRAG(id, func)** A column has been resized by the user.
- EVT_LIST_CACHE_HINT(id, func)** Prepare cache for a virtual list control

See also

wxListCtrl (p. 864)

wxListEvent::wxListEvent

wxListEvent(WXTYPE *commandType* = 0, int *id* = 0)

Constructor.

wxListEvent::GetCacheFrom

long GetCacheFrom() const

For `EVT_LIST_CACHE_HINT` event only: return the first item which the list control advises us to cache.

wxListEvent::GetCacheTo

long GetCacheTo() const

For `EVT_LIST_CACHE_HINT` event only: return the last item (inclusive) which the list control advises us to cache.

wxListEvent::GetKeyCode

int GetKeyCode() const

Key code if the event is a keypress event.

wxListEvent::GetIndex

long GetIndex() const

The item index.

wxListEvent::GetColumn**int GetColumn() const**

The column position: it is only used with `COL` events. For the column dragging events, it is the column to the left of the divider being dragged, for the column click events it may be -1 if the user clicked in the list control header outside any column.

wxListEvent::GetPoint**wxPoint GetPoint() const**

The position of the mouse pointer if the event is a drag event.

wxListEvent::GetLabel**const wxString& GetLabel() const**

The (new) item label for `EVT_LIST_END_LABEL_EDIT` event.

wxListEvent::GetText**const wxString& GetText() const**

The text.

wxListEvent::GetImage**int GetImage() const**

The image.

wxListEvent::GetData**long GetData() const**

The data.

wxListEvent::GetMask**long GetMask() const**

The mask.

wxListEvent::GetItem**const wxListItem& GetItem() const**

An item object, used by some events. See also *wxListCtrl::SetItem* (p. 879).

wxCommandEvent::IsEditCancelled**bool IsEditCancelled() const**

This method only makes sense for `EVT_LIST_END_LABEL_EDIT` message and returns `true` if the label editing has been cancelled by the user (*GetLabel* (p. 886) returns an empty string in this case but it doesn't allow the application to distinguish between really cancelling the edit and the admittedly rare case when the user wants to rename it to an empty string).

wxListItem

This class stores information about a `wxListCtrl` item or column.

Derived from

wxObject (p. **Error! Bookmark not defined.**)

Include files

<wx/listctrl.h>

wxListItem::wxListItem**wxListItem()**

Constructor.

wxListItem::Clear**void Clear()**

Resets the item state to the default.

wxListItem::GetAlign**wxListColumnFormat GetAlign() const**

Returns the alignment for this item. Can be one of `wxLIST_FORMAT_LEFT`, `wxLIST_FORMAT_RIGHT` or `wxLIST_FORMAT_CENTRE`.

wxListItem::GetBackgroundColour**wxColour GetBackgroundColour() const**

Returns the background colour for this item.

wxListItem::GetColumn

int GetColumn() const

Returns the zero-based column; meaningful only in report mode.

wxListItem::GetData**long GetData() const**

Returns client data associated with the control. Please note that client data is associated with the item and not with subitems.

wxListItem::GetFont**wxFont GetFont() const**

Returns the font used to display the item.

wxListItem::GetId**long GetId() const**

Returns the zero-based item position.

wxListItem::GetImage**int GetImage() const**

Returns the zero-based index of the image associated with the item into the image list.

wxListItem::GetMask**long GetMask() const**

Returns a bit mask indicating which fields of the structure are valid; can be any combination of the following values:

<code>wxLIST_MASK_STATE</code>	GetState is valid.
<code>wxLIST_MASK_TEXT</code>	GetText is valid.
<code>wxLIST_MASK_IMAGE</code>	GetImage is valid.
<code>wxLIST_MASK_DATA</code>	GetData is valid.
<code>wxLIST_MASK_WIDTH</code>	GetWidth is valid.
<code>wxLIST_MASK_FORMAT</code>	GetFormat is valid.

wxListItem::GetState**long GetState() const**

Returns a bit field representing the state of the item. Can be any combination of:

`wxLIST_STATE_DONTCARE` Don't care what the state is. Win32 only.

`wxLIST_STATE_DROPHILITED` The item is highlighted to receive a drop event.
Win32 only.

`wxLIST_STATE_FOCUSED` The item has the focus.

`wxLIST_STATE_SELECTED` The item is selected.

`wxLIST_STATE_CUT` The item is in the cut state. Win32 only.

`wxListItem::GetText`

`const wxString& GetText() const`

Returns the label/header text.

`wxListItem::GetTextColour`

`wxColour GetTextColour() const`

Returns the text colour.

`wxListItem::GetWidth`

`int GetWidth() const`

Meaningful only for column headers in report mode. Returns the column width.

`wxListItem::SetAlign`

`void SetAlign(wxListColumnFormat align)`

Sets the alignment for the item. See also `wxListItem::GetAlign` (p. 887)

`wxListItem::SetBackgroundColour`

`void SetBackgroundColour(const wxColour& colBack)`

Sets the background colour for the item.

`wxListItem::SetColumn`

`void SetColumn(int col)`

Sets the zero-based column. Meaningful only in report mode.

`wxListItem::SetData`

void SetData(long data)

void SetData(void* data)

Sets client data for the item. Please note that client data is associated with the item and not with subitems.

wxListItem::SetFont

void SetFont(const wxFont& font)

Sets the font for the item.

wxListItem::SetId

void SetId(long id)

Sets the zero-based item position.

wxListItem::SetImage

void SetImage(int image)

Sets the zero-based index of the image associated with the item into the image list.

wxListItem::SetMask

void SetMask(long mask)

Sets the mask of valid fields. See *wxListItem::GetMask* (p. 888).

wxListItem::SetState

void SetState(long state)

Sets the item state flags (note that the valid state flags are influenced by the value of the state mask, see *wxListItem::SetStateMask* (p. 890)). See *wxListItem::GetState* (p. 889) for valid flag values.

wxListItem::SetStateMask

void SetStateMask(long stateMask)

Sets the bitmask that is used to determine which of the state flags are to be set. See also *wxListItem::SetState* (p. 890).

wxListItem::SetText

void SetText(const wxString& text)

Sets the text label for the item.

wxListItem::SetTextColour

void SetTextColour(const wxColour& colText)

Sets the text colour for the item.

wxListItem::SetWidth

void SetWidth(int width)

Meaningful only for column headers in report mode. Sets the column width.

wxListItemAttr

Represents the attributes (color, font, ...) of *awxListCtrl* (p. 864) *wxListItem* (p. 864).

Include files

<wx/listctrl.h>

See also

wxListCtrl overview (p. **Error! Bookmark not defined.**), *wxListCtrl* (p. 864), *wxListItem* (p. 887)

wxListItemAttr::wxListItemAttr

wxListItemAttr()

Default constructor.

wxListItemAttr(const wxColour& colText, const wxColour& colBack, const wxFont& font)

Construct a *wxListItemAttr* with the specified foreground and background colors and font.

wxListItemAttr::GetBackgroundColour

const wxColour& GetBackgroundColour() const

Returns the currently set background color.

wxListItemAttr::GetFont

const wxFont& GetFont() const

Returns the currently set font.

wxListItemAttr::GetTextColour

const wxColour& GetTextColour() const

Returns the currently set text color.

wxListItemAttr::HasBackgroundColour

bool HasBackgroundColour() const

Returns `true` if the currently set background color is valid.

wxListItemAttr::HasFont

bool HasFont() const

Returns `true` if the currently set font is valid.

wxListItemAttr::HasTextColour

bool HasTextColour() const

Returns `true` if the currently set text color is valid.

wxListItemAttr::SetBackgroundColour

void SetBackgroundColour(const wxColour& colour)

Sets a new background color.

wxListItemAttr::SetFont

void SetFont(const wxFont& font)

Sets a new font.

wxListItemAttr::SetTextColour

void SetTextColour(const wxColour& colour)

Sets a new text color.

wxListView

This class currently simply presents a simpler to use interface for the `wxListCtrl` (p. 864) -- it can be thought of as a *façade* for that complicated class. Using it is preferable to using `wxListCtrl` (p. 864) directly whenever possible because in the future some ports

might implement `wxListView` but not the full set of `wxListCtrl` features.

Other than different interface, this class is identical to `wxListCtrl`. In particular, it uses the same events, same window styles and so on.

Derived from

`wxListCtrl` (p. 864)

`wxControl` (p. 218)

`wxWindow` (p. **Error! Bookmark not defined.**)

`wxEvtHandler` (p. 490)

`wxObject` (p. **Error! Bookmark not defined.**)

Include files

<wx/listctrl.h>

wxListView::ClearColumnImage

void ClearColumnImage(int col)

Resets the column image -- after calling this function, no image will be shown.

Parameters

col

the column to clear image for

See also

`SetColumnImage` (p. 895)

wxListView::Focus

void Focus(long index)

Sets focus to the item with the given *index*.

wxListView::GetFirstSelected

long GetFirstSelected() const

Returns the first selected item in a (presumably) multiple selection control. Together with `GetNextSelected` (p. 894) it can be used to iterate over all selected items in the control.

Return value

The first selected item, if any, -1 otherwise.

wxListView::GetFocusedItem

long GetFocusedItem() const

Returns the currently focused item or -1 if none.

See also

IsSelected (p. 894),
Focus (p. 893)

wxListView::GetNextSelected**long GetNextSelected(long item) const**

Used together with *GetFirstSelected* (p. 893) to iterate over all selected items in the control.

Return value

Returns the next selected item or -1 if there are no more of them.

wxListView::IsSelected**bool IsSelected(long index)**

Returns `true` if the item with the given *index* is selected, `false` otherwise.

See also

GetFirstSelected (p. 893),
GetNextSelected (p. 894)

wxListView::Select**void Select(long n, bool on = true)**

Selects or unselects the given item.

Parameters

n

the item to select or unselect

on

if `true` (default), selects the item, otherwise unselects it

See also

SetItemState (p. 882)

wxListView::SetColumnImage

void SetColumnImage(int col, int image)

Sets the column image for the specified column. To use the column images, the control must have a valid image list with at least one image.

Parameters

col

the column to set image for

image

the index of the column image in the controls image list

See also

ClearColumnImage (p. 893),

SetImageList (p. 879)

wxLocale

wxLocale class encapsulates all language-dependent settings and is a generalization of the C locale concept.

In wxWidgets this class manages message catalogs which contain the translations of the strings used to the current language.

wxPerl note: In wxPerl you can't use the '_' function name, so the Wx::Locale module can export the gettext and gettext_noop under any given name.

```
# this imports gettext ( equivalent to Wx::GetTranslation
# and gettext_noop ( a noop )
# into your module
use Wx::Locale qw(:default);

# ....

# use the functions
print gettext( ``Panic!' ' );

button = Wx::Button->new( window, -1, gettext( ``Label' ' ) );
```

If you need to translate a lot of strings, then adding gettext() around each one is a long task (that is why _() was introduced), so just choose a shorter name for gettext:

```
#
use Wx::Locale 'gettext' => 't',
               'gettext_noop' => 'gettext_noop';

# ...

# use the functions
print t( ``Panic!' ' );

# ...
```

Derived from

No base class

See also

Internationalization overview (p. **Error! Bookmark not defined.**),
Internat sample (p. **Error! Bookmark not defined.**)

Include files

<wx/intl.h>

Supported languages

See *list of recognized language constants* (p. **Error! Bookmark not defined.**). These constants may be used to specify the language in *Init* (p. 901) and are returned by *GetSystemLanguage* (p. 901):

wxLocale::wxLocale**wxLocale()**

This is the default constructor and it does nothing to initialize the object: *Init()* (p. 901) must be used to do that.

```
wxLocale(int language, int flags = wxLOCALE_LOAD_DEFAULT |  
wxLOCALE_CONV_ENCODING)
```

See *Init()* (p. 901) for parameters description.

```
wxLocale(const char *szName, const char *szShort = NULL, const char *szLocale =  
NULL, bool bLoadDefault = true, bool bConvertEncoding = false)
```

See *Init()* (p. 901) for parameters description.

The call of this function has several global side effects which you should understand: first of all, the application locale is changed - note that this will affect many of standard C library functions such as *printf()* or *strftime()*. Second, this *wxLocale* object becomes the new current global locale for the application and so all subsequent calls to *wxGetTranslation()* will try to translate the messages using the message catalogs for this locale.

wxLocale::~~wxLocale**~wxLocale()**

The destructor, like the constructor, also has global side effects: the previously set locale is restored and so the changes described in *Init* (p. 901) documentation are rolled back.

wxLocale::AddCatalog

bool AddCatalog(const char *szDomain)

bool AddCatalog(const char *szDomain, wxLanguage msgldLanguage, const char *msgldCharset)

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. 898).

All loaded catalogs will be used for message lookup by *GetString()* (p. 900) for the current locale.

Returns true if catalog was successfully loaded, false otherwise (which might mean that the catalog is not found or that it isn't in the correct format).

The second form of this method takes two additional arguments, *msgldLanguage* and *msgldCharset*.

msgldLanguage specifies the language of "msgid" strings in source code (i.e. arguments to *GetString* (p. 900), *wxGetTranslation* (p. **Error! Bookmark not defined.**) and the *_()* (p. **Error! Bookmark not defined.**) macro). It is used if AddCatalog cannot find any catalog for current language: if the language is same as source code language, then strings from source code are used instead.

msgldCharset lets you specify the charset used for msgids in sources in case they use 8-bit characters (e.g. German or French strings). This argument has no effect in Unicode build, because literals in sources are Unicode strings; you have to use compiler-specific method of setting the right charset when compiling with Unicode.

By default (i.e. when you use the first form), msgid strings are assumed to be in English and written only using 7-bit ASCII characters.

If you have to deal with non-English strings or 8-bit characters in the source code, see the instructions in *Writing non-English applications* (p. **Error! Bookmark not defined.**).

wxLocale::AddCatalogLookupPathPrefix

void AddCatalogLookupPathPrefix(const wxString& prefix)

Add a prefix to the catalog lookup path: the message catalog files will be looked up under prefix/<lang>/LC_MESSAGES, prefix/<lang> and prefix (in this order).

This only applies to subsequent invocations of *AddCatalog()*.

wxLocale::AddLanguage

static void AddLanguage(const wxLanguageInfo& info)

Adds custom, user-defined language to the database of known languages. This database is used in conjunction with the first form of *Init* (p. 901).

wxLanguageInfo is defined as follows:

```
struct WXDLL_EXPORT wxLanguageInfo
{
    int Language;                // wxLanguage id
    wxString CanonicalName;      // Canonical name, e.g. fr_FR
#ifdef __WIN32__
    wxUint32 WinLang, WinSublang; // Win32 language identifiers
                                   // (LANG_XXXX, SUBLANG_XXXX)
#endif
    wxString Description;        // human-readable name of the
    language
};
```

Language should be greater than wxLANGUAGE_USER_DEFINED.

wxPerl note: In wxPerl Wx::LanguageInfo has only one method:

Wx::LanguageInfo->new(language, canonicalName, WinLang, WinSubLang,
Description)

wxLocale::FindLanguageInfo

static wxLanguageInfo * FindLanguageInfo(const wxString& locale)

This function may be used to find the language description structure for the given locale, specified either as a two letter ISO language code (for example, "pt"), a language code followed by the country code ("pt_BR") or a full, human readable, language description ("Portuguese-Brazil").

Returns the information for the given language or NULL if this language is unknown. Note that even if the returned pointer is valid, the caller should *not* delete it.

See also

GetLanguageInfo (p. 899)

wxLocale::GetCanonicalName

wxString GetCanonicalName() const

Returns the canonical form of current locale name. Canonical form is the one that is used on UNIX systems: it is a two- or five-letter string in xx or xx_YY format, where xx is ISO 639 code of language and YY is ISO 3166 code of the country. Examples are "en", "en_GB", "en_US" or "fr_FR".

This form is internally used when looking up message catalogs.

Compare *GetSysName* (p. 900).

wxLocale::GetLanguage

int GetLanguage() const

Returns *wxLanguage* (p. 896) constant of current language. Note that you can call this function only if you used the form of *Init* (p. 901) that takes *wxLanguage* argument.

wxLocale::GetLanguageInfo

static wxLanguageInfo * GetLanguageInfo(int lang) const

Returns a pointer to *wxLanguageInfo* structure containing information about the given language or *NULL* if this language is unknown. Note that even if the returned pointer is valid, the caller should *not* delete it.

See *AddLanguage* (p. 898) for the *wxLanguageInfo* description.

As with *Init* (p. 901), *wxLANGUAGE_DEFAULT* has the special meaning if passed as an argument to this function and in this case the result of *GetSystemLanguage()* (p. 901) is used.

wxLocale::GetLanguageName

static wxString GetLanguageName(int lang) const

Returns English name of the given language or empty string if this language is unknown.

See *GetLanguageInfo* (p. 899) for a remark about special meaning of *wxLANGUAGE_DEFAULT*.

wxLocale::GetLocale

const char* GetLocale() const

Returns the locale name as passed to the constructor or *Init()* (p. 901). This is full, human-readable name, e.g. "English" or "French".

wxLocale::GetName

const wxString& GetName() const

Returns the current short name for the locale (as given to the constructor or the *Init()* function).

wxLocale::GetString

const char* GetString(const char *szOrigString, const char *szDomain = NULL) const

const char* GetString(const char *szOrigString, const char *szOrigString2, size_t n, const char *szDomain = NULL) const

Retrieves the translation for a string in all loaded domains unless the *szDomain* parameter is specified (and then only this catalog/domain is searched).

Returns original string if translation is not available (in this case an error message is generated the first time a string is not found; use *wxLogNull* (p. **Error! Bookmark not defined.**) to suppress it).

The second form is used when retrieving translation of string that has different singular and plural form in English or different plural forms in some other language. It takes two extra arguments: *szOrigString* parameter must contain the singular form of the string to be converted. It is also used as the key for the search in the catalog. The *szOrigString2* parameter is the plural form (in English). The parameter *n* is used to determine the plural form. If no message catalog is found *szOrigString* is returned if 'n == 1', otherwise *szOrigString2*. See GNU gettext manual (http://www.gnu.org/manual/gettext/html_chapter/gettext_10.html#SEC150) for additional information on plural forms handling.

This method is called by the *wxGetTranslation* (p. **Error! Bookmark not defined.**) function and *_()* (p. **Error! Bookmark not defined.**) macro.

Remarks

Domains are searched in the last to first order, i.e. catalogs added later override those added before.

wxLocale::GetHeaderValue

wxString GetHeaderValue(const char *szHeader, const char *szDomain = NULL) const

Returns the header value for header *szHeader*. The search for *szHeader* is case sensitive. If an *szDomain* is passed, this domain is searched. Else all domains will be searched until a header has been found. The return value is the value of the header if found. Else this will be empty.

wxLocale::GetSysName

wxString GetSysName() const

Returns current platform-specific locale name as passed to *setlocale()*.

Compare *GetCanonicalName* (p. 899).

wxLocale::GetSystemEncoding

static wxFontEncoding GetSystemEncoding() const

Tries to detect the user's default font encoding. Returns *wxFontEncoding* (p. 561) value or **wxFONTENCODING_SYSTEM** if it couldn't be determined.

wxLocale::GetSystemEncodingName

static wxString GetSystemEncodingName() const

Tries to detect the name of the user's default font encoding. This string isn't particularly useful for the application as its form is platform-dependent and so you should probably use *GetSystemEncoding* (p. 901) instead.

Returns a user-readable string value or an empty string if it couldn't be determined.

wxLocale::GetSystemLanguage

static int GetSystemLanguage() const

Tries to detect the user's default language setting. Returns *wxLanguage* (p. 896) value or **wxLANGUAGE_UNKNOWN** if the language-guessing algorithm failed.

wxLocale::Init

bool Init(int language = wxLANGUAGE_DEFAULT, int flags = wxLOCALE_LOAD_DEFAULT | wxLOCALE_CONV_ENCODING)

bool Init(const char *szName, const char *szShort = NULL, const char *szLocale = NULL, bool bLoadDefault = true, bool bConvertEncoding = false)

The second form is deprecated, use the first one unless you know what you are doing.

Parameters

language

wxLanguage (p. 896) identifier of the locale. **wxLANGUAGE_DEFAULT** has special meaning -- *wxLocale* will use system's default language (see *GetSystemLanguage* (p. 901)).

flags

Combination of the following:

wxLOCALE_LOAD_DEFAULT Load the message catalog for the given locale containing the translations of standard *wxWidgets* messages automatically.

wxLOCALE_CONV_ENCODING Automatically convert message catalogs to platform's default encoding. Note that it will do only basic conversion between well-known pair like iso8859-1 and windows-1252 or iso8859-2 and windows-1250. See *Writing non-English applications* (p. **Error! Bookmark not defined.**) for detailed description of this behaviour. Note that this flag is meaningless in Unicode build.

szName

The name of the locale. Only used in diagnostic messages.

szShort

The standard 2 letter locale abbreviation; it is used as the directory prefix when looking for the message catalog files.

szLocale

The parameter for the call to `setlocale()`. Note that it is platform-specific.

bLoadDefault

May be set to false to prevent loading of the message catalog for the given locale containing the translations of standard wxWidgets messages. This parameter would be rarely used in normal circumstances.

bConvertEncoding

May be set to true to do automatic conversion of message catalogs to platform's native encoding. Note that it will do only basic conversion between well-known pair like iso8859-1 and windows-1252 or iso8859-2 and windows-1250. See *Writing non-English applications* (p. **Error! Bookmark not defined.**) for detailed description of this behaviour.

The call of this function has several global side effects which you should understand: first of all, the application locale is changed - note that this will affect many of standard C library functions such as `printf()` or `strftime()`. Second, this `wxLocale` object becomes the new current global locale for the application and so all subsequent calls to `wxGetTranslation()` (p. **Error! Bookmark not defined.**) 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. 897)

wxLocale::IsOk

bool IsOk() const

Returns true if the locale could be set successfully.

wxLog

`wxLog` class defines the interface for the *log targets* used by wxWidgets logging

functions as explained in the *wxLog overview* (p. **Error! Bookmark not defined.**). 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. **Error! Bookmark not defined.**) for the descriptions of *wxWidgets* logging facilities.

Derived from

No base class

Include files

<wx/log.h>

Static functions

The functions in this section work with and manipulate the active log target. The *OnLog()* (p. 906) is called by the *wxLogXXX()* functions and invokes the *DoLog()* (p. 907) of the active log target if any. Get/Set methods are used to install/query the current active target and, finally, *DontCreateOnDemand()* (p. 908) 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. 906)

GetActiveTarget (p. 906)

SetActiveTarget (p. 906)

DontCreateOnDemand (p. 908)

Suspend (p. 907)

Resume (p. 907)

Logging functions

There are two functions which must be implemented by any derived class to actually process the log messages: *DoLog* (p. 907) and *DoLogString* (p. 907). The second function receives a string which just has to be output in some way and the easiest way to write a new log target is to override just this function in the derived class. If more control over the output format is needed, then the first function must be overridden which allows to construct custom messages depending on the log level or even do completely different things depending on the message severity (for example, throw away all messages except warnings and errors, show warnings on the screen and forward the error messages to the user's (or programmer's) cell phone - maybe depending on whether the timestamp tells us if it is day or night in the current time zone).

There also functions to support message buffering. Why are they needed? Some of wxLog implementations, most notably the standard wxLogGui class, buffer the messages (for example, to avoid showing the user a zillion of modal message boxes one after another -- which would be really annoying). *Flush()* (p. 908) shows them all and clears the buffer contents. This function doesn't do anything if the buffer is already empty.

Flush (p. 908)

FlushActive (p. 908)

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. **Error! Bookmark not defined.**). 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. 909) and *GetTraceMask* (p. 909) and using *wxLogTrace* (p. **Error! Bookmark not defined.**) which takes an integer mask or by using *AddTraceMask* (p. 906) 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. **Error! Bookmark not defined.**) documentation.

Finally, the `wxLog::DoLog()` function automatically prepends a time stamp to all the messages. The format of the time stamp may be changed: it can be any string with % specifications fully described in the documentation of the standard `strftime()` function. For example, the default format is "[%d/%b/%y %H:%M:%S] " which gives something like "[17/Sep/98 22:10:16] " (without quotes) for the current date. Setting an empty string as the time format disables timestamping of the messages completely.

NB: Timestamping is disabled for Visual C++ users in debug builds by default because otherwise it would be impossible to directly go to the line from which the log message was generated by simply clicking in the debugger window on the corresponding error message. If you wish to enable it, please use `SetTimestamp` (p. 909) explicitly.

`AddTraceMask` (p. 906)
`RemoveTraceMask` (p. 909)
`ClearTraceMasks` (p. 906)
`GetTraceMasks` (p. 906)
`IsAllowedTraceMask` (p. 909)
`SetVerbose` (p. 908)
`GetVerbose` (p. 908)
`SetTimestamp` (p. 909)
`GetTimestamp` (p. 909)
`SetTraceMask` (p. 909)
`GetTraceMask` (p. 909)

wxLog::AddTraceMask

static void AddTraceMask(const wxString& mask)

Add the *mask* to the list of allowed masks for `wxLogTrace` (p. **Error! Bookmark not defined.**).

See also

`RemoveTraceMask` (p. 909) `GetTraceMasks` (p. 906)

wxLog::ClearTraceMasks

static void ClearTraceMasks()

Removes all trace masks previously set with `AddTraceMask` (p. 906).

See also

`RemoveTraceMask` (p. 909)

wxLog::GetTraceMasks

static const wxString & GetTraceMasks()

Returns the currently allowed list of string trace masks.

See also

AddTraceMask (p. 906).

wxLog::OnLog

static void OnLog(wxLogLevel *level*, const char * *message*)

Forwards the message at specified level to the *DoLog()* function of the active log target if there is any, does nothing otherwise.

wxLog::GetActiveTarget

static wxLog * GetActiveTarget()

Returns the pointer to the active log target (may be NULL).

wxLog::SetActiveTarget

static wxLog * SetActiveTarget(wxLog * *logtarget*)

Sets the specified log target as the active one. Returns the pointer to the previous active log target (may be NULL). To suppress logging use a new instance of wxLogNull not NULL. If the active log target is set to NULL a new default log target will be created when logging occurs.

wxLog::Suspend

static void Suspend()

Suspends the logging until *Resume* (p. 907) is called. Note that the latter must be called the same number of times as the former to undo it, i.e. if you call *Suspend()* twice you must call *Resume()* twice as well.

Note that suspending the logging means that the log sink won't be flushed periodically, it doesn't have any effect if the current log target does the logging immediately without waiting for *Flush* (p. 908) to be called (the standard GUI log target only shows the log dialog when it is flushed, so *Suspend()* works as expected with it).

See also

Resume (p. 907),
wxLogNull (p. **Error! Bookmark not defined.**)

wxLog::Resume

static void Resume()

Resumes logging previously suspended by a call to *Suspend* (p. 907). All messages logged in the meanwhile will be flushed soon.

wxLog::DoLog**virtual void DoLog(wxLogLevel level, const wxChar *msg, time_t timestamp)**

Called to process the message of the specified severity. *msg* is the text of the message as specified in the call of *wxLogXXX()* function which generated it and *timestamp* is the moment when the message was generated.

The base class version prepends the timestamp to the message, adds a prefix corresponding to the log level and then calls *DoLogString* (p. 907) with the resulting string.

wxLog::DoLogString**virtual void DoLogString(const wxChar *msg, time_t timestamp)**

Called to log the specified string. The timestamp is already included into the string but still passed to this function.

A simple implementation may just send the string to `stdout` or, better, `stderr`.

wxLog::DontCreateOnDemand**static void DontCreateOnDemand()**

Instructs *wxLog* to not create new log targets on the fly if there is none currently. (Almost) for internal use only: it is supposed to be called by the application shutdown code.

Note that this function also calls *ClearTraceMasks* (p. 906).

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. 908)

wxLog::SetVerbose**static void SetVerbose(bool verbose = true)**

Activates or deactivates verbose mode in which the verbose messages are logged as the normal ones instead of being silently dropped.

wxLog::GetVerbose

static bool GetVerbose()

Returns whether the verbose mode is currently active.

wxLog::SetLogLevel

static void SetLogLevel(wxLogLevel logLevel)

Specifies that log messages with level > logLevel should be ignored and not sent to the active log target.

wxLog::GetLogLevel

static wxLogLevel GetLogLevel()

Returns the current log level limit.

wxLog::SetTimestamp

void SetTimestamp(const char * format)

Sets the timestamp format prepended by the default log targets to all messages. The string may contain any normal characters as well as %prefixed format 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. 904) section for details.

wxLog::GetTraceMask

Returns the current trace mask, see *Customization* (p. 904) section for details.

wxLog::IsAllowedTraceMask

static bool IsAllowedTraceMask(const wxChar *mask)

Returns true if the *mask* is one of allowed masks for *wxLogTrace* (p. **Error! Bookmark not defined.**).

See also: *AddTraceMask* (p. 906), *RemoveTraceMask* (p. 909)

wxLog::RemoveTraceMask

static void RemoveTraceMask(const wxString& mask)

Remove the *mask* from the list of allowed masks for *wxLogTrace* (p. **Error! Bookmark not defined.**).

See also: *AddTraceMask* (p. 906)

wxLogChain

This simple class allows to chain log sinks, that is to install a new sink but keep passing log messages to the old one instead of replacing it completely as *SetActiveTarget* (p. 906) does.

It is especially useful when you want to divert the logs somewhere (for example to a file or a log window) but also keep showing the error messages using the standard dialogs as *wxLogGui* (p. **Error! Bookmark not defined.**) does by default.

Example of usage:

```
wxLogChain *logChain = new wxLogChain(new wxLogStderr);

// all the log messages are sent to stderr and also processed as
// usually
...

// don't delete logChain directly as this would leave a dangling
// pointer as active log target, use SetActiveTarget() instead
delete wxLog::SetActiveTarget(...something else or NULL...);
```

Derived from

wxLog (p. 903)

Include files

<wx/log.h>

wxLogChain::wxLogChain

wxLogChain(wxLog *logger)

Sets the specified *logger* (which may be `NULL`) as the default log target but the log messages are also passed to the previous log target if any.

wxLogChain::~~wxLogChain**~wxLogChain()**

Destroys the previous log target.

wxLogChain::GetOldLog**wxLog * GetOldLog() const**

Returns the pointer to the previously active log target (which may be `NULL`).

wxLogChain::IsPassingMessages**bool IsPassingMessages() const**

Returns `true` if the messages are passed to the previously active log target (default) or `false` if *PassMessages* (p. 911) had been called.

wxLogChain::PassMessages**void PassMessages(bool passMessages)**

By default, the log messages are passed to the previously active log target. Calling this function with `false` parameter disables this behaviour (presumably temporarily, as you shouldn't use `wxLogChain` at all otherwise) and it can be reenabled by calling it again with *passMessages* set to `true`.

wxLogChain::SetLog**void SetLog(wxLog *logger)**

Sets another log target to use (may be `NULL`). The log target specified in the *constructor* (p. 910) or in a previous call to this function is deleted.

This doesn't change the old log target value (the one the messages are forwarded to) which still remains the same as was active when `wxLogChain` object was created.

wxLogGui

This is the default log target for the GUI `wxWidgets` applications. It is passed to *wxLog::SetActiveTarget* (p. 906) at the program startup and is deleted by `wxWidgets` during the program shut down.

Derived from

wxLog (p. 903)

Include files

<wx/log.h>

wxLogGui::wxLogGui

wxLogGui()

Default constructor.

wxLogNull

This class allows to temporarily suspend logging. All calls to the log functions during the life time of an object of this class are just ignored.

In particular, it can be used to suppress the log messages given by wxWidgets itself but it should be noted that it is rarely the best way to cope with this problem as **all** log messages are suppressed, even if they indicate a completely different error than the one the programmer wanted to suppress.

For instance, the example of the overview:

```
wxFile file;

// wxFile.Open() normally complains if file can't be opened, we
// don't want it
{
    wxLogNull logNo;
    if ( !file.Open("bar") )
        ... process error ourselves ...
} // ~wxLogNull called, old log sink restored

wxLogMessage("..."); // ok
```

would be better written as:

```
wxFile file;

// don't try to open file if it doesn't exist, we are prepared
// to deal with
// this ourselves - but all other errors are not expected
if ( wxFile::Exists("bar") )
{
    // gives an error message if the file couldn't be opened
    file.Open("bar");
}
else
{
    ...
}
```

Derived from

wxLog (p. 903)

Include files

<wx/log.h>

wxLogNull::wxLogNull**wxLogNull()**

Suspends logging.

wxLogNull::~~wxLogNull

Resumes logging.

wxLogPassThrough

A special version of *wxLogChain* (p. 909) which uses itself as the new log target. Maybe more clearly, it means that this is a log target which forwards the log messages to the previously installed one in addition to processing them itself.

Unlike *wxLogChain* (p. 909) which is usually used directly as is, this class must be derived from to implement *DoLog* (p. 907) and/or *DoLogString* (p. 907) methods.

Derived from

wxLogChain (p. 909)

Include files

<wx/log.h>

wxLogPassThrough::wxLogPassThrough

Default ctor installs this object as the current active log target.

wxLogStderr

This class can be used to redirect the log messages to a C file stream (not to be confused with C++ streams). It is the default log target for the non-GUI wxWidgets applications which send all the output to *stderr*.

Derived from

wxLog (p. 903)

Include files

<wx/log.h>

See also

wxLogStream (p. 914)

wxLogStderr::wxLogStderr

wxLogStderr(FILE *fp = NULL)

Constructs a log target which sends all the log messages to the given `FILE`. If it is `NULL`, the messages are sent to `stderr`.

wxLogStream

This class can be used to redirect the log messages to a C++ stream.

Please note that this class is only available if `wxWidgets` was compiled with the standard `iostream` library support (`wxUSE_STD_Iostream` must be on).

Derived from

wxLog (p. 903)

Include files

<wx/log.h>

See also

wxLogStderr (p. 913),
wxStreamToTextRedirector (p. **Error! Bookmark not defined.**)

wxLogStream::wxLogStream

wxLogStream(std::ostream *ostr = NULL)

Constructs a log target which sends all the log messages to the given output stream. If it is `NULL`, the messages are sent to `cerr`.

wxLogTextCtrl

Using these target all the log messages can be redirected to a text control. The text control must have been created with `wxTE_MULTILINE` style by the caller previously.

Derived from

wxLog (p. 903)

Include files

<wx/log.h>

See also

wxLogTextCtrl (p. 914),
wxStreamToTextRedirector (p. **Error! Bookmark not defined.**)

wxLogTextCtrl::wxLogTextCtrl

wxLogTextCtrl(wxTextCtrl **textctrl*)

Constructs a log target which sends all the log messages to the given text control. The *textctrl* parameter cannot be `NULL`.

wxLogWindow

This class represents a background log window: to be precise, it collects all log messages in the log frame which it manages but also passes them on to the log target which was active at the moment of its creation. This allows, for example, to show all the log messages in a frame but still continue to process them normally by showing the standard log dialog.

Derived from

wxLogPassThrough (p. 913)

Include files

<wx/log.h>

See also

wxLogTextCtrl (p. 914)

wxLogWindow::wxLogWindow

wxLogWindow(wxFrame **parent*, const wxChar **title*, bool *show* = *true*, bool *passToOld* = *true*)

Creates the log frame window and starts collecting the messages in it.

Parameters

parent

The parent window for the log frame, may be `NULL`

title

The title for the log frame

show

`true` to show the frame initially (default), otherwise `wxLogWindow::Show` (p. 916) must be called later.

passToOld

`true` to process the log messages normally in addition to logging them in the log frame (default), `false` to only log them in the log frame.

wxLogWindow::Show

void Show(bool *show* = `true`)

Shows or hides the frame.

wxLogWindow::GetFrame

wxFrame * GetFrame() const

Returns the associated log frame window. This may be used to position or resize it but use `wxLogWindow::Show` (p. 916) to show or hide it.

wxLogWindow::OnFrameCreate

virtual void OnFrameCreate(wxFrame **frame*)

Called immediately after the log frame creation allowing for any extra initializations.

wxLogWindow::OnFrameClose

virtual bool OnFrameClose(wxFrame **frame*)

Called if the user closes the window interactively, will not be called if it is destroyed for another reason (such as when program exits).

Return `true` from here to allow the frame to close, `false` to prevent this from happening.

See also

`wxLogWindow::OnFrameDelete` (p. 916)

wxLogWindow::OnFrameDelete

virtual void OnFrameDelete(wxFrame **frame*)

Called right before the log frame is going to be deleted: will always be called unlike `OnFrameClose` (p. 916).

wxLongLong

This class represents a signed 64 bit long number. It is implemented using the native 64 bit type where available (machines with 64 bit longs or compilers which have (an analog of) *long long* type) and uses the emulation code in the other cases which ensures that it is the most efficient solution for working with 64 bit integers independently of the architecture.

wxLongLong defines all usual arithmetic operations such as addition, subtraction, bitwise shifts and logical operations as well as multiplication and division (not yet for the machines without native *long long*). It also has operators for implicit construction from and conversion to the native *long long* type if it exists and *long*.

You would usually use this type in exactly the same manner as any other (built-in) arithmetic type. Note that wxLongLong is a signed type, if you want unsigned values use wxULongLong which has exactly the same API as wxLongLong except when explicitly mentioned otherwise.

If a native (i.e. supported directly by the compiler) 64 bit integer type was found to exist, *wxLongLong_t* macro will be defined to correspond to it. Also, in this case only, two additional macros will be defined: *wxLongLongFmtSpec* (p. **Error! Bookmark not defined.**) for printing 64 bit integers using the standard `printf()` function (but see also *ToString()* (p. 919) for a more portable solution) and *wxLL* (p. **Error! Bookmark not defined.**) for defining 64 bit integer compile-time constants.

Derived from

No base class

Include files

<wx/longlong.h>

wxLongLong::wxLongLong

wxLongLong()

Default constructor initializes the object to 0.

wxLongLong::wxLongLong

wxLongLong(wxLongLong_t ll)

Constructor from native long long (only for compilers supporting it).

wxLongLong::wxLongLong

wxLongLong(long hi, unsigned long lo)

Constructor from 2 longs: the high and low part are combined into one wxLongLong.

wxLongLong::operator=**wxLongLong& operator operator=(wxLongLong_t //)**

Assignment operator from native long long (only for compilers supporting it).

wxLongLong::operator=**wxLongLong& operator operator=(wxULongLong_t //)**

Assignment operator from native unsigned long long (only for compilers supporting it).

wxLongLong::operator=**wxLongLong& operator operator=(long //)**

Assignment operator from long.

wxLongLong::operator=**wxLongLong& operator operator=(unsigned long //)**

Assignment operator from unsigned long.

wxLongLong::operator=**wxLongLong& operator operator=(const wxULongLong & //)**

Assignment operator from unsigned long long. The sign bit will be copied too.

wxLongLong::Abs**wxLongLong Abs() const****wxLongLong& Abs()**

Returns an absolute value of wxLongLong - either making a copy (const version) or modifying it in place (the second one). Not in wxULongLong.

wxLongLong::Assign**wxLongLong& Assign(double d)**

This allows to convert a double value to wxLongLong type. Such conversion is not always possible in which case the result will be silently truncated in a platform-dependent way. Not in wxULongLong.

wxLongLong::GetHi**long GetHi() const**

Returns the high 32 bits of 64 bit integer.

wxLongLong::GetLo

unsigned long GetLo() const

Returns the low 32 bits of 64 bit integer.

wxLongLong::GetValue

wxLongLong_t GetValue() const

Convert to native long long (only for compilers supporting it)

wxLongLong::ToDouble

double ToDouble() const

Returns the value as `double`.

wxLongLong::ToLong

long ToLong() const

Truncate `wxLongLong` to `long`. If the conversion loses data (i.e. the `wxLongLong` value is outside the range of built-in `long` type), an assert will be triggered in debug mode.

wxLongLong::ToString

wxString ToString() const

Returns the string representation of a `wxLongLong`.

wxLongLong::operator+

wxLongLong operator+(const wxLongLong& //) const

Adds 2 `wxLongLong`s 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. Not in wxULongLong.

wxLongLong::operator-

wxLongLong operator-(const wxLongLong& l) const

Subtracts 2 wxLongLongs and returns the result.

wxLongLong::operator-=

wxLongLong& operator-(const wxLongLong& l)

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. **Error! Bookmark not defined.**)

Include files

<wx/bitmap.h>

Remarks

A mask may be associated with a *wxBitmap* (p. 84). It is used in *wxDC::Blit* (p. 373) when the source device context is a *wxMemoryDC* (p. **Error! Bookmark not defined.**) with *wxBitmap* selected into it that contains a mask.

See also

wxBitmap (p. 84), *wxDC::Blit* (p. 373), *wxMemoryDC* (p. **Error! Bookmark not defined.**)

wxMask::wxMask

wxMask()

Default constructor.

wxMask(const wxBitmap (p. 84)& bitmap)

Constructs a mask from a monochrome bitmap.

wxPython note: This is the default constructor for *wxMask* in *wxPython*.

wxMask(const wxBitmap (p. 84)& bitmap, const wxColour (p. 168)& colour)

Constructs a mask from a bitmap and a colour that indicates the background.

wxPython note: *wxPython* has an alternate *wxMask* constructor matching this form called *wxMaskColour*.

wxMask(const wxBitmap& bitmap, int index)

Constructs a mask from a bitmap and a palette index that indicates the background. Not yet implemented for GTK.

Parameters

bitmap

A valid bitmap.

colour

A colour specifying the transparency RGB values.

index

Index into a palette, specifying the transparency colour.

wxMask::~~wxMask

~wxMask()

Destroys the *wxMask* object and the underlying bitmap data.

wxMask::Create

bool Create(const wxBitmap& bitmap)

Constructs a mask from a monochrome bitmap.

bool Create(const wxBitmap& *bitmap*, const wxColour& *colour*)

Constructs a mask from a bitmap and a colour that indicates the background.

bool Create(const wxBitmap& *bitmap*, int *index*)

Constructs a mask from a bitmap and a palette index that indicates the background. Not yet implemented for GTK.

Parameters

bitmap

A valid bitmap.

colour

A colour specifying the transparency RGB values.

index

Index into a palette, specifying the transparency colour.

wxMaximizeEvent

An event being sent when the frame is maximized or restored.

Derived from

wxEvent (p. 487)

wxObject (p. **Error! Bookmark not defined.**)

Include files

<wx/event.h>

Event table macros

To process a maximize event, use this event handler macro to direct input to a member function that takes a `wxMaximizeEvent` argument.

EVT_MAXIMIZE(func) Process a `wxEVT_MAXIMIZE` event.

See also

Event handling overview (p. **Error! Bookmark not defined.**),

wxTopLevelWindow::Maximize (p. **Error! Bookmark not defined.**),

wxTopLevelWindow::IsMaximized (p. **Error! Bookmark not defined.**)

wxMaximizeEvent::wxMaximizeEvent

wxMaximizeEvent(int *id* = 0)

Constructor.

wxMBConv

This class is the base class of a hierarchy of classes capable of converting text strings between multibyte (SBCS or DBCS) encodings and Unicode.

In the documentation for this and related classes please notice that *length* of the string refers to the number of characters in the string not counting the terminating `NUL`, if any. While the *size* of the string is the total number of bytes in the string, including any trailing `NUL`s. Thus, length of wide character string `L"foo"` is 3 while its size can be either 8 or 16 depending on whether `wchar_t` is 2 bytes (as under Windows) or 4 (Unix).

Global variables

There are several predefined instances of this class:
wxConvLibc Uses the standard ANSI C `mbstowcs()` and `wcstombs()` functions to perform the conversions; thus depends on the current locale.

wxConvFile

The appropriate conversion for the file names, depends on the system.

Derived from

No base class

Include files

<wx/strconv.h>

See also

wxCSSConv (p. 229), *wxEncodingConverter* (p. 482), *wxMBConv classes overview* (p. **Error! Bookmark not defined.**)

wxMBConv::wxMBConv

wxMBConv()

Constructor.

wxMBConv::MB2WC

virtual size_t MB2WC(wchar_t *out, const char *in, size_t outLen) const

Converts from a string *in* in multibyte encoding to Unicode putting up to *outLen* characters into the buffer *out*.

If *out* is `NULL`, only the length of the string which would result from the conversion is

calculated and returned. Note that this is the length and not size, i.e. the returned value does *not* include the trailing `NUL`. But when the function is called with a non-`NULL` *out* buffer, the *outLen* parameter should be one more to allow to properly `NUL`-terminate the string.

Parameters

out

The output buffer, may be `NULL` if the caller is only interested in the length of the resulting string

in

The `NUL`-terminated input string, cannot be `NULL`

outLen

The length of the output buffer but *including* `NUL`, ignored if *out* is `NULL`

Return value

The length of the converted string *excluding* the trailing `NUL`