Additional Functionality with Qt

Qt 3.0

Contents

QApplication Class Reference

The QApplication class manages the GUI application's control flow and main settings.

`#include <qapplication.h>`

Inherits QObject [p. 123].

Inherited by QXtApplication [Widgets with Qt].

Public Members

- **QApplication** (int & argc, char ** argv)
- **QApplication** (int & argc, char ** argv, bool GUIenabled)
- enum **Type** { Tty, GuiClient, GuiServer }
- **QApplication** (int & argc, char ** argv, Type type)
- **QApplication** (Display * dpy, HANDLE visual = 0, HANDLE colormap = 0)
- **QApplication** (Display * dpy, int argc, char ** argv, HANDLE visual = 0, HANDLE colormap = 0)
- virtual **~QApplication** ()
- int **argc** () const
- char ** **argv** () const
- Type **type** () const
- enum **ColorSpec** { NormalColor = 0, CustomColor = 1, ManyColor = 2 }
- QWidget * **mainWidget** () const
- virtual void **setMainWidget** (QWidget * mainWidget)
- virtual void **polish** (QWidget * w)
- QWidget * **focusWidget** () const
- QWidget * **activeWindow** () const
- int **exec** ()
- void **processEvents** ()
- void **processEvents** (int maxtime)
- void **processOneEvent** ()
- bool **hasPendingEvents** ()
- int **enter_loop** ()
- void **exit_loop** ()
- int **loopLevel** () const
- virtual bool **notify** (QObject * receiver, QEvent * e)
- void **setDefaultCodec** (QTextCodec * codec)
- QTextCodec * **defaultCodec** () const
- void **installTranslator** (QTranslator * mf)
- void **removeTranslator** (QTranslator * mf)
- enum **Encoding** { DefaultCodec, UnicodeUTF8 }

- QString **translate** (const char * context, const char * sourceText, const char * comment = 0, Encoding encoding = DefaultCodec) const
- virtual bool **winEventFilter** (MSG *)
- void **winFocus** (QWidget * widget, bool gotFocus)
- bool **isSessionRestored** () const
- QString **sessionId** () const
- virtual void **commitData** (QSessionManager & sm)
- virtual void **saveState** (QSessionManager & sm)
- void **wakeUpGuiThread** ()
- void **lock** ()
- void **unlock** (bool wakeUpGui = TRUE)
- bool **locked** ()
- bool **tryLock** ()
- void **setEnableRemoteControl** (bool enable, const QUuid appId = QUuid ( ))
- bool **remoteControlEnabled** () const
- QUuid **applicationId** () const

Public Slots

- void **quit** ()
- void **closeAllWindows** ()

Signals

- void **lastWindowClosed** ()
- void **aboutToQuit** ()
- void **guiThreadAwake** ()

Static Public Members

- QStyle & **style** ()
- void **setStyle** (QStyle * style)
- QStyle * **setStyle** (const QString & style)
- int **colorSpec** ()
- void **setColorSpec** (int spec)
- QCursor * **overrideCursor** ()
- void **setOverrideCursor** (const QCursor & cursor, bool replace = FALSE)
- void **restoreOverrideCursor** ()
- bool **hasGlobalMouseTracking** ()
- void **setGlobalMouseTracking** (bool enable)
- QPalette **palette** (const QWidget * w = 0)
- void **setPalette** (const QPalette & palette, bool informWidgets = FALSE, const char * className = 0)
- QFont **font** (const QWidget * w = 0)
- void **setFont** (const QFont & font, bool informWidgets = FALSE, const char * className = 0)
- QFontMetrics **fontMetrics** ()
- QWidgetList * **allWidgets** ()
- QWidgetList * **topLevelWidgets** ()

- QDesktopWidget * **desktop** ()
- QWidget * **activePopupWidget** ()
- QWidget * **activeModalWidget** ()
- QClipboard * **clipboard** ()
- QWidget * **widgetAt** (int x, int y, bool child = FALSE)
- QWidget * **widgetAt** (const QPoint & pos, bool child = FALSE)
- void **exit** (int retcode = 0)
- bool **sendEvent** (QObject * receiver, QEvent * event)
- void **postEvent** (QObject * receiver, QEvent * event)
- void **sendPostedEvents** (QObject * receiver, int event_type)
- void **sendPostedEvents** ()
- void **removePostedEvents** (QObject * receiver)
- bool **startingUp** ()
- bool **closingDown** ()
- void **flushX** ()
- void **flush** ()
- void **syncX** ()
- void **beep** ()
- void setWinStyleHighlightColor (const QColor & c) *(obsolete)*
- const QColor & winStyleHighlightColor () *(obsolete)*
- void **setDesktopSettingsAware** (bool on)
- bool **desktopSettingsAware** ()
- void **setCursorFlashTime** (int msecs)
- int **cursorFlashTime** ()
- void **setDoubleClickInterval** (int ms)
- int **doubleClickInterval** ()
- void **setWheelScrollLines** (int n)
- int **wheelScrollLines** ()
- void **setGlobalStrut** (const QSize & strut)
- QSize **globalStrut** ()
- void **setLibraryPaths** (const QStringList & paths)
- QStringList **libraryPaths** ()
- void **addLibraryPath** (const QString & path)
- void **removeLibraryPath** (const QString & path)
- void **setStartDragTime** (int ms)
- int **startDragTime** ()
- void **setStartDragDistance** (int l)
- int **startDragDistance** ()
- void **setReverseLayout** (bool b)
- bool **reverseLayout** ()
- int **horizontalAlignment** (int align)
- bool **isEffectEnabled** (Qt::UIEffect effect)
- void **setEffectEnabled** (Qt::UIEffect effect, bool enable = TRUE)
- WindowsVersion **winVersion** ()

Related Functions

- void **qAddPostRoutine** (QtCleanUpFunction p)
- const char * **qVersion** ()
- bool **qSysInfo** (int * wordSize, bool * bigEndian)
- void **qDebug** (const char * msg, ...)
- void **qWarning** (const char * msg, ...)
- void **qFatal** (const char * msg, ...)
- void **qSystemWarning** (const char * msg, int code)
- void **Q_ASSERT** (bool test)
- void **Q_CHECK_PTR** (void * p)
- QtMsgHandler **qInstallMsgHandler** (QtMsgHandler h)

Detailed Description

The QApplication class manages the GUI application's control flow and main settings.

It contains the main event loop, where all events from the window system and other sources are processed and dispatched. It also handles the application initialization and finalization, and provides session management. Finally, it handles most system-wide and application-wide settings.

For any GUI application that uses Qt, there is precisely one QApplication object, no matter whether the application has 0, 1, 2 or more windows at any time.

The QApplication object is accessible through the global variable `qApp`. Its main areas of responsibility are:

- It initializes the application with the user's desktop settings such as palette(), font() and doubleClickInterval(). It keeps track of these properties in case the user changes the desktop globally, for example through some kind of control panel.
- It performs event handling, meaning that it receives events from the underlying window system and dispatches them to the relevant widgets. By using sendEvent() and postEvent() you can send your own events to widgets.
- It parses common command line arguments and sets its internal state accordingly. See the constructor documentation below for more details about this.
- It defines the application's look and feel, which is encapsulated in a QStyle object. This can be changed at runtime with setStyle().
- It specifies how the application is to allocate colors. See setColorSpec() for details.
- It specifies the default text encoding (see setDefaultCodec()) and provides localization of strings that are visible to the user via translate().
- It provides some magical objects like the desktop() and the clipboard().
- It knows about the application's windows. You can ask which widget is at a certain position using widgetAt(), get a list of topLevelWidgets() and closeAllWindows(), etc.
- It manages the application's mouse cursor handling, see setOverrideCursor() and setGlobalMouseTracking().
- On the X window system, it provides functions to flush and sync the communication stream, see flushX() and syncX().
- It provides support for sophisticated session management. This makes it possible for applications to terminate gracefully when the user logs out, to cancel a shutdown process if termination isn't possible and even to preserve the entire application state for a future session. See isSessionRestored(), sessionId() and commitData() and saveState() for details.

The Application walk-through example contains a typical complete main() that does the usual things with QApplication.

Since the QApplication object does so much initialization, it **must** be created before any other objects related to the user interface are created.

Since it also deals with common command line arguments, it is usually a good idea to create it *before* any interpretation or modification of `argv` is done in the application itself. (Note also that for X11, setMainWidget() may change the main widget according to the `-geometry` option. To preserve this functionality, you must set your defaults before setMainWidget() and any overrides after.)

Groups of functions:

- System settings: desktopSettingsAware(), setDesktopSettingsAware(), cursorFlashTime(), setCursorFlashTime(), doubleClickInterval(), setDoubleClickInterval(), wheelScrollLines(), setWheelScrollLines(), palette(), setPalette(), font(), setFont(), fontMetrics().
- Event handling: exec(), processEvents(), enter_loop(), exit_loop(), exit(), quit(). sendEvent(), postEvent(), sendPostedEvents(), removePostedEvents(), notify(), macEventFilter(), x11EventFilter(), x11ProcessEvent(), winEventFilter().
- GUI Styles: style(), setStyle(), polish().
- Color usage: colorSpec(), setColorSpec().
- Text handling: setDefaultCodec(), installTranslator(), removeTranslator() translate().
- Widgets: mainWidget(), setMainWidget(), allWidgets(), topLevelWidgets(), desktop(), activePopupWidget(), activeModalWidget(), clipboard(), focusWidget(), activeWindow(), widgetAt().
- Advanced cursor handling: hasGlobalMouseTracking(), setGlobalMouseTracking(), overrideCursor(), setOverrideCursor(), restoreOverrideCursor().
- X Window System synchronization: flushX(), syncX().
- Session management: isSessionRestored(), sessionId(), commitData(), saveState()
- Miscellaneous: closeAllWindows(), startingUp(), closingDown(),

Non-GUI programs: While Qt is not optimized or designed for writing non-GUI programs, it's possible to use some of its classes without creating a QApplication. This can be useful if you wish to share code between a non-GUI server and a GUI client.

See also Main Window and Related Classes.

Member Type Documentation

QApplication::ColorSpec

- `QApplication::NormalColor` - the default color allocation policy
- `QApplication::CustomColor` - the same as NormalColor for X11; allocates colors to a palette on demand under Windows
- `QApplication::ManyColor` - the choice for applications that use thousands of colors

See setColorSpec() for full details.

QApplication::Encoding

This enum type defines the 8-bit encoding of character string arguments to translate():

- `QApplication::DefaultCodec` - the defaultCodec()'s encoding (Latin-1 if none is set)

- `QApplication::UnicodeUTF8` - UTF-8

See also QObject::tr() [p. 137], QObject::trUtf8() [p. 137] and QString::fromUtf8() [Datastructures and String Handling with Qt].

QApplication::Type

- `QApplication::Tty` - a console application
- `QApplication::GuiClient` - a GUI client application
- `QApplication::GuiServer` - a GUI server application

Member Function Documentation

QApplication::QApplication (int & argc, char ** argv)

Initializes the window system and constructs an application object with the command line arguments *argc* and *argv*.

The global `qApp` pointer refers to this application object. Only one application object should be created.

This application object must be constructed before any paint devices (includes widgets, pixmaps, bitmaps etc.)

Note that *argc* and *argv* might be changed. Qt removes command line arguments that it recognizes. The original *argc* and *argv* can be accessed later with `qApp->argc()` and `qApp->argv()`. The documentation for argv() contains a detailed description of how to process command line arguments.

Qt debugging options (not available if Qt was compiled with the QT_NO_DEBUG flag defined):

- -nograb, tells Qt that it must never grab the mouse or the keyboard.
- -dograb (only under X11), running under a debugger can cause an implicit -nograb, use -dograb to override.
- -sync (only under X11), switches to synchronous mode for debugging.

See Debugging Techniques for a more detailed explanation.

All Qt programs automatically support the following command line options:

- -style= *style*, sets the application GUI style. Possible values are `motif`, `windows`, and `platinum`. If you compiled Qt with additional styles or have additional styles as plugins these will be available to the `-style` command line option.
- -session= *session*, restores the application from an earlier session.

The X11 version of Qt also supports some traditional X11 command line options:

- -display *display*, sets the X display (default is $DISPLAY).
- -geometry *geometry*, sets the client geometry of the main widget.
- -fn or `-font` *font*, defines the application font. The font should be specified using an X logical font description.
- -bg or `-background` *color*, sets the default background color and an application palette (light and dark shades are calculated).
- -fg or `-foreground` *color*, sets the default foreground color.
- -btn or `-button` *color*, sets the default button color.
- -name *name*, sets the application name.
- -title *title*, sets the application title (caption).

- -visual `TrueColor`, forces the application to use a TrueColor visual on an 8-bit display.
- -ncols *count*, limits the number of colors allocated in the color cube on an 8-bit display, if the application is using the QApplication::ManyColor color specification. If *count* is 216 then a 6x6x6 color cube is used (ie. 6 levels of red, 6 of green, and 6 of blue); for other values, a cube approximately proportional to a 2x3x1 cube is used.
- -cmap, causes the application to install a private color map on an 8-bit display.

See also argc() [p. 12] and argv() [p. 13].

QApplication::QApplication (int & argc, char ** argv, bool GUIenabled)

Constructs an application object with the command line arguments *argc* and *argv*. If *GUIenabled* is TRUE, a GUI application is constructed, otherwise a non-GUI (console) application is created.

Set *GUIenabled* to FALSE for programs without a graphical user interface that should be able to run without a window system.

On X11, the window system is initialized if *GUIenabled* is TRUE. If *GUIenabled* is FALSE, the application does not connect to the X-server. On Windows and Macintosh, currently the window system is always initialized, regardless of the value of GUIenabled. This may change in future versions of Qt.

For threaded configurations (i.e. when Qt has been built as a threaded library), the application global mutex will be locked in the constructor and unlocked when entering the event loop with exec(). You must unlock the mutex explicitly if you don't call exec(), otherwise you might get warnings on application exit.

The following example shows how to create an application that uses a graphical interface when available.

```
  int main( int argc, char **argv )
  {
#ifdef Q_WS_X11
    bool useGUI = getenv( "DISPLAY" ) != 0;
#else
    bool useGUI = TRUE;
#endif
    QApplication app(argc, argv, useGUI);

    if ( useGUI ) {
       //start GUI version
       ...
    } else {
       //start non-GUI version
       ...
    }
    return app.exec();
  }
```

QApplication::QApplication (int & argc, char ** argv, Type type)

Constructs an application object with the command line arguments *argc* and *argv*.

For Qt/Embedded, passing QApplication::GuiServer for *type* makes this application the server (equivalent to running with the -qws option).

QApplication::QApplication (Display * dpy, HANDLE visual = 0, HANDLE colormap = 0)

Create an application, given an already open display *dpy*. If *visual* and *colormap* are non-zero, the application will use those as the default Visual and Colormap contexts.

This is available only on X11.

QApplication::QApplication (Display * dpy, int argc, char ** argv, HANDLE visual = 0, HANDLE colormap = 0)

Create an application, given an already open display *dpy* and using *argc* command line arguments in *argv*. If *visual* and *colormap* are non-zero, the application will use those as the default Visual and Colormap contexts.

This is available only on X11.

QApplication::~QApplication () [virtual]

Cleans up any window system resources that were allocated by this application. Sets the global variable `qApp` to null.

void QApplication::aboutToQuit () [signal]

This signal is emitted when the application is about to quit the main event loop. This may happen either after a call to quit() from inside the application or when the users shuts down the entire desktop session.

The signal is particularly useful if your application has to do some last-second cleanups. Note that no user interaction is possible in this state.

See also quit() [p. 21].

QWidget * QApplication::activeModalWidget () [static]

Returns the active modal widget.

A modal widget is a special top level widget which is a subclass of QDialog that specifies the modal parameter of the constructor as TRUE. A modal widget must be closed before the user can continue with other parts of the program.

Modal widgets are organized in a stack. This function returns the active modal widget at the top of the stack.

See also activePopupWidget() [p. 11] and topLevelWidgets() [p. 29].

QWidget * QApplication::activePopupWidget () [static]

Returns the active popup widget.

A popup widget is a special top level widget that sets the WType_Popup widget flag, e.g. the QPopupMenu widget. When the application opens a popup widget, all events are sent to the popup. Normal widgets and modal widgets cannot be accessed before the popup widget is closed.

Only other popup widgets may be opened when a popup widget is shown. The popup widgets are organized in a stack. This function returns the active popup widget at the top of the stack.

See also activeModalWidget() [p. 11] and topLevelWidgets() [p. 29].

QWidget * QApplication::activeWindow () const

Returns the application top-level window that has the keyboard input focus, or null if no application window has the focus. Note that there might be an activeWindow even if there is no focusWidget(), for example if no widget in that window accepts key events.

See also QWidget::setFocus() [Widgets with Qt], QWidget::focus [Widgets with Qt] and focusWidget() [p. 16].

Example: network/mail/smtp.cpp.

void QApplication::addLibraryPath (const QString & path) [static]

Append *path* to the end of the library path list. If *path* is empty or already in the path list, the path list is not changed.

See also removeLibraryPath() [p. 21], libraryPaths() [p. 18] and setLibraryPaths() [p. 26].

QWidgetList * QApplication::allWidgets () [static]

Returns a list of all the widgets in the application.

The list is created using `new` and must be deleted by the caller.

The list is empty (QPtrList::isEmpty()) if there are no widgets.

Note that some of the widgets may be hidden.

Example that updates all widgets:

```
QWidgetList  *list = QApplication::allWidgets();
QWidgetListIt it( *list );         // iterate over the widgets
QWidget * w;
while ( (w=it.current()) != 0 ) {  // for each widget...
    ++it;
    w->update();
}
delete list;                      // delete the list, not the widgets
```

The QWidgetList class is defined in the qwidgetlist.h header file.

Warning: Delete the list as soon as you have finished using it. The widgets in the list may be deleted by someone else at any time.

See also topLevelWidgets() [p. 29], QWidget::visible [Widgets with Qt] and QPtrList::isEmpty() [Datastructures and String Handling with Qt].

QUuid QApplication::applicationId () const

Returns the application id that was set with setEnableRemoteControl.

int QApplication::argc () const

Returns the number of command line arguments.

The documentation for argv() contains a detailed description of how to process command line arguments.

See also argv() [p. 13] and QApplication::QApplication() [p. 9].

Example: scribble/scribble.cpp.

char ** QApplication::argv () const

Returns the command line argument vector.

`argv()[0]` is the program name, `argv()[1]` is the first argument and `argv()[argc()-1]` is the last argument.

A QApplication object is constructed by passing *argc* and *argv* from the `main()` function. Some of the arguments may be recognized as Qt options and removed from the argument vector. For example, the X11 version of Qt knows about `-display`, `-font` and a few more options.

Example:

```
// showargs.cpp - displays program arguments in a list box

#include <qapplication.h>
#include <qlistbox.h>

int main( int argc, char **argv )
{
    QApplication a( argc, argv );
    QListBox b;
    a.setMainWidget( &b );
    for ( int i=0; i<a.argc(); i++ )        // a.argc() == argc
        b.insertItem( a.argv()[i] );        // a.argv()[i] == argv[i]
    b.show();
    return a.exec();
}
```

If you run `showargs -display unix:0 -font 9x15bold hello world` under X11, the list box contains the three strings "showargs", "hello" and "world".

See also argc() [p. 12] and QApplication::QApplication() [p. 9].

Example: scribble/scribble.cpp.

void QApplication::beep () [static]

Sounds the bell, using the default volume and sound.

QClipboard * QApplication::clipboard () [static]

Returns a pointer to the application global clipboard.

Example: showimg/showimg.cpp.

void QApplication::closeAllWindows () [slot]

Closes all top-level windows.

This function is particularly useful for applications with many top-level windows. It could for example be connected to a "Quit" entry in the file menu as shown in the following code example:

```
// the "Quit" menu entry should try to close all windows
QPopupMenu* file = new QPopupMenu( this );
file->insertItem( "&Quit", qApp, SLOT(closeAllWindows()), CTRL+Key_Q );

// when the last window is closed, the application should quit
```

```
    connect( qApp, SIGNAL( lastWindowClosed() ), qApp, SLOT( quit() ) );
```

The windows are closed in random order, until one window does not accept the close event.

See also QWidget::close() [Widgets with Qt], QWidget::closeEvent() [Widgets with Qt], lastWindowClosed() [p. 18], quit() [p. 21], topLevelWidgets() [p. 29] and QWidget::isTopLevel [Widgets with Qt].

Examples: action/application.cpp, application/application.cpp, helpviewer/helpwindow.cpp, mdi/application.cpp and qwerty/qwerty.cpp.

bool QApplication::closingDown () [static]

Returns TRUE if the application objects are being destroyed.

See also startingUp() [p. 29].

int QApplication::colorSpec () [static]

Returns the color specification.

See also QApplication::setColorSpec() [p. 23].

Example: showimg/showimg.cpp.

void QApplication::commitData (QSessionManager & sm) [virtual]

This function deals with session management. It is invoked when the QSessionManager wants the application to commit all its data.

Usually this means saving all open files, after getting permission from the user. Furthermore you may want to provide a means by which the user can cancel the shutdown.

Note that you should not exit the application within this function. Instead, the session manager may or may not do this afterwards, depending on the context.

Important Within this function, no user interaction is possible, *unless* you ask the session manager *sm* for explicit permission. See QSessionManager::allowsInteraction() and QSessionManager::allowsErrorInteraction() for details and example usage.

The default implementation requests interaction and sends a close event to all visible top level widgets. If any event was rejected, the shutdown is cancelled.

See also isSessionRestored() [p. 18], sessionId() [p. 23] and saveState() [p. 22].

int QApplication::cursorFlashTime () [static]

Returns the text cursor's flash time in milliseconds. The flash time is the time required to display, invert and restore the caret display.

The default value on X11 is 1000 milliseconds. On Windows, the control panel value is used.

Widgets should not cache this value since it may vary any time the user changes the global desktop settings.

See also setCursorFlashTime() [p. 24].

QTextCodec * QApplication::defaultCodec () const

Returns the default codec (see setDefaultCodec()). Returns 0 by default (no codec).

QDesktopWidget * QApplication::desktop () [static]

Returns the desktop widget (also called the root window).

The desktop widget is useful for obtaining the size of the screen. It may also be possible to draw on the desktop. We recommend against assuming that it's possible to draw on the desktop, as it works on some operating systems and not on others.

```
    QDesktopWidget *d = QApplication::desktop();
    int w=d->width();                   // returns desktop width
    int h=d->height();                  // returns desktop height
```

Examples: desktop/desktop.cpp, helpviewer/main.cpp, i18n/main.cpp, qmag/qmag.cpp, qwerty/main.cpp, qwerty/qwerty.cpp and scribble/main.cpp.

bool QApplication::desktopSettingsAware () [static]

Returns the value set by setDesktopSettingsAware(), by default TRUE.

See also setDesktopSettingsAware() [p. 24].

int QApplication::doubleClickInterval () [static]

Returns the maximum duration for a double click.

The default value on X11 is 400 milliseconds. On Windows, the control panel value is used.

See also setDoubleClickInterval() [p. 25].

int QApplication::enter_loop ()

This function enters the main event loop (recursively). Do not call it unless you really know what you are doing.

See also exit_loop() and loopLevel() [p. 18].

int QApplication::exec ()

Enters the main event loop and waits until exit() is called or the main widget is destroyed, and returns the value that was set to exit() (which is 0 if exit() is called via quit()).

It is necessary to call this function to start event handling. The main event loop receives events from the window system and dispatches these to the application widgets.

Generally speaking, no user interaction can take place before calling exec(). As a special case, modal widgets like QMessageBox can be used before calling exec(), because modal widgets call exec() to start a local event loop.

To make your application perform idle processing, i.e. executing a special function whenever there are no pending events, use a QTimer with 0 timeout. More advanced idle processing schemes can be achieved using processEvents().

See also quit() [p. 21], exit() [p. 16], processEvents() [p. 20] and setMainWidget() [p. 26].

Examples: action/actiongroup/main.cpp, biff/main.cpp, fonts/simple-qfont-demo/simple-qfont-demo.cpp, life/main.cpp, t1/main.cpp, t4/main.cpp and xml/outliner/main.cpp.

void QApplication::exit (int retcode = 0) [static]

Tells the application to exit with a return code.

After this function has been called, the application leaves the main event loop and returns from the call to exec(). The exec() function returns *retcode*.

By convention, *retcode* 0 means success. Any non-zero value indicates an error.

Note that unlike the C library function of the same name, this function *does* return to the caller - it is event processing that stops.

See also quit() [p. 21] and exec() [p. 15].

Example: picture/picture.cpp.

void QApplication::exit_loop ()

This function exits from a recursive call to the main event loop. Do not call it unless you are an expert.

See also enter_loop() and loopLevel() [p. 18].

void QApplication::flush () [static]

Flushes the window system specific event queues.

If you are doing graphical changes inside a loop that does not return to the event loop on asynchronous window systems like X11 or double buffered window systems like MacOS X, and you want to visualize these changes immediately (e.g. Splash Screens), call this function.

See also flushX() [p. 16], sendPostedEvents() [p. 22] and QPainter::flush() [Graphics with Qt].

void QApplication::flushX () [static]

Flushes the X event queue in the X11 implementation. This normally returns almost immediately. Does nothing on other platforms.

See also syncX() [p. 29].

Example: xform/xform.cpp.

QWidget * QApplication::focusWidget () const

Returns the application widget that has the keyboard input focus, or null if no widget in this application has the focus.

See also QWidget::setFocus() [Widgets with Qt], QWidget::focus [Widgets with Qt] and activeWindow() [p. 12].

QFont QApplication::font (const QWidget * w = 0) [static]

Returns the default font for the widget. Basically this function uses w->className() to find the font.

If *w* is 0 the default application font is returned.

See also setFont() [p. 25], fontMetrics() [p. 17] and QWidget::font [Widgets with Qt].

Examples: qfd/fontdisplayer.cpp, themes/metal.cpp and themes/themes.cpp.

QFontMetrics QApplication::fontMetrics () [static]

Returns display (screen) font metrics for the application font.

See also font() [p. 16], setFont() [p. 25], QWidget::fontMetrics() [Widgets with Qt] and QPainter::fontMetrics() [Graphics with Qt].

QSize QApplication::globalStrut () [static]

Returns the application's global strut.

The strut is a size object whose dimensions are the minimum that any GUI element that the user can interact with should have. For example no button should be resized to be smaller than the global strut size.

See also setGlobalStrut() [p. 26].

void QApplication::guiThreadAwake () [signal]

This signal is emitted when the GUI thread is about to process a cycle of the event loop.

See also wakeUpGuiThread() [p. 30].

bool QApplication::hasGlobalMouseTracking () [static]

Returns TRUE if global mouse tracking is enabled, otherwise FALSE.

See also setGlobalMouseTracking() [p. 25].

bool QApplication::hasPendingEvents ()

This function returns TRUE if there are pending events, and returns FALSE if there are not. Pending events can be either from the window system or posted events using QApplication::postEvent().

int QApplication::horizontalAlignment (int align) [static]

Strips out vertical alignment flags and transforms an alignment *align* of AlignAuto into AlignLeft or AlignRight according to the language used. The other horizontal alignment flags are left untouched.

void QApplication::installTranslator (QTranslator * mf)

Adds the message file *mf* to the list of message files to be used for translations.

Multiple message files can be installed. Translations are searched for in the last installed message file, then the one from last, and so on, back to the first installed message file. The search stops as soon as a matching translation is found.

See also removeTranslator() [p. 21], translate() [p. 30] and QTranslator::load() [Accessibility and Internationalization with Qt].

Example: i18n/main.cpp.

bool QApplication::isEffectEnabled (Qt::UIEffect effect) [static]

Returns TRUE if *effect* is enabled, otherwise FALSE.

By default, Qt will try to use the desktop settings, and setDesktopSettingsAware() must be called to prevent this.

See also setEffectEnabled() [p. 25] and Qt::UIEffect [p. 192].

bool QApplication::isSessionRestored () const

Returns TRUE if the application has been restored from an earlier session.

See also sessionId() [p. 23], commitData() [p. 14] and saveState() [p. 22].

void QApplication::lastWindowClosed () [signal]

This signal is emitted when the user has closed the last top level window.

The signal is very useful when your application has many top level widgets but no main widget. You can then connect it to the quit() slot.

For convenience, this signal is *not* emitted for transient top level widgets such as popup menus and dialogs.

See also mainWidget() [p. 19], topLevelWidgets() [p. 29], QWidget::isTopLevel [Widgets with Qt] and QWidget::close() [Widgets with Qt].

Examples: action/main.cpp, addressbook/main.cpp, application/main.cpp, helpviewer/main.cpp, mdi/main.cpp, qwerty/main.cpp and showimg/main.cpp.

QStringList QApplication::libraryPaths () [static]

Returns a list of paths that the application will search when dynamically loading libraries.

See also setLibraryPaths() [p. 26], addLibraryPath() [p. 12], removeLibraryPath() [p. 21] and QLibrary [Plugins with Qt].

void QApplication::lock ()

Lock the Qt library mutex. If another thread has already locked the mutex, the calling thread will block until the other thread has unlocked the mutex.

See also unlock() [p. 30] and locked() [p. 18].

bool QApplication::locked ()

Returns TRUE if the Qt library mutex is locked by a different thread, otherwise returns FALSE.

Warning: Due to differing implementations of recursive mutexes on supported platforms, calling this function from the same thread that previous locked the mutex will give undefined results.

See also lock() [p. 18] and unlock() [p. 30].

int QApplication::loopLevel () const

Returns the current loop level

See also enter_loop() and exit_loop().

QWidget * QApplication::mainWidget () const

Returns the main application widget, or a null pointer if there is not a defined main widget.

See also setMainWidget() [p. 26].

bool QApplication::notify (QObject * receiver, QEvent * e) [virtual]

Sends event *e* to *receiver*: *receiver*->event(*e*). Returns the value that is returned from the receiver's event handler.

For certain types of events (e.g. mouse and key events), the event will be propagated to the receiver's parent and so on up to the top-level object if the receiver is not interested in the event (i.e., it returns FALSE).

Reimplementing this virtual function is one of five ways to process an event:

1. Reimplementing this function. Very powerful, you get complete control, but of course only one subclass can be qApp.
2. Installing an event filter on qApp. Such an event filter gets to process all events for all widgets, so it's just as powerful as reimplementing notify(), and in this way it's possible to have more than one application-global event filter. Global event filters get to see even mouse events for disabled widgets, and if global mouse tracking is enabled, mouse move events for all widgets.
3. Reimplementing QObject::event() (as QWidget does). If you do this you get tab key presses, and you get to see the events before any widget-specific event filters.
4. Installing an event filter on the object. Such an even filter gets all the events except Tab and Shift-Tab key presses.
5. Finally, reimplementing paintEvent(), mousePressEvent() and so on. This is the normal, easiest and least powerful way.

See also QObject::event() [p. 131] and installEventFilter() [p. 132].

QCursor * QApplication::overrideCursor () [static]

Returns the active application override cursor.

This function returns 0 if no application cursor has been defined (i.e. the internal cursor stack is empty).

See also setOverrideCursor() [p. 26] and restoreOverrideCursor() [p. 21].

QPalette QApplication::palette (const QWidget * w = 0) [static]

Returns a pointer to the default application palette. There is always an application palette, i.e. the returned pointer is guaranteed to be non-null.

If a widget is passed at *w*, the default palette for the widget's class is returned. This may or may not be the application palette. In most cases there isn't a special palette for certain types of widgets, but one notable exception is the popup menu under Windows, if the user has defined a special background color for menus in the display settings.

See also setPalette() [p. 27] and QWidget::palette [Widgets with Qt].

Examples: desktop/desktop.cpp, themes/metal.cpp and themes/wood.cpp.

void QApplication::polish (QWidget * w) [virtual]

Initialization of the appearance of the widget *w before* it is first shown.

Usually widgets call this automatically when they are polished. It may be used to do some style-based central customization of widgets.

Note that you are not limited to the public functions of QWidget. Instead, based on meta information like QObject::className() you are able to customize any kind of widget.

See also QStyle::polish() [Events, Actions, Layouts and Styles with Qt], QWidget::polish() [Widgets with Qt], setPalette() [p. 27] and setFont() [p. 25].

void QApplication::postEvent (QObject * receiver, QEvent * event) [static]

Adds the event *event* with the object *receiver* as the reciever of the event to an event queue and returns immediately.

The event must be allocated on the heap since the post event queue will take ownership of the event and delete it once it has been posted.

When control returns to the main event loop, all events that are stored in the queue will be sent using the notify() function.

See also sendEvent() [p. 22], QThread::postEvent() [Threading with Qt] and notify() [p. 19].

void QApplication::processEvents ()

Processes pending events, for 3 seconds or until there are no more events to process, whichever is shorter.

You can call this function occasionally when your program is busy performing a long operation (e.g. copying a file).

See also exec() [p. 15] and QTimer [p. 205].

Example: fileiconview/qfileiconview.cpp.

void QApplication::processEvents (int maxtime)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Processes pending events for *maxtime* milliseconds or until there are no more events to process, whichever is shorter.

You can call this function occasionally when you program is busy doing a long operation (e.g. copying a file).

See also exec() [p. 15] and QTimer [p. 205].

void QApplication::processOneEvent ()

Waits for an event to occur, processes it, then returns.

This function is useful for adapting Qt to situations where the event processing must be grafted into existing program loops.

Using this function in new applications may be an indication of design problems.

See also processEvents() [p. 20], exec() [p. 15] and QTimer [p. 205].

void QApplication::quit () [slot]

Tells the application to exit with return code 0 (success). Equivalent to calling QApplication::exit(0).

It's common to connect the lastWindowClosed() signal to quit(), and you also often connect e.g. QButton::clicked() or signals in QAction, QPopupMenu or QMenuBar to it.

Example:

```
    QPushButton *quitButton = new QPushButton( "Quit" );
    connect( quitButton, SIGNAL(clicked()), qApp, SLOT(quit()) );
```

See also exit() [p. 16], aboutToQuit() [p. 11], lastWindowClosed() [p. 18] and QAction [Events, Actions, Layouts and Styles with Qt].

Examples: addressbook/main.cpp, helpviewer/main.cpp, qwerty/main.cpp, showimg/main.cpp, t2/main.cpp, t4/main.cpp and t6/main.cpp.

bool QApplication::remoteControlEnabled () const

Returns TRUE if remote control access is enabled for the application; otherwise returns FALSE.

void QApplication::removeLibraryPath (const QString & path) [static]

Removes *path* from the library path list. If *path* is empty or not in the path list, the list is not changed.

See also addLibraryPath() [p. 12], libraryPaths() [p. 18] and setLibraryPaths() [p. 26].

void QApplication::removePostedEvents (QObject * receiver) [static]

Removes all events posted using postEvent() for *receiver*.

The events are *not* dispatched, instead they are removed from the queue. You should never need to call this function. If you do call it, be aware that killing events may cause *receiver* to break one or more invariants.

void QApplication::removeTranslator (QTranslator * mf)

Removes the message file *mf* from the list of message files used by this application. (It does not delete the message file from the file system.)

See also installTranslator() [p. 17], translate() [p. 30] and QObject::tr() [p. 137].

Example: i18n/main.cpp.

void QApplication::restoreOverrideCursor () [static]

Undoes the last setOverrideCursor().

If setOverrideCursor() has been called twice, calling restoreOverrideCursor() will activate the first cursor set. Calling this function a second time restores the original widgets cursors.

See also setOverrideCursor() [p. 26] and overrideCursor() [p. 19].

Example: showimg/showimg.cpp.

bool QApplication::reverseLayout () [static]

Returns TRUE if all dialogs and widgets will be laid out in a mirrored fashion.

See also setReverseLayout() [p. 27].

void QApplication::saveState (QSessionManager & sm) [virtual]

This function deals with session management. It is invoked when the session manager wants the application to preserve its state for a future session.

For a text editor this would mean creating a temporary file that includes the current contents of the edit buffers, the location of the cursor and other aspects of the current editing session.

Note that you should never exit the application within this function. Instead, the session manager may or may not do this afterwards, depending on the context. Futhermore, most session managers will very likely request a saved state immediately after the application has been started. This permits the session manager to learn about the application's restart policy.

Important Within this function, no user interaction is possible, *unless* you ask the session manager *sm* for explicit permission. See QSessionManager::allowsInteraction() and QSessionManager::allowsErrorInteraction() for details.

See also isSessionRestored() [p. 18], sessionId() [p. 23] and commitData() [p. 14].

bool QApplication::sendEvent (QObject * receiver, QEvent * event) [static]

Sends event *event* directly to receiver *receiver*, using the notify() function. Returns the value that was returned from the event handler.

The event is *not* deleted when the event has been sent. The normal approach is to create the event on the stack, e.g.

```
    QMouseEvent me( QEvent::MouseButtonPress, pos, 0, 0 );
    QApplication::sendEvent( mainWindow, &me );
```

If you create the event on the heap you must delete it.

See also postEvent() [p. 20] and notify() [p. 19].

Example: popup/popup.cpp.

void QApplication::sendPostedEvents (QObject * receiver, int event_type) [static]

Immediately dispatches all events which have been previously queued with QApplication::postEvent() and which are for the object *receiver* and have the event type *event_type*.

Note that events from the window system are *not* dispatched by this function, but by processEvents().

void QApplication::sendPostedEvents () [static]

Dispatches all posted events, i.e. empties the event queue. This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

QString QApplication::sessionId () const

Returns the identifier of the current session.

If the application has been restored from an earlier session, this identifier is the same as it was in that previous session.

The session identifier is guaranteed to be unique both for different applications and for different instances of the same application.

See also isSessionRestored() [p. 18], commitData() [p. 14] and saveState() [p. 22].

void QApplication::setColorSpec (int spec) [static]

Sets the color specification for the application to *spec*.

The color specification controls how your application allocates colors when run on a display with a limited amount of colors, i.e. 8 bit / 256 color displays.

The color specification must be set before you create the QApplication object.

The choices are:

- QApplication::NormalColor. This is the default color allocation strategy. Use this choice if your application uses buttons, menus, texts and pixmaps with few colors. With this choice, the application uses system global colors. This works fine for most applications under X11, but on Windows machines it may cause dithering of non-standard colors.
- QApplication::CustomColor. Use this choice if your application needs a small number of custom colors. On X11, this choice is the same as NormalColor. On Windows, Qt creates a Windows palette, and allocates colors to it on demand.
- QApplication::ManyColor. Use this choice if your application is very color hungry (e.g. it wants thousands of colors). Under X11 the effect is:
  - For 256-color displays which have at best a 256 color true color visual, the default visual is used, and colors are allocated from a color cube. The color cube is the 6x6x6 (216 color) "Web palette", but the number of colors can be changed by the *-ncols* option. The user can force the application to use the true color visual by the -visual option.
  - For 256-color displays which have a true color visual with more than 256 colors, use that visual. Silicon Graphics X servers have this feature, for example. They provide an 8 bit visual by default but can deliver true color when asked.

  On Windows, Qt creates a Windows palette, and fills it with a color cube.

Be aware that the CustomColor and ManyColor choices may lead to colormap flashing: The foreground application gets (most) of the available colors, while the background windows will look less attractive.

Example:

```
int main( int argc, char **argv )
{
    QApplication::setColorSpec( QApplication::ManyColor );
    QApplication a( argc, argv );
    ...
}
```

QColor provides more functionality for controlling color allocation and freeing up certain colors. See QColor::enterAllocContext() for more information.

To see what mode you end up with, you can call QColor::numBitPlanes() once the QApplication object exists. A value greater than 8 (typically 16, 24 or 32) means true color.

The color cube used by Qt has all those colors with red, green, and blue components of either 0x00, 0x33, 0x66, 0x99, 0xCC, or 0xFF.

See also colorSpec() [p. 14], QColor::numBitPlanes() [Graphics with Qt] and QColor::enterAllocContext() [Graphics with Qt].

Examples: helpviewer/main.cpp, showimg/main.cpp, t9/main.cpp, tetrix/tetrix.cpp and themes/main.cpp.

void QApplication::setCursorFlashTime (int msecs) [static]

Sets the text cursor's flash time to *msecs* milliseconds. The flash time is the time required to display, invert and restore the caret display: A full flash cycle. Usually, the text cursor is displayed for *msecs/2* milliseconds, then hidden for *msecs/2* milliseconds, but this may vary.

Note that on Microsoft Windows, calling this function sets the cursor flash time for all windows.

See also cursorFlashTime() [p. 14].

void QApplication::setDefaultCodec (QTextCodec * codec)

Sets the default codec of the application to *codec*.

If the literal quoted text in the program is not in the Latin1 encoding, this function can be used to set the appropriate encoding. For example, software developed by Korean programmers might use eucKR for all the text in the program, in which case the main() function might look like this:

```
    int main(int argc, char** argv)
    {
        QApplication app(argc, argv);
        ... install any additional codecs ...
        app.setDefaultCodec( QTextCodec::codecForName("eucKR") );
        ...
    }
```

Note that this is *not* the way to select the encoding that the *user* has chosen. For example, to convert an application containing literal English strings to Korean, all that is needed is for the English strings to be passed through tr() and for translation files to be loaded. For details of internationalization, see the Qt internationalization documentation.

Note also that some Qt built-in classes call tr() with various strings. These strings are in English, so for a full translation, a codec would be required for these strings.

void QApplication::setDesktopSettingsAware (bool on) [static]

By default, Qt will try to use the current standard colors, fonts etc. from the underlying window system's desktop settings, and use them for all relevant widgets. This behavior can be switched off by calling this function with *on* set to FALSE.

This static function must be called before creating the QApplication object, like this:

```
  int main( int argc, char** argv ) {
    QApplication::setDesktopSettingsAware( FALSE ); // I know better than the user
    QApplication myApp( argc, argv ); // give me default fonts & colors
    ...
  }
```

See also desktopSettingsAware() [p. 15].

void QApplication::setDoubleClickInterval (int ms) [static]

Sets the time limit that distinguishes a double click from two consecutive mouse clicks to *ms* milliseconds.

Note that on Microsoft Windows, calling this function sets the double click interval for all windows.

See also doubleClickInterval() [p. 15].

void QApplication::setEffectEnabled (Qt::UIEffect effect, bool enable = TRUE) [static]

Enables the UI effect *effect* if *enable* is TRUE, otherwise the effect will not be used.

See also isEffectEnabled() [p. 18], Qt::UIEffect [p. 192] and setDesktopSettingsAware() [p. 24].

void QApplication::setEnableRemoteControl (bool enable, const QUuid appId = QUuid ())

Enables remote access to the application if *enable* is set to TRUE. You can use the *appId* to give your application a unique identification that can be used by the remote control. If *enable* is set to FALSE a currently remote access is terminated. Remote control access is disabled by default. You can call this function any time after having created the application.

void QApplication::setFont (const QFont & font, bool informWidgets = FALSE, const char * className = 0) [static]

Changes the default application font to *font*. If *informWidgets* is TRUE, then existing widgets are informed about the change and may adjust themselves to the new application setting. Otherwise the change only affects newly created widgets. If *className* is passed, the change applies only to classes that inherit *className* (as reported by QObject::inherits()).

On application start-up, the default font depends on the window system. It can vary both with window system version and with locale. This function lets you override the default font; but overriding may be a bad idea, for example some locales need extra-large fonts to support their special characters.

See also font() [p. 16], fontMetrics() [p. 17] and QWidget::font [Widgets with Qt].

Examples: desktop/desktop.cpp, i18n/main.cpp, qfd/qfd.cpp, showimg/main.cpp, themes/metal.cpp and themes/themes.cpp.

void QApplication::setGlobalMouseTracking (bool enable) [static]

Enables global mouse tracking if *enable* is TRUE or disables it if *enable* is FALSE.

Enabling global mouse tracking makes it possible for widget event filters or application event filters to get all mouse move events, even when no button is depressed. This is useful for special GUI elements, e.g. tool tips.

Global mouse tracking does not affect widgets and their mouseMoveEvent(). For a widget to get mouse move events when no button is depressed, it must do QWidget::setMouseTracking(TRUE).

This function uses an internal counter. Each setGlobalMouseTracking(TRUE) must have a corresponding setGlobalMouseTracking(FALSE):

```
    // at this point global mouse tracking is off
    QApplication::setGlobalMouseTracking( TRUE );
    QApplication::setGlobalMouseTracking( TRUE );
    QApplication::setGlobalMouseTracking( FALSE );
    // at this point it's still on
    QApplication::setGlobalMouseTracking( FALSE );
    // but now it's off
```

See also hasGlobalMouseTracking() [p. 17] and QWidget::mouseTracking [Widgets with Qt].

void QApplication::setGlobalStrut (const QSize & strut) [static]

Sets the application's global strut to *strut*.

The strut is a size object whose dimensions are the minimum that any GUI element that the user can interact with should have. For example no button should be resized to be smaller than the global strut size.

The strut size should be considered when reimplementing GUI controls that may be used on touch-screens or similar IO-devices.

Example:

```
  QSize& WidgetClass::sizeHint() const
  {
      return QSize( 80, 25 ).expandedTo( QApplication::globalStrut() );
  }
```

See also globalStrut() [p. 17].

void QApplication::setLibraryPaths (const QStringList & paths) [static]

Sets the list of directories to search when loading libraries to *paths*. If *paths* is empty, the path list is unchanged, otherwise all existing paths will be deleted and the path list will consist of the paths given in *paths*.

See also libraryPaths() [p. 18], addLibraryPath() [p. 12], removeLibraryPath() [p. 21] and QLibrary [Plugins with Qt].

void QApplication::setMainWidget (QWidget * mainWidget) [virtual]

Sets the main widget of the application to *mainWidget*.

The main widget is like any other, in most respects except that if it is deleted, the application exits.

You need not have a main widget; connecting lastWindowClosed() to quit() is another alternative.

For X11, this function also resizes and moves the main widget according to the *-geometry* command-line option, so you should set the default geometry (using QWidget::setGeometry()) before calling setMainWidget().

See also mainWidget() [p. 19], exec() [p. 15] and quit() [p. 21].

Examples: action/actiongroup/main.cpp, biff/main.cpp, fonts/simple-qfont-demo/simple-qfont-demo.cpp, life/main.cpp, t1/main.cpp, t4/main.cpp and xml/outliner/main.cpp.

void QApplication::setOverrideCursor (const QCursor & cursor, bool replace = FALSE) [static]

Sets the application override cursor to *cursor*.

Application override cursors are intended for showing the user that the application is in a special state, for example during an operation that might take some time.

This cursor will be displayed in all the widgets of the application until restoreOverrideCursor() or another setOverrideCursor() is called.

Application cursors are stored on an internal stack. setOverrideCursor() pushes the cursor onto the stack, and restoreOverrideCursor() pops the active cursor off the stack. Every setOverrideCursor() must eventually be followed by a corresponding restoreOverrideCursor(), otherwise the stack will never be emptied.

If *replace* is TRUE, the new cursor will replace the last override cursor (the stack keeps its depth). If *replace* is FALSE, the new stack is pushed onto the top of the stack.

Example:

```
    QApplication::setOverrideCursor( Qt::WaitCursor );
    calculateHugeMandelbrot();                // lunch time...
    QApplication::restoreOverrideCursor();
```

See also overrideCursor() [p. 19], restoreOverrideCursor() [p. 21] and QWidget::cursor [Widgets with Qt].

Example: showimg/showimg.cpp.

void QApplication::setPalette (const QPalette & palette, bool informWidgets = FALSE, const char * className = 0) [static]

Changes the default application palette to *palette*. If *informWidgets* is TRUE, then existing widgets are informed about the change and may adjust themselves to the new application setting. Otherwise the change only affects newly created widgets. If *className* is passed, the change applies only to classes that inherit *className* (as reported by QObject::inherits()).

The palette may be changed according to the current GUI style in QStyle::polish().

See also QWidget::palette [Widgets with Qt], palette() [p. 19] and QStyle::polish() [Events, Actions, Layouts and Styles with Qt].

Examples: i18n/main.cpp, themes/metal.cpp, themes/themes.cpp and themes/wood.cpp.

void QApplication::setReverseLayout (bool b) [static]

If *b* is TRUE, all dialogs and widgets will be laid out in a mirrored fashion, as required by right to left languages such as Hebrew and Arabic.

See also reverseLayout() [p. 22].

void QApplication::setStartDragDistance (int l) [static]

Sets the distance after which a drag should start to *l* ms.

See also startDragDistance() [p. 28].

void QApplication::setStartDragTime (int ms) [static]

Sets the time after which a drag should start to *ms* ms.

See also startDragTime() [p. 29].

void QApplication::setStyle (QStyle * style) [static]

Sets the application GUI style to *style*. Ownership of the style object is transferred to QApplication, so QApplication will delete the style object on application exit or when a new style is set.

Example usage:

```
    QApplication::setStyle( new QWindowStyle );
```

When switching application styles, the color palette is set back to the initial colors or the system defaults. This is necessary since certain styles have to adapt the color palette to be fully style-guide compliant.

See also style() [p. 29], QStyle [Events, Actions, Layouts and Styles with Qt], setPalette() [p. 27] and desktopSettingsAware() [p. 15].

Example: themes/themes.cpp.

QStyle * QApplication::setStyle (const QString & style) [static]

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Uses QStyleFactory to create a QStyle object for *style*.

void QApplication::setWheelScrollLines (int n) [static]

Sets the number of lines to scroll when the mouse wheel is rotated to *n*.

If this number exceeds the number of visible lines in a certain widget, the widget should interpret the scroll operation as a single page up / page down operation instead.

See also wheelScrollLines() [p. 31].

void QApplication::setWinStyleHighlightColor (const QColor & c) [static]

This function is obsolete. It is provided to keep old source working. We strongly advise against using it in new code.

Sets the color used to mark selections in windows style for all widgets in the application. Will repaint all widgets if the color is changed.

The default color is `darkBlue`.

See also winStyleHighlightColor() [p. 31].

int QApplication::startDragDistance () [static]

If you support drag and drop in you application and a drag should start after a mouse click and after moving the mouse a certain distance, you should use the value which this method returns as the distance. So if the mouse position of the click is stored in `startPos` and the current position (e.g. in the mouse move event) is `currPos`, you can find out if a drag should be started with code like this:

```
  if ( ( startPos - currPos ).manhattanLength() > QApplication::startDragDistance() )
      startTheDrag();
```

Qt internally uses this value too, e.g. in the QFileDialog.

The default value is 4 pixels.

See also setStartDragDistance() [p. 27], startDragTime() [p. 29] and QPoint::manhattanLength() [Graphics with Qt].

int QApplication::startDragTime () [static]

If you support drag and drop in you application and a drag should start after a mouse click and after a certain time elapsed, you should use the value which this method returns as delay (in ms).

Qt internally uses also this delay e.g. in QTextView or QLineEdit for starting a drag.

The default value is 500 ms.

See also setStartDragTime() [p. 27] and startDragDistance() [p. 28].

bool QApplication::startingUp () [static]

Returns TRUE if an application object has not been created yet.

See also closingDown() [p. 14].

QStyle & QApplication::style () [static]

Returns the style object of the application.

See also setStyle() [p. 28] and QStyle [Events, Actions, Layouts and Styles with Qt].

void QApplication::syncX () [static]

Synchronizes with the X server in the X11 implementation. This normally takes some time. Does nothing on other platforms.

See also flushX() [p. 16].

QWidgetList * QApplication::topLevelWidgets () [static]

Returns a list of the top level widgets in the application.

The list is created using `new` and must be deleted by the caller.

The list is empty (QPtrList::isEmpty()) if there are no top level widgets.

Note that some of the top level widgets may be hidden, for example the tooltip if no tooltip is currently shown.

Example:

```
// Show all hidden top level widgets.
QWidgetList  *list = QApplication::topLevelWidgets();
QWidgetListIt it( *list );  // iterate over the widgets
QWidget * w;
while ( (w=it.current()) != 0 ) {   // for each top level widget...
    ++it;
    if ( !w->isVisible() )
        w->show();
}
delete list;                // delete the list, not the widgets
```

Warning: Delete the list as soon you have finished using it. The widgets in the list may be deleted by someone else at any time.

See also allWidgets() [p. 12], QWidget::isTopLevel [Widgets with Qt], QWidget::visible [Widgets with Qt] and QPtrList::isEmpty() [Datastructures and String Handling with Qt].

QString QApplication::translate (const char * context, const char * sourceText, const char * comment = 0, Encoding encoding = DefaultCodec) const

Returns the translation text for *sourceText*, by querying the installed messages files. The message files are searched from the most recently installed message file back to the first installed message file.

QObject::tr() and QObject::trUtf8() provide this functionality more conveniently.

context is typically a class name (e.g., "MyDialog") and *sourceText* is either English text or a short marker text, if the output text will be very long (as for help texts).

comment is a disambiguating comment, for when the same *sourceText* is used in different roles within one context. By default, it is null.

See the QTranslator [Accessibility and Internationalization with Qt] documentation for more information about contexts and comments.

If none of the message files contain a translation for *sourceText* in *context*, this function returns a QString equivalent of *sourceText*. The encoding of *sourceText* is specified by *encoding*; it defaults to DefaultCodec.

This function is not virtual. You can use alternative translation techniques by subclassing QTranslator.

See also QObject::tr() [p. 137], installTranslator() [p. 17] and defaultCodec() [p. 14].

bool QApplication::tryLock ()

Attempts to lock the Qt library mutex. If the lock was obtained, this function returns TRUE. If another thread has locked the mutex, this function returns FALSE, instead of waiting for the lock to become available.

The mutex must be unlocked with unlock() before another thread can successfully lock it.

See also lock() [p. 18] and unlock() [p. 30].

Type QApplication::type () const

Returns the type of application, Tty, GuiClient or GuiServer.

void QApplication::unlock (bool wakeUpGui = TRUE)

Unlock the Qt library mutex. if *wakeUpGui* is TRUE (the default), then the GUI thread will be woken with QApplication::wakeUpGuiThread().

See also lock() [p. 18] and locked() [p. 18].

void QApplication::wakeUpGuiThread ()

Wakes up the GUI thread.

See also guiThreadAwake() [p. 17].

int QApplication::wheelScrollLines () [static]

Returns the number of lines to scroll when the mouse wheel is rotated.

See also setWheelScrollLines() [p. 28].

QWidget * QApplication::widgetAt (int x, int y, bool child = FALSE) [static]

Returns a pointer to the widget at global screen position *(x,y)*, or a null pointer if there is no Qt widget there.

If *child* is FALSE and there is a child widget at position *(x,y)*, the top-level widget containing it is returned. If *child* is TRUE the child widget at position *(x,y)* is returned.

This function is normally rather slow.

See also QCursor::pos() [Graphics with Qt], QWidget::grabMouse() [Widgets with Qt] and QWidget::grabKeyboard() [Widgets with Qt].

QWidget * QApplication::widgetAt (const QPoint & pos, bool child = FALSE) [static]

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns a pointer to the widget at global screen position *pos*, or a null pointer if there is no Qt widget there.

If *child* is FALSE and there is a child widget at position *pos*, the top-level widget containing it is returned. If *child* is TRUE the child widget at position *pos* is returned.

bool QApplication::winEventFilter (MSG *) [virtual]

The message procedure calls this function for every message received. Reimplement this function if you want to process window messages *msg* that are not processed by Qt.

void QApplication::winFocus (QWidget * widget, bool gotFocus)

If *gotFocus* is TRUE, *widget* will become the active window. Otherwise, the active window is reset to NULL.

const QColor & QApplication::winStyleHighlightColor () [static]

This function is obsolete. It is provided to keep old source working. We strongly advise against using it in new code.

Returns the color used to mark selections in windows style.

See also setWinStyleHighlightColor() [p. 28].

WindowsVersion QApplication::winVersion () [static]

Returns the version of the Windows operating system running:

- Qt::WV_95 - Windows 95
- Qt::WV_98 - Windows 98
- Qt::WV_ME - Windows ME
- Qt::WV_NT - Windows NT 4.x

- Qt::WV_2000 - Windows 2000 (NT5)
- Qt::WV_XP - Windows XP

Note that this function is implemented for the Windows version of Qt only.

Related Functions

void Q_ASSERT (bool test)

Prints a warning message containing the source code file name and line number if *test* is FALSE.

This is really a macro defined in qglobal.h.

Q_ASSERT is useful for testing required conditions in your program.

Example:

```
//
// File: div.cpp
//

#include <qglobal.h>

int divide( int a, int b )
{
    Q_ASSERT( b != 0 );                     // this is line 9
    return a/b;
}
```

If b is zero, the Q_ASSERT statement will output the following message using the qWarning() function:

```
ASSERT: "b == 0" in div.cpp (9)
```

See also qWarning() [p. 35] and Debugging [Programming with Qt].

void Q_CHECK_PTR (void * p)

If *p* is null, a fatal messages says that the program ran out of memory and exits. If *p* is not null, nothing happens.

This is really a macro defined in qglobal.h.

Example:

```
int *a;
Q_CHECK_PTR( a = new int[80] );     // never do this!
  // do this instead:
a = new int[80];
Q_CHECK_PTR( a );                   // this is fine
```

See also qFatal() [p. 34] and Debugging [Programming with Qt].

void qAddPostRoutine (QtCleanUpFunction p)

Adds a global routine that will be called from the QApplication destructor. This function is normally used to add cleanup routines for program-wide functionality.

The function given by *p* should take no arguments and return nothing, like this:

```
static int *global_ptr = 0;

static void cleanup_ptr()
{
    delete [] global_ptr;
    global_ptr = 0;
}

void init_ptr()
{
    global_ptr = new int[100];      // allocate data
    qAddPostRoutine( cleanup_ptr ); // delete later
}
```

Note that for an application- or module-wide cleanup, qAddPostRoutine() is often not suitable. People have a tendency to make such modules dynamically loaded, and then unload those modules long before the QApplication destructor is called, for example.

For modules and libraries, using a reference-counted initialization manager or Qt' parent-child delete mechanism may be better. Here is an example of a private class which uses the parent-child mechanism to call a cleanup function at the right time:

```
class MyPrivateInitStuff: public QObject {
private:
    MyPrivateInitStuff( QObject * parent ): QObject( parent) {
        // initialization goes here
    }
    MyPrivateInitStuff * p;

public:
    static MyPrivateInitStuff * initStuff( QObject * parent ) {
        if ( !p )
            p = new MyPrivateInitStuff( parent );
        return p;
    }

    ~MyPrivateInitStuff() {
        // cleanup (the "post routine") goes here
    }
}
```

By selecting the right parent widget/object, this can often be made to clean up the module's data at the exact right moment.

void qDebug (const char * msg, ...)

Prints a debug message *msg*, or calls the message handler (if it has been installed).

This function takes a format string and a list of arguments, similar to the C printf() function.

Example:

```
qDebug( "my window handle = %x", myWidget->id() );
```

Under X11, the text is printed to stderr. Under Windows, the text is sent to the debugger.

Warning: The internal buffer is limited to 8196 bytes (including the 0-terminator).

See also qWarning() [p. 35], qFatal() [p. 34], qInstallMsgHandler() [p. 34] and Debugging [Programming with Qt].

void qFatal (const char * msg, ...)

Prints a fatal error message *msg* and exits, or calls the message handler (if it has been installed).

This function takes a format string and a list of arguments, similar to the C printf() function.

Example:

```
int divide( int a, int b )
{
    if ( b == 0 )                          // program error
        qFatal( "divide: cannot divide by zero" );
    return a/b;
}
```

Under X11, the text is printed to stderr. Under Windows, the text is sent to the debugger.

Warning: The internal buffer is limited to 8196 bytes (including the 0-terminator).

See also qDebug() [p. 33], qWarning() [p. 35], qInstallMsgHandler() [p. 34] and Debugging [Programming with Qt].

QtMsgHandler qInstallMsgHandler (QtMsgHandler h)

Installs a Qt message handler *h*. Returns a pointer to the message handler previously defined.

The message handler is a function that prints out debug messages, warnings and fatal error messages. The Qt library (debug version) contains hundreds of warning messages that are printed when internal errors (usually invalid function arguments) occur. If you implement your own message handler, you get total control of these messages.

The default message handler prints the message to the standard output under X11 or to the debugger under Windows. If it is a fatal message, the application aborts immediately.

Only one message handler can be defined, since this is usually done on an application-wide basis to control debug output.

To restore the message handler, call `qInstallMsgHandler(0)`.

Example:

```
#include <qapplication.h>
#include
#include

void myMessageOutput( QtMsgType type, const char *msg )
{
    switch ( type ) {
        case QtDebugMsg:
            fprintf( stderr, "Debug: %s\n", msg );
            break;
        case QtWarningMsg:
            fprintf( stderr, "Warning: %s\n", msg );
            break;
        case QtFatalMsg:
            fprintf( stderr, "Fatal: %s\n", msg );
            abort();                        // dump core on purpose
    }
}

int main( int argc, char **argv )
{
    qInstallMsgHandler( myMessageOutput );
    QApplication a( argc, argv );
    ...
    return a.exec();
}
```

See also qDebug() [p. 33], qWarning() [p. 35], qFatal() [p. 34] and Debugging [Programming with Qt].

bool qSysInfo (int * wordSize, bool * bigEndian)

Obtains information about the system.

The system's word size in bits (typically 32) is returned in *wordSize*. The *bigEndian* is set to TRUE if this is a big-endian machine, or to FALSE if this is a little-endian machine.

In debug mode, this function calls qFatal() with a message if the computer is truly weird (i.e. different endianness for 16 bit and 32 bit integers), in release mode it returns FALSE.

void qSystemWarning (const char * msg, int code)

Prints the message *msg* and uses *code* to get a system specific error message. When *code* is -1 (default), the system's last error code will be used if possible. Use this method to handle failures in platform specific API calls.

This function does nothing when Qt is built with Q_NO_DEBUG defined.

const char * qVersion ()

Returns the Qt version number for the library, typically "1.44" or "2.3.0".

void qWarning (const char * msg, ...)

Prints a warning message *msg*, or calls the message handler (if it has been installed).

This function takes a format string and a list of arguments, similar to the C printf() function.

Example:

```
void f( int c )
{
    if ( c > 200 )
        qWarning( "f: bad argument, c == %d", c );
}
```

Under X11, the text is printed to stderr. Under Windows, the text is sent to the debugger.

Warning: The internal buffer is limited to 8196 bytes (including the 0-terminator).

See also qDebug() [p. 33], qFatal() [p. 34], qInstallMsgHandler() [p. 34] and Debugging [Programming with Qt].

QDate Class Reference

The QDate class provides date functions.

`#include <qdatetime.h>`

Public Members

- **QDate** ()
- **QDate** (int y, int m, int d)
- bool **isNull** () const
- bool **isValid** () const
- int **year** () const
- int **month** () const
- int **day** () const
- int **dayOfWeek** () const
- int **dayOfYear** () const
- int **daysInMonth** () const
- int **daysInYear** () const
- QString **toString** (Qt::DateFormat f = Qt::TextDate) const
- QString **toString** (const QString & format) const
- bool **setYMD** (int y, int m, int d)
- QDate **addDays** (int ndays) const
- QDate **addMonths** (int nmonths) const
- QDate **addYears** (int nyears) const
- int **daysTo** (const QDate & d) const
- bool **operator==** (const QDate & d) const
- bool **operator!=** (const QDate & d) const
- bool **operator<** (const QDate & d) const
- bool **operator<=** (const QDate & d) const
- bool **operator>** (const QDate & d) const
- bool **operator>=** (const QDate & d) const

Static Public Members

- QString monthName (int month) *(obsolete)*
- QString dayName (int weekday) *(obsolete)*
- QString **shortMonthName** (int month)
- QString **shortDayName** (int weekday)
- QString **longMonthName** (int month)

- QString **longDayName** (int weekday)
- QDate **currentDate** ()
- QDate **fromString** (const QString & s, Qt::DateFormat f = Qt::TextDate)
- bool **isValid** (int y, int m, int d)
- bool **leapYear** (int y)

Related Functions

- QDataStream & **operator<<** (QDataStream & s, const QDate & d)
- QDataStream & **operator>>** (QDataStream & s, QDate & d)

Detailed Description

The QDate class provides date functions.

A QDate object contains a calendar date, i.e. year, month, and day numbers in the modern western (Gregorian) calendar. It can read the current date from the system clock. It provides functions for comparing dates and for manipulating dates, e.g. by adding a number of days or months or years.

A QDate object is typically created either by giving the year, month and day numbers explicitly, or by using the static function currentDate(), which makes a QDate object which contains the system clock's date. An explicit date can also be set using setYMD(). The fromString() function returns a QDate given a string and a date format which is used to interpret the date within the string.

The year(), month(), and day() functions provide access to the year, month, and day numbers. Also, dayOfWeek() and dayOfYear() functions are provided. The same information is provided in textual format by the toString(), shortDayName(), longDayName(), shortMonthName() and longMonthName() functions.

QDate provides a full set of operators to compare two QDate objects where smaller means earlier and larger means later.

You can increment (or decrement) a date by a given number of days using addDays(). Similarly you can use addMonths() and addYears(). The daysTo() function returns the number of days between two dates.

The daysInMonth() and daysInYear() functions return how many days there are in this date's month and year, respectively. The leapYear() function indicates whether this date is in a leap year.

Note that QDate should not be used for date calculations for dates prior to the introduction of the Gregorian calendar. This calendar was adopted by England from 14th September 1752 (hence this is the earliest valid QDate), and subsequently by most other western countries, until 1923.

The end of time is reached around 8000, by which time we expect Qt to be obsolete.

See also QTime [p. 198], QDateTime [p. 44], QDateEdit [Widgets with Qt], QDateTimeEdit [Widgets with Qt] and Time and Date.

Member Function Documentation

QDate::QDate ()

Constructs a null date. Null dates are invalid.

See also isNull() [p. 39] and isValid() [p. 39].

QDate::QDate (int y, int m, int d)

Constructs a date with year *y*, month *m* and day *d*.

y must be in the range 1752..8000, *m* must be in the range 1..12, and *d* must be in the range 1..31. Exception: if *y* is in the range 0..99, it is interpreted as 1900..1999.

See also isValid() [p. 39].

QDate QDate::addDays (int ndays) const

Returns a QDate object containing a date *ndays* later than the date of this object (or earlier if *ndays* is negative).

See also daysTo() [p. 39].

QDate QDate::addMonths (int nmonths) const

Returns a QDate object containing a date *nmonths* later than the date of this object (or earlier if *nmonths* is negative).

QDate QDate::addYears (int nyears) const

Returns a QDate object containing a date *nyears* later than the date of this object (or earlier if *nyears* is negative).

QDate QDate::currentDate () [static]

Returns the current date, as reported by the system clock.

See also QTime::currentTime() [p. 200] and QDateTime::currentDateTime() [p. 46].

Example: dclock/dclock.cpp.

int QDate::day () const

Returns the day of the month (1..31) of this date.

See also year() [p. 42], month() [p. 40] and dayOfWeek() [p. 38].

Example: dclock/dclock.cpp.

QString QDate::dayName (int weekday) [static]

This function is obsolete. It is provided to keep old source working. We strongly advise against using it in new code.

Use shortDayName() instead.

int QDate::dayOfWeek () const

Returns the weekday (Monday=1..Sunday=7) for this date.

See also day() [p. 38] and dayOfYear() [p. 39].

int QDate::dayOfYear () const

Returns the day of the year (1..365) for this date.

See also day() [p. 38] and dayOfWeek() [p. 38].

int QDate::daysInMonth () const

Returns the number of days in the month (28..31) for this date.

See also day() [p. 38] and daysInYear() [p. 39].

int QDate::daysInYear () const

Returns the number of days in the year (365 or 366) for this date.

See also day() [p. 38] and daysInMonth() [p. 39].

int QDate::daysTo (const QDate & d) const

Returns the number of days from this date to *d* (which is negative if *d* is earlier than this date).

Example:

```
    QDate d1( 1995, 5, 17 );  // May 17th 1995
    QDate d2( 1995, 5, 20 );  // May 20th 1995
    d1.daysTo( d2 );          // returns 3
    d2.daysTo( d1 );          // returns -3
```

See also addDays() [p. 38].

QDate QDate::fromString (const QString & s, Qt::DateFormat f = Qt::TextDate) [static]

Returns the QDate represented by the string *s*, using the format *f*, or an invalid date if this is not possible.

Qt::LocalDate cannot be used here.

Note for Qt::TextDate: It is recommended to use the English short month names (e.g. Jan). Localized month names may also be used, but they depend on the user's locale settings.

bool QDate::isNull () const

Returns TRUE if the date is null; otherwise returns FALSE. A null date is invalid.

See also isValid() [p. 39].

bool QDate::isValid () const

Returns TRUE if this date is valid; otherwise returns FALSE.

See also isNull() [p. 39].

bool QDate::isValid (int y, int m, int d) [static]

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns TRUE if the specified date (year *y*, month *m* and day *d*) is valid.

Example:

```
    QDate::isValid( 2002, 5, 17 );  // TRUE   May 17th 2002 is valid
    QDate::isValid( 2002, 2, 30 );  // FALSE  Feb 30th does not exist
    QDate::isValid( 2004, 2, 29 );  // TRUE   2004 is a leap year
    QDate::isValid( 1202, 6, 6 );   // FALSE  1202 is pre-Gregorian
```

Note that a *y* value in the range 00..99 is interpreted as 1900..1999.

See also isNull() [p. 39] and setYMD() [p. 41].

bool QDate::leapYear (int y) [static]

Returns TRUE if the specified year *y* is a leap year.

QString QDate::longDayName (int weekday) [static]

Returns the long name of the *weekday*.

1 = "Monday", 2 = "Tuesday", ... 7 = "Sunday"

The day names will be localized according to the system's locale settings.

See also toString() [p. 42], shortDayName() [p. 41], shortMonthName() [p. 41] and longMonthName() [p. 40].

QString QDate::longMonthName (int month) [static]

Returns the long name of the *month*.

1 = "January", 2 = "February", ... 12 = "December"

The month names will be localized according to the system's locale settings.

See also toString() [p. 42], shortMonthName() [p. 41], shortDayName() [p. 41] and longDayName() [p. 40].

int QDate::month () const

Returns the month (January=1..December=12) of this date.

See also year() [p. 42] and day() [p. 38].

Example: dclock/dclock.cpp.

QString QDate::monthName (int month) [static]

This function is obsolete. It is provided to keep old source working. We strongly advise against using it in new code.

Use shortMonthName() instead.

bool QDate::operator!= (const QDate & d) const

Returns TRUE if this date is different from *d*; otherwise returns FALSE.

bool QDate::operator< (const QDate & d) const

Returns TRUE if this date is earlier than *d*, otherwise returns FALSE.

bool QDate::operator<= (const QDate & d) const

Returns TRUE if this date is earlier than or equal to *d*, otherwise returns FALSE.

bool QDate::operator== (const QDate & d) const

Returns TRUE if this date is equal to *d*; otherwise returns FALSE.

bool QDate::operator> (const QDate & d) const

Returns TRUE if this date is later than *d*, otherwise returns FALSE.

bool QDate::operator>= (const QDate & d) const

Returns TRUE if this date is later than or equal to *d*, otherwise returns FALSE.

bool QDate::setYMD (int y, int m, int d)

Sets the date's year *y*, month *m* and day *d*.

y must be in the range 1752..8000, *m* must be in the range 1..12, and *d* must be in the range 1..31. Exception: if *y* is in the range 0..99, it is interpreted as 1900..1999.

Returns TRUE if the date is valid, otherwise returns FALSE.

QString QDate::shortDayName (int weekday) [static]

Returns the name of the *weekday*.

1 = "Mon", 2 = "Tue", ... 7 = "Sun"

The day names will be localized according to the system's locale settings.

See also toString() [p. 42], shortMonthName() [p. 41], longMonthName() [p. 40] and longDayName() [p. 40].

QString QDate::shortMonthName (int month) [static]

Returns the name of the *month*.

1 = "Jan", 2 = "Feb", ... 12 = "Dec"

The month names will be localized according to the system's locale settings.

See also toString() [p. 42], longMonthName() [p. 40], shortDayName() [p. 41] and longDayName() [p. 40].

QString QDate::toString (const QString & format) const

Returns the datetime as a string. The *format* parameter determines the format of the result string.

These expressions may be used:

- *d* - the day as number without a leading zero (1-31)
- *dd* - the day as number with a leading zero (01-31)
- *ddd* - the abbrevated day name (Mon - Sun). Uses QDate::shortDayName().
- *dddd* - the long day name (Monday - Sunday). Uses QDate::longDayName().
- *M* - the month as number without a leading zero (1-12)
- *MM* - the month as number with a leading zero (01-12)
- *MMM* - the abbrevated month name (Jan - Dec). Uses QDate::shortMonthName().
- *MMMM* - the long month name (January - December). Uses QDate::longMonthName().
- *yy* - the year as two digit number (00-99)
- *yyyy* - the year as four digit number (0000-9999)

All other input characters will be ignored.

Example format Strings (assumed that the QDate is 21. May 2001)

- "dd.MM.yyyy" will result in "21.05.2001"
- "ddd MMMM d yy" will result in "Tue May 21 01"

See also QDate::toString() [p. 42] and QTime::toString() [p. 203].

QString QDate::toString (Qt::DateFormat f = Qt::TextDate) const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns the date as a string. The *f* parameter determines the format of the string.

If *f* is Qt::TextDate, the string format is "Sat May 20 1995" (using the shortDayName() and shortMonthName() functions to generate the string).

If *f* is Qt::ISODate, the string format corresponds to the ISO 8601 specification for representations of dates, which is YYYY-MM-DD where YYYY is the year, MM is the month of the year (between 01 and 12), and DD is the day of the month between 01 and 31.

If *f* is Qt::LocalDate, the string format depends on the locale settings of the system.

See also shortDayName() [p. 41] and shortMonthName() [p. 41].

int QDate::year () const

Returns the year (1752..8000) of this date.

See also month() [p. 40] and day() [p. 38].

Related Functions

QDataStream & operator<< (QDataStream & s, const QDate & d)

Writes the date, *d*, to the data stream, *s*.

See also Format of the QDataStream operators [Input/Output and Networking with Qt].

QDataStream & operator>> (QDataStream & s, QDate & d)

Reads a date from the stream *s* into *d*.

See also Format of the QDataStream operators [Input/Output and Networking with Qt].

QDateTime Class Reference

The QDateTime class provides date and time functions.

`#include <qdatetime.h>`

Public Members

- **QDateTime** ()
- **QDateTime** (const QDate & date)
- **QDateTime** (const QDate & date, const QTime & time)
- bool **isNull** () const
- bool **isValid** () const
- QDate **date** () const
- QTime **time** () const
- void **setDate** (const QDate & date)
- void **setTime** (const QTime & time)
- void **setTime_t** (uint secsSince1Jan1970UTC)
- QString **toString** (Qt::DateFormat f = Qt::TextDate) const
- QString **toString** (const QString & format) const
- QDateTime **addDays** (int ndays) const
- QDateTime **addMonths** (int nmonths) const
- QDateTime **addYears** (int nyears) const
- QDateTime **addSecs** (int nsecs) const
- int **daysTo** (const QDateTime & dt) const
- int **secsTo** (const QDateTime & dt) const
- bool **operator==** (const QDateTime & dt) const
- bool **operator!=** (const QDateTime & dt) const
- bool **operator<** (const QDateTime & dt) const
- bool **operator<=** (const QDateTime & dt) const
- bool **operator>** (const QDateTime & dt) const
- bool **operator>=** (const QDateTime & dt) const

Static Public Members

- QDateTime **currentDateTime** ()
- QDateTime **fromString** (const QString & s, Qt::DateFormat f = Qt::TextDate)

Related Functions

- QDataStream & **operator<<** (QDataStream & s, const QDateTime & dt)
- QDataStream & **operator>>** (QDataStream & s, QDateTime & dt)

Detailed Description

The QDateTime class provides date and time functions.

A QDateTime object contains a calendar date and a clock time (a "datetime"). It is a combination of the QDate and QTime classes. It can read the current datetime from the system clock. It provides functions for comparing datetimes and for manipulating a datetime by adding a number of seconds, days, months or years.

A QDateTime object is typically created either by giving a date and time explicitly in the constructor, or by using the static function currentDateTime(), which returns a QDateTime object set to the system clock's time. The date and time can be changed with setDate() and setTime(). A datetime can also be set using the setTime_t() function, which takes a POSIX-standard "number of seconds since 00:00:00 on January 1, 1970" value. The fromString() function returns a QDate given a string and a date format which is used to interpret the date within the string.

The date() and time() functions provide access to the date and time parts of the datetime. The same information is provided in textual format by the toString() function.

QDateTime provides a full set of operators to compare two QDateTime objects where smaller means earlier and larger means later.

You can increment (or decrement) a datetime by a given number of seconds using addSecs() or days using addDays(). Similarly you can use addMonths() and addYears(). The daysTo() function returns the number of days between two datetimes, and sectTo() returns the number of seconds between two datetimes.

The range of a datetime object is constrained to the ranges of the QDate and QTime objects which it embodies.

See also QDate [p. 36], QTime [p. 198], QDateTimeEdit [Widgets with Qt] and Time and Date.

Member Function Documentation

QDateTime::QDateTime ()

Constructs a null datetime (i.e. null date and null time). A null datetime is invalid, since the date is invalid.

See also isValid() [p. 47].

QDateTime::QDateTime (const QDate & date)

Constructs a datetime with date *date* and null time (00:00:00.000).

QDateTime::QDateTime (const QDate & date, const QTime & time)

Constructs a datetime with date *date* and time *time*.

QDateTime QDateTime::addDays (int ndays) const

Returns a QDateTime object containing a datetime *ndays* days later than the datetime of this object (or earlier if *ndays* is negative).

See also daysTo() [p. 46], addMonths() [p. 46], addYears() [p. 46] and addSecs() [p. 46].

QDateTime QDateTime::addMonths (int nmonths) const

Returns a QDateTime object containing a datetime *nmonths* months later than the datetime of this object (or earlier if *nmonths* is negative).

See also daysTo() [p. 46], addDays() [p. 45], addYears() [p. 46] and addSecs() [p. 46].

QDateTime QDateTime::addSecs (int nsecs) const

Returns a QDateTime object containing a datetime *nsecs* seconds later than the datetime of this object (or earlier if *nsecs* is negative).

See also secsTo() [p. 47], addDays() [p. 45], addMonths() [p. 46] and addYears() [p. 46].

Example: listviews/listviews.cpp.

QDateTime QDateTime::addYears (int nyears) const

Returns a QDateTime object containing a datetime *nyears* years later than the datetime of this object (or earlier if *nyears* is negative).

See also daysTo() [p. 46], addDays() [p. 45], addMonths() [p. 46] and addSecs() [p. 46].

QDateTime QDateTime::currentDateTime () [static]

Returns the current datetime, as reported by the system clock.

See also QDate::currentDate() [p. 38] and QTime::currentTime() [p. 200].

Example: listviews/listviews.cpp.

QDate QDateTime::date () const

Returns the date part of the datetime.

See also setDate() [p. 48] and time() [p. 48].

int QDateTime::daysTo (const QDateTime & dt) const

Returns the number of days from this datetime to *dt* (which is negative if *dt* is earlier than this datetime).

See also addDays() [p. 45] and secsTo() [p. 47].

QDateTime QDateTime::fromString (const QString & s, Qt::DateFormat f = Qt::TextDate) [static]

Returns the QDateTime represented by the string *s*, using the format *f*, or an invalid datetime if this is not possible.

Note that Qt::LocalDate cannot be used here.

Note for Qt::TextDate: It is recommended to use the English short month names (e.g. Jan). Localized month names may also be used, but they depend on the user's locale settings.

bool QDateTime::isNull () const

Returns TRUE if both the date and the time are null; otherwise returns FALSE. A null datetime is invalid.

See also QDate::isNull() [p. 39] and QTime::isNull() [p. 201].

bool QDateTime::isValid () const

Returns TRUE if both the date and the time are valid; otherwise returns FALSE.

See also QDate::isValid() [p. 39] and QTime::isValid() [p. 201].

bool QDateTime::operator!= (const QDateTime & dt) const

Returns TRUE if this datetime is different from *dt*; otherwise returns FALSE.

See also operator==() [p. 47].

bool QDateTime::operator< (const QDateTime & dt) const

Returns TRUE if this datetime is earlier than *dt*, otherwise returns FALSE.

bool QDateTime::operator<= (const QDateTime & dt) const

Returns TRUE if this datetime is earlier than or equal to *dt*, otherwise returns FALSE.

bool QDateTime::operator== (const QDateTime & dt) const

Returns TRUE if this datetime is equal to *dt*; otherwise returns FALSE.

See also operator!=() [p. 47].

bool QDateTime::operator> (const QDateTime & dt) const

Returns TRUE if this datetime is later than *dt*, otherwise returns FALSE.

bool QDateTime::operator>= (const QDateTime & dt) const

Returns TRUE if this datetime is later than or equal to *dt*, otherwise returns FALSE.

int QDateTime::secsTo (const QDateTime & dt) const

Returns the number of seconds from this datetime to *dt* (which is negative if *dt* is earlier than this datetime).

Example:

```
    QDateTime dt = QDateTime::currentDateTime();
    QDateTime xmas( QDate(dt.year(),12,24), QTime(17,00) );
    qDebug( "There are %d seconds to Christmas", dt.secsTo(xmas) );
```

See also addSecs() [p. 46], daysTo() [p. 46] and QTime::secsTo() [p. 202].

void QDateTime::setDate (const QDate & date)

Sets the date part of this datetime to *date*.

See also date() [p. 46] and setTime() [p. 48].

void QDateTime::setTime (const QTime & time)

Sets the time part of this datetime to *time*.

See also time() [p. 48] and setDate() [p. 48].

void QDateTime::setTime_t (uint secsSince1Jan1970UTC)

Sets the date and time to local time given the number of seconds that have passed since 00:00:00 on January 1, 1970, Coordinated Universal Time (UTC). On systems that do not support timezones this function will behave as if local time were UTC.

Note that Microsoft Windows supports only a limited range of values for *secsSince1Jan1970UTC*.

QTime QDateTime::time () const

Returns the time part of the datetime.

See also setTime() [p. 48] and date() [p. 46].

QString QDateTime::toString (const QString & format) const

Returns the datetime as a string. The *format* parameter determines the format of the result string.

These expressions may be used for the date:

- *d* - the day as number without a leading zero (1-31)
- *dd* - the day as number with a leading zero (01-31)
- *ddd* - the abbrevated day name (Mon - Sun). Uses QDate::shortDayName().
- *dddd* - the long day name (Monday - Sunday). Uses QDate::longDayName().
- *M* - the month as number without a leading zero (1-12)
- *MM* - the month as number with a leading zero (01-12)
- *MMM* - the abbrevated month name (Jan - Dec). Uses QDate::shortMonthName().
- *MMMM* - the long month name (January - December). Uses QDate::longMonthName().
- *yy* - the year as two digit number (00-99)
- *yyyy* - the year as four digit number (0000-9999)

These expressions may be used for the time:

- *h* - the hour without a leading zero (0-23 or 1-12 if AM/PM display)
- *hh* - the hour with a leading zero (00-23 or 01-12 if AM/PM display)
- *m* - the minute without a leading zero (0-59)
- *mm* - the minute with a leading zero (00-59)
- *s* - the second whithout a leading zero (0-59)

- *ss* - the second whith a leading zero (00-59)
- *z* - the milliseconds without leading zeroes (0-999)
- *zzz* - the milliseconds with leading zeroes (000-999)
- *AP* - switch to AM/PM display. *AP* will be replaced by either "AM" or "PM".
- *ap* - switch to AM/PM display. *ap* will be replaced by either "am" or "pm".

All other input characters will be ignored.

Example format Strings (assumed that the QDateTime is 21. May 2001 14:13:09)

- "dd.MM.yyyy" will result in "21.05.2001"
- "ddd MMMM d yy" will result in "Tue May 21 01"
- "hh:mm:ss.zzz" will result in "14:13:09.042"
- "h:m:s ap" will result in "2:13:9 pm"

See also QDate::toString() [p. 42] and QTime::toString() [p. 203].

QString QDateTime::toString (Qt::DateFormat f = Qt::TextDate) const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns the datetime as a string. The *f* parameter determines the format of the string.

If *f* is Qt::TextDate, the string format is "Wed May 20 03:40:13 1998" (using QDate::shortDayName(), QDate::shortMonthName(), and QTime::toString() to generate the string).

If *f* is Qt::ISODate, the string format corresponds to the ISO 8601 specification for representations of dates and times, which is YYYY-MM-DDTHH:MM:SS.

If *f* is Qt::LocalDate, the string format depends on the locale settings of the system.

If the format *f* is invalid, toString() returns a null string.

See also QDate::toString() [p. 42] and QTime::toString() [p. 203].

Related Functions

QDataStream & operator<< (QDataStream & s, const QDateTime & dt)

Writes the datetime *dt* to the stream *s*.

See also Format of the QDataStream operators [Input/Output and Networking with Qt].

QDataStream & operator>> (QDataStream & s, QDateTime & dt)

Reads a datetime from the stream *s* into *dt*.

See also Format of the QDataStream operators [Input/Output and Networking with Qt].

QDoubleValidator Class Reference

The QDoubleValidator class provides range checking of floating-point numbers.

`#include <qvalidator.h>`

Inherits QValidator [p. 208].

Public Members

- **QDoubleValidator** (QObject * parent, const char * name = 0)
- **QDoubleValidator** (double bottom, double top, int decimals, QObject * parent, const char * name = 0)
- **~QDoubleValidator** ()
- virtual QValidator::State **validate** (QString & input, int &) const
- virtual void **setRange** (double minimum, double maximum, int decimals = 0)
- void **setBottom** (double)
- void **setTop** (double)
- void **setDecimals** (int)
- double **bottom** () const
- double **top** () const
- int **decimals** () const

Properties

- double **bottom** — the validator's minimum acceptable value
- int **decimals** — the validator's maximum number of digits after the decimal point
- double **top** — the validator's maximum acceptable value

Detailed Description

The QDoubleValidator class provides range checking of floating-point numbers.

QDoubleValidator provides an upper bound, a lower bound and a limit on the number of digits after the decimal point. It does not provide a fixup() function.

You can set the acceptable range in one call with setRange(), or with setBottom() and setTop(). Set the number of decimal places with setDecimals(). The validate() function returns the validation state.

See also QIntValidator [p. 113], QRegExpValidator [p. 153] and Miscellaneous Classes.

Member Function Documentation

QDoubleValidator::QDoubleValidator (QObject * parent, const char * name = 0)

Constructs a validator object with parent *parent*, called *name*, which accepts any double.

QDoubleValidator::QDoubleValidator (double bottom, double top, int decimals, QObject * parent, const char * name = 0)

Constructs a validator object with parent *parent*, called *name*. This validator will accept doubles from *bottom* to *top* inclusive, with up to *decimals* digits after the decimal point.

QDoubleValidator::~QDoubleValidator ()

Destroys the validator, freeing any resources used.

double QDoubleValidator::bottom () const

Returns the validator's minimum acceptable value. See the "bottom" [p. 52] property for details.

int QDoubleValidator::decimals () const

Returns the validator's maximum number of digits after the decimal point. See the "decimals" [p. 52] property for details.

void QDoubleValidator::setBottom (double)

Sets the validator's minimum acceptable value. See the "bottom" [p. 52] property for details.

void QDoubleValidator::setDecimals (int)

Sets the validator's maximum number of digits after the decimal point. See the "decimals" [p. 52] property for details.

void QDoubleValidator::setRange (double minimum, double maximum, int decimals = 0) [virtual]

Sets the validator to accept doubles from *minimum* up to and including *maximum* with at most *decimals* digits after the decimal point.

void QDoubleValidator::setTop (double)

Sets the validator's maximum acceptable value. See the "top" [p. 52] property for details.

double QDoubleValidator::top () const

Returns the validator's maximum acceptable value. See the "top" [p. 52] property for details.

QValidator::State QDoubleValidator::validate (QString & input, int &) const [virtual]

Returns Acceptable if the string *input* contains a double that is within the valid range and is in the correct format.

Returns Intermediate if *input* contains a double that is outside the range or is in the wrong format, e.g. with too many digits after the decimal point or is empty.

Returns Invalid if the *input* is not a double.

Reimplemented from QValidator [p. 209].

Property Documentation

double bottom

This property holds the validator's minimum acceptable value.

Set this property's value with setBottom() and get this property's value with bottom().

See also setRange() [p. 51].

int decimals

This property holds the validator's maximum number of digits after the decimal point.

Set this property's value with setDecimals() and get this property's value with decimals().

See also setRange() [p. 51].

double top

This property holds the validator's maximum acceptable value.

Set this property's value with setTop() and get this property's value with top().

See also setRange() [p. 51].

QEditorFactory Class Reference

The QEditorFactory class is used to create editor widgets for QVariant data types.

`#include <qeditorfactory.h>`

Inherits QObject [p. 123].

Inherited by QSqlEditorFactory [Databases with Qt].

Public Members

- **QEditorFactory** (QObject * parent = 0, const char * name = 0)
- **~QEditorFactory** ()
- virtual QWidget * **createEditor** (QWidget * parent, const QVariant & v)

Static Public Members

- QEditorFactory * **defaultFactory** ()
- void **installDefaultFactory** (QEditorFactory * factory)

Detailed Description

The QEditorFactory class is used to create editor widgets for QVariant data types.

Each editor factory provides the createEditor() function which given a QVariant will create and return a QWidget that can edit that QVariant. For example if you have a QVariant::String type, a QLineEdit would be the default editor returned, whereas a QVariant::Int's default editor would be a QSpinBox.

If you want to create different editors for fields with the same data type, subclass QEditorFactory and reimplement the createEditor() function.

See also Advanced Widgets.

Member Function Documentation

QEditorFactory::QEditorFactory (QObject * parent = 0, const char * name = 0)

Constructs an editor factory with parent *parent* and name *name*.

QEditorFactory::~QEditorFactory ()

Destroys the object and frees any allocated resources.

QWidget * QEditorFactory::createEditor (QWidget * parent, const QVariant & v) [virtual]

Creates and returns the appropriate editor for the QVariant *v*. If the QVariant is invalid, 0 is returned. The *parent* is passed to the appropriate editor's constructor.

Reimplemented in QSqlEditorFactory.

QEditorFactory * QEditorFactory::defaultFactory () [static]

Returns an instance of a default editor factory.

void QEditorFactory::installDefaultFactory (QEditorFactory * factory) [static]

Replaces the default editor factory with *factory*. *QEditorFactory takes ownership of factory, and destroys it when it is no longer needed.*

QErrorMessage Class Reference

The QErrorMessage class provides an error message display dialog.

`#include <qerrormessage.h>`

Inherits QDialog [Dialogs and Windows with Qt].

Public Members

- **QErrorMessage** (QWidget * parent, const char * name = 0)
- **~QErrorMessage** ()

Public Slots

- void **message** (const QString & m)

Static Public Members

- QErrorMessage * **qtHandler** ()

Detailed Description

The QErrorMessage class provides an error message display dialog.

This is basically a QLabel and a "show this message again" checkbox which remembers what not to show.

There are two ways to use this class:

1. For production applications. In this context the class can be used to display messages which you don't need the user to see more than once. To use QErrorMessage like this, you create the dialog in the usual way and call the message() slot, or connect signals to it.
2. For developers. In this context the static qtHandler() installs a message handler using qInstallMsgHandler() and creates a QErrorMessage that displays qDebug(), qWarning() and qFatal() messages.

In both cases QErrorMessage will queue pending messages, and display them (or not) in order, as soon as the user presses Enter or clicks OK after seeing each message.

See also QMessageBox [Dialogs and Windows with Qt], QStatusBar::message() [Widgets with Qt], Dialog Classes and Miscellaneous Classes.

Member Function Documentation

QErrorMessage::QErrorMessage (QWidget * parent, const char * name = 0)

Constructs and installs an error handler window. The parent *parent* and name *name* are passed on to the QDialog constructor.

QErrorMessage::~QErrorMessage ()

Destroys the object and frees any allocated resources. Notably, the list of "do not show again" messages is deleted.

void QErrorMessage::message (const QString & m) [slot]

Shows message *m* and returns immediately. If the user has requested that *m* not be shown, this function does nothing.

Normally, *m* is shown at once, but if there are pending messages, *m* is queued for later display.

QErrorMessage * QErrorMessage::qtHandler () [static]

Returns a pointer to a QErrorMessage object that outputs the default Qt messages. This function creates such an object, if there isn't one already.

QFocusData Class Reference

The QFocusData class maintains the list of widgets in the focus chain.

`#include <qfocusdata.h>`

Public Members

- QWidget * **focusWidget** () const
- QWidget * **home** ()
- QWidget * **next** ()
- QWidget * **prev** ()
- int **count** () const

Detailed Description

The QFocusData class maintains the list of widgets in the focus chain.

This read-only list always contains at least one widget (i.e. the top-level widget). It provides a simple cursor which can be reset to the current focus widget using home(), or moved to its neighboring widgets using next() and prev(). You can also retrieve the count() of the number of widgets in the list. The list is a loop, so if you keep iterating, for example using next(), you will never come to the end.

Some widgets in the list may not accept focus. Widgets are added to the list as necessary, but not removed from it. This lets widgets change focus policy dynamically without disrupting the focus chain the user experiences. When a widget disables and re-enables tab focus, its position in the focus chain does not change.

When reimplementing QWidget::focusNextPrevChild() to provide special focus flow, you will usually call QWidget::focusData() to retrieve the focus data stored at the top-level widget. A top-level widget's focus data contains the focus list for its hierarchy of widgets.

The cursor may change at any time.

This class is *not* thread-safe.

See also QWidget::focusNextPrevChild() [Widgets with Qt], QWidget::setTabOrder() [Widgets with Qt], QWidget::focusPolicy [Widgets with Qt] and Miscellaneous Classes.

Member Function Documentation

int QFocusData::count () const

Returns the number of widgets in the focus chain.

QWidget * QFocusData::focusWidget () const

Returns the widgets in the hierarchy that are in the focus chain.

QWidget * QFocusData::home ()

Moves the cursor to the focusWidget() and returns that widget. You must call this before next() or prev() to iterate meaningfully.

QWidget * QFocusData::next ()

Moves the cursor to the next widget in the focus chain. There is *always* a next widget because the list is a loop.

QWidget * QFocusData::prev ()

Moves the cursor to the previous widget in the focus chain. There is *always* a previous widget because the list is a loop.

QFont Class Reference

The QFont class specifies a font used for drawing text.

`#include <qfont.h>`

Public Members

- enum **StyleHint** { Helvetica, SansSerif = Helvetica, Times, Serif = Times, Courier, TypeWriter = Courier, OldEnglish, Decorative = OldEnglish, System, AnyStyle }
- enum **StyleStrategy** { PreferDefault = 0x0001, PreferBitmap = 0x0002, PreferDevice = 0x0004, PreferOutline = 0x0008, ForceOutline = 0x0010, PreferMatch = 0x0020, PreferQuality = 0x0040, PreferAntialias = 0x0080, NoAntialias = 0x0100 }
- enum **Weight** { Light = 25, Normal = 50, DemiBold = 63, Bold = 75, Black = 87 }
- **QFont** ()
- **QFont** (const QString & family, int pointSize = 12, int weight = Normal, bool italic = FALSE)
- **QFont** (const QFont & font)
- **~QFont** ()
- QString **family** () const
- void **setFamily** (const QString & family)
- int **pointSize** () const
- float **pointSizeFloat** () const
- void **setPointSize** (int pointSize)
- void **setPointSizeFloat** (float pointSize)
- int **pixelSize** () const
- void **setPixelSize** (int pixelSize)
- void setPixelSizeFloat (float pixelSize) *(obsolete)*
- int **weight** () const
- void **setWeight** (int weight)
- bool **bold** () const
- void **setBold** (bool enable)
- bool **italic** () const
- void **setItalic** (bool enable)
- bool **underline** () const
- void **setUnderline** (bool enable)
- bool **strikeOut** () const
- void **setStrikeOut** (bool enable)
- bool **fixedPitch** () const
- void **setFixedPitch** (bool enable)
- StyleHint **styleHint** () const
- StyleStrategy **styleStrategy** () const
- void **setStyleHint** (StyleHint hint, StyleStrategy strategy = PreferDefault)

- void **setStyleStrategy** (StyleStrategy s)
- bool **rawMode** () const
- void **setRawMode** (bool enable)
- bool **exactMatch** () const
- QFont & **operator=** (const QFont & font)
- bool **operator==** (const QFont & f) const
- bool **operator!=** (const QFont & f) const
- bool **isCopyOf** (const QFont & f) const
- HFONT **handle** () const
- void **setRawName** (const QString & name)
- QString **rawName** () const
- QString **key** () const
- QString **toString** () const
- bool **fromString** (const QString & descrip)
- enum **Script** { Latin, Greek, Cyrillic, Armenian, Georgian, Runic, Ogham, SpacingModifiers, CombiningMarks, Hebrew, Arabic, Syriac, Thaana, Devanagari, Bengali, Gurmukhi, Gujarati, Oriya, Tamil, Telugu, Kannada, Malayalam, Sinhala, Thai, Lao, Tibetan, Myanmar, Khmer, Han, Hiragana, Katakana, Hangul, Bopomofo, Yi, Ethiopic, Cherokee, CanadianAboriginal, Mongolian, CurrencySymbols, LetterlikeSymbols, NumberForms, MathematicalOperators, TechnicalSymbols, GeometricSymbols, MiscellaneousSymbols, EnclosedAndSquare, Braille, Unicode, NScripts, UnknownScript = NScripts, NoScript, HanX11, LatinBasic = Latin, LatinExtendedA_2 = HanX11 + 1, LatinExtendedA_3, LatinExtendedA_4, LatinExtendedA_14, LatinExtendedA_15, LastPrivateScript }
- QString **defaultFamily** () const
- QString **lastResortFamily** () const
- QString **lastResortFont** () const

Static Public Members

- QString **substitute** (const QString & familyName)
- QStringList **substitutes** (const QString & familyName)
- QStringList **substitutions** ()
- void **insertSubstitution** (const QString & familyName, const QString & substituteName)
- void **insertSubstitutions** (const QString & familyName, const QStringList & substituteNames)
- void **removeSubstitution** (const QString & familyName)
- QFont defaultFont () *(obsolete)*
- void setDefaultFont (const QFont & f) *(obsolete)*

Protected Members

- bool **dirty** () const
- int **deciPointSize** () const

Related Functions

- QDataStream & **operator<<** (QDataStream & s, const QFont & font)
- QDataStream & **operator>>** (QDataStream & s, QFont & font)

Detailed Description

The QFont class specifies a font used for drawing text.

When you create a QFont object you specify various attributes that you want the font to have. Qt will use the font with the specified attributes, or if no matching font exists, Qt will use the closest matching installed font. The attributes of the font that is actually used are retrievable from a QFontInfo object. If the window system provides an exact match exactMatch() returns TRUE. Use QFontMetrics to get measurements, e.g. the pixel length of a string using QFontMetrics::width().

Use QApplication::setFont() to set the application's default font.

If a choosen X11 font does not include all the characters that need to be displayed, QFont will try to find the characters in the nearest equivalent fonts. When a QPainter draws a character from a font the QFont will report whether or not it has the character; if it does not, QPainter will draw an unfilled square.

Create QFonts like this:

```
    QFont serifFont( "Times", 10, Bold );
    QFont sansFont( "Helvetica [Cronyx]", 12 );
```

The attributes set in the constructor can also be set later, e.g. setFamily(), setPointSize(), setPointSizeFloat(), setWeight() and setItalic(). The remaining attributes must be set after contstruction, e.g. setBold(), setUnderline(), setStrikeOut() and setFixedPitch(). QFontInfo objects should be created *after* the font's attributes have been set. A QFontInfo object will not change, even if you change the font's attributes. The corresponding "get" functions, e.g. family(), pointSize(), etc., return the values that were set, even though the values used may differ. The actual values are available from a QFontInfo object.

If the requested font family is unavailable you can influence the font matching algorithm by choosing a particular QFont::StyleHint and QFont::StyleStrategy with setStyleHint(). The default family (corresponding to the current style hint) is returned by defaultFamily().

The font-matching algorithm has a lastResortFamily() and lastResortFont() in cases where a suitable match cannot be found. You can provide substitutions for font family names using insertSubstitution() and insertSubstitutions(). Substitutions can be removed with removeSubstitution(). Use substitute() to retrieve a family's first substitute, or the family name itself if it has no substitutes. Use substitutes() to retrieve a list of a family's substitutes (which may be empty).

Every QFont has a key() which you can use, for example, as the key in a cache or dictionary. If you want to store a user's font preferences you could use QSettings, writing the font information with toString() and reading it back with fromString(). The operator<<() and operator>>() functions are also available, but they work on a data stream.

It is possible to set the height of characters shown on the screen to a specified number of pixels with setPixelSize(); however using setPointSize() has a similar effect and provides device independence.

Under the X Window System you can set a font using its system specific name with setRawName().

Loading fonts can be expensive, especially on X11. QFont contains extensive optimizations to make the copying of QFont objects fast, and to cache the results of the slow window system functions it depends upon.

The font matching algorithm works as follows:

1. The specified font family is searched for.
2. If not found, the styleHint() is used to select a replacement family.
3. Each replacement font family is searched for.
4. If none of these are found or there was no styleHint(), "helvetica" will be searched for.
5. If "helvetica" isn't found Qt will try the lastResortFamily().
6. If the lastResortFamily() isn't found Qt will try the lastResortFont() which will always return a name of some kind.

Once a font is found, the remaining attributes are matched in order of priority:

1. fixedPitch()
2. pointSize() (see below)
3. weight()
4. italic()

If you have a font which matches on family, even if none of the other attributes match, this font will be chosen in preference to a font which doesn't match on family but which does match on the other attributes. This is because font family is the dominant search criteria.

The point size is defined to match if it is within 20% of the requested point size. When several fonts match and are only distinguished by point size, the font with the closest point size to the one requested will be chosen.

The actual family, font size, weight and other font attributes used for drawing text will depend on what's available for the chosen family under the window system. A QFontInfo object can be used to determine the actual values used for drawing the text.

Examples:

```
    QFont f("Helvetica");
```

If you had both an Adobe and a Cronyx Helvetica, you might get either.

```
    QFont f1( "Helvetica [Cronyx]" );  // Qt 3.x
    QFont f2( "Cronyx-Helvetica" );    // Qt 2.x compatibility
```

You can specify the foundry you want in the family name. Both fonts, f1 and f2, in the above example will be set to "Helvetica [Cronyx]".

To determine the attributes of the font actually used in the window system, use a QFontInfo object, e.g.

```
    QFontInfo info( f1 );
    QString family = info.family();
```

To find out font metrics use a QFontMetrics object, e.g.

```
    QFontMetrics fm( f1 );
    int pixelWidth = fm.width( "How many pixels wide is this text?" );
    int pixelHeight = fm.height();
```

For more general information on fonts, see the comp.fonts FAQ. Information on encodings can be found from Roman Czyborra's page.

See also QFontMetrics [p. 90], QFontInfo [p. 85], QFontDatabase [p. 75], QApplication::setFont() [p. 25], QWidget::font [Widgets with Qt], QPainter::setFont() [Graphics with Qt], QFont::StyleHint [p. 65], QFont::Weight [p. 65], Widget Appearance and Style, Graphics Classes and Implicitly and Explicitly Shared Classes.

Member Type Documentation

QFont::Script

This enum represents Unicode allocated scripts. For exhaustive coverage see The Unicode Standard Version 3.0. The following scripts are supported:

Modern European alphabetic scripts (left to right):

- `QFont::Latin` - consists of most alphabets based on the original Latin alphabet.
- `QFont::Greek` - covers ancient and modern Greek and Coptic.
- `QFont::Cyrillic` - covers the Slavic and non-Slavic languages using cyrillic alphabets.
- `QFont::Armenian` - contains the Armenian alphabet used with the Armenian language.
- `QFont::Georgian` - covers at least the language Georgian.
- `QFont::Runic` - covers the known constituents of the Runic alphabets used by the early and medieval societies in the Germanic, Scandinavian, and Anglo-Saxon areas.
- `QFont::Ogham` - is an alphabetical script used to write a very early form of Irish.
- `QFont::SpacingModifiers` - are small signs indicating modifications to the preceeding letter.
- `QFont::CombiningMarks` - consist of diacritical marks not specific to a particular alphabet, diacritical marks used in combination with mathematical and technical symbols, and glyph encodings applied to multiple letterforms.

Middle Eastern scripts (right to left):

- `QFont::Hebrew` - is used for writing Hebrew, Yiddish, and some other languages.
- `QFont::Arabic` - covers the Arabic language as well as Persian, Urdu, Kurdish and some others.
- `QFont::Syriac` - is used to write the active liturgical languages and dialects of several Middle Eastern and Southeast Indian communities.
- `QFont::Thaana` - is used to write the Maledivian Dhivehi language.

South and Southeast Asian scripts (left to right with few historical exceptions):

- `QFont::Devanagari` - covers classical Sanskrit and modern Hindi as well as several other languages.
- `QFont::Bengali` - is a relative to Devanagari employed to write the Bengali language used in West Bengal/India and Bangladesh as well as several minority languages.
- `QFont::Gurmukhi` - is another Devanagari relative used to write Punjabi.
- `QFont::Gujarati` - is closely related to Devanagari and used to write the Gujarati language of the Gujarat state in India.
- `QFont::Oriya` - is used to write the Oriya language of Orissa state/India.
- `QFont::Tamil` - is used to write the Tamil language of Tamil Nadu state/India, Sri Lanka, Singapore and parts of Malaysia as well as some minority languages.
- `QFont::Telugu` - is used to write the Telugu language of Andhra Pradesh state/India and some minority languages.
- `QFont::Kannada` - is another South Indian script used to write the Kannada language of Karnataka state/India and some minority languages.
- `QFont::Malayalam` - is used to write the Malayalam language of Kerala state/India.
- `QFont::Sinhala` - is used for Sri Lanka's majority language Sinhala and is also employed to write Pali, Sanskrit, and Tamil.
- `QFont::Thai` - is used to write Thai and other Southeast Asian languages.
- `QFont::Lao` - is a language and script quite similar to Thai.
- `QFont::Tibetan` - is the script used to write Tibetan in several countries like Tibet, the bordering Indian regions and Nepal. It is also used in the Buddist philosophy and liturgy of the Mongolian cultural area.
- `QFont::Myanmar` - is mainly used to write the Burmese language of Myanmar (former Burma).
- `QFont::Khmer` - is the official language of Kampuchea.

East Asian scripts (traditionally top-down, right to left, modern often horizontal left to right):

- `QFont::Han` - consists of the CJK (Chinese, Japanese, Korean) idiographic characters.

- `QFont::Hiragana` - is a cursive syllabary used to indicate phonetics and pronounciation of Japanese words.
- `QFont::Katakana` - is a non-cursive syllabic script used to write Japanese words with visual emphasis and non-Japanese words in a phonetical manner.
- `QFont::Hangul` - is a Korean script consisting of alphabetic components.
- `QFont::Bopomofo` - is a phonetic alphabet for Chinese (mainly Mandarin).
- `QFont::Yi` - (also called Cuan or Wei) is a syllabary used to write the Yi language of Southwestern China, Myanmar, Laos, and Vietnam.

Additional scripts that do not fit well into the script categories above:

- `QFont::Ethiopic` - is a syllabary used by several Central East African languages.
- `QFont::Cherokee` - is a left-to-right syllabic script used to write the Cherokee language.
- `QFont::CanadianAboriginal` - consists of the syllabics used by some Canadian aboriginal societies.
- `QFont::Mongolian` - is the traditional (and recently reintroduced) script used to write Mongolian.

Symbols:

- `QFont::CurrencySymbols` - contains currency symbols not encoded in other scripts.
- `QFont::LetterlikeSymbols` - consists of symbols derived from ordinary letters of an alphabetical script.
- `QFont::NumberForms` - are provided for compatibility with other existing character sets.
- `QFont::MathematicalOperators` - consists of encodings for operators, relations and other symbols like arrows used in a mathematical context.
- `QFont::TechnicalSymbols` - contains representations for control codes, the space symbol, APL symbols and other symbols mainly used in the context of electronic data processing.
- `QFont::GeometricSymbols` - covers block elements and geometric shapes.
- `QFont::MiscellaneousSymbols` - consists of a heterogeneous collection of symbols that do not fit any other Unicode character block, e.g. Dingbats.
- `QFont::EnclosedAndSquare` - is provided for compatibility with some East Asian standards.
- `QFont::Braille` - is an international writing system used by blind people. This script encodes the 256 eight-dot patterns with the 64 six-dot patterns as a subset.

- `QFont::Unicode` - includes all the above scripts.

The values below are provided for completeness and must not be used in user programs.

- `QFont::HanX11` - For internal use only.
- `QFont::LatinBasic` - For internal use only.
- `QFont::LatinExtendedA_2` - For internal use only.
- `QFont::LatinExtendedA_3` - For internal use only.
- `QFont::LatinExtendedA_4` - For internal use only.
- `QFont::LatinExtendedA_14` - For internal use only.
- `QFont::LatinExtendedA_15` - For internal use only.
- `QFont::LastPrivateScript` - For internal use only.
- `QFont::NScripts` - For internal use only.
- `QFont::NoScript` - For internal use only.
- `QFont::UnknownScript` - For internal use only.

QFont::StyleHint

Style hints are used by the font matching algorithm to find an appropriate default family if a selected font family is not available.

- `QFont::AnyStyle` - leaves the font matching algorithm to choose the family. This is the default.
- `QFont::SansSerif` - the font matcher prefer sans serif fonts.
- `QFont::Helvetica` - is a synonym for SansSerif.
- `QFont::Serif` - the font matcher prefers serif fonts.
- `QFont::Times` - is a synonym for Serif.
- `QFont::TypeWriter` - the font matcher prefers fixed pitch fonts.
- `QFont::Courier` - a synonym for TypeWriter.
- `QFont::OldEnglish` - the font matcher prefers decorative fonts.
- `QFont::Decorative` - is a synonym for OldEnglish.
- `QFont::System` - the font matcher prefers system fonts.

QFont::StyleStrategy

The style strategy tells the font matching algorithm what type of fonts should be used to find an appropriate default family.

The following strategies are available:

- `QFont::PreferDefault` - the default style strategy. It does not prefer any type of font.
- `QFont::PreferBitmap` - prefers bitmap fonts (as opposed to outline fonts).
- `QFont::PreferDevice` - prefers device fonts.
- `QFont::PreferOutline` - prefers outline fonts (as opposed to bitmap fonts).
- `QFont::ForceOutline` - forces the use of outline fonts.
- `QFont::NoAntialias` - don't antialias the fonts.
- `QFont::PreferAntialias` - antialias if possible.

Any of these may be OR-ed with one of these flags:

- `QFont::PreferMatch` - prefer an exact match. The font matcher will try to use the exact font size that has been specified.
- `QFont::PreferQuality` - prefer the best quality font. The font matcher will use the nearest standard point size that the font supports.

Whilst all strategies work on Windows, they are currently ignored under X11.

QFont::Weight

Qt uses a weighting scale from 0 to 99 similar to, but not the same as, the scales used in Windows or CSS. A weight of 0 is ultralight, whilst 99 will be an extremely black.

This enum contains the predefined font weights:

- `QFont::Light` - 25
- `QFont::Normal` - 50
- `QFont::DemiBold` - 63
- `QFont::Bold` - 75
- `QFont::Black` - 87

Member Function Documentation

QFont::QFont ()

Constructs a font object that uses the application's default font.

See also QApplication::setFont() [p. 25] and QApplication::font() [p. 16].

QFont::QFont (const QString & family, int pointSize = 12, int weight = Normal, bool italic = FALSE)

Constructs a font object with the specified *family*, *pointSize*, *weight* and *italic* settings.

If *pointSize* is <= 0 it is set to 1.

The *family* name may optionally also include a foundry name, e.g. "Helvetica [Cronyx]". (The Qt 2.x syntax, i.e. "Cronyx-Helvetica", is also supported.) If the *family* is available from more than one foundry and the foundry isn't specified, an arbitrary foundry is chosen. If the family isn't available a family will be set using the font matching algorithm.

See also Weight [p. 65], setFamily() [p. 70], setPointSize() [p. 71], setWeight() [p. 72], setItalic() [p. 70], setStyleHint() [p. 72] and QApplication::font() [p. 16].

QFont::QFont (const QFont & font)

Constructs a font that is a copy of *font*.

QFont::~QFont ()

Destroys the font object and frees all allocated resources.

bool QFont::bold () const

Returns TRUE if weight() is a value greater than QFont::Normal; otherwise returns FALSE.

See also weight() [p. 73], setBold() [p. 70] and QFontInfo::bold() [p. 86].

int QFont::deciPointSize () const [protected]

Returns the point size in 1/10ths of a point.

The returned value will be -1 if the font size has been specified in pixels.

See also pointSize() [p. 69] and pointSizeFloat() [p. 69].

QString QFont::defaultFamily () const

Returns the family name that corresponds to the current style hint.

See also StyleHint [p. 65], styleHint() [p. 72] and setStyleHint() [p. 72].

QFont QFont::defaultFont () [static]

This function is obsolete. It is provided to keep old source working. We strongly advise against using it in new code.

Please use QApplication::font() instead.

bool QFont::dirty () const [protected]

Returns TRUE if the font attributes have been changed and the font has to be (re)loaded; otherwise returns FALSE.

bool QFont::exactMatch () const

Returns TRUE if a window system font exactly matching the settings of this font is available.

See also QFontInfo [p. 85].

QString QFont::family () const

Returns the requested font family name, i.e. the name set in the constructor or the last setFont() call.

See also setFamily() [p. 70], substitutes() [p. 73] and substitute() [p. 73].

Example: fonts/simple-qfont-demo/viewer.cpp.

bool QFont::fixedPitch () const

Returns TRUE if fixed pitch has been set; otherwise returns FALSE.

See also setFixedPitch() [p. 70] and QFontInfo::fixedPitch() [p. 86].

bool QFont::fromString (const QString & descrip)

Sets this font to match the description *descrip*. The description is a comma-separated list of the font attributes, as returned by toString().

See also toString() [p. 73] and operator>>() [p. 74].

HFONT QFont::handle () const

Returns the window system handle to the font, for low-level access. Using this function is *not* portable.

Example: i18n/main.cpp.

void QFont::insertSubstitution (const QString & familyName, const QString & substituteName) [static]

Inserts the family name *substituteName* into the substitution table for *familyName*.

See also insertSubstitutions() [p. 68], removeSubstitution() [p. 70], substitutions() [p. 73], substitute() [p. 73] and substitutes() [p. 73].

Example: fonts/simple-qfont-demo/viewer.cpp.

void QFont::insertSubstitutions (const QString & familyName, const QStringList & substituteNames) [static]

Inserts the list of families *substituteNames* into the substitution list for *familyName*.

See also insertSubstitution() [p. 67], removeSubstitution() [p. 70], substitutions() [p. 73] and substitute() [p. 73].

Example: fonts/simple-qfont-demo/viewer.cpp.

bool QFont::isCopyOf (const QFont & f) const

Returns TRUE if this font and *f* are copies of each other, i.e. one of them was created as a copy of the other and neither has been modified since. This is much stricter than equality.

See also operator=() [p. 69] and operator==() [p. 69].

bool QFont::italic () const

Returns TRUE if italic has been set; otherwise returns FALSE.

See also setItalic() [p. 70].

QString QFont::key () const

Returns the font's key, a textual representation of a font. It is typically used as the key for a cache or dictionary of fonts.

See also QMap [Datastructures and String Handling with Qt].

QString QFont::lastResortFamily () const

Returns the "last resort" font family name.

The current implementation tries a wide variety of common fonts, returning the first one it finds. Is is possible that no family is found in which case a null string is returned.

See also lastResortFont() [p. 68].

QString QFont::lastResortFont () const

Returns a "last resort" font name for the font matching algorithm. This is used if the last resort family is not available. It will always return a name, if necessary returning something like "fixed" or "system".

The current implementation tries a wide variety of common fonts, returning the first one it finds. This implementation may change at any time, but this function will always return a string containing something.

It is theoretically possible that there really isn't a lastResortFont() in which case Qt will abort with an error message. We have not been able to identify a case where this happens. Please report it as a bug if it does, preferably with a list of the fonts you have installed.

See also lastResortFamily() [p. 68] and rawName() [p. 70].

bool QFont::operator!= (const QFont & f) const

Returns TRUE if this font is different from *f*; otherwise returns FALSE.

Two QFonts are considered to be different if their font attributes are different. If rawMode() is enabled for both fonts, only the family fields are compared.

See also operator==() [p. 69].

QFont & QFont::operator= (const QFont & font)

Assigns *font* to this font and returns a reference to it.

bool QFont::operator== (const QFont & f) const

Returns TRUE if this font is equal to *f*; otherwise returns FALSE.

Two QFonts are considered equal if their font attributes are equal. If rawMode() is enabled for both fonts, only the family fields are compared.

See also operator!=() [p. 69] and isCopyOf() [p. 68].

int QFont::pixelSize () const

Returns the pixel size of the font if it was set with setPixelSize(). Returns -1 if the size was set with setPointSize() or setPointSizeFloat().

See also setPixelSize() [p. 71], pointSize() [p. 69], QFontInfo::pointSize() [p. 87] and QFontInfo::pixelSize() [p. 87].

int QFont::pointSize () const

Returns the point size of the font. Returns -1 if the font size was specified in pixels.

See also setPointSize() [p. 71], deciPointSize() [p. 66] and pointSizeFloat() [p. 69].

Example: fonts/simple-qfont-demo/viewer.cpp.

float QFont::pointSizeFloat () const

Returns the point size of the font. Returns -1 if the font size was specified in pixels.

See also pointSize() [p. 69], setPointSizeFloat() [p. 71], pixelSize() [p. 69], QFontInfo::pointSize() [p. 87] and QFontInfo::pixelSize() [p. 87].

bool QFont::rawMode () const

Returns TRUE if raw mode is used for font name matching; otherwise returns FALSE.

See also setRawMode() [p. 71] and rawName() [p. 70].

QString QFont::rawName () const

Returns the name of the font within the underlying window system. On Windows, this is usually just the family name of a truetype font. Under X, it is an XLFD (X Logical Font Description). Using the return value of this function is usually *not portable*.

See also setRawName() [p. 71].

void QFont::removeSubstitution (const QString & familyName) [static]

Removes all the substitutions for *familyName*.

See also insertSubstitutions() [p. 68], insertSubstitution() [p. 67], substitutions() [p. 73] and substitute() [p. 73].

void QFont::setBold (bool enable)

If *enable* is true sets the font's weight to QFont::Bold; otherwise sets the weight to QFont::Normal.

For finer boldness control use setWeight().

See also bold() [p. 66] and setWeight() [p. 72].

Examples: menu/menu.cpp and themes/metal.cpp.

void QFont::setDefaultFont (const QFont & f) [static]

This function is obsolete. It is provided to keep old source working. We strongly advise against using it in new code.

Please use QApplication::setFont() instead.

void QFont::setFamily (const QString & family)

Sets the family name of the font. The name is case insensitive and may include a foundry name.

The *family* name may optionally also include a foundry name, e.g. "Helvetica [Cronyx]". (The Qt 2.x syntax, i.e. "Cronyx-Helvetica", is also supported.) If the *family* is available from more than one foundry and the foundry isn't specified, an arbitrary foundry is chosen. If the family isn't available a family will be set using the font matching algorithm.

See also family() [p. 67], setStyleHint() [p. 72] and QFontInfo [p. 85].

void QFont::setFixedPitch (bool enable)

If *enable* is TRUE, sets fixed pitch on; otherwise sets fixed pitch off.

See also fixedPitch() [p. 67] and QFontInfo [p. 85].

void QFont::setItalic (bool enable)

If *enable* is TRUE, italic is set on; otherwise italic is set off.

See also italic() [p. 68] and QFontInfo [p. 85].

Examples: fileiconview/qfileiconview.cpp, fonts/simple-qfont-demo/viewer.cpp and themes/metal.cpp.

void QFont::setPixelSize (int pixelSize)

Sets the font size to *pixelSize* pixels.

Using this function makes the font device dependent. Use setPointSize() or setPointSizeFloat() to set the size of the font in a device independent manner.

See also pixelSize() [p. 69].

Example: qwerty/qwerty.cpp.

void QFont::setPixelSizeFloat (float pixelSize)

This function is obsolete. It is provided to keep old source working. We strongly advise against using it in new code.

Sets the logical pixel height of font characters when shown on the screen to *pixelSize*.

void QFont::setPointSize (int pointSize)

Sets the point size to *pointSize*. The point size must be greater than zero.

See also pointSize() [p. 69] and setPointSizeFloat() [p. 71].

Example: fonts/simple-qfont-demo/viewer.cpp.

void QFont::setPointSizeFloat (float pointSize)

Sets the point size to *pointSize*. The point size must be greater than zero. The requested precision may not be achieved on all platforms.

See also pointSizeFloat() [p. 69], setPointSize() [p. 71] and setPixelSize() [p. 71].

void QFont::setRawMode (bool enable)

If *enable* is TRUE, turns raw mode on; otherwise turns raw mode off. This function only has an affect under X11.

If raw mode is enabled, Qt will search for an X font with a complete font name matching the family name, ignoring all other values set for the QFont. If the font name matches several fonts, Qt will use the first font returned by X. QFontInfo *cannot* be used to fetch information about a QFont using raw mode (it will return the values set in the QFont for all parameters, including the family name).

Warning: Do not use raw mode unless you really, really need it! In most (if not all) cases, setRawName() is a much better choice.

See also rawMode() [p. 69] and setRawName() [p. 71].

void QFont::setRawName (const QString & name)

Sets a font by its system specific name. The function is in particular useful under X, where system font settings (for example X resources) are usually available in XLFD (X Logical Font Description) form only. You can pass an XLFD as *name* to this function.

In Qt 2.0 and later, a font set with setRawName() is still a full-featured QFont. It can be queried (for example with italic()) or modified (for example with setItalic()) and is therefore also suitable for rendering rich text.

If Qt's internal font database cannot resolve the raw name, the font becomes a raw font with *name* as its family.

Note that the present implementation does not handle wildcards in XLFDs well, and that font aliases (file `fonts.alias` in the font directory on X11) are not supported.

See also rawName() [p. 70], setRawMode() [p. 71] and setFamily() [p. 70].

void QFont::setStrikeOut (bool enable)

If *enable* is TRUE, sets strikeout on; otherwise sets strikeout off.

See also strikeOut() [p. 72] and QFontInfo [p. 85].

void QFont::setStyleHint (StyleHint hint, StyleStrategy strategy = PreferDefault)

Sets the style hint and strategy to *hint* and *strategy*, respectively.

If these aren't set explicitly the style hint will default to AnyStyle and the style strategy to PreferDefault.

See also StyleHint [p. 65], styleHint() [p. 72], StyleStrategy [p. 65], styleStrategy() [p. 73] and QFontInfo [p. 85].

Examples: desktop/desktop.cpp and fonts/simple-qfont-demo/viewer.cpp.

void QFont::setStyleStrategy (StyleStrategy s)

Sets the style strategy for the font to *s*.

See also QFont::StyleStrategy [p. 65].

void QFont::setUnderline (bool enable)

If *enable* is TRUE, sets underline on; otherwise sets underline off.

See also underline() [p. 73] and QFontInfo [p. 85].

Examples: fonts/simple-qfont-demo/viewer.cpp and menu/menu.cpp.

void QFont::setWeight (int weight)

Sets the weight the font to *weight*, which should be a value from the QFont::Weight enumeration.

See also weight() [p. 73] and QFontInfo [p. 85].

Example: fonts/simple-qfont-demo/viewer.cpp.

bool QFont::strikeOut () const

Returns TRUE if strikeout has been set; otherwise returns FALSE.

See also setStrikeOut() [p. 72] and QFontInfo::strikeOut().

StyleHint QFont::styleHint () const

Returns the StyleHint.

The style hint affects the font matching algorithm. See QFont::StyleHint for the list of strategies.

See also setStyleHint() [p. 72], QFont::StyleStrategy [p. 65] and QFontInfo::styleHint() [p. 87].

StyleStrategy QFont::styleStrategy () const

Returns the StyleStrategy.

The style strategy affects the font matching algorithm. See QFont::StyleStrategy for the list of strategies.

See also setStyleHint() [p. 72] and QFont::StyleHint [p. 65].

QString QFont::substitute (const QString & familyName) [static]

Returns the first family name to be used whenever *familyName* is specified. The lookup is case insensitive.

If there is no substitution for *familyName*, *familyName* is returned.

To obtain a list of substitutions use substitutes().

See also setFamily() [p. 70], insertSubstitutions() [p. 68], insertSubstitution() [p. 67] and removeSubstitution() [p. 70].

QStringList QFont::substitutes (const QString & familyName) [static]

Returns a list of family names to be used whenever *familyName* is specified. The lookup is case insensitive.

If there is no substitution for *familyName*, an empty list is returned.

See also substitute() [p. 73], insertSubstitutions() [p. 68], insertSubstitution() [p. 67] and removeSubstitution() [p. 70].

Example: fonts/simple-qfont-demo/viewer.cpp.

QStringList QFont::substitutions () [static]

Returns a sorted list of substituted family names.

See also insertSubstitution() [p. 67], removeSubstitution() [p. 70] and substitute() [p. 73].

QString QFont::toString () const

Returns a description of the font. The description is a comma-separated list of the attributes, perfectly suited for use in QSettings.

See also fromString() [p. 67] and operator<<() [p. 74].

bool QFont::underline () const

Returns TRUE if underline has been set; otherwise returns FALSE.

See also setUnderline() [p. 72] and QFontInfo::underline().

int QFont::weight () const

Returns the weight of the font which is one of the enumerated values from QFont::Weight.

See also setWeight() [p. 72], Weight [p. 65] and QFontInfo [p. 85].

Related Functions

QDataStream & operator<< (QDataStream & s, const QFont & font)

Writes the font *font* to the data stream *s*. (toString() writes to a text stream.)

See also Format of the QDataStream operators [Input/Output and Networking with Qt].

QDataStream & operator>> (QDataStream & s, QFont & font)

Reads the font *font* from the data stream *s*. (fromString() reads from a text stream.)

See also Format of the QDataStream operators [Input/Output and Networking with Qt].

QFontDatabase Class Reference

The QFontDatabase class provides information about the fonts available in the underlying window system.

```
#include <qfontdatabase.h>
```

Public Members

- **QFontDatabase** ()
- QStringList **families** () const
- QStringList **styles** (const QString & family) const
- QValueList<int> **pointSizes** (const QString & family, const QString & style = QString::null)
- QValueList<int> **smoothSizes** (const QString & family, const QString & style)
- QString **styleString** (const QFont & f)
- QFont **font** (const QString & family, const QString & style, int pointSize)
- bool **isBitmapScalable** (const QString & family, const QString & style = QString::null) const
- bool **isSmoothlyScalable** (const QString & family, const QString & style = QString::null) const
- bool **isScalable** (const QString & family, const QString & style = QString::null) const
- bool **isFixedPitch** (const QString & family, const QString & style = QString::null) const
- bool **italic** (const QString & family, const QString & style) const
- bool **bold** (const QString & family, const QString & style) const
- int **weight** (const QString & family, const QString & style) const
- QStringList families (bool) const *(obsolete)*
- QStringList styles (const QString & family, const QString &) const *(obsolete)*
- QValueList<int> pointSizes (const QString & family, const QString & style, const QString &) *(obsolete)*
- QValueList<int> smoothSizes (const QString & family, const QString & style, const QString &) *(obsolete)*
- QFont font (const QString & familyName, const QString & style, int pointSize, const QString &) *(obsolete)*
- bool isBitmapScalable (const QString & family, const QString & style, const QString &) const *(obsolete)*
- bool isSmoothlyScalable (const QString & family, const QString & style, const QString &) const *(obsolete)*
- bool isScalable (const QString & family, const QString & style, const QString &) const *(obsolete)*
- bool isFixedPitch (const QString & family, const QString & style, const QString &) const *(obsolete)*
- bool italic (const QString & family, const QString & style, const QString &) const *(obsolete)*
- bool bold (const QString & family, const QString & style, const QString &) const *(obsolete)*
- int weight (const QString & family, const QString & style, const QString &) const *(obsolete)*

Static Public Members

- QValueList<int> **standardSizes** ()
- QString **scriptName** (QFont::Script script)
- QString **scriptSample** (QFont::Script script)

Detailed Description

The QFontDatabase class provides information about the fonts available in the underlying window system.

The most common uses of this class are to query the database for the list of font families() and the pointSizes() and styles() that are available for each family. An alternative to pointSizes() is smoothSizes() which returns the sizes at which a given family and style will look attractive.

If the font family is available from two or more foundries the foundry name is included in the family name, e.g. "Helvetica [Adobe]" and "Helvetica [Cronyx]". When you specify a family you can either use the hyphenated "foundry-family" format, e.g. "Cronyx-Helvetica", or the bracketed format, e.g. "Helvetica [Cronyx]". If the family has a foundry it is always returned, e.g. by families(), using the bracketed format.

The font() function returns a QFont given a family, style and point size.

A family and style combination can be checked to see if it is italic() or bold(), and to retrieve its weight(). Similarly we can call isBitmapScalable(), isSmoothlyScalable(), isScalable() and isFixedPitch().

A text version of a style is given by styleString().

The QFontDatabase class also supports some static functions, for example, standardSizes(). You can retrieve the Unicode 3.0 description of a script using scriptName(), and a sample of characters in a script with scriptSample().

Example:

```
#include <qapplication.h>
#include <qfontdatabase.h>

int main( int argc, char **argv )
{
    QApplication app( argc, argv );
    QFontDatabase fdb;
    QStringList families = fdb.families();
    for ( QStringList::Iterator f = families.begin(); f != families.end(); ++f ) {
        QString family = *f;
        qDebug( family );
        QStringList styles = fdb.styles( family );
        for ( QStringList::Iterator s = styles.begin(); s != styles.end(); ++s ) {
            QString style = *s;
            QString dstyle = "\t" + style + " (";
            QValueList smoothies = fdb.smoothSizes( family, style );
            for ( QValueList::Iterator points = smoothies.begin(); points != smoothies.end(); ++points ) {
                dstyle += QString::number( *points ) + " ";
            }
            dstyle = dstyle.left( dstyle.length() - 1 ) + ")";
            qDebug( dstyle );
        }
    }
    return 0;
}
```

This example gets the list of font families, then the list of styles for each family and the point sizes that are available for each family/style combination.

See also Environment Classes and Graphics Classes.

Member Function Documentation

QFontDatabase::QFontDatabase ()

Creates a font database object.

bool QFontDatabase::bold (const QString & family, const QString & style) const

Returns TRUE if the font that has family *family* and style *style* is bold; otherwise returns FALSE.

See also italic() [p. 78] and weight() [p. 80].

bool QFontDatabase::bold (const QString & family, const QString & style, const QString &) const

This function is obsolete. It is provided to keep old source working. We strongly advise against using it in new code.

QStringList QFontDatabase::families () const

Returns a list of the names of the available font families.

If a family exists in several foundries, the returned name for that font is in the form "family [foundry]". Examples: "Times [Adobe]", "Times [Cronyx]", "Palatino".

QStringList QFontDatabase::families (bool) const

This function is obsolete. It is provided to keep old source working. We strongly advise against using it in new code.

QFont QFontDatabase::font (const QString & family, const QString & style, int pointSize)

Returns a QFont object that has family *family*, style *style* and point size *pointSize*. If no matching font could be created, a QFont object that uses the application's default font is returned.

QFont QFontDatabase::font (const QString & familyName, const QString & style, int pointSize, const QString &)

This function is obsolete. It is provided to keep old source working. We strongly advise against using it in new code.

bool QFontDatabase::isBitmapScalable (const QString & family, const QString & style = QString::null) const

Returns TRUE if the font that has family *family* and style *style* is a scalable bitmap font; otherwise returns FALSE. Scaling a bitmap font usually produces an unattractive hardly readable result, because the pixels of the font are scaled. If you need to scale a bitmap font it is better to scale it to one of the fixed sizes returned by smoothSizes().

See also isScalable() [p. 78] and isSmoothlyScalable() [p. 78].

bool QFontDatabase::isBitmapScalable (const QString & family, const QString & style, const QString &) const

This function is obsolete. It is provided to keep old source working. We strongly advise against using it in new code.

bool QFontDatabase::isFixedPitch (const QString & family, const QString & style = QString::null) const

Returns TRUE if the font that has family *family* and style *style* is fixed pitch; otherwise returns FALSE.

bool QFontDatabase::isFixedPitch (const QString & family, const QString & style, const QString &) const

This function is obsolete. It is provided to keep old source working. We strongly advise against using it in new code.

bool QFontDatabase::isScalable (const QString & family, const QString & style = QString::null) const

Returns TRUE if the font that has family *family* and style *style* is scalable; otherwise returns FALSE.

See also isBitmapScalable() [p. 77] and isSmoothlyScalable() [p. 78].

bool QFontDatabase::isScalable (const QString & family, const QString & style, const QString &) const

This function is obsolete. It is provided to keep old source working. We strongly advise against using it in new code.

bool QFontDatabase::isSmoothlyScalable (const QString & family, const QString & style = QString::null) const

Returns TRUE if the font that has family *family* and style *style* is smoothly scalable; otherwise returns FALSE. If this function returns TRUE, it's safe to scale this font to any size, and the result will always look attractive.

See also isScalable() [p. 78] and isBitmapScalable() [p. 77].

bool QFontDatabase::isSmoothlyScalable (const QString & family, const QString & style, const QString &) const

This function is obsolete. It is provided to keep old source working. We strongly advise against using it in new code.

bool QFontDatabase::italic (const QString & family, const QString & style) const

Returns TRUE if the font that has family *family* and style *style* is italic; otherwise returns FALSE.

See also weight() [p. 80] and bold() [p. 77].

bool QFontDatabase::italic (const QString & family, const QString & style, const QString &) const

This function is obsolete. It is provided to keep old source working. We strongly advise against using it in new code.

QValueList<int> QFontDatabase::pointSizes (const QString & family, const QString & style = QString::null)

Returns a list of the point sizes available for the font that has family *family* and style *style*. The list may be empty.

See also smoothSizes() [p. 79] and standardSizes() [p. 79].

QValueList<int> QFontDatabase::pointSizes (const QString & family, const QString & style, const QString &)

This function is obsolete. It is provided to keep old source working. We strongly advise against using it in new code.

QString QFontDatabase::scriptName (QFont::Script script) [static]

Returns a string that gives a default description of the *script* (e.g. for displaying to the user in a dialog). The name matches the name of the script as indicated by the Unicode 3.0 standard.

See also QFont::Script [p. 62].

QString QFontDatabase::scriptSample (QFont::Script script) [static]

Returns a string with sample characters from *script*.

See also QFont::Script [p. 62].

QValueList<int> QFontDatabase::smoothSizes (const QString & family, const QString & style)

Returns the point sizes of a font that has family *family* and style *style* that will look attractive. The list may be empty. For non-scalable fonts and smoothly scalable fonts, this function is equivalent to pointSizes().

See also pointSizes() [p. 79] and standardSizes() [p. 79].

QValueList<int> QFontDatabase::smoothSizes (const QString & family, const QString & style, const QString &)

This function is obsolete. It is provided to keep old source working. We strongly advise against using it in new code.

QValueList<int> QFontDatabase::standardSizes () [static]

Returns a list of standard font sizes.

See also smoothSizes() [p. 79] and pointSizes() [p. 79].

QString QFontDatabase::styleString (const QFont & f)

Returns a string that describes the style of the font *f*. For example, "Bold Italic", "Bold", "Italic" or "Normal". An empty string may be returned.

QStringList QFontDatabase::styles (const QString & family) const

Returns a list of the styles available for the font family, *family*. Some example styles: "Light", "Light Italic", "Bold", "Oblique", "Demi". The list may be empty.

QStringList QFontDatabase::styles (const QString & family, const QString &) const

This function is obsolete. It is provided to keep old source working. We strongly advise against using it in new code.

int QFontDatabase::weight (const QString & family, const QString & style) const

Returns the weight of the font that has family *family* and style *style*. If there is no such family and style combination, returns -1.

See also italic() [p. 78] and bold() [p. 77].

int QFontDatabase::weight (const QString & family, const QString & style, const QString &) const

This function is obsolete. It is provided to keep old source working. We strongly advise against using it in new code.

QFontDialog Class Reference

The QFontDialog class provides a dialog widget for selecting a font.

`#include <qfontdialog.h>`

Inherits QDialog [Dialogs and Windows with Qt].

Signals

- void **fontSelected** (const QFont & font)
- void **fontHighlighted** (const QFont & font)

Static Public Members

- QFont **getFont** (bool * ok, const QFont & initial, QWidget * parent = 0, const char * name = 0)
- QFont **getFont** (bool * ok, QWidget * parent = 0, const char * name = 0)

Protected Members

- virtual bool **eventFilter** (QObject * o, QEvent * e)
- QListBox * **familyListBox** () const
- virtual void **updateFamilies** ()
- QListBox * **styleListBox** () const
- virtual void **updateStyles** ()
- QListBox * **sizeListBox** () const
- virtual void **updateSizes** ()
- QComboBox * **scriptCombo** () const
- virtual void **updateScripts** ()

Protected Slots

- void **sizeChanged** (const QString & s)

Detailed Description

The QFontDialog class provides a dialog widget for selecting a font.

The usual way to use this class is to call one of the static convenience functions, getFont(), e.g.

Examples:

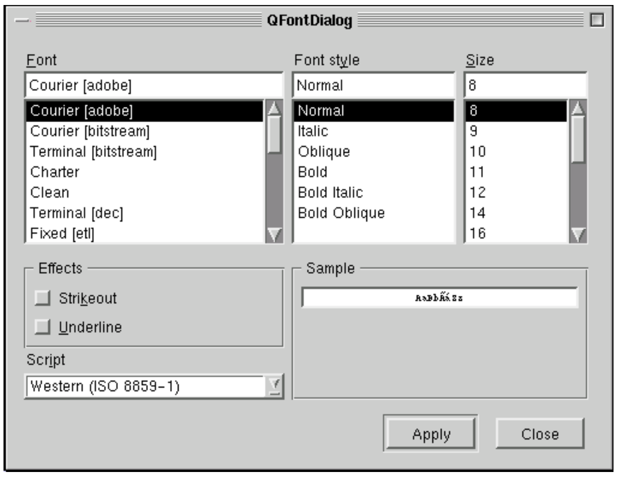
```
bool ok;
QFont font = QFontDialog::getFont( &ok, QFont( "Helvetica [Cronyx]", 10 ), this );
if ( ok ) {
    // font is set to the font the user selected
} else {
    // the user cancelled the dialog; font is set to the initial
    // value, in this case Helvetica [Cronyx], 10
}
```

The dialog can also be used to set a widget's font directly:

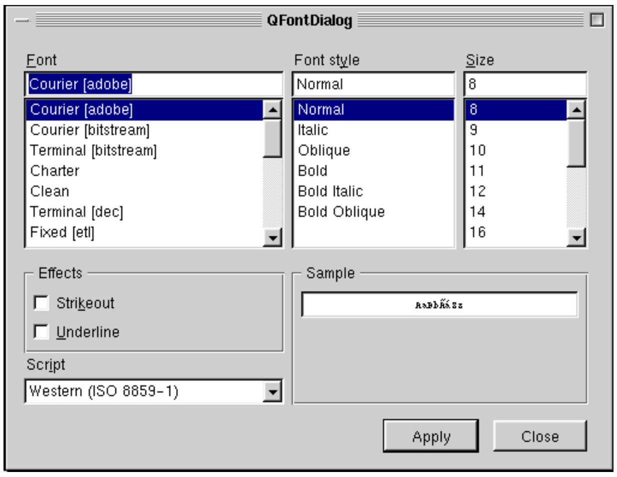
```
aWidget.setFont( QFontDialog::getFont( 0, aWidget.font() ) );
```

If the user clicks OK the font they chose will be used for aWidget, and if they click cancel the original font is kept.

See also QFont [p. 59], QFontInfo [p. 85], QFontMetrics [p. 90] and Dialog Classes.



Font: Courier [adobe]
Font style: Normal
Size: 8
Script: Western (ISO 8859-1)



Font: Courier [adobe]
Font style: Normal
Size: 8
Script: Western (ISO 8859-1)

Member Function Documentation

bool QFontDialog::eventFilter (QObject * o, QEvent * e) [virtual protected]

An event filter to make the Up, Down, PageUp and PageDown keys work correctly in the line edits. The source of the event is the object *o* and the event is *e*.

QListBox * QFontDialog::familyListBox () const [protected]

Returns a pointer to the "font family" list box. This is mainly useful if you reimplement updateFontFamilies();

void QFontDialog::fontHighlighted (const QFont & font) [signal]

This signal is emitted when the user changed a setting in the dialog. The font that is highlighted is passed in *font*.

void QFontDialog::fontSelected (const QFont & font) [signal]

This signal is emitted when the user has chosen a font and clicked OK. The font that was selected is passed in *font*.

QFont QFontDialog::getFont (bool * ok, const QFont & initial, QWidget * parent = 0, const char * name = 0) [static]

Executes a modal font dialog and returns a font.

If the user clicks OK, the selected font is returned. If the user clicks Cancel, the *initial* font is returned.

The dialog has parent *parent* and is called *name*. *initial* is the initial selected font. If the *ok* parameter is not-null, **ok* is set to TRUE if the user clicked OK, and set to FALSE if the user clicked Cancel.

This static function is less flexible than the full QFontDialog object, but is convenient and easy to use.

Examples:

```
bool ok;
QFont font = QFontDialog::getFont( &ok, QFont( "Times", 12 ), this );
if ( ok ) {
    // font is set to the font the user selected
} else {
    // the user cancelled the dialog; font is set to the initial
    // value, in this case Times, 12.
}
```

The dialog can also be used to set a widget's font directly:

```
myWidget.setFont( QFontDialog::getFont( 0, myWidget.font() ) );
```

In this example, if the user clicks OK the font they chose will be used, and if they click cancel the original font is kept.

Examples: qfd/fontdisplayer.cpp, qwerty/qwerty.cpp and xform/xform.cpp.

QFont QFontDialog::getFont (bool * ok, QWidget * parent = 0, const char * name = 0) [static]

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Executes a modal font dialog and returns a font.

If the user clicks OK, the selected font is returned. If the user clicks Cancel, the Qt default font is returned.

The dialog has parent *parent* and is called *name*. If the *ok* parameter is not-null, ** ok* is set to TRUE if the user clicked OK, and FALSE if the user clicked Cancel.

This static function is less functional than the full QFontDialog object, but is convenient and easy to use.

Example:

```
bool ok;
QFont font = QFontDialog::getFont( &ok, this );
if ( ok ) {
    // font is set to the font the user selected
} else {
    // the user cancelled the dialog; font is set to the default
    // application font, QApplication::font()
}
```

QComboBox * QFontDialog::scriptCombo () const [protected]

Returns a pointer to the "font style" list box. This is mainly useful if you reimplement updateFontStyles();

void QFontDialog::sizeChanged (const QString & s) [protected slot]

This slot is called if the user changes the font size. The size is passed in the *s* argument as a *string*.

QListBox * QFontDialog::sizeListBox () const [protected]

Returns a pointer to the "font size" list box. This is mainly useful if you reimplement updateFontSizes();

QListBox * QFontDialog::styleListBox () const [protected]

Returns a pointer to the "font style" list box. This is mainly useful if you reimplement updateFontStyles();

void QFontDialog::updateFamilies () [virtual protected]

Updates the contents of the "font family" list box. This function can be reimplemented if you have special requirements.

void QFontDialog::updateScripts () [virtual protected]

Updates the contents of the "font script" combo box. This function can be reimplemented if you have special requirements.

void QFontDialog::updateSizes () [virtual protected]

Updates the contents of the "font size" list box. This function can be reimplemented if you have special requirements.

void QFontDialog::updateStyles () [virtual protected]

Updates the contents of the "font style" list box. This function can be reimplemented if you have special requirements.

QFontInfo Class Reference

The QFontInfo class provides general information about fonts.

`#include <qfontinfo.h>`

Public Members

- **QFontInfo** (const QFont & font)
- **QFontInfo** (const QFontInfo & fi)
- **~QFontInfo** ()
- QFontInfo & **operator=** (const QFontInfo & fi)
- QString **family** () const
- int **pixelSize** () const
- int **pointSize** () const
- bool **italic** () const
- int **weight** () const
- bool **bold** () const
- bool **fixedPitch** () const
- QFont::StyleHint **styleHint** () const
- bool **rawMode** () const
- bool **exactMatch** () const

Detailed Description

The QFontInfo class provides general information about fonts.

The QFontInfo class provides the same access functions as QFont, e.g. family(), pointSize(), italic(), weight(), fixedPitch(), styleHint() etc. But whilst the QFont access functions return the values that were set, a QFontInfo object returns the values that apply to the font that will actually be used to draw the text.

For example, when the program asks for a 25pt Courier font on a machine that has a 24pt Courier font but not a scalable one, QFont will (normally) use the 24pt Courier for rendering. In this case, QFont::pointSize() returns 25 and QFontInfo::pointSize() 24.

There are three ways to create a QFontInfo object.

1. Calling the QFontInfo constructor with a QFont creates a font info object for a screen-compatible font, i.e. the font cannot be a printer font*. If the font is changed later, the font info object is *not* updated.
2. QWidget::fontInfo() returns the font info for a widget's font. This is equivalent to calling QFontInfo(widget->font()). If the widget's font is changed later, the font info object is *not* updated.
3. QPainter::fontInfo() returns the font info for a painter's current font. The font info object is *automatically* updated if you set a new painter font.

* If you use a printer font the values returned will almost certainly be inaccurate. Printer fonts are not always accessible so the nearest screen font is used if a printer font is supplied.

See also QFont [p. 59], QFontMetrics [p. 90], QFontDatabase [p. 75], Graphics Classes and Implicitly and Explicitly Shared Classes.

Member Function Documentation

QFontInfo::QFontInfo (const QFont & font)

Constructs a font info object for *font*.

The font must be screen-compatible, i.e. a font you use when drawing text in widgets or pixmaps.

The font info object holds the information for the font that is passed in the constructor at the time it is created, and is not updated if the font's attributes are changed later.

Use the QPainter::fontInfo() function to get the font info when painting. This is a little slower than using this constructor, but it always gives correct results because the font info data is updated.

QFontInfo::QFontInfo (const QFontInfo & fi)

Constructs a copy of *fi*.

QFontInfo::~QFontInfo ()

Destroys the font info object.

bool QFontInfo::bold () const

Returns TRUE if weight() would return a value greater than QFont::Normal; otherwise returns FALSE.

See also weight() [p. 87] and QFont::bold() [p. 66].

bool QFontInfo::exactMatch () const

Returns TRUE if the matched window system font is exactly the same as the one specified by the font.

See also QFont::exactMatch() [p. 67].

QString QFontInfo::family () const

Returns the family name of the matched window system font.

See also QFont::family() [p. 67].

Example: fonts/simple-qfont-demo/viewer.cpp.

bool QFontInfo::fixedPitch () const

Returns the fixed pitch value of the matched window system font.

See also QFont::fixedPitch() [p. 67].

bool QFontInfo::italic () const

Returns the italic value of the matched window system font.

See also QFont::italic() [p. 68].

QFontInfo & QFontInfo::operator= (const QFontInfo & fi)

Assigns the font info in *fi*.

int QFontInfo::pixelSize () const

Returns the pixel size of the matched window system font.

See also QFont::pointSize() [p. 69].

int QFontInfo::pointSize () const

Returns the point size of the matched window system font.

See also QFont::pointSize() [p. 69].

Example: fonts/simple-qfont-demo/viewer.cpp.

bool QFontInfo::rawMode () const

Returns TRUE if the font is a raw mode font.

If it is a raw mode font, all other functions in QFontInfo will return the same values set in the QFont, regardless of the font actually used.

See also QFont::rawMode() [p. 69].

QFont::StyleHint QFontInfo::styleHint () const

Returns the style of the matched window system font.

Currently only returns the style hint set in QFont.

See also QFont::styleHint() [p. 72] and QFont::StyleHint [p. 65].

int QFontInfo::weight () const

Returns the weight of the matched window system font.

See also QFont::weight() [p. 73] and bold() [p. 86].

QFontManager Class Reference

The QFontManager class implements font management in Qt/Embedded.

```
#include <qfontmanager_qws.h>
```

Public Members

- **QFontManager** ()
- **~QFontManager** ()
- QDiskFont * **get** (const QFontDef & f)

Static Public Members

- void **initialize** ()
- void **cleanup** ()

Detailed Description

The QFontManager class implements font management in Qt/Embedded.

There is one and only one QFontManager per Qt/Embedded application (qt_fontmanager is a global variable that points to it). It keeps a list of font factories, a cache of rendered fonts and a list of fonts available on disk. QFontManager is called when a new font needs to be rendered from a Freetype-compatible or BDF font on disk; this only happens if there isn't an appropriate QPF font already available.

See also Qt/Embedded.

Member Function Documentation

QFontManager::QFontManager ()

Creates a font manager. This method reads in the font definition file from $QTDIR/lib/fonts/fontdir (or /usr/local/qt-embedded/lib/fonts/fontdir if QTDIR isn't defined) and creates a list of QDiskFonts to hold the information in the file. It also constructs any defined font factories.

QFontManager::~QFontManager ()

Destroys the QFontManager and sets qt_fontmanager to 0.

void QFontManager::cleanup () [static]

Destroys the font manager

QDiskFont * QFontManager::get (const QFontDef & f)

Returns the QDiskFont that best matches *f*, based on family, weight, italicity and font size.

void QFontManager::initialize () [static]

Creates a new QFontManager and points qt_fontmanager to it

QFontMetrics Class Reference

The QFontMetrics class provides font metrics information.

`#include <qfontmetrics.h>`

Public Members

- **QFontMetrics** (const QFont & font)
- **QFontMetrics** (const QFontMetrics & fm)
- **~QFontMetrics** ()
- QFontMetrics & **operator=** (const QFontMetrics & fm)
- int **ascent** () const
- int **descent** () const
- int **height** () const
- int **leading** () const
- int **lineSpacing** () const
- int **minLeftBearing** () const
- int **minRightBearing** () const
- int **maxWidth** () const
- bool **inFont** (QChar ch) const
- int **leftBearing** (QChar ch) const
- int **rightBearing** (QChar ch) const
- int **width** (const QString & str, int len = -1) const
- int **width** (QChar ch) const
- int width (char c) const *(obsolete)*
- int **charWidth** (const QString & str, int pos) const
- QRect **boundingRect** (const QString & str, int len = -1) const
- QRect **boundingRect** (QChar ch) const
- QRect **boundingRect** (int x, int y, int w, int h, int flgs, const QString & str, int len = -1, int tabstops = 0, int * tabarray = 0, QTextParag ** intern = 0) const
- QSize **size** (int flgs, const QString & str, int len = -1, int tabstops = 0, int * tabarray = 0, QTextParag ** intern = 0) const
- int **underlinePos** () const
- int **strikeOutPos** () const
- int **lineWidth** () const

Detailed Description

The QFontMetrics class provides font metrics information.

QFontMetrics functions calculate size of characters and strings for a given font. There are three ways you can create a QFontMetrics object:

1. Calling the QFontMetrics constructor with a QFont creates a font metrics object for a screen-compatible font, i.e. the font cannot be a printer font*. If the font is changed later, the font metrics object is *not* updated.
2. QWidget::fontMetrics() returns the font metrics for a widget's font. This is equivalent to QFontMetrics(widget->font()). If the widget's font is changed later, the font metrics object is *not* updated.
3. QPainter::fontMetrics() returns the font metrics for a painter's current font. The font metrics object is *automatically* updated if you set a new painter font.

* If you use a printer font the values returned will almost certainly be inaccurate. Printer fonts are not always accessible so the nearest screen font is used if a printer font is supplied.

Once created, the object provides functions to access the individual metrics of the font, its characters, and for strings rendered in the font.

There are several functions that operate on the font: ascent(), descent(), height(), leading() and lineSpacing() return the basic size properties of the font. The underlinePos(), strikeOutPos() and lineWidth() functions, return the properties of the line that underlines or strikes out the characters. These functions are all fast.

There are also some functions that operate on the set of glyphs in the font: minLeftBearing(), minRightBearing() and maxWidth(). These are by necessity slow, and we recommend avoiding them if possible.

For each character, you can get its width(), leftBearing() and rightBearing() and find out whether it is in the font using inFont(). You can also treat the character as a string, and use the string functions on it.

The string functions include width(), to return the width of a string in pixels (or points, for a printer), boundingRect(), to return a rectangle large enough to contain the rendered string, and size(), to return the size of that rectangle.

Example:

```
    QFont font( "times", 24 );
    QFontMetrics fm (font );
    int pixelsWide = fm.width( "What's the width of this text?" );
    int pixelsHigh = fm.height();
```

See also QFont [p. 59], QFontInfo [p. 85], QFontDatabase [p. 75], Graphics Classes and Implicitly and Explicitly Shared Classes.

Member Function Documentation

QFontMetrics::QFontMetrics (const QFont & font)

Constructs a font metrics object for *font*.

The font must be screen-compatible, i.e. a font you use when drawing text in QWidget or QPixmap objects, not QPicture or QPrinter.

The font metrics object holds the information for the font that is passed in the constructor at the time it is created, and is not updated if the font's attributes are changed later.

Use QPainter::fontMetrics() to get the font metrics when painting. This is a little slower than using this constructor, but it always gives correct results because the font info data is updated.

QFontMetrics::QFontMetrics (const QFontMetrics & fm)

Constructs a copy of *fm*.

QFontMetrics::~QFontMetrics ()

Destroys the font metrics object and frees all allocated resources.

int QFontMetrics::ascent () const

Returns the maximum ascent of the font.

The ascent is the distance from the base line to the uppermost line where pixels may be drawn.

See also descent() [p. 93].

Examples: drawdemo/drawdemo.cpp and scrollview/scrollview.cpp.

QRect QFontMetrics::boundingRect (const QString & str, int len = -1) const

Returns the bounding rectangle of the first *len* characters of *str*, which is the set of pixels the text would cover if drawn at (0,0).

If *len* is negative (the default), the entire string is used.

Note that the bounding rectangle may extend to the left of (0,0), e.g. for italicized fonts, and that the text output may cover *all* pixels in the bounding rectangle.

Newline characters are processed as normal characters, *not* as linebreaks.

Due to the different actual character heights, the height of the bounding rectangle of e.g. "Yes" and "yes" may be different.

See also width() [p. 96] and QPainter::boundingRect() [Graphics with Qt].

Example: xform/xform.cpp.

QRect QFontMetrics::boundingRect (QChar ch) const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns the bounding rectangle of the character *ch* relative to the left-most point on the base line.

Note that the bounding rectangle may extend to the left of (0,0), e.g. for italicized fonts, and that the text output may cover *all* pixels in the bounding rectangle.

Note that the rectangle usually extends both above and below the base line.

See also width() [p. 96].

QRect QFontMetrics::boundingRect (int x, int y, int w, int h, int flgs, const QString & str, int len = -1, int tabstops = 0, int * tabarray = 0, QTextParag ** intern = 0) const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns the bounding rectangle of the first *len* characters of *str*, which is the set of pixels the text would cover if drawn at (0,0). The drawing, and hence the bounding rectangle, is constrained to the rectangle (*x*, *y*, *w*, *h*).

If *len* is negative (which is the default), the entire string is used.

The *flgs* argument is the bitwise OR of the following flags:

- `AlignAuto` aligns to the left border for all languages except Hebrew and Arabic where it aligns to the right.
- `AlignLeft` aligns to the left border.
- `AlignRight` aligns to the right border.
- `AlignJustify` produces justified text.
- `AlignHCenter` aligns horizontally centered.
- `AlignTop` aligns to the top border.
- `AlignBottom` aligns to the bottom border.
- `AlignVCenter` aligns vertically centered
- `AlignCenter` (= `AlignHCenter` | AlignVCenter)
- `SingleLine` ignores newline characters in the text.
- `ExpandTabs` expands tabs (see below)
- `ShowPrefix` interprets "&x" as "x" underlined.
- `WordBreak` breaks the text to fit the rectangle.

Horizontal alignment defaults to AlignAuto and vertical alignment defaults to AlignTop.

If several of the horizontal or several of the vertical alignment flags are set, the resulting alignment is undefined.

These flags are defined in qnamespace.h.

If `ExpandTabs` is set in *flgs*, then: if *tabarray* is non-null, it specifies a 0-terminated sequence of pixel-positions for tabs; otherwise if *tabstops* is non-zero, it is used as the tab spacing (in pixels).

Note that the bounding rectangle may extend to the left of (0,0), e.g. for italicized fonts, and that the text output may cover *all* pixels in the bounding rectangle.

Newline characters are processed as linebreaks.

Despite the different actual character heights, the heights of the bounding rectangles of "Yes" and "yes" are the same.

The bounding rectangle given by this function is somewhat larger than that calculated by the simpler boundingRect() function. This function uses the maximum left and right font bearings as is necessary for multi-line text to align correctly. Also, fontHeight() and lineSpacing() are used to calculate the height, rather than individual character heights.

The *intern* argument should not be used.

See also width() [p. 96], QPainter::boundingRect() [Graphics with Qt] and Qt::AlignmentFlags [p. 179].

int QFontMetrics::charWidth (const QString & str, int pos) const

Returns the width of the character at position *pos* in the string *str*.

The whole string is needed, as the glyph drawn may change depending on the context (the letter before and after the current one) for some languages (e.g. Arabic).

This function also takes non spacing marks and ligatures into account.

int QFontMetrics::descent () const

Returns the maximum descent of the font.

The descent is the distance from the base line to the lowermost line where pixels may be drawn. (Note that this is different from X, which adds 1 pixel.)

See also ascent() [p. 92].

Examples: drawdemo/drawdemo.cpp and hello/hello.cpp.

int QFontMetrics::height () const

Returns the height of the font.

This is always equal to ascent()+descent()+1 (the 1 is for the base line).

See also leading() [p. 94] and lineSpacing() [p. 94].

Examples: grapher/grapher.cpp, hello/hello.cpp and qfd/fontdisplayer.cpp.

bool QFontMetrics::inFont (QChar ch) const

Returns TRUE if character *ch* is a valid character in the font; otherwise returns FALSE.

Example: qfd/fontdisplayer.cpp.

int QFontMetrics::leading () const

Returns the leading of the font.

This is the natural inter-line spacing.

See also height() [p. 94] and lineSpacing() [p. 94].

int QFontMetrics::leftBearing (QChar ch) const

Returns the left bearing of character *ch* in the font.

The left bearing is the right-ward distance of the left-most pixel of the character from the logical origin of the character. This value is negative if the pixels of the character extend to the left of the logical origin.

See width(QChar) for a graphical description of this metric.

See also rightBearing() [p. 95], minLeftBearing() [p. 95] and width() [p. 96].

Example: qfd/fontdisplayer.cpp.

int QFontMetrics::lineSpacing () const

Returns the distance from one base line to the next.

This value is always equal to leading()+height().

See also height() [p. 94] and leading() [p. 94].

Examples: action/application.cpp, application/application.cpp, mdi/application.cpp, qfd/fontdisplayer.cpp, qwerty/qwerty.cpp and scrollview/scrollview.cpp.

int QFontMetrics::lineWidth () const

Returns the width of the underline and strikeout lines, adjusted for the point size of the font.

See also underlinePos() [p. 96] and strikeOutPos() [p. 96].

int QFontMetrics::maxWidth () const

Returns the width of the widest character in the font.

Example: qfd/fontdisplayer.cpp.

int QFontMetrics::minLeftBearing () const

Returns the minimum left bearing of the font.

This is the smallest leftBearing(char) of all characters in the font.

Note that this function can be very slow if the font is large.

See also minRightBearing() [p. 95] and leftBearing() [p. 94].

Example: qfd/fontdisplayer.cpp.

int QFontMetrics::minRightBearing () const

Returns the minimum right bearing of the font.

This is the smallest rightBearing(char) of all characters in the font.

Note that this function can be very slow if the font is large.

See also minLeftBearing() [p. 95] and rightBearing() [p. 95].

Example: qfd/fontdisplayer.cpp.

QFontMetrics & QFontMetrics::operator= (const QFontMetrics & fm)

Assigns the font metrics *fm*.

int QFontMetrics::rightBearing (QChar ch) const

Returns the right bearing of character *ch* in the font.

The right bearing is the left-ward distance of the right-most pixel of the character from the logical origin of a subsequent character. This value is negative if the pixels of the character extend to the right of the width() of the character.

See width() for a graphical description of this metric.

See also leftBearing() [p. 94], minRightBearing() [p. 95] and width() [p. 96].

Example: qfd/fontdisplayer.cpp.

QSize QFontMetrics::size (int flgs, const QString & str, int len = -1, int tabstops = 0, int * tabarray = 0, QTextParag ** intern = 0) const

Returns the size in pixels of the first *len* characters of *str*.

If *len* is negative (the default), the entire string is used.

The *flgs* argument is the bitwise OR of the following flags:

- `SingleLine` ignores newline characters.
- `ExpandTabs` expands tabs (see below)
- `ShowPrefix` interprets "&x" as "x" underlined.
- `WordBreak` breaks the text to fit the rectangle.

These flags are defined in qnamespace.h.

If `ExpandTabs` is set in *flgs*, then: if *tabarray* is non-null, it specifies a 0-terminated sequence of pixel-positions for tabs; otherwise if *tabstops* is non-zero, it is used as the tab spacing (in pixels).

Newline characters are processed as linebreaks.

Despite the different actual character heights, the heights of the bounding rectangles of "Yes" and "yes" are the same.

The *intern* argument should not be used.

See also boundingRect() [p. 92].

int QFontMetrics::strikeOutPos () const

Returns the distance from the base line to where the strikeout line should be drawn.

See also underlinePos() [p. 96] and lineWidth() [p. 94].

int QFontMetrics::underlinePos () const

Returns the distance from the base line to where an underscore should be drawn.

See also strikeOutPos() [p. 96] and lineWidth() [p. 94].

int QFontMetrics::width (const QString & str, int len = -1) const

Returns the width in pixels of the first *len* characters of *str*. If *len* is negative (the default), the entire string is used.

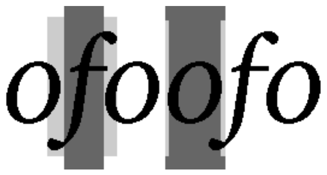
Note that this value is *not* equal to boundingRect().width(); boundingRect() returns a rectangle describing the pixels this string will cover whereas width() returns the distance to where the next string should be drawn.

See also boundingRect() [p. 92].

Examples: drawdemo/drawdemo.cpp, hello/hello.cpp, movies/main.cpp, qfd/fontdisplayer.cpp and scrollview/scrollview.cpp.

int QFontMetrics::width (QChar ch) const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.



Returns the logical width of character *ch* in pixels. This is a distance appropriate for drawing a subsequent character after *ch*.

Some of the metrics are described in the image to the right. The tall dark rectangle covers the logical width() of a character. The shorter pale rectangles cover leftBearing() and rightBearing() of the characters. Notice that the bearings of "f" in this particular font are both negative, while the bearings of "o" are both positive.

Warning: This function will produce incorrect results for Arabic characters or non spacing marks in the middle of a string, as the glyph shaping and positioning of marks that happens when processing strings cannot be taken into account. Use charWidth() instead if you aren't looking for the width of isolated characters.

See also boundingRect() [p. 92] and charWidth() [p. 93].

int QFontMetrics::width (char c) const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

This function is obsolete. It is provided to keep old source working. We strongly advise against using it in new code.

Provided to aid porting from Qt 1.x.

QGuardedPtr Class Reference

The QGuardedPtr class is a template class that provides guarded pointers to QObjects.

```
#include <qguardedptr.h>
```

Public Members

- **QGuardedPtr** ()
- **QGuardedPtr** (T * p)
- **QGuardedPtr** (const QGuardedPtr<T> & p)
- **~QGuardedPtr** ()
- QGuardedPtr<T> & **operator=** (const QGuardedPtr<T> & p)
- QGuardedPtr<T> & **operator=** (T * p)
- bool **operator==** (const QGuardedPtr<T> & p) const
- bool **operator!=** (const QGuardedPtr<T> & p) const
- bool **isNull** () const
- T * **operator->** () const
- T & **operator*** () const
- **operator T *** () const

Detailed Description

The QGuardedPtr class is a template class that provides guarded pointers to QObjects.

A guarded pointer, QGuardedPtr<*X*>, behaves like a normal C++ pointer *X**, except that it is automatically set to null when the referenced object is destroyed (unlike normal C++ pointers, which become "dangling pointers" in such cases). *X* must be a subclass of QObject.

Guarded pointers are useful whenever you need to store a pointer to a QObject that is owned by someone else and therefore might be destroyed while you still keep a reference to it. You can safely test the pointer for validity.

Example:

```
QGuardedPtr label = new QLabel( 0,"label" );
label->setText("I like guarded pointers");

delete (QLabel*) label; // emulate somebody destroying the label

if ( label)
    label->show();
else
    qDebug("The label has been destroyed");
```

The program will output

```
    The label has been destroyed
```

rather than dereferencing an invalid address in `label->show()`.

The functions and operators available with a QGuardedPtr are the same as those available with a normal unguarded pointer, except the pointer arithmetic operators (++, --, -, and +), which are normally used only with arrays of objects. Use them like normal pointers and you will not need to read this class documentation.

For creating guarded pointers, you can construct or assign to them from an X* or from another guarded pointer of the same type. You can compare them with each other for equality (==) and inequality (!=), or test for null with isNull(). Finally, you can dereference them using either the `*x` or the `x->member` notation.

A guarded pointer will automatically cast to an X*, so you can freely mix guarded and unguarded pointers. This means that if you have a QGuardedPtr, you can pass it to a function that requires a QWidget*. For this reason, it is of little value to declare functions to take a QGuardedPtr as a parameter - just use normal pointers. Use a QGuardedPtr when you are storing a pointer over time.

Note again that class *X* must inherit QObject, or a compilation or link error will result.

See also Object Model.

Member Function Documentation

QGuardedPtr::QGuardedPtr ()

Constructs a null guarded pointer.

See also isNull() [p. 99].

QGuardedPtr::QGuardedPtr (T * p)

Constructs a guarded pointer that points to same object as *p* points to.

QGuardedPtr::QGuardedPtr (const QGuardedPtr<T> & p)

Copy one guarded pointer from another. The constructed guarded pointer points to the same object that *p* points to (which may be null).

QGuardedPtr::~QGuardedPtr ()

Destroys the guarded pointer. Just like a normal pointer, destroying a guarded pointer does *not* destroy the object being pointed to.

bool QGuardedPtr::isNull () const

Returns `TRUE` if the referenced object has been destroyed or if there is no referenced object.

QGuardedPtr::operator T * () const

Cast operator; implements pointer semantics. Because of this function you can pass a QGuardedPtr to a function where an X* is required.

bool QGuardedPtr::operator!= (const QGuardedPtr<T> & p) const

Inequality operator; implements pointer semantics, the negation of operator==. Returns TRUE if *p* and this guarded pointer are not pointing to the same object; otherwise returns FALSE.

T & QGuardedPtr::operator* () const

Dereference operator; implements pointer semantics. Just use this operator as you would with a normal C++ pointer.

T * QGuardedPtr::operator-> () const

Overloaded arrow operator; implements pointer semantics. Just use this operator as you would with a normal C++ pointer.

QGuardedPtr<T> & QGuardedPtr::operator= (const QGuardedPtr<T> & p)

Assignment operator. This guarded pointer then points to the same object as *p* points to.

QGuardedPtr<T> & QGuardedPtr::operator= (T * p)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Assignment operator. This guarded pointer then points to same object as *p* points to.

bool QGuardedPtr::operator== (const QGuardedPtr<T> & p) const

Equality operator; implements traditional pointer semantics. Returns TRUE if both *p* and this guarded pointer are null, or if both *p* and this point to the same object; otherwise returns FALSE.

See also operator!=() [p. 100].

QHeader Class Reference

The QHeader class provides a header row or column, e.g. for tables and listviews.

`#include <qheader.h>`

Inherits QWidget [Widgets with Qt].

Public Members

- **QHeader** (QWidget * parent = 0, const char * name = 0)
- **QHeader** (int n, QWidget * parent = 0, const char * name = 0)
- **~QHeader** ()
- int **addLabel** (const QString & s, int size = -1)
- int **addLabel** (const QIconSet & iconset, const QString & s, int size = -1)
- void **removeLabel** (int section)
- virtual void **setLabel** (int section, const QString & s, int size = -1)
- virtual void **setLabel** (int section, const QIconSet & iconset, const QString & s, int size = -1)
- QString **label** (int section) const
- QIconSet * **iconSet** (int section) const
- virtual void **setOrientation** (Orientation)
- Orientation **orientation** () const
- virtual void **setTracking** (bool enable)
- bool **tracking** () const
- virtual void **setClickEnabled** (bool enable, int section = -1)
- virtual void **setResizeEnabled** (bool enable, int section = -1)
- virtual void **setMovingEnabled** (bool)
- virtual void **setStretchEnabled** (bool b, int section)
- void **setStretchEnabled** (bool b)
- bool **isClickEnabled** (int section = -1) const
- bool **isResizeEnabled** (int section = -1) const
- bool **isMovingEnabled** () const
- bool **isStretchEnabled** () const
- bool **isStretchEnabled** (int section) const
- void **resizeSection** (int section, int s)
- int **sectionSize** (int section) const
- int **sectionPos** (int section) const
- int **sectionAt** (int pos) const
- int **count** () const
- int **headerWidth** () const
- QRect **sectionRect** (int section) const
- virtual void setCellSize (int section, int s) *(obsolete)*

- int cellSize (int i) const *(obsolete)*
- int cellPos (int i) const *(obsolete)*
- int cellAt (int pos) const *(obsolete)*
- int **offset** () const
- int **mapToSection** (int index) const
- int **mapToIndex** (int section) const
- int mapToLogical (int a) const *(obsolete)*
- int mapToActual (int l) const *(obsolete)*
- void **moveSection** (int section, int toIndex)
- virtual void moveCell (int fromIdx, int toIdx) *(obsolete)*
- void **setSortIndicator** (int section, bool increasing = TRUE)
- void **adjustHeaderSize** ()

Public Slots

- virtual void **setOffset** (int pos)

Signals

- void **clicked** (int section)
- void **pressed** (int section)
- void **released** (int section)
- void **sizeChange** (int section, int oldSize, int newSize)
- void **indexChange** (int section, int fromIndex, int toIndex)
- void sectionClicked (int index) *(obsolete)*
- void moved (int fromIndex, int toIndex) *(obsolete)*

Properties

- int **count** — the number of sections in the header *(read only)*
- bool **moving** — whether the header sections can be moved
- int **offset** — the header's leftmost (or topmost) visible pixel
- Orientation **orientation** — the header's physical orientation
- bool **stretching** — whether the header sections always take up the full width (or height) of the header
- bool **tracking** — whether the sizeChange() signal is emitted continuously

Protected Members

- QRect **sRect** (int index)
- virtual void **paintSection** (QPainter * p, int index, const QRect & fr)
- virtual void **paintSectionLabel** (QPainter * p, int index, const QRect & fr)

Detailed Description

The QHeader class provides a header row or column, e.g. for tables and listviews.

This class provides a header, e.g. a vertical header to display row labels, or a horizontal header to display column labels. It is used by QTable and QListView for example.

A header is composed of one or more *sections*, each of which may display a text label and an iconset. A sort indicator (an arrow) may also be displayed using setSortIndicator().

Sections are added with addLabel() and removed with removeLabel(). The label and iconset are set in addLabel() and can be changed later with setLabel(). Use count() to retrieve the number of sections in the header.

The orientation of the header is set with setOrientation(). If setStretchEnabled() is TRUE, the sections will expand to take up the full width (height for vertical headers) of the header. The user can resize the sections manually if setResizeEnabled() is TRUE. Call adjustHeaderSize() to have the sections resize to occupy the full width (or height).

A section can be moved with moveSection(). If setMovingEnabled() is TRUE the user may drag a section from one position to another. If a section is moved, the index positions at which sections were added (with addLabel()), may not be the same after the move. You don't have to worry about this in practice because the QHeader API works in terms of section numbers, so it doesn't matter where a particular section has been moved to.

If you want the current index position of a section call mapToIndex() giving it the section number. (This is the number returned by the addLabel() call which created the section.) If you want to get the section number of a section at a particular index position call mapToSection() giving it the index number.

Here's an example to clarify mapToSection() and mapToIndex():

Index positions			
0	1	2	3
Original section ordering			
Sect 0	Sect 1	Sect 2	Sect 3
Ordering after the user moves a section			
Sect 0	Sect 2	Sect 3	Sect 1

k	**mapToSection(*k*)**	**mapToIndex(*k*)**
0	0	0
1	2	3
2	3	1
3	1	2

In the example above, if we wanted to find out which section is at index position 3 we'd call mapToSection(3) and get a section number of 1 since section 1 was moved. Similarly, if we wanted to know which index position section 2 occupied we'd call mapToIndex(2) and get an index of 1.

QHeader provides the clicked(), pressed() and released() signals. If the user changes the size of a section, the sizeChange() signal is emitted. If you want to have a sizeChange() signal emitted continuously whilst the user is resizing (rather than just after the resizing is finished), use setTracking(). If the user moves a section the indexChange() signal is emitted.

See also QListView [Widgets with Qt], QTable [Widgets with Qt] and Advanced Widgets.

Member Function Documentation

QHeader::QHeader (QWidget * parent = 0, const char * name = 0)

Constructs a horizontal header called *name*, with parent *parent*.

QHeader::QHeader (int n, QWidget * parent = 0, const char * name = 0)

Constructs a horizontal header called *name*, with *n* sections and parent *parent*.

QHeader::~QHeader ()

Destroys the header and all its sections.

int QHeader::addLabel (const QString & s, int size = -1)

Adds a new section with label text *s*. Returns the index position where the section was added (at the right for horizontal headers, at the bottom for vertical headers). The section's width is set to *size*. If *size* < 0, an appropriate size for the text *s* is chosen.

int QHeader::addLabel (const QIconSet & iconset, const QString & s, int size = -1)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Adds a new section with iconset *iconset* and label text *s*. Returns the index position where the section was added (at the right for horizontal headers, at the bottom for vertical headers). The section's width is set to *size*, unless size is negative in which case the size is calculated taking account of the size of the text.

void QHeader::adjustHeaderSize ()

Adjusts the size of the sections to fit the size of the header as completely as possible. Only sections for which isStretchEnabled() is TRUE will be resized.

int QHeader::cellAt (int pos) const

This function is obsolete. It is provided to keep old source working. We strongly advise against using it in new code.

Use sectionAt() instead.

Returns the index at which the section is displayed, which contains *pos* in widget coordinates, or -1 if *pos* is outside the header sections.

int QHeader::cellPos (int i) const

This function is obsolete. It is provided to keep old source working. We strongly advise against using it in new code.

Use sectionPos() instead.

Returns the position in pixels of the section that is displayed at the index *i*. The position is measured from the start of the header.

int QHeader::cellSize (int i) const

This function is obsolete. It is provided to keep old source working. We strongly advise against using it in new code.

Use sectionSize() instead.

Returns the size in pixels of the section that is displayed at the index *i*.

void QHeader::clicked (int section) [signal]

If isClickEnabled() is TRUE, this signal is emitted when the user clicks section *section*.

See also pressed() [p. 108] and released() [p. 108].

int QHeader::count () const

Returns the number of sections in the header. See the "count" [p. 111] property for details.

int QHeader::headerWidth () const

Returns the total width of all the header columns.

QIconSet * QHeader::iconSet (int section) const

Returns the icon set for section *section*. If the section does not exist, 0 is returned.

void QHeader::indexChange (int section, int fromIndex, int toIndex) [signal]

This signal is emitted when the user moves section *section* from index position *fromIndex*, to index position *toIndex*.

bool QHeader::isClickEnabled (int section = -1) const

Returns TRUE if section *section* is clickable; otherwise returns FALSE.

If *section* is out of range (negative or larger than count() - 1), TRUE is returned if all sections are clickable; otherwise returns FALSE.

See also setClickEnabled() [p. 109].

bool QHeader::isMovingEnabled () const

Returns TRUE if the header sections can be moved; otherwise returns FALSE. See the "moving" [p. 111] property for details.

bool QHeader::isResizeEnabled (int section = -1) const

Returns TRUE if section *section* is resizeable; otherwise returns FALSE.

If *section* is -1 then this function applies to all sections, i.e. TRUE is returned if all sections are resizeable; otherwise returns FALSE.

See also setResizeEnabled() [p. 110].

bool QHeader::isStretchEnabled () const

Returns TRUE if the header sections always take up the full width (or height) of the header; otherwise returns FALSE. See the "stretching" [p. 111] property for details.

bool QHeader::isStretchEnabled (int section) const

Returns TRUE if section *section* will resize to take up the full width (or height) of the header; otherwise returns FALSE. If at least one section has stretch enabled the sections will always take up the full width of the header.

See also setStretchEnabled() [p. 110].

QString QHeader::label (int section) const

Returns the text for section *section*. If the section does not exist, a null string is returned.

int QHeader::mapToActual (int l) const

This function is obsolete. It is provided to keep old source working. We strongly advise against using it in new code.

Use mapToIndex() instead.

Translates from logical index *l* to actual index (index at which the section *l* is displayed) . Returns -1 if *l* is outside the legal range.

See also mapToLogical() [p. 106].

int QHeader::mapToIndex (int section) const

Returns the index at which the section *section* is currently displayed.

For more explanation see the mapTo example.

int QHeader::mapToLogical (int a) const

This function is obsolete. It is provided to keep old source working. We strongly advise against using it in new code.

Use mapToSection() instead.

Translates from actual index *a* (index at which the section is displayed) to logical index of the section. Returns -1 if *a* is outside the legal range.

See also mapToActual() [p. 106].

int QHeader::mapToSection (int index) const

Returns the section that is displayed at index position *index*.

For more explanation see the mapTo example.

void QHeader::moveCell (int fromIdx, int toIdx) [virtual]

This function is obsolete. It is provided to keep old source working. We strongly advise against using it in new code.

Use moveSection() instead.

Moves the section that is currently displayed at index *fromIdx* to index *toIdx*.

void QHeader::moveSection (int section, int toIndex)

Moves section *section* to index position *toIndex*.

void QHeader::moved (int fromIndex, int toIndex) [signal]

This function is obsolete. It is provided to keep old source working. We strongly advise against using it in new code.

Use indexChange() instead.

This signal is emitted when the user has moved the section which is displayed at the index *fromIndex* to the index *toIndex*.

int QHeader::offset () const

Returns the header's leftmost (or topmost) visible pixel. See the "offset" [p. 111] property for details.

Orientation QHeader::orientation () const

Returns the header's physical orientation. See the "orientation" [p. 111] property for details.

void QHeader::paintSection (QPainter * p, int index, const QRect & fr) [virtual protected]

Paints the section at position *index*, inside rectangle *fr* (which uses widget coordinates) using painter *p*.

Calls paintSectionLabel().

void QHeader::paintSectionLabel (QPainter * p, int index, const QRect & fr) [virtual protected]

Paints the label of the section at position *index*, inside rectangle *fr* (which uses widget coordinates) using painter *p*.

Called by paintSection()

void QHeader::pressed (int section) [signal]

This signal is emitted when the user presses section *section* down.

See also released() [p. 108].

void QHeader::released (int section) [signal]

This signal is emitted when section *section* is released.

See also pressed() [p. 108].

void QHeader::removeLabel (int section)

Removes section *section*. If the section does not exist, nothing happens.

void QHeader::resizeSection (int section, int s)

Resizes section *section* to *s* pixels wide (or high).

QRect QHeader::sRect (int index) [protected]

Returns the rectangle covered by the section at index *index*.

int QHeader::sectionAt (int pos) const

Returns the index which contains the position *pos* given in pixels.

See also offset [p. 111].

void QHeader::sectionClicked (int index) [signal]

This function is obsolete. It is provided to keep old source working. We strongly advise against using it in new code.

Use clicked() instead.

This signal is emitted when a part of the header is clicked. *index* is the index at which the section is displayed.

In a list view this signal would typically be connected to a slot that sorts the specified column (or row).

int QHeader::sectionPos (int section) const

Returns the position (in pixels) at which the *section* starts.

See also offset [p. 111].

QRect QHeader::sectionRect (int section) const

Returns the rectangle covered by section *section*.

int QHeader::sectionSize (int section) const

Returns the width (or height) of the *section* in pixels.

void QHeader::setCellSize (int section, int s) [virtual]

This function is obsolete. It is provided to keep old source working. We strongly advise against using it in new code.

Use resizeSection() instead.

Sets the size of the section *section* to *s* pixels.

Warning: does not repaint or send out signals

void QHeader::setClickEnabled (bool enable, int section = -1) [virtual]

If *enable* is TRUE, any clicks on section *section* will result in clicked() signals being emitted; otherwise the section will ignore clicks.

If *section* is -1 (the default) then the *enable* value is set for all existing sections and will be applied to any new sections that are added.

See also moving [p. 111] and setResizeEnabled() [p. 110].

void QHeader::setLabel (int section, const QString & s, int size = -1) [virtual]

Sets the text of section *section* to *s*. The section's width is set to *size* if *size* >= 0; otherwise it is left unchanged. Any icon set that has been set for this section remains unchanged.

If the section does not exist, nothing happens.

void QHeader::setLabel (int section, const QIconSet & iconset, const QString & s, int size = -1) [virtual]

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Sets the icon for section *section* to *iconset* and the text to *s*. The section's width is set to *size* if *size* >= 0; otherwise it is left unchanged.

If the section does not exist, nothing happens.

void QHeader::setMovingEnabled (bool) [virtual]

Sets whether the header sections can be moved. See the "moving" [p. 111] property for details.

void QHeader::setOffset (int pos) [virtual slot]

Sets the header's leftmost (or topmost) visible pixel to *pos*. See the "offset" [p. 111] property for details.

void QHeader::setOrientation (Orientation) [virtual]

Sets the header's physical orientation. See the "orientation" [p. 111] property for details.

void QHeader::setResizeEnabled (bool enable, int section = -1) [virtual]

If *enable* is TRUE the user may resize section *section*; otherwise the section may not be manually resized.

If *section* is negative (the default) then the *enable* value is set for all existing sections and will be applied to any new sections that are added. Example:

```
    // Allow resizing of all current and future sections
    header->setResizeEnabled(TRUE);
    // Disable resizing of section 3, (the fourth section added)
    header->setResizeEnabled(FALSE, 3);
```

If the user resizes a section, a sizeChange() signal is emitted.

See also moving [p. 111], setClickEnabled() [p. 109] and tracking [p. 111].

void QHeader::setSortIndicator (int section, bool increasing = TRUE)

Because the QHeader is often used together with table or list widgets, QHeader can indicate a sort order. This is achieved by displaying an arrow at the right edge of a section.

If *increasing* is TRUE (the default) the arrow will point downwards; otherwise it will point upwards.

Only one section can show a sort indicator at any one time. If you don't want any section to show a sort indicator pass a *section* number of -1.

void QHeader::setStretchEnabled (bool b, int section) [virtual]

If *b* is TRUE, section *section* will be resized when the header is resized, so that the sections take up the full width (or height for vertical headers) of the header; otherwise section *section* will be set to be unstretchable and will not resize when the header is resized.

If *section* is -1, and if *b* is TRUE, then all sections will be resized equally when the header is resized so that they take up the full width (or height for vertical headers) of the header; otherwise all the sections will be set to be unstretchable and will not resize when the header is resized.

See also adjustHeaderSize() [p. 104].

void QHeader::setStretchEnabled (bool b)

Sets whether the header sections always take up the full width (or height) of the header to *b*. See the "stretching" [p. 111] property for details.

void QHeader::setTracking (bool enable) [virtual]

Sets whether the sizeChange() signal is emitted continuously to *enable*. See the "tracking" [p. 111] property for details.

void QHeader::sizeChange (int section, int oldSize, int newSize) [signal]

This signal is emitted when the user has changed the size of a *section* from *oldSize* to *newSize*. This signal is typically connected to a slot that repaints the table or list that contains the header.

bool QHeader::tracking () const

Returns TRUE if the sizeChange() signal is emitted continuously; otherwise returns FALSE. See the "tracking" [p. 111] property for details.

Property Documentation

int count

This property holds the number of sections in the header.

Get this property's value with count().

bool moving

This property holds whether the header sections can be moved.

If this property is TRUE the user may move sections. If the user moves a section the indexChange() signal is emitted.

See also setClickEnabled() [p. 109] and setResizeEnabled() [p. 110].

Set this property's value with setMovingEnabled() and get this property's value with isMovingEnabled().

int offset

This property holds the header's leftmost (or topmost) visible pixel.

Setting this property will scroll the header so that *offset* becomes the leftmost (or topmost for vertical headers) visible pixel.

Set this property's value with setOffset() and get this property's value with offset().

Orientation orientation

This property holds the header's physical orientation.

The orientation is either QHeader::Vertical or QHeader::Horizontal (the default).

Call setOrientation() before adding labels if you don't provide a size parameter otherwise the sizes will be incorrect.

Set this property's value with setOrientation() and get this property's value with orientation().

bool stretching

This property holds whether the header sections always take up the full width (or height) of the header.

Set this property's value with setStretchEnabled() and get this property's value with isStretchEnabled().

bool tracking

This property holds whether the sizeChange() signal is emitted continuously.

If tracking is on, the sizeChange() signal is emitted continuously while the mouse is moved (i.e. when the header is resized), otherwise it is only emitted when the mouse button is released at the end of resizing.

Tracking defaults to FALSE.

Set this property's value with setTracking() and get this property's value with tracking().

QIntValidator Class Reference

The QIntValidator class provides a validator which ensures that a string contains a valid integer within a specified range.

`#include <qvalidator.h>`

Inherits QValidator [p. 208].

Public Members

- **QIntValidator** (QObject * parent, const char * name = 0)
- **QIntValidator** (int minimum, int maximum, QObject * parent, const char * name = 0)
- **~QIntValidator** ()
- virtual QValidator::State **validate** (QString & input, int &) const
- void **setBottom** (int)
- void **setTop** (int)
- virtual void **setRange** (int minimum, int maximum)
- int **bottom** () const
- int **top** () const

Properties

- int **bottom** — the validator's lowest acceptable value
- int **top** — the validator's highest acceptable value

Detailed Description

The QIntValidator class provides a validator which ensures that a string contains a valid integer within a specified range.

The validate() function returns Acceptable, Intermediate or Invalid. Acceptable means that the string is a valid integer within the specified range. Intermediate means that the string is a valid integer but is not within the specified range. Invalid means that the string is not a valid integer.

Example of use:

```
    QIntValidator v( 0, 100, this );
    QLineEdit* edit = new QLineEdit( this );

    // the edit lineedit will only accept integers between 0 and 100
    edit->setValidator( &v );
```

Below we present some examples of validators. In practice they would normally be associated with a widget as in the example above.

```
    QString s;
    QIntValidator v( 0, 100, this );

    s = "10";
    v.validate( s, 0 ); // returns Acceptable
    s = "35";
    v.validate( s, 0 ); // returns Acceptable

    s = "105";
    v.validate( s, 0 ); // returns Intermediate

    s = "-763";
    v.validate( s, 0 ); // returns Invalid
    s = "abc";
    v.validate( s, 0 ); // returns Invalid
    s = "12v";
    v.validate( s, 0 ); // returns Invalid
```

The minimum and maximum values are set in one call with setRange() or individually with setBottom() and setTop().

See also QDoubleValidator [p. 50], QRegExpValidator [p. 153] and Miscellaneous Classes.

Member Function Documentation

QIntValidator::QIntValidator (QObject * parent, const char * name = 0)

Constructs a validator that accepts all integers and has parent *parent* and name *name*.

QIntValidator::QIntValidator (int minimum, int maximum, QObject * parent, const char * name = 0)

Constructs a validator that accepts all integers from and including *minimum* up to and including *maximum* with parent *parent* and name *name*.

QIntValidator::~QIntValidator ()

Destroys the validator, freeing any resources allocated.

int QIntValidator::bottom () const

Returns the validator's lowest acceptable value. See the "bottom" [p. 115] property for details.

void QIntValidator::setBottom (int)

Sets the validator's lowest acceptable value. See the "bottom" [p. 115] property for details.

void QIntValidator::setRange (int minimum, int maximum) [virtual]

Sets the range of the validator to accept only integers between *minimum* and *maximum* inclusive.

void QIntValidator::setTop (int)

Sets the validator's highest acceptable value. See the "top" [p. 115] property for details.

int QIntValidator::top () const

Returns the validator's highest acceptable value. See the "top" [p. 115] property for details.

QValidator::State QIntValidator::validate (QString & input, int &) const [virtual]

Returns Acceptable if the *input* is an integer within the valid range, Intermediate if the *input* is an integer outside the valid range and Invalid if the *input* is not an integer.

```
    s = "35";
    v.validate( s, 0 ); // returns Acceptable

    s = "105";
    v.validate( s, 0 ); // returns Intermediate

    s = "abc";
    v.validate( s, 0 ); // returns Invalid
```

Reimplemented from QValidator [p. 209].

Property Documentation

int bottom

This property holds the validator's lowest acceptable value.

Set this property's value with setBottom() and get this property's value with bottom().

See also setRange() [p. 115].

int top

This property holds the validator's highest acceptable value.

Set this property's value with setTop() and get this property's value with top().

See also setRange() [p. 115].

QMetaObject Class Reference

The QMetaObject class contains meta information about Qt objects.

```
#include <qmetaobject.h>
```

Public Members

- const char * **className** () const
- const char * **superClassName** () const
- QMetaObject * **superClass** () const
- bool **inherits** (const char * clname) const
- int **numSlots** (bool super = FALSE) const
- int **numSignals** (bool super = FALSE) const
- QStrList **slotNames** (bool super = FALSE) const
- QStrList **signalNames** (bool super = FALSE) const
- int **numClassInfo** (bool super = FALSE) const
- const QClassInfo * **classInfo** (int index, bool super = FALSE) const
- const char * **classInfo** (const char * name, bool super = FALSE) const
- const QMetaProperty * **property** (int index, bool super = FALSE) const
- int **findProperty** (const char * name, bool super = FALSE) const
- QStrList **propertyNames** (bool super = FALSE) const
- int **numProperties** (bool super = FALSE) const

Detailed Description

The QMetaObject class contains meta information about Qt objects.

The Meta Object System in Qt is responsible for the signal/slot mechanism for communication between objects, runtime type information and the property system. All meta information in Qt is kept in a single instance of QMetaObject per class.

In general, you will not have to use this class directly in any application program. However, if you write meta applications such as scripting engines or GUI builders, you might find these functions useful:

- className() to get the name of a class.
- superClassName() to get the name of the superclass.
- inherits(), the function called by QObject::inherits().
- superClass() to access the meta object of the superclass.
- numSlots(), numSignals(), slotNames(), and signalNames() to get information about a class's signals and slots.

- property() and propertyNames() to receive information about a class's properties.
- classInfo() and numClassInfo() to access additional class information.

See also Object Model.

Member Function Documentation

const QClassInfo * QMetaObject::classInfo (int index, bool super = FALSE) const

Returns the class information with index *index* or 0 if no such information exists.

If *super* is TRUE, inherited class information is included.

const char * QMetaObject::classInfo (const char * name, bool super = FALSE) const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns the class information with name *name* or 0 if no such information exists.

If *super* is TRUE, inherited class information is included.

const char * QMetaObject::className () const

Returns the class name.

See also QObject::className() [p. 127] and superClassName() [p. 119].

int QMetaObject::findProperty (const char * name, bool super = FALSE) const

Returns the index for the property with name *name* or -1 if no such property exists.

If *super* is TRUE, inherited properties are included.

See also property() [p. 118] and propertyNames() [p. 118].

bool QMetaObject::inherits (const char * clname) const

Returns TRUE if this class inherits *clname* within the meta object inheritance chain.

(A class is considered to inherit itself.)

int QMetaObject::numClassInfo (bool super = FALSE) const

Returns the number of class information available for this class.

If *super* is TRUE, inherited class information is included.

int QMetaObject::numProperties (bool super = FALSE) const

Returns the number of properties for this class.

If *super* is TRUE, inherited properties are included.

See also propertyNames() [p. 118].

int QMetaObject::numSignals (bool super = FALSE) const

Returns the number of signals for this class.

If *super* is TRUE, inherited signals are included.

See also signalNames() [p. 118].

int QMetaObject::numSlots (bool super = FALSE) const

Returns the number of slots for this class.

If *super* is TRUE, inherited slots are included.

See also slotNames() [p. 118].

const QMetaProperty * QMetaObject::property (int index, bool super = FALSE) const

Returns the property meta data for the property at index *index* or 0 if no such property exists.

If *super* is TRUE, inherited properties are included.

See also propertyNames() [p. 118].

QStrList QMetaObject::propertyNames (bool super = FALSE) const

Returns a list with the names of all properties for this class.

If *super* is TRUE, inherited properties are included.

See also property() [p. 118].

QStrList QMetaObject::signalNames (bool super = FALSE) const

Returns a list with the names of all signals for this class.

If *super* is TRUE, inherited signals are included.

QStrList QMetaObject::slotNames (bool super = FALSE) const

Returns a list with the names of all slots for this class.

If *super* is TRUE, inherited slots are included.

See also numSlots() [p. 118].

QMetaObject * QMetaObject::superClass () const

Returns the meta object of the super class or 0 if there is no such object.

const char * QMetaObject::superClassName () const

Returns the class name of the superclass or 0 if there is no superclass in the QObject hierachy.

See also className() [p. 117].

QMetaProperty Class Reference

The QMetaProperty class stores meta data about a property.

`#include <qmetaobject.h>`

Public Members

- const char * **type** () const
- const char * **name** () const
- bool **writable** () const
- bool **isSetType** () const
- bool **isEnumType** () const
- QStrList **enumKeys** () const
- int **keyToValue** (const char * key) const
- const char * **valueToKey** (int value) const
- int **keysToValue** (const QStrList & keys) const
- QStrList **valueToKeys** (int value) const
- bool **designable** (QObject * o) const
- bool **scriptable** (QObject * o) const
- bool **stored** (QObject * o) const
- bool **reset** (QObject * o) const

Detailed Description

The QMetaProperty class stores meta data about a property.

Property meta data includes type(), name(), and whether a property is writable(), designable() and stored().

The functions isSetType(), isEnumType() and enumKeys() provide further information about a property's type. The conversion functions keyToValue(), valueToKey(), keysToValue() and valueToKeys() allow conversion between the integer representation of an enumeration or set value and its literal representation.

Actual property values are set and received through QObject's set and get functions. See QObject::setProperty() and QObject::property() for details.

You receive meta property data through an object's meta object. See QMetaObject::property() and QMetaObject::propertyNames() for details.

See also Object Model.

Member Function Documentation

bool QMetaProperty::designable (QObject * o) const

Returns TRUE if the property is designable for object *o*; otherwise returns FALSE.

QStrList QMetaProperty::enumKeys () const

Returns the possible enumeration keys if this property is an enumeration type (or a set type).

See also isEnumType() [p. 121].

bool QMetaProperty::isEnumType () const

Returns whether the property's type is an enumeration value.

See also isSetType() [p. 121] and enumKeys() [p. 121].

bool QMetaProperty::isSetType () const

Returns whether the property's type is an enumeration value that is used as set, i.e. whether the enumeration values can be OR'ed together. A set type is implicitly also an enum type.

See also isEnumType() [p. 121] and enumKeys() [p. 121].

int QMetaProperty::keyToValue (const char * key) const

Converts the enumeration key *key* to its integer value.

For set types, use keysToValue().

See also valueToKey() [p. 122], isSetType() [p. 121] and keysToValue() [p. 121].

int QMetaProperty::keysToValue (const QStrList & keys) const

Converts the list of keys *keys* to their combined integer value.

See also isSetType() [p. 121] and valueToKey() [p. 122].

const char * QMetaProperty::name () const

Returns the name of the property.

bool QMetaProperty::reset (QObject * o) const

Tries to reset the property for object *o* with a reset method. On success, returns TRUE; otherwise returns FALSE.

Reset methods are optional, usually only a few properties support them.

bool QMetaProperty::scriptable (QObject * o) const

Returns TRUE if the property is scriptable for object *o*; otherwise returns FALSE.

bool QMetaProperty::stored (QObject * o) const

Returns TRUE if the property shall be stored for object *o*; otherwise returns FALSE.

const char * QMetaProperty::type () const

Returns the type of the property.

const char * QMetaProperty::valueToKey (int value) const

Converts the enumeration value *value* to its literal key.

For set types, use valueToKeys().

See also isSetType() [p. 121] and valueToKeys() [p. 122].

QStrList QMetaProperty::valueToKeys (int value) const

Converts the set value *value* to a list of keys.

See also isSetType() [p. 121] and valueToKey() [p. 122].

bool QMetaProperty::writable () const

Returns whether the property is writable or not.

QObject Class Reference

The QObject class is the base class of all Qt objects.

`#include <qobject.h>`

Inherits Qt [p. 176].

Inherited by QAccel [Events, Actions, Layouts and Styles with Qt], QAccessibleObject [Accessibility and Internationalization with Qt], QAction [Events, Actions, Layouts and Styles with Qt], QApplication [p. 4], QStyle [Events, Actions, Layouts and Styles with Qt], QDataPump, QWidget [Widgets with Qt], QCanvas [Graphics with Qt], QClipboard [Input/Output and Networking with Qt], QCopChannel [Embedded Applications with Qt], QDns [Input/Output and Networking with Qt], QLayout [Events, Actions, Layouts and Styles with Qt], QDragObject [Events, Actions, Layouts and Styles with Qt], QEditorFactory [p. 53], QFileIconProvider [Dialogs and Windows with Qt], QNetworkProtocol [Input/Output and Networking with Qt], QServerSocket [Input/Output and Networking with Qt], QWSKeyboardHandler [Embedded Applications with Qt], QNetworkOperation [Input/Output and Networking with Qt], QNPInstance, QObjectCleanupHandler [Events, Actions, Layouts and Styles with Qt], QProcess [Input/Output and Networking with Qt], QSessionManager [Input/Output and Networking with Qt], QSignal [p. 168], QSignalMapper [p. 171], QSocket [Input/Output and Networking with Qt], QSocketNotifier [Input/Output and Networking with Qt], QSound [p. 173], QSqlDatabase [Databases with Qt], QSqlDriver [Databases with Qt], QSqlForm [Databases with Qt], QStyleSheet [Events, Actions, Layouts and Styles with Qt], QTimer [p. 205], QToolTipGroup [Dialogs and Windows with Qt], QTranslator [Accessibility and Internationalization with Qt], QUrlOperator [Input/Output and Networking with Qt], QValidator [p. 208] and QWSMouseHandler [Embedded Applications with Qt].

Public Members

- **QObject** (QObject * parent = 0, const char * name = 0)
- virtual **~QObject** ()
- const char * **className** () const
- QString **tr** (const char * sourceText, const char * comment) const
- QString **trUtf8** (const char * sourceText, const char * comment) const
- QMetaObject * **metaObject** () const
- virtual bool **event** (QEvent * e)
- virtual bool **eventFilter** (QObject * watched, QEvent * e)
- bool **isA** (const char * clname) const
- bool **inherits** (const char * clname) const
- const char * **name** () const
- const char * **name** (const char * defaultName) const
- virtual void **setName** (const char * name)
- bool **isWidgetType** () const
- bool **highPriority** () const
- bool **signalsBlocked** () const
- void **blockSignals** (bool block)

- int **startTimer** (int interval)
- void **killTimer** (int id)
- void **killTimers** ()
- QObject * **child** (const char * objName, const char * inheritsClass = 0, bool recursiveSearch = TRUE)
- const QObjectList * **children** () const
- QObjectList * **queryList** (const char * inheritsClass = 0, const char * objName = 0, bool regexpMatch = TRUE, bool recursiveSearch = TRUE) const
- virtual void **insertChild** (QObject * obj)
- virtual void **removeChild** (QObject * obj)
- void **installEventFilter** (const QObject * obj)
- void **removeEventFilter** (const QObject * obj)
- bool **connect** (const QObject * sender, const char * signal, const char * member) const
- bool **disconnect** (const char * signal = 0, const QObject * receiver = 0, const char * member = 0)
- bool **disconnect** (const QObject * receiver, const char * member = 0)
- void **dumpObjectTree** ()
- void **dumpObjectInfo** ()
- virtual bool **setProperty** (const char * name, const QVariant & value)
- virtual QVariant **property** (const char * name) const
- QObject * **parent** () const

Public Slots

- void **deleteLater** ()

Signals

- void **destroyed** ()
- void **destroyed** (QObject * obj)

Static Public Members

- const QObjectList * **objectTrees** ()
- bool **connect** (const QObject * sender, const char * signal, const QObject * receiver, const char * member)
- bool **disconnect** (const QObject * sender, const char * signal, const QObject * receiver, const char * member)

Properties

- QCString **name** — the name of this object

Protected Members

- const QObject * **sender** ()
- virtual void **timerEvent** (QTimerEvent *)
- virtual void **childEvent** (QChildEvent *)
- virtual void **customEvent** (QCustomEvent *)

- virtual void **connectNotify** (const char * signal)
- virtual void **disconnectNotify** (const char * signal)
- virtual bool **checkConnectArgs** (const char * signal, const QObject * receiver, const char * member)

Static Protected Members

- QCString **normalizeSignalSlot** (const char * signalSlot)

Related Functions

- void * **qt_find_obj_child** (QObject * parent, const char * type, const char * name)

Detailed Description

The QObject class is the base class of all Qt objects.

QObject is the heart of the Qt object model. The central feature in this model is a very powerful mechanism for seamless object communication called signals and slots. You can can connect a signal to a slot with connect() and destroy the connection with disconnect(). To avoid never ending notification loops you can temporarily block signals with blockSignals(). The protected functions connectNotify() and disconnectNotify() make it possible to track connections.

QObjects organize themselves in object trees. When you create a QObject with another object as parent, the object will automatically do an insertChild() on the parent and thus show up in the parent's children() list. The parent takes ownership of the object i.e. it will automatically delete its children in its destructor. You can look for an object by name and optionally type using child() or queryList(), and get the list of tree roots using objectTrees().

Every object has an object name() and can report its className() and whether it inherits() another class in the QObject inheritance hierarchy.

When an object is deleted, it emits a destroyed() signal. You can catch this signal to avoid dangling references to QObjects. The QGuardedPtr class provides an elegant way to use this feature.

QObjects can receive events through event() and filter the events of other objects. See installEventFilter() and eventFilter() for details. A convenience handler childEvent() can be reimplemented to catch child events.

Last but not least, QObject provides the basic timer support in Qt; see QTimer for high-level support for timers.

Notice that the Q_OBJECT macro is mandatory for any object that implements signals, slots or properties. You also need to run the moc program (Meta Object Compiler) on the source file. We strongly recommend the use of this macro in *all* subclasses of QObject regardless of whether or not they actually use signals, slots and properties, since failure to do so may lead certain functions to exhibit undefined behaviour.

All Qt widgets inherit QObject. The convenience function isWidgetType() returns whether an object is actually a widget. It is much faster than inherits("QWidget").

Some QObject functions, e.g. children(), objectTrees() and queryList() return a QObjectList. A QObjectList is a QPtrList of QObjects. QObjectLists support the same operations as QPtrLists and have an iterator class, QObjectListIt.

See also Object Model.

Member Function Documentation

QObject::QObject (QObject * parent = 0, const char * name = 0)

Constructs an object with the parent object *parent* and a *name*.

The parent of an object may be viewed as the object's owner. For instance, a dialog box is the parent of the "OK" and "Cancel" buttons it contains.

The destructor of a parent object destroys all child objects.

Setting *parent* to 0 constructs an object with no parent. If the object is a widget, it will become a top-level window.

The object name is some text that can be used to identify a QObject. It's particularly useful in conjunction with *Qt Designer*. You can find an object by name (and type) using child(). To find several objects use queryList().

See also parent() [p. 134], name [p. 137], child() [p. 126] and queryList() [p. 134].

QObject::~QObject () [virtual]

Destroys the object, deleting all its child objects.

All signals to and from the object are automatically disconnected.

Warning: All child objects are deleted. If any of these objects are on the stack or global, sooner or later your program will crash. We do not recommend holding pointers to child objects from outside the parent. If you still do, the QObject::destroyed() signal gives you an opportunity to detect when an object is destroyed.

void QObject::blockSignals (bool block)

Blocks signals if *block* is TRUE, or unblocks signals if *block* is FALSE.

Emitted signals disappear into hyperspace if signals are blocked.

Example: rot13/rot13.cpp.

bool QObject::checkConnectArgs (const char * signal, const QObject * receiver, const char * member) [virtual protected]

Returns TRUE if the *signal* and the *member* arguments are compatible; otherwise returns FALSE. (The *receiver* argument is currently ignored.)

Warning: We recommend that you use the default implementation and do not reimplement this function.

QObject * QObject::child (const char * objName, const char * inheritsClass = 0, bool recursiveSearch = TRUE)

Searches the children and optionally grandchildren of this object, and returns a child that is called *objName* that inherits *inheritsClass*. If *inheritsClass* is 0 (the default), any class matches.

If *recursiveSearch* is TRUE (the default), child() performs a depth-first search of the object's children.

If there is no such object, this function returns 0. If there are more than one, the first one found is retured; if you need all of them, use queryList().

void QObject::childEvent (QChildEvent *) [virtual protected]

This event handler can be reimplemented in a subclass to receive child events.

Child events are sent to objects when children are inserted or removed.

Note that events with QEvent::type() QEvent::ChildInserted are posted (with QApplication::postEvent()) to make sure that the child's construction is completed before this function is called.

Note that if a child is removed immediately after it is inserted, the `ChildInserted` event may be suppressed, but the `ChildRemoved` event will always be sent. In this case there will be a `ChildRemoved` event without a corresponding `ChildInserted` event.

If you change state based on `ChildInserted` events, call QWidget::constPolish(), or do

```
    QApplication::sendPostedEvents( this, QEvent::ChildInserted );
```

in functions that depend on the state. One notable example is QWidget::sizeHint().

See also event() [p. 131] and QChildEvent [Events, Actions, Layouts and Styles with Qt].

Reimplemented in QMainWindow and QSplitter.

const QObjectList * QObject::children () const

Returns a list of child objects, or 0 if this object has no children.

The QObjectList class is defined in the qobjectlist.h header file.

The first child added is the first object in the list and the last child added is the last object in the list, i.e. new children are appended at the end.

Note that the list order changes when QWidget children are raised or lowered. A widget that is raised becomes the last object in the list, and a widget that is lowered becomes the first object in the list.

See also child() [p. 126], queryList() [p. 134], parent() [p. 134], insertChild() [p. 132] and removeChild() [p. 135].

const char * QObject::className () const

Returns the class name of this object.

This function is generated by the Meta Object Compiler.

Warning: This function will return a wrong name if the class definition lacks the Q_OBJECT macro.

See also name [p. 137], inherits() [p. 131], isA() [p. 133] and isWidgetType() [p. 133].

Example: sql/overview/custom1/main.cpp.

bool QObject::connect (const QObject * sender, const char * signal, const QObject * receiver, const char * member) [static]

Connects *signal* from the *sender* object to *member* in object *receiver*, and returns TRUE if the connection succeeds; otherwise returns FALSE.

You must use the SIGNAL() and SLOT() macros when specifying the *signal* and the *member*, for example:

```
    QLabel     *label  = new QLabel;
    QScrollBar *scroll = new QScrollBar;
```

```
QObject::connect( scroll, SIGNAL(valueChanged(int)),
                  label,  SLOT(setNum(int)) );
```

This example ensures that the label always displays the current scroll bar value.

A signal can also be connected to another signal:

```
class MyWidget : public QWidget
{
    Q_OBJECT
public:
    MyWidget();

signals:
    void myUsefulSignal();

private:
    QPushButton *aButton;
};

MyWidget::MyWidget()
{
    aButton = new QPushButton( this );
    connect( aButton, SIGNAL(clicked()), SIGNAL(myUsefulSignal()) );
}
```

In this example, the MyWidget constructor relays a signal from a private member variable, and makes it available under a name that relates to MyWidget.

A signal can be connected to many slots and signals. Many signals can be connected to one slot.

If a signal is connected to several slots, the slots are activated in an arbitrary order when the signal is emitted.

The function returns TRUE if it successfully connects the signal to the slot. It will return FALSE if it cannot create the connection, for example, if QObject is unable to verify the existence of either *signal* or *member*, or if their signatures aren't compatible.

See also disconnect() [p. 129].

Examples: action/actiongroup/editor.cpp, action/main.cpp, addressbook/main.cpp, application/main.cpp, iconview/main.cpp, mdi/main.cpp and t2/main.cpp.

bool QObject::connect (const QObject * sender, const char * signal, const char * member) const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Connects *signal* from the *sender* object to this object's *member*.

Equivalent to: `QObject::connect(sender, signal, this, member)`.

See also disconnect() [p. 129].

void QObject::connectNotify (const char * signal) [virtual protected]

This virtual function is called when something has been connected to *signal* in this object.

Warning: This function violates the object-oriented principle of modularity. However, it might be useful when you need to perform expensive initialization only if something is connected to a signal.

See also connect() [p. 127] and disconnectNotify() [p. 130].

void QObject::customEvent (QCustomEvent *) [virtual protected]

This event handler can be reimplemented in a subclass to receive custom events. Custom events are user-defined events with a type value at least as large as the "User" item of the QEvent::Type enum, and is typically a QCustomEvent or QCustomEvent subclass.

See also event() [p. 131] and QCustomEvent [Events, Actions, Layouts and Styles with Qt].

void QObject::deleteLater () [slot]

Delete this object deferred.

Instead of an immediate deletion this function schedules a deferred delete event for processing when Qt returns to the main event loop.

void QObject::destroyed () [signal]

This signal is emitted immediately before the object is destroyed.

All the objects's children are destroyed immediately after this signal is emitted.

void QObject::destroyed (QObject * obj) [signal]

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

This signal is emitted immediately before the object *obj* is destroyed.

All the objects's children are destroyed immediately after this signal is emitted.

bool QObject::disconnect (const QObject * sender, const char * signal, const QObject * receiver, const char * member) [static]

Disconnects *signal* in object *sender* from *member* in object *receiver*.

A signal-slot connection is removed when either of the objects involved are destroyed.

disconnect() is typically used in three ways, as the following examples demonstrate.

1. Disconnect everything connected to an object's signals:

   ```
   disconnect( myObject, 0, 0, 0 );
   ```

   equivalent to the non-static overloaded function

   ```
   myObject->disconnect();
   ```

2. Disconnect everything connected to a specific signal:

   ```
   disconnect( myObject, SIGNAL(mySignal()), 0, 0 );
   ```

   equivalent to the non-static overloaded function

   ```
   myObject->disconnect( SIGNAL(mySignal()) );
   ```

3. Disconnect a specific receiver:

   ```
   disconnect( myObject, 0, myReceiver, 0 );
   ```

   equivalent to the non-static overloaded function

   ```
   myObject->disconnect(  myReceiver );
   ```

0 may be used as a wildcard, meaning "any signal", "any receiving object", or "any slot in the receiving object", respectively.

The *sender* may never be 0. (You cannot disconnect signals from more than one object.)

If *signal* is 0, it disconnects *receiver* and *member* from any signal. If not, only the specified signal is disconnected.

If *receiver* is 0, it disconnects anything connected to *signal*. If not, slots in objects other than *receiver* are not disconnected.

If *member* is 0, it disconnects anything that is connected to *receiver*. If not, only slots named *member* will be disconnected, and all other slots are left alone. The *member* must be 0 if *receiver* is left out, so you cannot disconnect a specifically-named slot on all objects.

See also connect() [p. 127].

bool QObject::disconnect (const char * signal = 0, const QObject * receiver = 0, const char * member = 0)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Disconnects *signal* from *member* of *receiver*.

A signal-slot connection is removed when either of the objects involved are destroyed.

bool QObject::disconnect (const QObject * receiver, const char * member = 0)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Disconnects all signals in this object from *receiver*'s *member*.

A signal-slot connection is removed when either of the objects involved are destroyed.

void QObject::disconnectNotify (const char * signal) [virtual protected]

This virtual function is called when something has been disconnected from *signal* in this object.

Warning: This function violates the object-oriented principle of modularity. However, it might be useful for optimizing access to expensive resources.

See also disconnect() [p. 129] and connectNotify() [p. 128].

void QObject::dumpObjectInfo ()

Dumps information about signal connections, etc. for this object to the debug output.

This function is useful for debugging, but does nothing if the library has been compiled in release mode (i.e. without debugging information).

void QObject::dumpObjectTree ()

Dumps a tree of children to the debug output.

This function is useful for debugging, but does nothing if the library has been compiled in release mode (i.e. without debugging information).

bool QObject::event (QEvent * e) [virtual]

This virtual function receives events to an object and should return TRUE if the event *e* was recognized and processed.

The event() function can be reimplemented to customize the behavior of an object.

See also installEventFilter() [p. 132], timerEvent() [p. 137], QApplication::sendEvent() [p. 22], QApplication::postEvent() [p. 20] and QWidget::event() [Widgets with Qt].

Reimplemented in QWidget.

bool QObject::eventFilter (QObject * watched, QEvent * e) [virtual]

Filters events if this object has been installed as an event filter for the *watched* object.

In your reimplementation of this function, if you want to filter the event *e*, out, i.e. stop it being handled further, return TRUE; otherwise return FALSE.

Warning: If you delete the receiver object in this function, be sure to return TRUE. Otherwise, Qt will forward the event to the deleted object and the program might crash.

See also installEventFilter() [p. 132].

Reimplemented in QAccel, QScrollView and QSpinBox.

bool QObject::highPriority () const

Returns TRUE if the object is a high-priority object, or FALSE if it is a standard-priority object.

High-priority objects are placed first in QObject's list of children on the assumption that they will be referenced very often.

bool QObject::inherits (const char * clname) const

Returns TRUE if this object is an instance of a class that inherits *clname*, and *clname* inherits QObject; otherwise returns FALSE.

A class is considered to inherit itself.

Example:

```
    QTimer *t = new QTimer;         // QTimer inherits QObject
    t->inherits( "QTimer" );        // returns TRUE
    t->inherits( "QObject" );       // returns TRUE
    t->inherits( "QButton" );       // returns FALSE

    // QScrollBar inherits QWidget and QRangeControl
    QScrollBar *s = new QScrollBar( 0 );
    s->inherits( "QWidget" );       // returns TRUE
    s->inherits( "QRangeControl" ); // returns FALSE
```

(QRangeControl is not a QObject.)

See also isA() [p. 133] and metaObject() [p. 133].

Examples: themes/metal.cpp and themes/wood.cpp.

void QObject::insertChild (QObject * obj) [virtual]

Inserts an object *obj* into the list of child objects.

Warning: This function cannot be used to make one widget the child widget of another widget. Child widgets can only be created by setting the parent widget in the constructor or by calling QWidget::reparent().

See also removeChild() [p. 135] and QWidget::reparent() [Widgets with Qt].

void QObject::installEventFilter (const QObject * obj)

Installs an event filter *obj* on this object.

An event filter is an object that receives all events that are sent to this object. The filter can either stop the event or forward it to this object. The event filter *obj* receives events via its eventFilter() function. The eventFilter() function must return TRUE if the event should be filtered, (i.e. stopped); otherwise it must return FALSE.

If multiple event filters are installed on a single object, the filter that was installed last is activated first.

Example:

```
#include <qwidget.h>

class MyWidget : public QWidget
{
    Q_OBJECT
public:
    MyWidget( QWidget *parent = 0, const char *name = 0 );

protected:
    bool eventFilter( QObject *, QEvent * );
};

MyWidget::MyWidget( QWidget *parent, const char *name )
    : QWidget( parent, name )
{
    // install a filter on the parent (if any)
    if ( parent )
        parent->installEventFilter( this );
}

bool MyWidget::eventFilter( QObject *o, QEvent *e )
{
    if ( e->type() == QEvent::KeyPress ) {
        // special processing for key press
        QKeyEvent *k = (QKeyEvent *)e;
        qDebug( "Ate key press %d", k->key() );
        return TRUE; // eat event
    } else {
        // standard event processing
        return QWidget::eventFilter( o, e );
    }
}
```

The QAccel class, for example, uses this technique to intercept accelerator key presses.

Warning: If you delete the receiver object in your eventFilter() function, be sure to return TRUE. If you return FALSE, Qt sends the event to the deleted object and the program will crash.

See also removeEventFilter() [p. 135], eventFilter() [p. 131] and event() [p. 131].

bool QObject::isA (const char * clname) const

Returns TRUE if this object is an instance of the class *clname*; otherwise returns FALSE.

Example:

```
    QTimer *t = new QTimer; // QTimer inherits QObject
    t->isA( "QTimer" );     // returns TRUE
    t->isA( "QObject" );    // returns FALSE
```

See also inherits() [p. 131] and metaObject() [p. 133].

bool QObject::isWidgetType () const

Returns TRUE if the object is a widget; otherwise returns FALSE.

Calling this function is equivalent to calling inherits("QWidget"), except that it is much faster.

void QObject::killTimer (int id)

Kills the timer with timer identifier, *id*.

The timer identifier is returned by startTimer() when a timer event is started.

See also timerEvent() [p. 137], startTimer() [p. 136] and killTimers() [p. 133].

void QObject::killTimers ()

Kills all timers that this object has started.

Note that using this function can cause hard-to-find bugs: it kills timers started by sub- and superclasses as well as those started by you, which is often not what you want. We recommend using a QTimer or perhaps killTimer().

See also timerEvent() [p. 137], startTimer() [p. 136] and killTimer() [p. 133].

QMetaObject * QObject::metaObject () const

Returns a pointer to the meta object of this object.

A meta object contains information about a class that inherits QObject, e.g. class name, superclass name, properties, signals and slots. Every class that contains the Q_OBJECT macro will also have a meta object.

The meta object information is required by the signal/slot connection mechanism and the property system. The functions isA() and inherits() also make use of the meta object.

const char * QObject::name () const

Returns the name of this object. See the "name" [p. 137] property for details.

const char * QObject::name (const char * defaultName) const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns the name of this object, or *defaultName* if the object does not have a name.

QCString QObject::normalizeSignalSlot (const char * signalSlot) [static protected]

Normlizes the signal or slot definition *signalSlot* by removing unnecessary whitespace.

const QObjectList * QObject::objectTrees () [static]

Returns a pointer to the list of all object trees (their root objects), or 0 if there are no objects.

The QObjectList class is defined in the qobjectlist.h header file.

The latest root object created is the first object in the list and the first root object added is the last object in the list.

See also children() [p. 127], parent() [p. 134], insertChild() [p. 132] and removeChild() [p. 135].

QObject * QObject::parent () const

Returns a pointer to the parent object.

See also children() [p. 127].

QVariant QObject::property (const char * name) const [virtual]

Returns the value of the object's *name* property.

If no such property exists, the returned variant is invalid.

Information about all available properties are provided through the metaObject().

See also setProperty() [p. 136], QVariant::isValid() [Datastructures and String Handling with Qt], metaObject() [p. 133], QMetaObject::propertyNames() [p. 118] and QMetaObject::property() [p. 118].

QObjectList * QObject::queryList (const char * inheritsClass = 0, const char * objName = 0, bool regexpMatch = TRUE, bool recursiveSearch = TRUE) const

Searches the children and optionally grandchildren of this object, and returns a list of those objects that are named or that match *objName* and inherit *inheritsClass*. If *inheritsClass* is 0 (the default), all classes match. If *objName* is 0 (the default), all object names match.

If *regexpMatch* is TRUE (the default), *objName* is a regular expression that the objects's names must match. The syntax is that of a QRegExp. If *regexpMatch* is FALSE, *objName* is a string and object names must match it exactly.

Note that *inheritsClass* uses single inheritance from QObject, the way inherits() does. According to inherits(), QMenuBar inherits QWidget but not QMenuData. This does not quite match reality, but is the best that can be done on the wide variety of compilers Qt supports.

Finally, if *recursiveSearch* is TRUE (the default), queryList() searches *n*th-generation as well as first-generation children.

If all this seems a bit complex for your needs, the simpler child() function may be what you want.

This somewhat contrived example disables all the buttons in this window:

```
QObjectList *l = topLevelWidget()->queryList( "QButton" );
QObjectListIt it( *l ); // iterate over the buttons
QObject *obj;

while ( (obj = it.current()) != 0 ) {
    // for each found object...
    ++it;
    ((QButton*)obj)->setEnabled( FALSE );
}
delete l; // delete the list, not the objects
```

Warning: Delete the list as soon you have finished using it. The list contains pointers that may become invalid at almost any time without notice (as soon as the user closes a window you may have dangling pointers, for example).

See also child() [p. 126], children() [p. 127], parent() [p. 134], inherits() [p. 131], name [p. 137] and QRegExp [p. 139].

void QObject::removeChild (QObject * obj) [virtual]

Removes the child object *obj* from the list of children.

Warning: This function will not remove a child widget from the screen. It will only remove it from the parent widget's list of children.

See also insertChild() [p. 132] and QWidget::reparent() [Widgets with Qt].

void QObject::removeEventFilter (const QObject * obj)

Removes an event filter object *obj* from this object. The request is ignored if such an event filter has not been installed.

All event filters for this object are automatically removed when this object is destroyed.

It is always safe to remove an event filter, even during event filter activation (i.e. from the eventFilter() function).

See also installEventFilter() [p. 132], eventFilter() [p. 131] and event() [p. 131].

const QObject * QObject::sender () [protected]

Returns a pointer to the object that sent the signal, if called in a slot before any function call or signal emission. Returns an undefined value in all other cases.

Warning: This function will return something apparently correct in other cases as well. However, its value may change during any function call, depending on what signal-slot connections are activated during that call. In Qt 3.0 the value will change more often than in 2.x.

Warning: This function violates the object-oriented principle of modularity. However, getting access to the sender might be practical when many signals are connected to a single slot. The sender is undefined if the slot is called as a normal C++ function.

void QObject::setName (const char * name) [virtual]

Sets the object's name to *name*.

bool QObject::setProperty (const char * name, const QVariant & value) [virtual]

Sets the object's property *name* to *value*.

Returns TRUE if the operation was successful; otherwise returns FALSE.

Information about all available properties is provided through the metaObject().

See also property() [p. 134], metaObject() [p. 133], QMetaObject::propertyNames() [p. 118] and QMetaObject::property() [p. 118].

bool QObject::signalsBlocked () const

Returns TRUE if signals are blocked; otherwise returns FALSE.

Signals are not blocked by default.

See also blockSignals() [p. 126].

int QObject::startTimer (int interval)

Starts a timer and returns a timer identifier, or returns zero if it could not start a timer.

A timer event will occur every *interval* milliseconds until killTimer() or killTimers() is called. If *interval* is 0, then the timer event occurs once every time there are no more window system events to process.

The virtual timerEvent() function is called with the QTimerEvent event parameter class when a timer event occurs. Reimplement this function to get timer events.

If multiple timers are running, the QTimerEvent::timerId() can be used to find out which timer was activated.

Example:

```
class MyObject : public QObject
{
    Q_OBJECT
public:
    MyObject( QObject *parent = 0, const char *name = 0 );

protected:
    void timerEvent( QTimerEvent * );
};

MyObject::MyObject( QObject *parent, const char *name )
    : QObject( parent, name )
{
    startTimer( 50 );    // 50-millisecond timer
    startTimer( 1000 );  // 1-second timer
    startTimer( 60000 ); // 1-minute timer
}

void MyObject::timerEvent( QTimerEvent *e )
{
    qDebug( "timer event, id %d", e->timerId() );
}
```

There is practically no upper limit for the interval value (more than one year is possible). Note that QTimer's accuracy depends on the underlying operating system and hardware. Most platforms support an accuracy of 20ms; some provide more. If Qt is unable to deliver the requested number of timer clicks, it will silently discard some.

The QTimer class provides a high-level programming interface with one-shot timers and timer signals instead of events.

See also timerEvent() [p. 137], killTimer() [p. 133] and killTimers() [p. 133].

void QObject::timerEvent (QTimerEvent *) [virtual protected]

This event handler can be reimplemented in a subclass to receive timer events for the object.

QTimer provides a higher-level interface to the timer functionality, and also more general information about timers.

See also startTimer() [p. 136], killTimer() [p. 133], killTimers() [p. 133] and event() [p. 131].

Examples: biff/biff.cpp, dclock/dclock.cpp, forever/forever.cpp, grapher/grapher.cpp, qmag/qmag.cpp and xform/xform.cpp.

QString QObject::tr (const char * sourceText, const char * comment) const

Returns a translated version of *sourceText*, or *sourceText* itself if there is no appropriate translated version. The translation context is QObject with *comment* (null by default). All QObject subclasses using the Q_OBJECT macro automatically have a reimplementation of this function with the subclass name as context.

See also trUtf8() [p. 137], QApplication::translate() [p. 30] and Internationalization with Qt [Accessibility and Internationalization with Qt].

Example: network/networkprotocol/view.cpp.

QString QObject::trUtf8 (const char * sourceText, const char * comment) const

Returns a translated version of *sourceText*, or QString::fromUtf8(*sourceText*) if there is no appropriate version. It is otherwise identical to tr(*sourceText*, *comment*).

See also tr() [p. 137] and QApplication::translate() [p. 30].

Property Documentation

QCString name

This property holds the name of this object.

You can find an object by name (and type) using child(). You can find a set of objects with queryList().

The object name is set by the constructor or by the setName() function. The object name is not very useful in the current version of Qt, but will become increasingly important in the future.

If the object does not have a name, the name() function returns "unnamed", so printf() (used in qDebug()) will not be asked to output a null pointer. If you want a null pointer to be returned for unnamed objects, you can call name(0).

```
    qDebug( "MyClass::setPrecision(): (%s) invalid precision %f",
            name(), newPrecision );
```

See also className() [p. 127], child() [p. 126] and queryList() [p. 134].

Set this property's value with setName() and get this property's value with name().

Related Functions

void * qt_find_obj_child (QObject * parent, const char * type, const char * name)

Returns a pointer to the object named *name* that inherits *type* and with a given *parent*.

Returns 0 if there is no such child.

```
QListBox *c = (QListBox *) qt_find_obj_child( myWidget, "QListBox",
                                              "my list box" );
if ( c )
    c->insertItem( "another string" );
```

QRegExp Class Reference

The QRegExp class provides pattern matching using regular expressions.

```
#include <qregexp.h>
```

Public Members

- **QRegExp** ()
- **QRegExp** (const QString & pattern, bool caseSensitive = TRUE, bool wildcard = FALSE)
- **QRegExp** (const QRegExp & rx)
- **~QRegExp** ()
- QRegExp & **operator=** (const QRegExp & rx)
- bool **operator==** (const QRegExp & rx) const
- bool **operator!=** (const QRegExp & rx) const
- bool **isEmpty** () const
- bool **isValid** () const
- QString **pattern** () const
- void **setPattern** (const QString & pattern)
- bool **caseSensitive** () const
- void **setCaseSensitive** (bool sensitive)
- bool **wildcard** () const
- void **setWildcard** (bool wildcard)
- bool **minimal** () const
- void **setMinimal** (bool minimal)
- bool **exactMatch** (const QString & str) const
- int match (const QString & str, int index = 0, int * len = 0, bool indexIsStart = TRUE) const *(obsolete)*
- int **search** (const QString & str, int start = 0) const
- int **searchRev** (const QString & str, int start = -1) const
- int **matchedLength** () const
- QStringList **capturedTexts** ()
- QString **cap** (int nth = 0)
- int **pos** (int nth = 0)

Detailed Description

The QRegExp class provides pattern matching using regular expressions.

Regular expressions, or "regexps", provide a way to find patterns within text. This is useful in many contexts, for example:

1. *Validation*. A regexp can be used to check whether a piece of text meets some criteria, e.g. is an integer or contains no whitespace.
2. *Searching*. Regexps provide a much more powerful means of searching text than simple string matching does. For example we can create a regexp which says "find one of the words 'mail', 'letter' or 'correspondence' but not any of the words 'email', 'mailman' 'mailer', 'letterbox' etc."
3. *Search and Replace*. A regexp can be used to replace a pattern with a piece of text, for example replace all occurrences of '&' with '&amp;' except where the '&' is already followed by 'amp;'.
4. *String Splitting*. A regexp can be used to identify where a string should be split into its component fields, e.g. splitting tab-delimited strings.

We present a very brief introduction to regexps, a description of Qt's regexp language, some code examples, and finally the function documentation. QRegExp is modeled on Perl's regexp language, and also fully supports Unicode. QRegExp may also be used in the weaker 'wildcard' (globbing) mode which works in a similar way to command shells. A good text on regexps is *Mastering Regular Expressions: Powerful Techniques for Perl and Other Tools* by Jeffrey E. Friedl, ISBN 1565922573.

Experienced regexp users may prefer to skip the introduction and go directly to the relevant information.

Introduction

Regexps are built up from expressions, quantifiers and assertions. The simplest form of expression is simply a character, e.g. **x** or **5**. An expression can also be a set of characters. For example, **[ABCD]**, will match an **A** or a **B** or a **C** or a **D**. As a shorthand we could write this as **[A-D]**. If we want to match any of the captital letters in the English alphabet we can write **[A-Z]**. A quantifier tells the regexp engine how many occurrences of the expression we want, e.g. **x{1,1}** means match an **x** which occurs at least once and at most once. We'll look at assertions and more complex expressions later.

Note that in general regexps cannot be used to check for balanced brackets or tags. For example if you want to match an opening html `<b>` and its closing `</b>` you can only use a regexp if you know that these tags are not nested; the html fragment, `<b>bold <b>bolder</b></b>` will not match as expected. If you know the maximum level of nesting it is possible to create a regexp that will match correctly, but for an unknown level of nesting regexps will fail.

We'll start by writing a regexp to match integers in the range 0 to 99. We will require at least one digit so we will start with **[0-9]{1,1}** which means match a digit exactly once. This regexp alone will match integers in the range 0 to 9. To match one or two digits we can increase the maximum number of occurrences so the regexp becomes **[0-9]{1,2}** meaning match a digit at least once and at most twice. However, this regexp as it stands will not match correctly. This regexp will match one or two digits *within* a string. To ensure that we match against the whole string we must use the anchor assertions. We need **^** (caret) which when it is the first character in the regexp means that the regexp must match from the beginning of the string. And we also need **$** (dollar) which when it is the last character in the regexp means that the regexp must match until the end of the string. So now our regexp is **^[0-9]{1,2}$**. Note that assertions, such as **^** and **$**, do not match any characters.

If you've seen regexps elsewhere they may have looked different from the ones above. This is because some sets of characters and some quantifiers are so common that they have special symbols to represent them. **[0-9]** can be replaced with the symbol **\d**. The quantifier to match exactly one occurrence, **{1,1}**, can be replaced with the expression itself. This means that **x{1,1}** is exactly the same as **x** alone. So our 0 to 99 matcher could be written **^\d{1,2}$**. Another way of writing it would be **^\d\d{0,1}$**, i.e. from the start of the string match a digit followed by zero or one digits. In practice most people would write it **^\d\d?$**. The **?** is a shorthand for the quantifier **{0,1}**, i.e. a minimum of no occurrences a maximum of one occurrence. This is used to make an expression optional. The regexp **^\d\d?$** means "from the beginning of the string match one digit followed by zero or one digits and then the end of the string".

Our second example is matching the words 'mail', 'letter' or 'correspondence' but without matching 'email', 'mailman', 'mailer', 'letterbox' etc. We'll start by just matching 'mail'. In full the regexp is, **m{1,1}a{1,1}i{1,1}l{1,1}**, but since each expression itself is automatically quantified by **{1,1}** we can simply write this as **mail**; an 'm' followed by an 'a' followed by an 'i' followed by an 'l'. The symbol '|' (bar) is used for *alternation*, so our regexp now becomes **mail|letter|correspondence** which means match 'mail' *or* 'letter' *or* 'correspondence'. Whilst this regexp will find the words we want it will also find words we don't want such as 'email'. We will start by putting our regexp in parentheses, **(mail|letter|correspondence)**. Parentheses have two effects, firstly they group expressions together and secondly they identify parts of the regexp that we wish to capture. Our regexp still matches any of the three words but now they are grouped together as a unit. This is useful for building up more complex regexps. It is also useful because it allows us to examine which of the words actually matched. We need to use another assertion, this time **\b** "word boundary": **\b(mail|letter|correspondence)\b**. This regexp means "match a word boundary followed by the expression in parentheses followed by another word boundary". The **\b** assertion matches at a *position* in the regexp not a *character* in the regexp. A word boundary is any non-word character such as a space a newline or the beginning or end of the string.

For our third example we want to replace ampersands with the HTML entity '&amp;'. The regexp to match is simple: **&**, i.e. match one ampersand. Unfortunately this will mess up our text if some of the ampersands have already been turned into HTML entities. So what we really want to say is replace an ampersand providing it is not followed by 'amp;'. For this we need the negative lookahead assertion and our regexp becomes: **&(?!amp;)**. The negative lookahead assertion is introduced with '(?!' and finishes at the ')'. It means that the text it contains, 'amp;' in our example, must *not* follow the expression that preceeds it.

Regexps provide a rich language that can be used in a variety of ways. For example suppose we want to count all the occurrences of 'Eric' and 'Eirik' in a string. Two valid regexps to match these are **\b(Eric|Eirik)\b** and **\bEi?ri[ck]\b**. We need the word boundary '\b' so we don't get 'Ericsson' etc. The second regexp actually matches more than we want, 'Eric', 'Erik', 'Eiric' and 'Eirik'.

We will implement some the examples above in the code examples section.

Characters and Abbreviations for Sets of Characters

- **c** Any character represents itself unless it has a special regexp meaning. Thus **c** matches the character *c*.
- **\c** A character that follows a backslash matches the character itself except where mentioned below. For example if you wished to match a literal caret at the beginning of a string you would write \^.
- **\a** This matches the ASCII bell character (BEL, 0x07).
- **\f** This matches the ASCII form feed character (FF, 0x0C).
- **\n** This matches the ASCII line feed character (LF, 0x0A, Unix newline).
- **\r** This matches the ASCII carriage return character (CR, 0x0D).
- **\t** This matches the ASCII horizontal tab character (HT, 0x09).
- **\v** This matches the ASCII vertical tab character (VT, 0x0B).
- **\xhhhh** This matches the Unicode character corresponding to the hexadecimal number hhhh (between 0x0000 and 0xFFFF). \0ooo (i.e., \zero ooo) matches the ASCII/Latin-1 character corresponding to the octal number ooo (between 0 and 0377).
- **. (dot)** This matches any character (including newline).
- **\d** This matches a digit (see QChar::isDigit()).
- **\D** This matches a non-digit.
- **\s** This matches a whitespace (see QChar::isSpace()).
- **\S** This matches a non-whitespace.
- **\w** This matches a word character (see QChar::isLetterOrNumber()).
- **\W** This matches a non-word character.
- **\n** The n-th backreference, e.g. \1, \2, etc.

Note that the C++ compiler transforms backslashes in strings so to include a \ in a regexp you will need to enter it twice, i.e. \\.

Sets of Characters

Square brackets are used to match any character in the set of characters contained within the square brackets. All the character set abbreviations described above can be used within square brackets. Apart from the character set abbreviations and the following two exceptions no characters have special meanings in square brackets.

- **^** The caret negates the character set if it occurs as the first character, i.e. immediately after the opening square bracket. For example, **[abc]** matches 'a' or 'b' or 'c', but **[^abc]** matches anything *except* 'a', 'b' and 'c'.
- **-** The dash is used to indicate a range of characters, for example **[W-Z]** matches 'W' or 'X' or 'Y' or 'Z'.

Using the predefined character set abbreviations is more portable than using character ranges across platforms and languages. For example, **[0-9]** matches a digit in Western alphabets but **\d** matches a digit in *any* alphabet.

Note that in most regexp literature sets of characters are called "character classes".

Quantifiers

By default an expression is automatically quantified by **{1,1}**, i.e. it should occur exactly once. In the following list ***E*** stands for any expression. An expression is a character or an abbreviation for a set of characters or a set of characters in square brackets or any parenthesised expression.

- ***E*** Matches zero or one occurrence of *E*. This quantifier means "the previous expression is optional" since it will match whether or not the expression occurs in the string. It is the same as ***E*{0,1}**. For example **dents?** will match 'dent' and 'dents'.
- ***E*+** Matches one or more occurrences of *E*. This is the same as ***E*{1,MAXINT}**. For example, **0+** will match '0', '00', '000', etc.
- **_E_*** Matches zero or more occurrences of *E*. This is the same as ***E*{0,MAXINT}**. The * quantifier is often used by a mistake. Since it matches *zero* or more occurrences it will match no occurrences at all. For example if we want to match strings that end in whitespace and use the regexp **\s*$** we would get a match on every string. This is because we have said find zero or more whitespace followed by the end of string, so even strings that don't end in whitespace will match. The regexp we want in this case is **\s+$** to match strings that have at least one whitespace at the end.
- ***E*{n}** Matches exactly *n* occurrences of the expression. This is the same as repeating the expression *n* times. For example, **x{5}** is the same as **xxxxx**. It is also the same as ***E*{n,n}**, e.g. **x{5,5}**.
- ***E*{n,}** Matches at least *n* occurrences of the expression. This is the same as ***E*{n,MAXINT}**.
- ***E*{,m}** Matches at most *m* occurrences of the expression. This is the same as ***E*{0,m}**.
- ***E*{n,m}** Matches at least *n* occurrences of the expression and at most *m* occurrences of the expression.

(MAXINT is implementation dependent but will not be smaller than 1024.)

If we wish to apply a quantifier to more than just the preceding character we can use parentheses to group characters together in an expression. For example, **tag+** matches a 't' followed by an 'a' followed by at least one 'g', whereas **(tag)+** matches at least one occurrence of 'tag'.

Note that quantifiers are "greedy". They will match as much text as they can. For example, **0+** will match as many zeros as it can from the first zero it finds, e.g. '2.0005'. Quantifiers can be made non-greedy, see setMinimal().

Capturing Text

Parentheses allow us to group elements together so that we can quantify and capture them. For example if we have the expression **mail|letter|correspondence** that matches a string we know that *one* of the words matched but not which one. Using parentheses allows us to "capture" whatever is matched within their bounds, so if we used

(mail|letter|correspondence) and matched this regexp against the string "I sent you some email" we can use the cap() or capturedTexts() functions to extract the matched characters, in this case 'mail'.

We can use captured text within the regexp itself. To refer to the captured text we use *backreferences* which are indexed from 1, the same as for cap(). For example we could search for duplicate words in a string using **\b(\w+)\W+\1\b** which means match a word boundary followed by one or more word characters followed by one or more non-word characters followed by the same text as the first parenthesised expression followed by a word boundary.

If we want to use parentheses purely for grouping and not for capturing we can use the non-capturing syntax, e.g. **(?:green|blue)**. Non-capturing parentheses begin '(?:' and end ')'. In this example we match either 'green' or 'blue' but we do not capture the match so we only know whether or not we matched but not which color we actually found. Using non-capturing parentheses is more efficient than using capturing parentheses since the regexp engine has to do less book-keeping.

Both capturing and non-capturing parentheses may be nested.

Assertions

Assertions make some statement about the text at the point where they occur in the regexp but they do not match any characters. In the following list ***E*** stands for any expression.

- **^** The caret signifies the beginning of the string. If you wish to match a literal ^ you must escape it by writing \^. For example, **^#include** will only match strings which *begin* with the characters '#include'. (When the caret is the first character of a character set it has a special meaning, see Sets of Characters.)
- **$** The dollar signifies the end of the string. For example **\d\s*$** will match strings which end with a digit optionally followed by whitespace. If you wish to match a literal $ you must escape it by writing \$.
- **\b** A word boundary. For example the regexp **\bOK**\b means match immediately after a word boundary (e.g. start of string or whitespace) the letter 'O' then the letter 'K' immediately before another word boundary (e.g. end of string or whitespace). But note that the assertion does not actually match any whitespace so if we write **(\bOK**\b) and we have a match it will only contain 'OK' even if the string is "Its <u>OK</u> now".
- **\B** A non-word boundary. This assertion is true wherever **\b** is false. For example if we searched for **\Bon**\B in "Left on" the match would fail (space and end of string aren't non-word boundaries), but it would match in "t<u>on</u>ne".
- **(?=*E*)** Positive lookahead. This assertion is true if the expression matches at this point in the regexp. This assertion does not match any characters. For example, **^#define\s+(\w+)(?=MAX)** will match strings which begin with '#define' followed by at least one whitespace followed by at least one word character followed by 'MAX'. The first set of parentheses will capture the word character(s) matched. This regexp will not match '#define DEBUG' but will match '#define <u>INT</u>MAX 32767'.
- **(?!*E*)** Negative lookahead. This assertion is true if the expression does not match at this point in the regexp. This assertion does not match any characters. For example, **^#define\s+(\w+)\s*$** will match strings which begin with '#define' followed by at least one whitespace followed by at least one word character optionally followed by whitespace. This regexp will match define's that exist but have no value, i.e. it will not match '#define INTMAX 32767' but it will match '#define <u>DEBUG</u>'.

Wildcard Matching (globbing)

Most command shells such as *bash* or *cmd* support "file globbing", the ability to identify a group of files by using wildcards. The setWildcard() function is used to switch between regexp and wildcard mode. Wildcard matching is much simpler than full regexps and has only four features:

- **c** Any character represents itself apart from those mentioned below. Thus **c** matches the character *c*.
- **?** This matches any single character. It is the same as **.** in full regexps.
- * This matches zero or more of any characters. It is the same as **.*** in full regexps.

- **[...]** Sets of characters can be represented in square brackets, similar to full regexps. Within the character class, like outside, backslash has no special meaning.

For example if we are in wildcard mode and have strings which contain filenames we could identify HTML files with ***.html**. This will match zero or more characters followed by a dot followed by 'h', 't', 'm' and 'l'.

Notes for Perl Users

Most of the character class abbreviations supported by Perl are supported by QRegExp, see characters and abbreviations for sets of characters.

In QRegExp, apart from within character classes, `^` always signifies the start of the string, so carets must always be escaped unless used for that purpose. In Perl the meaning of caret varies automagically depending on where it occurs so escaping it is rarely necessary. The same applies to `$` which in QRegExp always signifies the end of the string.

QRegExp's quantifiers are the same as Perl's greedy quantifiers. Non-greedy matching cannot be applied to individual quantifiers, but can be applied to all the quantifiers in the pattern. For example, to match the Perl regexp **ro+?m** requires:

```
QRegExp rx( "ro+m" );
rx.setMinimal( TRUE );
```

The equivalent of Perl's `/i` option is setCaseSensitive(FALSE).

Perl's `/g` option can be emulated using a loop.

In QRegExp **.** matches any character, therefore all QRegExp regexps have the equivalent of Perl's `/s` option. QRegExp does not have an equivalent to Perl's `/m` option, but this can be emulated in various ways for example by splitting the input into lines or by looping with a regexp that searches for newlines.

Because QRegExp is string oriented there are no \A, \Z or \z assertions. The \G assertion is not supported but can be emulated in a loop.

Perl's $& is cap(0) or capturedTexts()[0]. There are no QRegExp equivalents for $`, $' or $+. Perl's capturing variables, $1, $2, ... correspond to cap(1) or capturedTexts()[1], cap(2) or capturedTexts()[2], etc.

To substitute a pattern use QString::replace().

Perl's extended `/x` syntax is not supported, nor are regexp comments (?#comment) or directives, e.g. (?i).

Both zero-width positive and zero-width negative lookahead assertions (?=pattern) and (?!pattern) are supported with the same syntax as Perl. Perl's lookbehind assertions, "independent" subexpressions and conditional expressions are not supported.

Non-capturing parentheses are also supported, with the same (?:pattern) syntax.

See QStringList::split() and QStringList::join() for equivalents to Perl's split and join functions.

Note: because C++ transforms \'s they must be written *twice* in code, e.g. **\b** must be written **\\b**.

Code Examples

```
QRegExp rx( "^\\d\\d?$" );  // match integers 0 to 99
rx.search( "123" );         // returns -1 (no match)
rx.search( "-6" );          // returns -1 (no match)
rx.search( "6" );           // returns 0 (matched as position 0)
```

The third string matches '6'. This is a simple validation regexp for integers in the range 0 to 99.

```
QRegExp rx( "^\\S+$" );     // match strings without whitespace
rx.search( "Hello world" ); // returns -1 (no match)
rx.search( "This_is-OK" );  // returns 0 (matched at position 0)
```

The second string matches 'This_is-OK'. We've used the character set abbreviation '\S' (non-whitespace) and the anchors to match strings which contain no whitespace.

In the following example we match strings containing 'mail' or 'letter' or 'correspondence' but only match whole words i.e. not 'email'

```
QRegExp rx( "\\b(mail|letter|correspondence)\\b" );
rx.search( "I sent you an email" );     // returns -1 (no match)
rx.search( "Please write the letter" ); // returns 17
```

The second string matches "Please write the letter". The word 'letter' is also captured (because of the parentheses). We can see what text we've captured like this:

```
QString captured = rx.cap( 1 ); // captured contains "letter"
```

This will capture the text from the first set of capturing parentheses (counting capturing left parentheses from left to right). The parentheses are counted from 1 since cap(0) is the whole matched regexp (equivalent to '&' in most regexp engines).

```
QRegExp rx( "&(?!amp;)" );      // match ampersands but not &amp;
QString line1 = "This & that";
line1.replace( rx, "&amp;" );
// line1 == "This &amp; that"
QString line2 = "His &amp; hers & theirs";
line2.replace( rx, "&amp;" );
// line2 == "His &amp; hers &amp; theirs"
```

Here we've passed the QRegExp to QString's replace() function to replace the matched text with new text.

```
QString str = "One Eric another Eirik, and an Ericsson."
              " How many Eiriks, Eric?";
QRegExp rx( "\\b(Eric|Eirik)\\b" ); // match Eric or Eirik
int pos = 0;    // where we are in the string
int count = 0;  // how many Eric and Eirik's we've counted
while ( pos >= 0 ) {
    pos = rx.search( str, pos );
    if ( pos >= 0 ) {
        pos++;      // move along in str
        count++;    // count our Eric or Eirik
    }
}
```

We've used the search() function to repeatedly match the regexp in the string. Note that instead of moving forward by one character at a time `pos++` we could have written `pos += rx.matchedLength()` to skip over the already matched string. The count will equal 3, matching 'One Eric another Eirik, and an Ericsson. How many Eiriks, Eric?'; it doesn't match 'Ericsson' or 'Eiriks' because they are not bounded by non-word boundaries.

One common use of regexps is to split lines of delimited data into their component fields.

```
str = "Trolltech AS\twww.trolltech.com\tNorway";
QString company, web, country;
```

```
rx.setPattern( "^([^\t]+)\t([^\t]+)\t([^\t]+)$" );
if ( rx.search( str ) != -1 ) {
    company = rx.cap( 1 );
    web = rx.cap( 2 );
    country = rx.cap( 3 );
}
```

In this example our input lines have the format company name, web address and country. Unfortunately the regexp is rather long and not very versatile — the code will break if we add any more fields. A simpler and better solution is to look for the separator, '\t' in this case, and take the surrounding text. The QStringList split() function can take a separator string or regexp as an argument and split a string accordingly.

```
QStringList field = QStringList::split( "\t", str );
```

Here field[0] is the company, field[1] the web address and so on.

To imitate the matching of a shell we can use wildcard mode.

```
QRegExp rx( "*.html" );     // invalid regexp: * doesn't quantify anything
rx.setWildcard( TRUE );     // now it's a valid wildcard regexp
rx.search( "index.html" );  // returns 0 (matched at position 0)
rx.search( "default.htm" ); // returns -1 (no match)
rx.search( "readme.txt" );  // returns -1 (no match)
```

Wildcard matching can be convenient because of its simplicity, but any wildcard regexp can be defined using full regexps, e.g. **.*\.html$**. Notice that we can't match both `.html` and `.htm` files with a wildcard unless we use ***.htm*** which will also match 'test.html.bak'. A full regexp gives us the precision we need, **.*\.html?$**.

QRegExp can match case insensitively using setCaseSensitive(), and can use non-greedy matching, see setMinimal(). By default QRegExp uses full regexps but this can be changed with setWildcard(). Searching can be forward with search() or backward with searchRev(). Captured text can be accessed using capturedTexts() which returns a string list of all captured strings, or using cap() which returns the captured string for the given index. The pos() function takes a match index and returns the position in the string where the match was made (or -1 if there was no match).

See also QRegExpValidator [p. 153], QString [Datastructures and String Handling with Qt], QStringList [Datastructures and String Handling with Qt], Miscellaneous Classes, Implicitly and Explicitly Shared Classes and Non-GUI Classes.

Member Function Documentation

QRegExp::QRegExp ()

Constructs an empty regexp.

See also isValid() [p. 149].

QRegExp::QRegExp (const QString & pattern, bool caseSensitive = TRUE, bool wildcard = FALSE)

Constructs a regular expression object for the given *pattern* string. The pattern must be given using wildcard notation if *wildcard* is TRUE (default is FALSE). The pattern is case sensitive, unless *caseSensitive* is FALSE. Matching is greedy (maximal), but can be changed by calling setMinimal().

See also setPattern() [p. 151], setCaseSensitive() [p. 151], setWildcard() [p. 151] and setMinimal() [p. 151].

QRegExp::QRegExp (const QRegExp & rx)

Constructs a regular expression as a copy of *rx*.

See also operator=() [p. 150].

QRegExp::~QRegExp ()

Destroys the regular expression and cleans up its internal data.

QString QRegExp::cap (int nth = 0)

Returns the text captured by the *nth* subexpression. The entire match has index 0 and the parenthesized subexpressions have indices starting from 1 (excluding non-capturing parentheses).

```
QRegExp rxlen( "(\\d+)(?:\\s*)(cm|inch)" );
int pos = rxlen.search( "Length: 189cm" );
if ( pos > -1 ) {
    QString value = rxlen.cap( 1 ); // "189"
    QString unit = rxlen.cap( 2 );  // "cm"
    // ...
}
```

The order of elements matched by cap() is as follows. The first element, cap(0), is the entire matching string. Each subsequent element corresponds to the next capturing open left parentheses. Thus cap(1) is the text of the first capturing parentheses, cap(2) is the text of the second, and so on.

Some patterns may lead to a number of matches which cannot be determined in advance, for example:

```
QRegExp rx( "(\\d+)" );
str = "Offsets: 12 14 99 231 7";
QStringList list;
pos = 0;
while ( pos >= 0 ) {
    pos = rx.search( str, pos );
    if ( pos > -1 ) {
        list += rx.cap( 1 );
        pos  += rx.matchedLength();
    }
}
// list contains "12", "14", "99", "231", "7"
```

See also capturedTexts() [p. 147], pos() [p. 150], exactMatch() [p. 148], search() [p. 150] and searchRev() [p. 151].

QStringList QRegExp::capturedTexts ()

Returns a list of the captured text strings.

The first string in the list is the entire matched string. Each subsequent list element contains a string that matched a (capturing) subexpression of the regexp.

For example:

```
    QRegExp rx( "(\\d+)(\\s*)(cm|inch(es)?)" );
    int pos = rx.search( "Length: 36 inches" );
    QStringList list = rx.capturedTexts();
    // list is now ( "36 inches", "36", " ", "inches", "es" )
```

The above example also captures elements that may be present but which we have no interest in. This problem can be solved by using non-capturing parentheses:

```
    QRegExp rx( "(\\d+)(?:\\s*)(cm|inch(?:es)?)" );
    int pos = rx.search( "Length: 36 inches" );
    QStringList list = rx.capturedTexts();
    // list is now ( "36 inches", "36", "inches" )
```

Some regexps can match an indeterminate number of times. For example if the input string is "Offsets: 12 14 99 231 7" and the regexp, `rx`, is **(\d+)+**, we would hope to get a list of all the numbers matched. However, after calling `rx.search(str)`, capturedTexts() will return the list ("12", "12"), i.e. the entire match was "12" and the first subexpression matched was "12". The correct approach is to use cap() in a loop.

The order of elements in the string list is as follows. The first element is the entire matching string. Each subsequent element corresponds to the next capturing open left parentheses. Thus capturedTexts()[1] is the text of the first capturing parentheses, capturedTexts()[2] is the text of the second and so on (corresponding to $1, $2, etc., in some other regexp languages).

See also cap() [p. 147], pos() [p. 150], exactMatch() [p. 148], search() [p. 150] and searchRev() [p. 151].

bool QRegExp::caseSensitive () const

Returns TRUE if case sensitivity is enabled, otherwise FALSE. The default is TRUE.

See also setCaseSensitive() [p. 151].

bool QRegExp::exactMatch (const QString & str) const

Returns TRUE if *str* is matched exactly by this regular expression otherwise it returns FALSE. You can determine how much of the string was matched by calling matchedLength().

For a given regexp string, R, exactMatch("R") is the equivalent of search("^R$") since exactMatch() effectively encloses the regexp in the start of string and end of string anchors, except that it sets matchedLength() differently.

For example, if the regular expression is **blue**, then exactMatch() returns TRUE only for input `blue`. For inputs `bluebell`, `blutak` and `lightblue`, exactMatch() returns FALSE and matchedLength() will return 4, 3 and 0 respectively.

Although const, this function sets matchedLength(), capturedTexts() and pos().

See also search() [p. 150], searchRev() [p. 151] and QRegExpValidator [p. 153].

bool QRegExp::isEmpty () const

Returns TRUE if the pattern string is empty, otherwise FALSE.

If you call exactMatch() with an empty pattern on an empty string it will return TRUE; otherwise it returns FALSE since it operates over the whole string. If you call search() with an empty pattern on *any* string it will return the start position (0 by default) since it will match at the start position, because the empty pattern matches the 'emptiness' at the start of the string, and the length of the match returned by matchedLength() will be 0.

See QString::isEmpty().

bool QRegExp::isValid () const

Returns TRUE if the regular expression is valid, or FALSE if it's invalid. An invalid regular expression never matches.

The pattern **[a-z** is an example of an invalid pattern, since it lacks a closing square bracket.

Note that the validity of a regexp may also depend on the setting of the wildcard flag, for example ***.html** is a valid wildcard regexp but an invalid full regexp.

int QRegExp::match (const QString & str, int index = 0, int * len = 0, bool indexIsStart = TRUE) const

This function is obsolete. It is provided to keep old source working. We strongly advise against using it in new code.

Attempts to match in *str*, starting from position *index*. Returns the position of the match, or -1 if there was no match.

The length of the match is stored in **len*, unless *len* is a null pointer.

If *indexIsStart* is TRUE (the default), the position *index* in the string will match the start of string anchor, ^, in the regexp, if present. Otherwise, position 0 in *str* will match.

Use search() and matchedLength() instead of this function.

If you really need the *indexIsStart* functionality, try this:

```
QRegExp rx( "some pattern" );
int pos = rx.search( str.mid(index) );
if ( pos >= 0 )
    pos += index;
int len = rx.matchedLength();
```

Where performance is important, you can replace `str.mid(index)` by `QConstString(str.unicode() + index, str.length() - index).string()`, which avoids copying the character data.

See also QString::mid() [Datastructures and String Handling with Qt] and QConstString [Datastructures and String Handling with Qt].

Example: qmag/qmag.cpp.

int QRegExp::matchedLength () const

Returns the length of the last matched string, or -1 if there was no match.

See also exactMatch() [p. 148], search() [p. 150] and searchRev() [p. 151].

bool QRegExp::minimal () const

Returns TRUE if minimal (non-greedy) matching is enabled, otherwise returns FALSE.

See also setMinimal() [p. 151].

bool QRegExp::operator!= (const QRegExp & rx) const

Returns TRUE if this regular expression is not equal to *rx*, otherwise FALSE.

See also operator==() [p. 150].

QRegExp & QRegExp::operator= (const QRegExp & rx)

Copies the regular expression *rx* and returns a reference to the copy. The case sensitivity, wildcard and minimal matching options are copied as well.

bool QRegExp::operator== (const QRegExp & rx) const

Returns TRUE if this regular expression is equal to *rx*, otherwise returns FALSE.

Two QRegExp objects are equal if they have the same pattern strings and the same settings for case sensitivity, wildcard and minimal matching.

QString QRegExp::pattern () const

Returns the pattern string of the regular expression. The pattern has either regular expression syntax or wildcard syntax, depending on wildcard().

See also setPattern() [p. 151].

int QRegExp::pos (int nth = 0)

Returns the position of the *nth* captured text in the searched string. If *nth* is 0 (the default), pos() returns the position of the whole match.

Example:

```
    QRegExp rx( "/([a-z]+)/([a-z]+)" );
    rx.search( "Output /dev/null" );    // returns 7 (position of /dev/null)
    rx.pos( 0 );                        // returns 7 (position of /dev/null)
    rx.pos( 1 );                        // returns 8 (position of dev)
    rx.pos( 2 );                        // returns 12 (position of null)
```

For zero-length matches, pos() always returns -1. (For example, if cap(4) would return an empty string, pos(4) returns -1.) This is due to an implementation tradeoff.

See also capturedTexts() [p. 147], exactMatch() [p. 148], search() [p. 150] and searchRev() [p. 151].

int QRegExp::search (const QString & str, int start = 0) const

Attempts to find a match in *str* from position *start* (0 by default). If *start* is -1, the search starts at the last character; if -2, at the next to last character; etc.

Returns the position of the first match, or -1 if there was no match.

You might prefer to use QString::find(), QString::contains() or even QStringList::grep(). To replace matches use QString::replace().

Example:

```
    QString str = "offsets: 1.23 .50 71.00 6.00";
    QRegExp rx( "\\d*\\.\\d+" );    // primitive floating point matching
    int count = 0;
    int pos = 0;
    while ( pos >= 0 ) {
        pos = rx.search( str, pos );
        count++;
    }
    // pos will be 9, 14, 18 and finally 24; count will end up as 4
```

Although const, this function sets matchedLength(), capturedTexts() and pos().

See also searchRev() [p. 151] and exactMatch() [p. 148].

int QRegExp::searchRev (const QString & str, int start = -1) const

Attempts to find a match backwards in *str* from position *start*. If *start* is -1 (the default), the search starts at the last character; if -2, at the next to last character; etc.

Returns the position of the first match, or -1 if there was no match.

Although const, this function sets matchedLength(), capturedTexts() and pos().

Warning: Searching backwards is much slower than searching forwards.

See also search() [p. 150] and exactMatch() [p. 148].

void QRegExp::setCaseSensitive (bool sensitive)

Sets case sensitive matching to *sensitive*.

If *sensitive* is TRUE, **\.txt$** matches `readme.txt` but not `README.TXT`.

See also caseSensitive() [p. 148].

void QRegExp::setMinimal (bool minimal)

Enables or disables minimal matching. If *minimal* is FALSE, matching is greedy (maximal) which is the default.

For example, suppose we have the input string "We must be <b>bold</b>, very <b>bold</b>!" and the pattern **<b>.*</b>**. With the default greedy (maximal) matching, the match is "We must be <u><b>bold</b>, very <b>bold</b></u>!". But with minimal (non-greedy) matching the first match is: "We must be <u><b>bold</b></u>, very <b>bold</b>!" and the second match is "We must be <b>bold</b>, very <u><b>bold</b></u>!". In practice we might use the pattern **<b>[^<]+</b>**, although this will still fail for nested tags.

See also minimal() [p. 149].

void QRegExp::setPattern (const QString & pattern)

Sets the pattern string to *pattern* and returns a reference to this regular expression. The case sensitivity, wildcard and minimal matching options are not changed.

See also pattern() [p. 150].

void QRegExp::setWildcard (bool wildcard)

Sets the wildcard mode for the regular expression. The default is FALSE.

Setting *wildcard* to TRUE enables simple shell-like wildcard matching. (See wildcard matching (globbing).)

For example, **r*.txt** matches the string `readme.txt` in wildcard mode, but does not match `readme`.

See also wildcard() [p. 152].

bool QRegExp::wildcard () const

Returns TRUE if wildcard mode is enabled, otherwise FALSE. The default is FALSE.

See also setWildcard() [p. 151].

QRegExpValidator Class Reference

The QRegExpValidator class is used to check a string against a regular expression.

`#include <qvalidator.h>`

Inherits QValidator [p. 208].

Public Members

- **QRegExpValidator** (QObject * parent, const char * name = 0)
- **QRegExpValidator** (const QRegExp & rx, QObject * parent, const char * name = 0)
- **~QRegExpValidator** ()
- virtual QValidator::State **validate** (QString & input, int & pos) const
- void **setRegExp** (const QRegExp & rx)
- const QRegExp & **regExp** () const

Detailed Description

The QRegExpValidator class is used to check a string against a regular expression.

QRegExpValidator contains a regular expression, "regexp", used to determine whether an input string is Acceptable, Intermediate or Invalid.

The regexp is treated as if it begins with the start of string assertion, ^, and ends with the end of string assertion $ so the match is against the entire input string, or from the given position if a start position greater than zero is given.

For a brief introduction to Qt's regexp engine see QRegExp.

Example of use:

```
// regexp: optional '-' followed by between 1 and 3 digits
QRegExp rx( "-?\\d{1,3}" );
QRegExpValidator validator( rx, 0 );

QLineEdit *edit = new QLineEdit( split );
edit->setValidator( &validator );
```

Below we present some examples of validators. In practice they would normally be associated with a widget as in the example above.

```
// integers 1 to 9999
QRegExp rx( "[1-9]\\d{0,3}" );
// the validator treats the regexp as "^[1-9]\\d{0,3}$"
```

```
    QRegExpValidator v( rx, 0 );
    QString s;

    s = "0";     v.validate( s, 0 );    // returns Invalid
    s = "12345"; v.validate( s, 0 );    // returns Invalid
    s = "1";     v.validate( s, 0 );    // returns Acceptable

    rx.setPattern( "\\S+" );            // one or more non-whitespace characters
    v.setRegExp( rx );
    s = "myfile.txt";  v.validate( s, 0 ); // Returns Acceptable
    s = "my file.txt"; v.validate( s, 0 ); // Returns Invalid

    // A, B or C followed by exactly five digits followed by W, X, Y or Z
    rx.setPattern( "[A-C]\\d{5}[W-Z]" );
    v.setRegExp( rx );
    s = "a12345Z"; v.validate( s, 0 );  // Returns Invalid
    s = "A12345Z"; v.validate( s, 0 );  // Returns Acceptable
    s = "B12";     v.validate( s, 0 );  // Returns Intermediate

    // match most 'readme' files
    rx.setPattern( "read\\S?me(\.(txt|asc|1st))?" );
    rx.setCaseSensitive( FALSE );
    v.setRegExp( rx );
    s = "readme";      v.validate( s, 0 ); // Returns Acceptable
    s = "README.1ST";  v.validate( s, 0 ); // Returns Acceptable
    s = "read me.txt"; v.validate( s, 0 ); // Returns Invalid
    s = "readm";       v.validate( s, 0 ); // Returns Intermediate
```

See also QRegExp [p. 139], QIntValidator [p. 113], QDoubleValidator [p. 50] and Miscellaneous Classes.

Member Function Documentation

QRegExpValidator::QRegExpValidator (QObject * parent, const char * name = 0)

Constructs a validator that accepts any string (including an empty one) as valid. The object's parent is *parent* and its name is *name*.

QRegExpValidator::QRegExpValidator (const QRegExp & rx, QObject * parent, const char * name = 0)

Constructs a validator which accepts all strings that match the regular expression *rx*. The object's parent is *parent* and its name is *name*.

The match is made against the entire string, e.g. if the regexp is **[A-Fa-f0-9]+** it will be treated as **^[A-Fa-f0-9]+$**.

QRegExpValidator::~QRegExpValidator ()

Destroys the validator, freeing any resources allocated.

const QRegExp & QRegExpValidator::regExp () const

Returns the regular expression used for validation.

See also setRegExp() [p. 155].

void QRegExpValidator::setRegExp (const QRegExp & rx)

Sets the regular expression used for validation to *rx*.

See also regExp() [p. 154].

QValidator::State QRegExpValidator::validate (QString & input, int & pos) const [virtual]

Returns Acceptable if *input* is matched by the regular expression for this validator, Intermediate if it has matched partially (i.e. could be a valid match if additional valid characters are added), and Invalid if *input* is not matched.

The start position is the beginning of the string unless *pos* is given and is > 0 in which case the regexp is matched from *pos* until the end of the string.

For example, if the regular expression is **\w\d\d** (that is, word-character, digit, digit) then "A57" is Acceptable, "E5" is Intermediate and "+9" is Invalid.

See also QRegExp::match() [p. 149].

Reimplemented from QValidator [p. 209].

QSimpleRichText Class Reference

The QSimpleRichText class provides a small displayable piece of rich text.

`#include <qsimplerichtext.h>`

Public Members

- **QSimpleRichText** (const QString & text, const QFont & fnt, const QString & context = QString::null, const QStyleSheet * sheet = 0)
- **QSimpleRichText** (const QString & text, const QFont & fnt, const QString & context, const QStyleSheet * sheet, const QMimeSourceFactory * factory, int pageBreak = -1, const QColor & linkColor = Qt::blue, bool linkUnderline = TRUE)
- **~QSimpleRichText** ()
- void **setWidth** (int w)
- void **setWidth** (QPainter * p, int w)
- void **setDefaultFont** (const QFont & f)
- int **width** () const
- int **widthUsed** () const
- int **height** () const
- void **adjustSize** ()
- void **draw** (QPainter * p, int x, int y, const QRect & clipRect, const QColorGroup & cg, const QBrush * paper = 0) const
- void draw (QPainter * p, int x, int y, const QRegion & clipRegion, const QColorGroup & cg, const QBrush * paper = 0) const *(obsolete)*
- QString **context** () const
- QString **anchorAt** (const QPoint & pos) const
- bool **inText** (const QPoint & pos) const

Detailed Description

The QSimpleRichText class provides a small displayable piece of rich text.

This class encapsulates simple rich text usage in which a string is interpreted as rich text and can be drawn. This is particularly useful if you want to display some rich text in a custom widget. A QStyleSheet is needed actually to understand and format rich text. Qt provides a default HTML-like style sheet, but you may define custom style sheets.

Once created, the rich text object can be queried for its width(), height(), and the actual width used (see widthUsed()). Most importantly, it can be drawn on any given QPainter with draw(). QSimpleRichText can also be used to implement hypertext or active text facilities by using anchorAt(). A hit test through inText() makes it possible to use simple rich text for text objects in editable drawing canvases.

Once constructed from a string the contents cannot be changed, only resized. If the contents change, just throw the rich text object away and make a new one with the new contents.

For large documents use QTextEdit or QTextBrowser. For very small items of rich text you can use a QLabel.

See also Text Related Classes.

Member Function Documentation

QSimpleRichText::QSimpleRichText (const QString & text, const QFont & fnt, const QString & context = QString::null, const QStyleSheet * sheet = 0)

Constructs a QSimpleRichText from the rich text string *text* and the font *fnt*.

The font is used as a basis for the text rendering. When using rich text rendering on a widget *w*, you would normally specify the widget's font, for example:

```
  QSimpleRichText myrichtext( contents, mywidget->font() );
```

context is the optional context of the document. This becomes important if *text* contains relative references, for example within image tags. QSimpleRichText always uses the default mime source factory (see QMimeSourceFactory::defaultFactory()) to resolve those references. The context will then be used to calculate the absolute path. See QMimeSourceFactory::makeAbsolute() for details.

The *sheet* is an optional style sheet. If it is 0, the default style sheet will be used (see QStyleSheet::defaultSheet()).

QSimpleRichText::QSimpleRichText (const QString & text, const QFont & fnt, const QString & context, const QStyleSheet * sheet, const QMimeSourceFactory * factory, int pageBreak = -1, const QColor & linkColor = Qt::blue, bool linkUnderline = TRUE)

Constructs a QSimpleRichText from the rich text string *text* and the font *fnt*.

This is a slightly more complex constructor for QSimpleRichText that takes an additional mime source factory *factory*, a page break parameter *pageBreak* and a bool *linkUnderline*. *linkColor* is only provided for compatibility, but has no effect, as QColorGroup's QColorGroup::link() color is used now.

context is the optional context of the document. This becomes important if *text* contains relative references, for example within image tags. QSimpleRichText always uses the default mime source factory (see QMimeSourceFactory::defaultFactory()) to resolve those references. The context will then be used to calculate the absolute path. See QMimeSourceFactory::makeAbsolute() for details.

The *sheet* is an optional style sheet. If it is 0, the default style sheet will be used (see QStyleSheet::defaultSheet()).

This constructor is useful for creating a QSimpleRichText object suitable for printing. Set *pageBreak* to be the height of the contents area of the pages.

QSimpleRichText::~QSimpleRichText ()

Destroys the document, freeing memory.

void QSimpleRichText::adjustSize ()

Adjusts the richt text document to a reasonable size.

See also setWidth() [p. 158].

QString QSimpleRichText::anchorAt (const QPoint & pos) const

Returns the anchor at the requested position, *pos*. An empty string is returned if no anchor is specified for this position.

QString QSimpleRichText::context () const

Returns the context of the rich text document. If no context has been specified in the constructor, a null string is returned. The context is the path to use to look up relative links, such as image tags and anchor references.

void QSimpleRichText::draw (QPainter * p, int x, int y, const QRect & clipRect, const QColorGroup & cg, const QBrush * paper = 0) const

Draws the formatted text with painter *p*, at position (*x*, *y*), clipped to *clipRect*. The clipping rectangle is given in the document's coordinates translated by (*x*, *y*). Colors from the color group *cg* are used as needed, and if not 0, **paper* is used as the background brush.

Note that the display code is highly optimized to reduce flicker, so passing a brush for *paper* is preferable to simply clearing the area to be painted and then calling this without a brush.

Example: helpviewer/helpwindow.cpp.

void QSimpleRichText::draw (QPainter * p, int x, int y, const QRegion & clipRegion, const QColorGroup & cg, const QBrush * paper = 0) const

This function is obsolete. It is provided to keep old source working. We strongly advise against using it in new code.

Use the version with clipRect instead. The region version has problems with larger documents on some platforms (on X11 regions internally are represented with 16bit coordinates).

int QSimpleRichText::height () const

Returns the height of the document in pixels.

See also setWidth() [p. 158].

Example: helpviewer/helpwindow.cpp.

bool QSimpleRichText::inText (const QPoint & pos) const

Returns TRUE if *pos* is within a text line of the document; otherwise returns FALSE.

void QSimpleRichText::setDefaultFont (const QFont & f)

Sets the default font for the document to *f*

void QSimpleRichText::setWidth (QPainter * p, int w)

Sets the width of the document to *w* pixels, recalculating the layout as if it were to be drawn with painter *p*.

Passing a painter is useful when you intend to draw on devices other than the screen, for example a QPrinter.

See also height() [p. 158] and adjustSize() [p. 157].

Example: helpviewer/helpwindow.cpp.

void QSimpleRichText::setWidth (int w)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Sets the width of the document to *w* pixels.

See also height() [p. 158] and adjustSize() [p. 157].

int QSimpleRichText::width () const

Returns the set width of the document in pixels.

See also widthUsed() [p. 159].

int QSimpleRichText::widthUsed () const

Returns the width in pixels that is actually used by the document. This can be smaller or wider than the set width.

It may be wider, for example, if the text contains images or non-breakable words that are already wider than the available space. It's smaller when the document only consists of lines that do not fill the width completely.

See also width() [p. 159].

QSettings Class Reference

The QSettings class provides persistent platform-independent application settings.

`#include <qsettings.h>`

Public Members

- **QSettings** ()
- **~QSettings** ()
- enum **System** { Unix = 0, Windows, Mac }
- bool **writeEntry** (const QString & key, bool value)
- bool **writeEntry** (const QString & key, double value)
- bool **writeEntry** (const QString & key, int value)
- bool **writeEntry** (const QString & key, const QString & value)
- bool **writeEntry** (const QString & key, const QStringList & value)
- bool **writeEntry** (const QString & key, const QStringList & value, const QChar & separator)
- QStringList **entryList** (const QString & key) const
- QStringList **subkeyList** (const QString & key) const
- QStringList **readListEntry** (const QString & key, bool * ok = 0)
- QStringList **readListEntry** (const QString & key, const QChar & separator, bool * ok = 0)
- QString **readEntry** (const QString & key, const QString & def = QString::null, bool * ok = 0)
- int **readNumEntry** (const QString & key, int def = 0, bool * ok = 0)
- double **readDoubleEntry** (const QString & key, double def = 0, bool * ok = 0)
- bool **readBoolEntry** (const QString & key, bool def = 0, bool * ok = 0)
- bool **removeEntry** (const QString & key)
- void **insertSearchPath** (System s, const QString & path)
- void **removeSearchPath** (System s, const QString & path)

Detailed Description

The QSettings class provides persistent platform-independent application settings.

On Unix systems, QSettings uses text files to store settings. On Windows systems, QSettings uses the system registry. On Mac OS X, QSettings will behave as on Unix, and store to text files.

Each setting comprises an identifying key and the data associated with the key. A key is a unicode string which consists of *two* or more subkeys. A subkey is a slash, '/', followed by one or more unicode characters (excluding slashes, newlines, carriage returns and equals, '=', signs). The associated data, called the entry or value, may be a boolean, an integer, a double, a string or a list of strings. Entry strings may contain any unicode characters.

If you want to save and restore the entire desktop's settings, i.e. which applications are running, use QSettings to save the settings for each individual application and QSessionManager to save the desktop's session.

Example settings:

```
/MyCompany/MyApplication/background color
/MyCompany/MyApplication/foreground color
/MyCompany/MyApplication/geometry/x
/MyCompany/MyApplication/geometry/y
/MyCompany/MyApplication/geometry/width
/MyCompany/MyApplication/geometry/height
/MyCompany/MyApplication/recent files/1
/MyCompany/MyApplication/recent files/2
/MyCompany/MyApplication/recent files/3
```

Each line above is a complete key, made up of subkeys.

A typical usage pattern for application startup:

```
QSettings settings;
settings.insertSearchPath( QSettings::Windows, "/MyCompany" );
// No search path needed for Unix; see notes further on.
// Use default values if the keys don't exist
QString bgColor = settings.readEntry( "/MyApplication/background color", "white" );
int width = settings.readNumEntry( "/MyApplication/geometry/width", 640 );
// ...
```

A typical usage pattern for application exit or 'save preferences':

```
QSettings settings;
settings.insertSearchPath( QSettings::Windows, "/MyCompany" );
// No search path needed for Unix; see notes further on.
settings.writeEntry( "/MyApplication/background color", bgColor );
settings.writeEntry( "/MyApplication/geometry/width", width );
// ...
```

You can get a list of entry-holding keys by calling entryList(), and a list of key-holding keys using subkeyList().

```
QStringList keys = entryList( "/MyApplication" );
// keys contains 'background color' and 'foreground color'.

QStringList keys = entryList( "/MyApplication/recent files" );
// keys contains '1', '2' and '3'.

QStringList subkeys = subkeyList( "/MyApplication" );
// subkeys contains 'geometry' and 'recent files'

QStringList subkeys = subkeyList( "/MyApplication/recent files" );
// subkeys is empty.
```

If you wish to use a different search path call insertSearchPath() as often as necessary to add your preferred paths. Call removeSearchPath() to remove any unwanted paths.

Since settings for Windows are stored in the registry there are size limits as follows:

- A subkey may not exceed 255 characters.
- An entry's value may not exceed 16,300 characters.
- All the values of a key (for example, all the 'recent files' subkeys values), may not exceed 65,535 characters.

These limitations are not enforced on Unix.

Notes for Unix Applications

There is no universally accepted place for storing application settings under Unix. In the examples the settings file will be searched for in the following directories:

1. $QTDIR/etc
2. /opt/MyCompany/share/etc
3. /opt/MyCompany/share/MyApplication/etc
4. $HOME/.qt

When reading settings the files are searched in the order shown above, with later settings overriding earlier settings. Files for which the user doesn't have read permission are ignored. When saving settings QSettings works forwards in the order shown above writing to the first settings file for which the user has write permission. ($QTDIR is the directory where Qt was installed.)

If you want to put the settings in a particular place in the filesystem you could do this:

```
settings.insertSearchPath( QSettings::Unix, "/opt/MyCompany/share" );
```

But in practice you may prefer not to use a search path for Unix. For example the following code:

```
settings.writeEntry( "/MyApplication/geometry/width", width );
```

will end up writing the "geometry/width" setting to the file `$HOME/.qt/myapplicationrc` (assuming that the application is being run by an ordinary user, i.e. not by root).

For cross-platform applications you should ensure that the Windows size limitations are not exceeded.

See also Input/Output and Networking and Miscellaneous Classes.

Member Type Documentation

QSettings::System

- `QSettings::Mac` - Macintosh execution environments
- `QSettings::Unix` - Mac OS X, Unix, Linux and Unix-like execution environments
- `QSettings::Windows` - Windows execution environments

Member Function Documentation

QSettings::QSettings ()

Creates a settings object.

QSettings::~QSettings ()

Destroys the settings object. All modifications made to the settings will automatically be saved.

QStringList QSettings::entryList (const QString & key) const

Returns a list of the keys which contain entries under *key*. Does *not* return any keys that contain keys.

Example settings:

```
    /MyCompany/MyApplication/background color
    /MyCompany/MyApplication/foreground color
    /MyCompany/MyApplication/geometry/x
    /MyCompany/MyApplication/geometry/y
    /MyCompany/MyApplication/geometry/width
    /MyCompany/MyApplication/geometry/height

    QStringList keys = entryList( "/MyApplication" );
```

`keys` contains 'background color' and 'foreground color'. It does not contain 'geometry' because this key contains keys not entries.

To access the geometry values could either use subkeyList() to read the keys and then read each entry, or simply read each entry directly by specifying its full key, e.g. "/MyCompany/MyApplication/geometry/y".

See also subkeyList() [p. 165].

void QSettings::insertSearchPath (System s, const QString & path)

Inserts *path* into the settings search path. The semantics of *path* depends on the system *s*.

When *s* is *Windows* and the execution environment is *not* Windows the function does nothing. Similarly when *s* is *Unix* and the execution environment is *not* Unix the function does nothing.

When *s* is *Windows*, and the execution environment is Windows, the search path list will be used as the first subfolder of the "Software" folder in the registry.

When reading settings the folders are searched forwards from the first folder (listed below) to the last, with later settings overriding settings found earlier, and ignoring any folders for which the user doesn't have read permission.

1. HKEY_CURRENT_USER/Software/MyCompany/MyApplication
2. HKEY_CURRENT_USER/Software/MyApplication
3. HKEY_LOCAL_MACHINE/Software/MyCompany/MyApplication
4. HKEY_LOCAL_MACHINE/Software/MyApplication

```
  QSettings settings;
  settings.insertSearchPath( QSettings::Windows, "/MyCompany" );
  settings.writeEntry( "/MyApplication/Tip of the day", TRUE );
```

The code above will write the subkey "Tip of the day" into the *first* of the registry folders listed below that is found and for which the user has write permission.

1. HKEY_LOCAL_MACHINE/Software/MyApplication
2. HKEY_LOCAL_MACHINE/Software/MyCompany/MyApplication
3. HKEY_CURRENT_USER/Software/MyApplication
4. HKEY_CURRENT_USER/Software/MyCompany/MyApplication

When *s* is *Unix*, and the execution environment is Unix, the search path list will be used when trying to determine a suitable filename for reading and writing settings files. By default, there are two entries in the search path:

1. $QTDIR/etc - where $QTDIR is the directory where Qt was installed.
2. $HOME/.qt/ - where $HOME is the user's home directory.

All insertions into the search path will go before $HOME/.qt/. For example:

```
QSettings settings;
settings.insertSearchPath( QSettings::Unix, "/opt/MyCompany/share/etc" );
settings.insertSearchPath( QSettings::Unix, "/opt/MyCompany/share/MyApplication/etc" );
// ...
```

Will result in a search path of:

1. $QTDIR/etc
2. /opt/MyCompany/share/etc
3. /opt/MyCompany/share/MyApplication/etc
4. $HOME/.qt

When reading settings the files are searched in the order shown above, with later settings overriding earlier settings. Files for which the user doesn't have read permission are ignored. When saving settings QSettings works forwards in the order shown above writing to the first settings file for which the user has write permission. ($QTDIR is the directory where Qt was installed.)

Settings under Unix are stored in files whose names are based on the first subkey of the key (not including the search path). The algorithm for creating names is essentially: lowercase the first subkey, replace spaces with underscores and add 'rc', e.g. `/MyCompany/MyApplication/background color` will be stored in `myapplicationrc` (assuming that `/MyCompany` is part of the search path).

See also removeSearchPath() [p. 165].

bool QSettings::readBoolEntry (const QString & key, bool def = 0, bool * ok = 0)

Reads the entry specified by *key*, and returns a bool, or the default value, *def*, if the entry couldn't be read. If *ok* is non-null, *ok is set to TRUE if the key was read, FALSE otherwise.

See also readEntry() [p. 164], readNumEntry() [p. 165], readDoubleEntry() [p. 164], writeEntry() [p. 166] and removeEntry() [p. 165].

double QSettings::readDoubleEntry (const QString & key, double def = 0, bool * ok = 0)

Reads the entry specified by *key*, and returns a double, or the default value, *def*, if the entry couldn't be read. If *ok* is non-null, *ok is set to TRUE if the key was read, FALSE otherwise.

See also readEntry() [p. 164], readNumEntry() [p. 165], readBoolEntry() [p. 164], writeEntry() [p. 166] and removeEntry() [p. 165].

QString QSettings::readEntry (const QString & key, const QString & def = QString::null, bool * ok = 0)

Reads the entry specified by *key*, and returns a QString, or the default value, *def*, if the entry couldn't be read. If *ok* is non-null, *ok is set to TRUE if the key was read, FALSE otherwise.

See also readListEntry() [p. 165], readNumEntry() [p. 165], readDoubleEntry() [p. 164], readBoolEntry() [p. 164], writeEntry() [p. 166] and removeEntry() [p. 165].

QStringList QSettings::readListEntry (const QString & key, bool * ok = 0)

Reads the entry specified by *key* as a string. If *ok* is non-null, *ok is set to TRUE if the key was read, FALSE otherwise.

See also readEntry() [p. 164], readDoubleEntry() [p. 164], readBoolEntry() [p. 164], writeEntry() [p. 166], removeEntry() [p. 165] and QStringList::split() [Datastructures and String Handling with Qt].

QStringList QSettings::readListEntry (const QString & key, const QChar & separator, bool * ok = 0)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Reads the entry specified by *key* as a string. The *separator* is used to create a QStringList by calling QStringList::split(*separator*, entry). If *ok* is non-null, *ok is set to TRUE if the key was read, FALSE otherwise.

See also readEntry() [p. 164], readDoubleEntry() [p. 164], readBoolEntry() [p. 164], writeEntry() [p. 166], removeEntry() [p. 165] and QStringList::split() [Datastructures and String Handling with Qt].

int QSettings::readNumEntry (const QString & key, int def = 0, bool * ok = 0)

Reads the entry specified by *key*, and returns an integer, or the default value, *def*, if the entry couldn't be read. If *ok* is non-null, *ok is set to TRUE if the key was read, FALSE otherwise.

See also readEntry() [p. 164], readDoubleEntry() [p. 164], readBoolEntry() [p. 164], writeEntry() [p. 166] and removeEntry() [p. 165].

bool QSettings::removeEntry (const QString & key)

Removes the entry specified by *key*.

Returns TRUE if the entry existed and was removed; otherwise returns FALSE.

See also readEntry() [p. 164] and writeEntry() [p. 166].

void QSettings::removeSearchPath (System s, const QString & path)

Removes all occurrences of *path* (using exact matching) from the settings search path for system *s*. Note that the default search paths cannot be removed.

See also insertSearchPath() [p. 163].

QStringList QSettings::subkeyList (const QString & key) const

Returns a list of the keys which contain keys under *key*. Does *not* return any keys that contain entries.

Example settings:

```
    /MyCompany/MyApplication/background color
    /MyCompany/MyApplication/foreground color
    /MyCompany/MyApplication/geometry/x
    /MyCompany/MyApplication/geometry/y
    /MyCompany/MyApplication/geometry/width
    /MyCompany/MyApplication/geometry/height
    /MyCompany/MyApplication/recent files/1
```

```
/MyCompany/MyApplication/recent files/2
/MyCompany/MyApplication/recent files/3

QStringList keys = subkeyList( "/MyApplication" );
```

`keys` contains 'geometry' and 'recent files'. It does not contain 'background color' or 'foreground color' because they are keys which contain entries not keys. To get a list of keys that have values rather than subkeys use entryList().

See also entryList() [p. 163].

bool QSettings::writeEntry (const QString & key, bool value)

Writes the boolean entry *value* into key *key*. The *key* is created if it doesn't exist. Any previous value is overwritten by *value*.

If an error occurs the settings are left unchanged and FALSE is returned; otherwise TRUE is returned.

See also readListEntry() [p. 165], readNumEntry() [p. 165], readDoubleEntry() [p. 164], readBoolEntry() [p. 164] and removeEntry() [p. 165].

bool QSettings::writeEntry (const QString & key, double value)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Writes the double entry *value* into key *key*. The *key* is created if it doesn't exist. Any previous value is overwritten by *value*.

If an error occurs the settings are left unchanged and FALSE is returned; otherwise TRUE is returned.

See also readListEntry() [p. 165], readNumEntry() [p. 165], readDoubleEntry() [p. 164], readBoolEntry() [p. 164] and removeEntry() [p. 165].

bool QSettings::writeEntry (const QString & key, int value)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Writes the integer entry *value* into key *key*. The *key* is created if it doesn't exist. Any previous value is overwritten by *value*.

If an error occurs the settings are left unchanged and FALSE is returned; otherwise TRUE is returned.

See also readListEntry() [p. 165], readNumEntry() [p. 165], readDoubleEntry() [p. 164], readBoolEntry() [p. 164] and removeEntry() [p. 165].

bool QSettings::writeEntry (const QString & key, const QString & value)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Writes the string entry *value* into key *key*. The *key* is created if it doesn't exist. Any previous value is overwritten by *value*. If *value* is an empty string or a null string the key's value will be an empty string.

If an error occurs the settings are left unchanged and FALSE is returned; otherwise TRUE is returned.

See also readListEntry() [p. 165], readNumEntry() [p. 165], readDoubleEntry() [p. 164], readBoolEntry() [p. 164] and removeEntry() [p. 165].

bool QSettings::writeEntry (const QString & key, const QStringList & value)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Writes the string list entry *value* into key *key*. The *key* is created if it doesn't exist. Any previous value is overwritten by *value*.

If an error occurs the settings are left unchanged and FALSE is returned; otherwise TRUE is returned.

See also readListEntry() [p. 165], readNumEntry() [p. 165], readDoubleEntry() [p. 164], readBoolEntry() [p. 164] and removeEntry() [p. 165].

bool QSettings::writeEntry (const QString & key, const QStringList & value, const QChar & separator)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Writes the string list entry *value* into key *key*. The *key* is created if it doesn't exist. Any previous value is overwritten by *value*. The list is stored as a sequence of strings separated by *separator*, so none of the strings in the list should contain the separator. If the list is empty or null the key's value will be an empty string.

If an error occurs the settings are left unchanged and FALSE is returned; otherwise TRUE is returned.

See also readListEntry() [p. 165], readNumEntry() [p. 165], readDoubleEntry() [p. 164], readBoolEntry() [p. 164] and removeEntry() [p. 165].

QSignal Class Reference

The QSignal class can be used to send signals for classes that don't inherit QObject.

`#include <qsignal.h>`

Inherits QObject [p. 123].

Public Members

- **QSignal** (QObject * parent = 0, const char * name = 0)
- **~QSignal** ()
- bool **connect** (const QObject * receiver, const char * member)
- bool **disconnect** (const QObject * receiver, const char * member = 0)
- void **activate** ()
- bool isBlocked () const *(obsolete)*
- void block (bool b) *(obsolete)*
- void setParameter (int value) *(obsolete)*
- int parameter () const *(obsolete)*
- void **setValue** (const QVariant & value)
- QVariant **value** () const

Detailed Description

The QSignal class can be used to send signals for classes that don't inherit QObject.

If you want to send signals from a class that does not inherit QObject, you can create an internal QSignal object to emit the signal. You must also provide a function that connects the signal to an outside object slot. This is how we have implemented signals in the QMenuData class, which is not a QObject.

In general, we recommend inheriting QObject instead. QObject provides much more functionality.

You can set a single QVariant parameter for the signal with setValue().

Note that QObject is a *private* base class of QSignal, i.e. you cannot call any QObject member functions from a QSignal object.

Example:

```
#include <qsignal.h>

class MyClass
{
public:
    MyClass();
```

```
        ~MyClass();

        void doSomething();

        void connect( QObject *receiver, const char *member );

    private:
        QSignal *sig;
    };

    MyClass::MyClass()
    {
        sig = new QSignal;
    }

    MyClass::~MyClass()
    {
        delete sig;
    }

    void MyClass::doSomething()
    {
        // ... does something
        sig->activate(); // emits the signal
    }

    void MyClass::connect( QObject *receiver, const char *member )
    {
        sig->connect( receiver, member );
    }
```

See also Input/Output and Networking and Miscellaneous Classes.

Member Function Documentation

QSignal::QSignal (QObject * parent = 0, const char * name = 0)

Constructs a signal object with the parent object *parent* and a *name*. These arguments are passed directly to QObject.

QSignal::~QSignal ()

Destroys the signal. All connections are removed, as is the case with all QObjects.

void QSignal::activate ()

Emits the signal. If the platform supports QVariant and a parameter has been set with setValue(), this value is passed in the signal.

void QSignal::block (bool b)

This function is obsolete. It is provided to keep old source working. We strongly advise against using it in new code.

Blocks the signal if *b* is TRUE, or unblocks the signal if *b* is FALSE.

An activated signal disappears into hyperspace if it is blocked.

See also isBlocked() [p. 170], activate() [p. 169] and QObject::blockSignals() [p. 126].

bool QSignal::connect (const QObject * receiver, const char * member)

Connects the signal to *member* in object *receiver*.

See also disconnect() [p. 170] and QObject::connect() [p. 127].

bool QSignal::disconnect (const QObject * receiver, const char * member = 0)

Disonnects the signal from *member* in object *receiver*.

See also connect() [p. 170] and QObject::disconnect() [p. 129].

bool QSignal::isBlocked () const

This function is obsolete. It is provided to keep old source working. We strongly advise against using it in new code.

Returns TRUE if the signal is blocked, or FALSE if it is not blocked.

The signal is not blocked by default.

See also block() [p. 170] and QObject::signalsBlocked() [p. 136].

int QSignal::parameter () const

This function is obsolete. It is provided to keep old source working. We strongly advise against using it in new code.

void QSignal::setParameter (int value)

This function is obsolete. It is provided to keep old source working. We strongly advise against using it in new code.

void QSignal::setValue (const QVariant & value)

Sets the signal's parameter to *value*

QVariant QSignal::value () const

Returns the signal's parameter

QSignalMapper Class Reference

The QSignalMapper class bundles signals from identifiable senders.

`#include <qsignalmapper.h>`

Inherits QObject [p. 123].

Public Members

- **QSignalMapper** (QObject * parent, const char * name = 0)
- **~QSignalMapper** ()
- virtual void **setMapping** (const QObject * sender, int identifier)
- virtual void **setMapping** (const QObject * sender, const QString & identifier)
- void **removeMappings** (const QObject * sender)

Public Slots

- void **map** ()

Signals

- void **mapped** (int)
- void **mapped** (const QString &)

Detailed Description

The QSignalMapper class bundles signals from identifiable senders.

This class collects a set of parameterless signals, and re-emits them with integer or string parameters corresponding to the object that sent the signal.

See also Input/Output and Networking.

Member Function Documentation

QSignalMapper::QSignalMapper (QObject * parent, const char * name = 0)

Constructs a QSignalMapper with parent *parent* and name *name*. Like all QObjects, it will be deleted when the parent is deleted.

QSignalMapper::~QSignalMapper ()

Destroys the QSignalMapper.

void QSignalMapper::map () [slot]

This slot emits signals based on which object sends signals to it.

Example: i18n/main.cpp.

void QSignalMapper::mapped (int) [signal]

This signal is emitted when map() is signaled from an object that has an integer mapping set.

See also setMapping() [p. 172].

Example: i18n/main.cpp.

void QSignalMapper::mapped (const QString &) [signal]

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

This signal is emitted when map() is signaled from an object that has a string mapping set.

See also setMapping() [p. 172].

void QSignalMapper::removeMappings (const QObject * sender)

Removes all mappings for *sender*. This is done automatically when mapped objects are destroyed.

void QSignalMapper::setMapping (const QObject * sender, int identifier) [virtual]

Adds a mapping so that when map() is signaled from the given *sender*, the signal mapped(*identifier*) is emitted.

There may be at most one integer identifier for each object.

Example: i18n/main.cpp.

void QSignalMapper::setMapping (const QObject * sender, const QString & identifier) [virtual]

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Adds a mapping so that when map() is signaled from the given *sender*, the signal mapper(*identifier*) is emitted.

There may be at most one string identifier for each object, and it may not be null.

QSound Class Reference

The QSound class provides access to the platform audio facilities.

`#include <qsound.h>`

Inherits QObject [p. 123].

Public Members

- **QSound** (const QString & filename, QObject * parent = 0, const char * name = 0)
- **~QSound** ()
- int **loops** () const
- int **loopsRemaining** () const
- void **setLoops** (int l)
- QString **fileName** () const
- bool **isFinished** () const

Public Slots

- void **play** ()
- void **stop** ()

Static Public Members

- bool **isAvailable** ()
- void **play** (const QString & filename)
- bool **available** ()

Detailed Description

The QSound class provides access to the platform audio facilities.

Qt provides the most commonly required audio operation in GUI applications: playing a sound file asynchronously to the user. This is most simply accomplished with a single call:

```
    QSound::play("mysounds/bells.wav");
```

A second API is provided in which a QSound object is created from a sound file and is later played:

```
QSound bells("mysounds/bells.wav");

bells.play();
```

Sounds played by the second model may use more memory but play more immediately than sounds played using the first model, depending on the underlying platform audio facilities.

On Microsoft Windows the underlying multimedia system is used; only WAVE format sound files are supported.

On X11 the Network Audio System is used if available, otherwise all operations work silently. NAS supports WAVE and AU files.

On Macintosh, in an ironic turn of events we use QT (QuickTime) for sound, this means all QuickTime formats are supported by Qt/Mac.

On Qt/Embedded, a built-in mixing sound server is used, which accesses `/dev/dsp` directly. Only the WAVE format is supported.

The availability of sound can be tested with QSound::isAvailable().

See also Multimedia Classes.

Member Function Documentation

QSound::QSound (const QString & filename, QObject * parent = 0, const char * name = 0)

Constructs a QSound that can quickly play the sound in a file named *filename*.

This can use more memory than the static `play` function.

The *parent* and *name* arguments (default 0) are passed on to the QObject constructor.

QSound::~QSound ()

Destroys the sound object.

bool QSound::available () [static]

Returns TRUE if sound support is available; otherwise returns FALSE.

QString QSound::fileName () const

Returns the filename associated with the sound.

bool QSound::isAvailable () [static]

Returns TRUE if sound facilities exist on the platform; otherwise returns FALSE. An application may choose either to notify the user if sound is crucial to the application or to operate silently without bothering the user.

If no sound is available, all QSound operations work silently and quickly.

bool QSound::isFinished () const

Returns TRUE if the sound has finished playing; otherwise returns FALSE.

int QSound::loops () const

Returns the number of times the sound will play.

int QSound::loopsRemaining () const

Returns the number of times the sound will loop. This value decreases each time the sound loops.

void QSound::play (const QString & filename) [static]

Plays the sound in a file called *filename*.

void QSound::play () [slot]

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Starts the sound playing. The function returns immediately. Depending on the platform audio facilities, other sounds may stop or may be mixed with the new sound.

The sound can be played again at any time, possibly mixing or replacing previous plays of the sound.

void QSound::setLoops (int l)

Sets the sound to repeat *l* times when it is played. Passing the value -1 will cause the sound to loop indefinitely.

See also loops() [p. 175].

void QSound::stop () [slot]

Stops the sound playing.

See also play() [p. 175].

Qt Class Reference

The Qt class is a namespace for miscellaneous identifiers that need to be global-like.

`#include <qnamespace.h>`

Inherited by QObject [p. 123], QPixmap [Graphics with Qt], QBrush [Graphics with Qt], QCanvasItem [Graphics with Qt], QCursor [Graphics with Qt], QEvent [Events, Actions, Layouts and Styles with Qt], QIconViewItem [Widgets with Qt], QKeySequence [Events, Actions, Layouts and Styles with Qt], QListViewItem [Widgets with Qt], QCustomMenuItem [Dialogs and Windows with Qt], QPainter [Graphics with Qt], QPen [Graphics with Qt], QStyleSheetItem [Events, Actions, Layouts and Styles with Qt], QTab [Widgets with Qt], QTableItem [Widgets with Qt], QThread [Threading with Qt], QToolTip [Dialogs and Windows with Qt] and QWhatsThis [Widgets with Qt].

Public Members

- enum **ButtonState** { NoButton = 0x0000, LeftButton = 0x0001, RightButton = 0x0002, MidButton = 0x0004, MouseButtonMask = 0x00ff, ShiftButton = 0x0100, ControlButton = 0x0200, AltButton = 0x0400, MetaButton = 0x0800, KeyButtonMask = 0x0fff, Keypad = 0x4000 }
- enum **Orientation** { Horizontal = 0, Vertical }
- enum **AlignmentFlags** { AlignAuto = 0x0000, AlignLeft = 0x0001, AlignRight = 0x0002, AlignHCenter = 0x0004, AlignJustify = 0x0008, AlignHorizontal_Mask = AlignLeft | AlignRight | AlignHCenter | AlignJustify, AlignTop = 0x0010, AlignBottom = 0x0020, AlignVCenter = 0x0040, AlignVertical_Mask = AlignTop | AlignBottom | AlignVCenter, AlignCenter = AlignVCenter | AlignHCenter }
- enum **TextFlags** { SingleLine = 0x0080, DontClip = 0x0100, ExpandTabs = 0x0200, ShowPrefix = 0x0400, WordBreak = 0x0800, BreakAnywhere = 0x1000, DontPrint = 0x2000, NoAccel = 0x4000 }
- enum **WidgetState** { WState_Created = 0x00000001, WState_Disabled = 0x00000002, WState_Visible = 0x00000004, WState_ForceHide = 0x00000008, WState_OwnCursor = 0x00000010, WState_MouseTracking = 0x00000020, WState_CompressKeys = 0x00000040, WState_BlockUpdates = 0x00000080, WState_InPaintEvent = 0x00000100, WState_Reparented = 0x00000200, WState_ConfigPending = 0x00000400, WState_Resized = 0x00000800, WState_AutoMask = 0x00001000, WState_Polished = 0x00002000, WState_DND = 0x00004000, WState_Reserved0 = 0x00008000, WState_Reserved1 = 0x00010000, WState_Reserved2 = 0x00020000, WState_Reserved3 = 0x00040000, WState_Maximized = 0x00080000, WState_Minimized = 0x00100000, WState_ForceDisabled = 0x00200000, WState_Exposed = 0x00400000, WState_HasMouse = 0x00800000 }
- enum **WidgetFlags** { WType_TopLevel = 0x00000001, WType_Dialog = 0x00000002, WType_Popup = 0x00000004, WType_Desktop = 0x00000008, WType_Mask = 0x0000000f, WStyle_Customize = 0x00000010, WStyle_NormalBorder = 0x00000020, WStyle_DialogBorder = 0x00000040, WStyle_NoBorder = 0x00002000, WStyle_Title = 0x00000080, WStyle_SysMenu = 0x00000100, WStyle_Minimize = 0x00000200, WStyle_Maximize = 0x00000400, WStyle_MinMax = WStyle_Minimize | WStyle_Maximize, WStyle_Tool = 0x00000800, WStyle_StaysOnTop = 0x00001000, WStyle_ContextHelp = 0x00004000, WStyle_Reserved = 0x00008000, WStyle_Mask = 0x0000fff0, WDestructiveClose = 0x00010000, WPaintDesktop = 0x00020000, WPaintUnclipped = 0x00040000, WPaintClever = 0x00080000, WResizeNoErase = 0x00100000, WMouseNoMask = 0x00200000, WStaticContents = 0x00400000, WRepaintNoErase = 0x00800000, WX11BypassWM = 0x01000000, WWinOwnDC = 0x00000000, WGroupLeader = 0x02000000, WShowModal = 0x04000000, WNoMousePropagation =

0x08000000, WSubWindow = 0x10000000, WNorthWestGravity = WStaticContents, WType_Modal = WType_Dialog | WShowModal, WStyle_Dialog = WType_Dialog, WStyle_NoBorderEx = WStyle_NoBorder }

- enum **ImageConversionFlags** { ColorMode_Mask = 0x00000003, AutoColor = 0x00000000, ColorOnly = 0x00000003, MonoOnly = 0x00000002, AlphaDither_Mask = 0x0000000c, ThresholdAlphaDither = 0x00000000, OrderedAlphaDither = 0x00000004, DiffuseAlphaDither = 0x00000008, NoAlpha = 0x0000000c, Dither_Mask = 0x00000030, DiffuseDither = 0x00000000, OrderedDither = 0x00000010, ThresholdDither = 0x00000020, DitherMode_Mask = 0x000000c0, AutoDither = 0x00000000, PreferDither = 0x00000040, AvoidDither = 0x00000080 }
- enum **BGMode** { TransparentMode, OpaqueMode }
- enum **PaintUnit** { PixelUnit, LoMetricUnit, HiMetricUnit, LoEnglishUnit, HiEnglishUnit, TwipsUnit }
- enum GUIStyle { MacStyle, WindowsStyle, Win3Style, PMStyle, MotifStyle } *(obsolete)*
- enum **Modifier** { SHIFT = 0x00200000, CTRL = 0x00400000, ALT = 0x00800000, MODIFIER_MASK = 0x00e00000, UNICODE_ACCEL = 0x10000000, ASCII_ACCEL = UNICODE_ACCEL }
- enum **Key** { Key_Escape = 0x1000, Key_Tab = 0x1001, Key_Backtab = 0x1002, Key_BackTab = Key_Backtab, Key_Backspace = 0x1003, Key_BackSpace = Key_Backspace, Key_Return = 0x1004, Key_Enter = 0x1005, Key_Insert = 0x1006, Key_Delete = 0x1007, Key_Pause = 0x1008, Key_Print = 0x1009, Key_SysReq = 0x100a, Key_Home = 0x1010, Key_End = 0x1011, Key_Left = 0x1012, Key_Up = 0x1013, Key_Right = 0x1014, Key_Down = 0x1015, Key_Prior = 0x1016, Key_PageUp = Key_Prior, Key_Next = 0x1017, Key_PageDown = Key_Next, Key_Shift = 0x1020, Key_Control = 0x1021, Key_Meta = 0x1022, Key_Alt = 0x1023, Key_CapsLock = 0x1024, Key_NumLock = 0x1025, Key_ScrollLock = 0x1026, Key_F1 = 0x1030, Key_F2 = 0x1031, Key_F3 = 0x1032, Key_F4 = 0x1033, Key_F5 = 0x1034, Key_F6 = 0x1035, Key_F7 = 0x1036, Key_F8 = 0x1037, Key_F9 = 0x1038, Key_F10 = 0x1039, Key_F11 = 0x103a, Key_F12 = 0x103b, Key_F13 = 0x103c, Key_F14 = 0x103d, Key_F15 = 0x103e, Key_F16 = 0x103f, Key_F17 = 0x1040, Key_F18 = 0x1041, Key_F19 = 0x1042, Key_F20 = 0x1043, Key_F21 = 0x1044, Key_F22 = 0x1045, Key_F23 = 0x1046, Key_F24 = 0x1047, Key_F25 = 0x1048, Key_F26 = 0x1049, Key_F27 = 0x104a, Key_F28 = 0x104b, Key_F29 = 0x104c, Key_F30 = 0x104d, Key_F31 = 0x104e, Key_F32 = 0x104f, Key_F33 = 0x1050, Key_F34 = 0x1051, Key_F35 = 0x1052, Key_Super_L = 0x1053, Key_Super_R = 0x1054, Key_Menu = 0x1055, Key_Hyper_L = 0x1056, Key_Hyper_R = 0x1057, Key_Help = 0x1058, Key_Direction_L = 0x1059, Key_Direction_R = 0x1060, Key_Space = 0x20, Key_Any = Key_Space, Key_Exclam = 0x21, Key_QuoteDbl = 0x22, Key_NumberSign = 0x23, Key_Dollar = 0x24, Key_Percent = 0x25, Key_Ampersand = 0x26, Key_Apostrophe = 0x27, Key_ParenLeft = 0x28, Key_ParenRight = 0x29, Key_Asterisk = 0x2a, Key_Plus = 0x2b, Key_Comma = 0x2c, Key_Minus = 0x2d, Key_Period = 0x2e, Key_Slash = 0x2f, Key_0 = 0x30, Key_1 = 0x31, Key_2 = 0x32, Key_3 = 0x33, Key_4 = 0x34, Key_5 = 0x35, Key_6 = 0x36, Key_7 = 0x37, Key_8 = 0x38, Key_9 = 0x39, Key_Colon = 0x3a, Key_Semicolon = 0x3b, Key_Less = 0x3c, Key_Equal = 0x3d, Key_Greater = 0x3e, Key_Question = 0x3f, Key_At = 0x40, Key_A = 0x41, Key_B = 0x42, Key_C = 0x43, Key_D = 0x44, Key_E = 0x45, Key_F = 0x46, Key_G = 0x47, Key_H = 0x48, Key_I = 0x49, Key_J = 0x4a, Key_K = 0x4b, Key_L = 0x4c, Key_M = 0x4d, Key_N = 0x4e, Key_O = 0x4f, Key_P = 0x50, Key_Q = 0x51, Key_R = 0x52, Key_S = 0x53, Key_T = 0x54, Key_U = 0x55, Key_V = 0x56, Key_W = 0x57, Key_X = 0x58, Key_Y = 0x59, Key_Z = 0x5a, Key_BracketLeft = 0x5b, Key_Backslash = 0x5c, Key_BracketRight = 0x5d, Key_AsciiCircum = 0x5e, Key_Underscore = 0x5f, Key_QuoteLeft = 0x60, Key_BraceLeft = 0x7b, Key_Bar = 0x7c, Key_BraceRight = 0x7d, Key_AsciiTilde = 0x7e, Key_nobreakspace = 0x0a0, Key_exclamdown = 0x0a1, Key_cent = 0x0a2, Key_sterling = 0x0a3, Key_currency = 0x0a4, Key_yen = 0x0a5, Key_brokenbar = 0x0a6, Key_section = 0x0a7, Key_diaeresis = 0x0a8, Key_copyright = 0x0a9, Key_ordfeminine = 0x0aa, Key_guillemotleft = 0x0ab, Key_notsign = 0x0ac, Key_hyphen = 0x0ad, Key_registered = 0x0ae, Key_macron = 0x0af, Key_degree = 0x0b0, Key_plusminus = 0x0b1, Key_twosuperior = 0x0b2, Key_threesuperior = 0x0b3, Key_acute = 0x0b4, Key_mu = 0x0b5, Key_paragraph = 0x0b6, Key_periodcentered = 0x0b7, Key_cedilla = 0x0b8, Key_onesuperior = 0x0b9, Key_masculine = 0x0ba, Key_guillemotright = 0x0bb, Key_onequarter = 0x0bc, Key_onehalf = 0x0bd, Key_threequarters = 0x0be, Key_questiondown = 0x0bf, Key_Agrave = 0x0c0, Key_Aacute = 0x0c1, Key_Acircumflex = 0x0c2, Key_Atilde = 0x0c3, Key_Adiaeresis = 0x0c4, Key_Aring = 0x0c5, Key_AE = 0x0c6, Key_Ccedilla = 0x0c7, Key_Egrave = 0x0c8, Key_Eacute = 0x0c9, Key_Ecircumflex = 0x0ca, Key_Ediaeresis = 0x0cb, Key_Igrave = 0x0cc, Key_Iacute = 0x0cd, Key_Icircumflex = 0x0ce, Key_Idiaeresis = 0x0cf, Key_ETH = 0x0d0, Key_Ntilde = 0x0d1, Key_Ograve = 0x0d2, Key_Oacute = 0x0d3, Key_Ocircumflex = 0x0d4, Key_Otilde = 0x0d5, Key_Odiaeresis = 0x0d6, Key_multiply = 0x0d7, Key_Ooblique = 0x0d8, Key_Ugrave = 0x0d9, Key_Uacute = 0x0da, Key_Ucircumflex = 0x0db, Key_Udiaeresis = 0x0dc, Key_Yacute = 0x0dd, Key_THORN = 0x0de,

Key_ssharp = 0x0df, Key_agrave = 0x0e0, Key_aacute = 0x0e1, Key_acircumflex = 0x0e2, Key_atilde = 0x0e3, Key_adiaeresis = 0x0e4, Key_aring = 0x0e5, Key_ae = 0x0e6, Key_ccedilla = 0x0e7, Key_egrave = 0x0e8, Key_eacute = 0x0e9, Key_ecircumflex = 0x0ea, Key_ediaeresis = 0x0eb, Key_igrave = 0x0ec, Key_iacute = 0x0ed, Key_icircumflex = 0x0ee, Key_idiaeresis = 0x0ef, Key_eth = 0x0f0, Key_ntilde = 0x0f1, Key_ograve = 0x0f2, Key_oacute = 0x0f3, Key_ocircumflex = 0x0f4, Key_otilde = 0x0f5, Key_odiaeresis = 0x0f6, Key_division = 0x0f7, Key_oslash = 0x0f8, Key_ugrave = 0x0f9, Key_uacute = 0x0fa, Key_ucircumflex = 0x0fb, Key_udiaeresis = 0x0fc, Key_yacute = 0x0fd, Key_thorn = 0x0fe, Key_ydiaeresis = 0x0ff, Key_unknown = 0xffff }

- enum **ArrowType** { UpArrow, DownArrow, LeftArrow, RightArrow }
- enum **RasterOp** { CopyROP, OrROP, XorROP, NotAndROP, EraseROP = NotAndROP, NotCopyROP, NotOrROP, NotXorROP, AndROP, NotEraseROP = AndROP, NotROP, ClearROP, SetROP, NopROP, AndNotROP, OrNotROP, NandROP, NorROP, LastROP = NorROP }
- enum **PenStyle** { NoPen, SolidLine, DashLine, DotLine, DashDotLine, DashDotDotLine, MPenStyle = 0x0f }
- enum **PenCapStyle** { FlatCap = 0x00, SquareCap = 0x10, RoundCap = 0x20, MPenCapStyle = 0x30 }
- enum **PenJoinStyle** { MiterJoin = 0x00, BevelJoin = 0x40, RoundJoin = 0x80, MPenJoinStyle = 0xc0 }
- enum **BrushStyle** { NoBrush, SolidPattern, Dense1Pattern, Dense2Pattern, Dense3Pattern, Dense4Pattern, Dense5Pattern, Dense6Pattern, Dense7Pattern, HorPattern, VerPattern, CrossPattern, BDiagPattern, FDiagPattern, DiagCrossPattern, CustomPattern = 24 }
- enum **WindowsVersion** { WV_32s = 0x0001, WV_95 = 0x0002, WV_98 = 0x0003, WV_Me = 0x0004, WV_DOS_based = 0x000f, WV_NT = 0x0010, WV_2000 = 0x0020, WV_XP = 0x0030, WV_NT_based = 0x00f0 }
- enum **UIEffect** { UI_General, UI_AnimateMenu, UI_FadeMenu, UI_AnimateCombo, UI_AnimateTooltip, UI_FadeTooltip }
- enum **CursorShape** { ArrowCursor, UpArrowCursor, CrossCursor, WaitCursor, IbeamCursor, SizeVerCursor, SizeHorCursor, SizeBDiagCursor, SizeFDiagCursor, SizeAllCursor, BlankCursor, SplitVCursor, SplitHCursor, PointingHandCursor, ForbiddenCursor, WhatsThisCursor, LastCursor = WhatsThisCursor, BitmapCursor = 24 }
- enum **TextFormat** { PlainText, RichText, AutoText }
- enum **Dock** { DockUnmanaged, DockTornOff, DockTop, DockBottom, DockRight, DockLeft, DockMinimized, Unmanaged = DockUnmanaged, TornOff = DockTornOff, Top = DockTop, Bottom = DockBottom, Right = DockRight, Left = DockLeft, Minimized = DockMinimized }
- enum **DateFormat** { TextDate, ISODate, LocalDate }
- enum **BackgroundMode** { FixedColor, FixedPixmap, NoBackground, PaletteForeground, PaletteButton, PaletteLight, PaletteMidlight, PaletteDark, PaletteMid, PaletteText, PaletteBrightText, PaletteBase, PaletteBackground, PaletteShadow, PaletteHighlight, PaletteHighlightedText, PaletteButtonText, PaletteLink, PaletteLinkVisited, X11ParentRelative }
- enum **StringComparisonMode** { CaseSensitive = 0x00001, BeginsWith = 0x00002, EndsWith = 0x00004, Contains = 0x00008, ExactMatch = 0x00010 }

Detailed Description

The Qt class is a namespace for miscellaneous identifiers that need to be global-like.

Normally, you can ignore this class. QObject and a few other classes inherit it, so all the identifiers in the Qt namespace are usable without qualification.

However, you may occasionally need to say `Qt::black` instead of just `black`, particularly in static utility functions (such as many class factories).

See also Miscellaneous Classes.

Member Type Documentation

Qt::AlignmentFlags

This enum type is used to describe alignment. It contains horizontal and vertical flags. The horizontal flags are:

- `Qt::AlignAuto` - Aligns according to the language. Left for most, right for Hebrew and Arabic.
- `Qt::AlignLeft` - Aligns with the left edge.
- `Qt::AlignRight` - Aligns with the right edge.
- `Qt::AlignHCenter` - Centers horizontally in the available space.
- `Qt::AlignJustify` - Justifies the text in the available space. Does not work for everything and may be interpreted as AlignAuto in some cases.

The vertical flags are:

- `Qt::AlignTop` - Aligns with the top.
- `Qt::AlignBottom` - Aligns with the bottom.
- `Qt::AlignVCenter` - Centers vertically in the available space.

You can use only one of the horizontal flags at a time. There is one two-dimensional flag:

- `Qt::AlignCenter` - Centers in both dimensions.

You can use at most one horizontal and one vertical flag at a time. AlignCenter counts as both horizontal and vertical.

Masks:

- `Qt::AlignHorizontal_Mask`
- `Qt::AlignVertical_Mask`

Conflicting combinations of flags have undefined meanings.

Qt::ArrowType

- `Qt::UpArrow`
- `Qt::DownArrow`
- `Qt::LeftArrow`
- `Qt::RightArrow`

Qt::BGMode

Background mode

- `Qt::TransparentMode`
- `Qt::OpaqueMode`

Qt::BackgroundMode

This enum describes how the background of a widget changes, as the widget's palette changes.

The background is what the widget contains when paintEvent() is called. To minimize flicker, this should be the most common color or pixmap in the widget. For PaletteBackground, use colorGroup().brush(QColorGroup::Background), and so on. There are also three special values, listed at the end:

- `Qt::PaletteForeground`
- `Qt::PaletteBackground`
- `Qt::PaletteButton`
- `Qt::PaletteLight`
- `Qt::PaletteMidlight`
- `Qt::PaletteDark`
- `Qt::PaletteMid`
- `Qt::PaletteText`
- `Qt::PaletteBrightText`
- `Qt::PaletteButtonText`
- `Qt::PaletteBase`
- `Qt::PaletteShadow`
- `Qt::PaletteHighlight`
- `Qt::PaletteHighlightedText`
- `Qt::NoBackground` - the widget is not cleared before paintEvent(). If the widget's paint event always draws on all the pixels, using this mode can be both fast and flicker-free.
- `Qt::FixedColor` - the widget is cleared to a fixed color, normally different from all the ones in the palette(). Set using setPaletteBackgroundColor().
- `Qt::FixedPixmap` - the widget is cleared to a fixed pixmap, normally different from all the ones in the palette(). Set using setBackgroundPixmap().
- `Qt::PaletteLink`
- `Qt::` - PaletteLinkVisited
- `Qt::X11ParentRelative` - (internal use only)

Although FixedColor and FixedPixmap are sometimes just right, if you use them, make sure that your application looks right when the desktop color scheme has been changed. (On X11, a quick way to test this is e.g. "./myapp -bg paleblue". On Windows, you have to use the control panel.)

See also QWidget::backgroundMode [Widgets with Qt], QWidget::backgroundMode [Widgets with Qt], QWidget::setBackgroundPixmap() [Widgets with Qt] and QWidget::paletteBackgroundColor [Widgets with Qt].

Qt::BrushStyle

- `Qt::NoBrush`
- `Qt::SolidPattern`
- `Qt::Dense1Pattern`
- `Qt::Dense2Pattern`
- `Qt::Dense3Pattern`
- `Qt::Dense4Pattern`
- `Qt::Dense5Pattern`

- `Qt::Dense6Pattern`
- `Qt::Dense7Pattern`
- `Qt::HorPattern`
- `Qt::VerPattern`
- `Qt::CrossPattern`
- `Qt::BDiagPattern`
- `Qt::FDiagPattern`
- `Qt::DiagCrossPattern`
- `Qt::CustomPattern`

Qt::ButtonState

This enum type describes the state of the mouse buttons and the modifier buttons. The currently defined values are:

- `Qt::NoButton` - used when the button state does not refer to any button (see QMouseEvent::button()).
- `Qt::LeftButton` - set if the left button is pressed, or this event refers to the left button. Note that the left button may be the right button on left-handed mice.
- `Qt::RightButton` - the right button.
- `Qt::MidButton` - the middle button.
- `Qt::ShiftButton` - a Shift key on the keyboard is also pressed.
- `Qt::ControlButton` - a Ctrl key on the keyboard is also pressed.
- `Qt::AltButton` - an Alt (or Meta) key on the keyboard is also pressed.
- `Qt::MetaButton` - a Meta key on the keyboard is also pressed.
- `Qt::Keypad` - a keypad button is pressed.
- `Qt::KeyButtonMask` - is a mask for ShiftButton, ControlButton and AltButton.
- `Qt::MouseButtonMask` - is a mask for LeftButton, RightButton and MidButton.

Qt::CursorShape

This enum type defines the various cursors that can be used.

- `Qt::ArrowCursor` - standard arrow cursor
- `Qt::UpArrowCursor` - upwards arrow
- `Qt::CrossCursor` - crosshair
- `Qt::WaitCursor` - hourglass/watch
- `Qt::IbeamCursor` - ibeam/text entry
- `Qt::SizeVerCursor` - vertical resize
- `Qt::SizeHorCursor` - horizontal resize
- `Qt::SizeBDiagCursor` - diagonal resize (/)
- `Qt::SizeFDiagCursor` - diagonal resize (\)
- `Qt::SizeAllCursor` - all directions resize
- `Qt::BlankCursor` - blank/invisible cursor
- `Qt::SplitVCursor` - vertical splitting
- `Qt::SplitHCursor` - horziontal splitting

- `Qt::PointingHandCursor` - a pointing hand
- `Qt::ForbiddenCursor` - a slashed circle
- `Qt::WhatsThisCursor` - an arrow with a question mark
- `Qt::BitmapCursor`

ArrowCursor is the default for widgets in a normal state.

Qt::DateFormat

- `Qt::TextDate` - (default) Qt format
- `Qt::ISODate` - ISO 8601 format
- `Qt::LocalDate` - locale dependent format

Qt::Dock

Each dock window can be in one of the following positions:

- `Qt::DockTop` - above the central widget, below the menu bar.
- `Qt::DockBottom` - below the central widget, above the status bar.
- `Qt::DockLeft` - to the left of the central widget.
- `Qt::DockRight` - to the right of the central widget.
- `Qt::DockMinimized` - the dock window is not shown (this is effectively a 'hidden' dock area); the handles of all minimized dock windows are drawn in one row below the menu bar.
- `Qt::DockTornOff` - the dock window floats as its own top level window which always stays on top of the main window.
- `Qt::DockUnmanaged` - not managed by a QMainWindow.

Qt::GUIStyle

This type is obsolete. It is provided to keep old source working. We strongly advise against using it in new code.

- `Qt::WindowsStyle`
- `Qt::MotifStyle`
- `Qt::MacStyle`
- `Qt::Win3Style`
- `Qt::PMStyle`

Qt::ImageConversionFlags

The conversion flag is a bitwise-OR of the following values. The options marked "(default)" are the set if no other values from the list are included (since the defaults are zero):

Color/Mono preference (ignored for QBitmap)

- `Qt::AutoColor` - (default) - If the image has depth 1 and contains only black and white pixels, the pixmap becomes monochrome.
- `Qt::ColorOnly` - The pixmap is dithered/converted to the native display depth.

- `Qt::MonoOnly` - The pixmap becomes monochrome. If necessary, it is dithered using the chosen dithering algorithm.

Dithering mode preference for RGB channels

- `Qt::DiffuseDither` - (default) - A high-quality dither.
- `Qt::OrderedDither` - A faster, more ordered dither.
- `Qt::ThresholdDither` - No dithering; closest color is used.

Dithering mode preference for alpha channel

- `Qt::ThresholdAlphaDither` - (default) - No dithering.
- `Qt::OrderedAlphaDither` - A faster, more ordered dither.
- `Qt::DiffuseAlphaDither` - A high-quality dither.
- `Qt::NoAlpha` - Not supported.

Color matching versus dithering preference

- `Qt::PreferDither` - (default when converting to a pixmap) - Always dither 32-bit images when the image is converted to 8 bits.
- `Qt::AvoidDither` - (default when converting for the purpose of saving to file) - Dither 32-bit images only if the image has more than 256 colors and it is being converted to 8 bits.
- `Qt::AutoDither` - Not supported.

The following are not values that are used directly, but masks for the above classes:

- `Qt::ColorMode_Mask` - Mask for the color mode.
- `Qt::Dither_Mask` - Mask for the dithering mode for RGB channels.
- `Qt::AlphaDither_Mask` - Mask for the dithering mode for the alpha channel.
- `Qt::DitherMode_Mask` - Mask for the mode that determines the preference of color matching versus dithering.

Using 0 as the conversion flag sets all the default options.

Qt::Key

The key names used by Qt.

- `Qt::Key_Escape`
- `Qt::Key_Tab`
- `Qt::Key_Backtab`
- `Qt::Key_Backspace`
- `Qt::Key_Return`
- `Qt::Key_Enter`
- `Qt::Key_Insert`
- `Qt::Key_Delete`
- `Qt::Key_Pause`
- `Qt::Key_Print`
- `Qt::Key_SysReq`

- `Qt::Key_Home`
- `Qt::Key_End`
- `Qt::Key_Left`
- `Qt::Key_Up`
- `Qt::Key_Right`
- `Qt::Key_Down`
- `Qt::Key_Prior`
- `Qt::Key_Next`
- `Qt::Key_Shift`
- `Qt::Key_Control`
- `Qt::Key_Meta`
- `Qt::Key_Alt`
- `Qt::Key_CapsLock`
- `Qt::Key_NumLock`
- `Qt::Key_ScrollLock`
- `Qt::Key_F1`
- `Qt::Key_F2`
- `Qt::Key_F3`
- `Qt::Key_F4`
- `Qt::Key_F5`
- `Qt::Key_F6`
- `Qt::Key_F7`
- `Qt::Key_F8`
- `Qt::Key_F9`
- `Qt::Key_F10`
- `Qt::Key_F11`
- `Qt::Key_F12`
- `Qt::Key_F13`
- `Qt::Key_F14`
- `Qt::Key_F15`
- `Qt::Key_F16`
- `Qt::Key_F17`
- `Qt::Key_F18`
- `Qt::Key_F19`
- `Qt::Key_F20`
- `Qt::Key_F21`
- `Qt::Key_F22`
- `Qt::Key_F23`
- `Qt::Key_F24`
- `Qt::Key_F25`
- `Qt::Key_F26`
- `Qt::Key_F27`
- `Qt::Key_F28`
- `Qt::Key_F29`

- Qt::Key_F30
- Qt::Key_F31
- Qt::Key_F32
- Qt::Key_F33
- Qt::Key_F34
- Qt::Key_F35
- Qt::Key_Super_L
- Qt::Key_Super_R
- Qt::Key_Menu
- Qt::Key_Hyper_L
- Qt::Key_Hyper_R
- Qt::Key_Help
- Qt::Key_Space
- Qt::Key_Any
- Qt::Key_Exclam
- Qt::Key_QuoteDbl
- Qt::Key_NumberSign
- Qt::Key_Dollar
- Qt::Key_Percent
- Qt::Key_Ampersand
- Qt::Key_Apostrophe
- Qt::Key_ParenLeft
- Qt::Key_ParenRight
- Qt::Key_Asterisk
- Qt::Key_Plus
- Qt::Key_Comma
- Qt::Key_Minus
- Qt::Key_Period
- Qt::Key_Slash
- Qt::Key_0
- Qt::Key_1
- Qt::Key_2
- Qt::Key_3
- Qt::Key_4
- Qt::Key_5
- Qt::Key_6
- Qt::Key_7
- Qt::Key_8
- Qt::Key_9
- Qt::Key_Colon
- Qt::Key_Semicolon
- Qt::Key_Less
- Qt::Key_Equal
- Qt::Key_Greater

- `Qt::Key_Question`
- `Qt::Key_At`
- `Qt::Key_A`
- `Qt::Key_B`
- `Qt::Key_C`
- `Qt::Key_D`
- `Qt::Key_E`
- `Qt::Key_F`
- `Qt::Key_G`
- `Qt::Key_H`
- `Qt::Key_I`
- `Qt::Key_J`
- `Qt::Key_K`
- `Qt::Key_L`
- `Qt::Key_M`
- `Qt::Key_N`
- `Qt::Key_O`
- `Qt::Key_P`
- `Qt::Key_Q`
- `Qt::Key_R`
- `Qt::Key_S`
- `Qt::Key_T`
- `Qt::Key_U`
- `Qt::Key_V`
- `Qt::Key_W`
- `Qt::Key_X`
- `Qt::Key_Y`
- `Qt::Key_Z`
- `Qt::Key_BracketLeft`
- `Qt::Key_Backslash`
- `Qt::Key_BracketRight`
- `Qt::Key_AsciiCircum`
- `Qt::Key_Underscore`
- `Qt::Key_QuoteLeft`
- `Qt::Key_BraceLeft`
- `Qt::Key_Bar`
- `Qt::Key_BraceRight`
- `Qt::Key_AsciiTilde`
- `Qt::Key_nobreakspace`
- `Qt::Key_exclamdown`
- `Qt::Key_cent`
- `Qt::Key_sterling`
- `Qt::Key_currency`
- `Qt::Key_yen`

- `Qt::Key_brokenbar`
- `Qt::Key_section`
- `Qt::Key_diaeresis`
- `Qt::Key_copyright`
- `Qt::Key_ordfeminine`
- `Qt::Key_guillemotleft`
- `Qt::Key_notsign`
- `Qt::Key_hyphen`
- `Qt::Key_registered`
- `Qt::Key_macron`
- `Qt::Key_degree`
- `Qt::Key_plusminus`
- `Qt::Key_twosuperior`
- `Qt::Key_threesuperior`
- `Qt::Key_acute`
- `Qt::Key_mu`
- `Qt::Key_paragraph`
- `Qt::Key_periodcentered`
- `Qt::Key_cedilla`
- `Qt::Key_onesuperior`
- `Qt::Key_masculine`
- `Qt::Key_guillemotright`
- `Qt::Key_onequarter`
- `Qt::Key_onehalf`
- `Qt::Key_threequarters`
- `Qt::Key_questiondown`
- `Qt::Key_Agrave`
- `Qt::Key_Aacute`
- `Qt::Key_Acircumflex`
- `Qt::Key_Atilde`
- `Qt::Key_Adiaeresis`
- `Qt::Key_Aring`
- `Qt::Key_AE`
- `Qt::Key_Ccedilla`
- `Qt::Key_Egrave`
- `Qt::Key_Eacute`
- `Qt::Key_Ecircumflex`
- `Qt::Key_Ediaeresis`
- `Qt::Key_Igrave`
- `Qt::Key_Iacute`
- `Qt::Key_Icircumflex`
- `Qt::Key_Idiaeresis`
- `Qt::Key_ETH`
- `Qt::Key_Ntilde`

- `Qt::Key_Ograve`
- `Qt::Key_Oacute`
- `Qt::Key_Ocircumflex`
- `Qt::Key_Otilde`
- `Qt::Key_Odiaeresis`
- `Qt::Key_multiply`
- `Qt::Key_Ooblique`
- `Qt::Key_Ugrave`
- `Qt::Key_Uacute`
- `Qt::Key_Ucircumflex`
- `Qt::Key_Udiaeresis`
- `Qt::Key_Yacute`
- `Qt::Key_THORN`
- `Qt::Key_ssharp`
- `Qt::Key_agrave`
- `Qt::Key_aacute`
- `Qt::Key_acircumflex`
- `Qt::Key_atilde`
- `Qt::Key_adiaeresis`
- `Qt::Key_aring`
- `Qt::Key_ae`
- `Qt::Key_ccedilla`
- `Qt::Key_egrave`
- `Qt::Key_eacute`
- `Qt::Key_ecircumflex`
- `Qt::Key_ediaeresis`
- `Qt::Key_igrave`
- `Qt::Key_iacute`
- `Qt::Key_icircumflex`
- `Qt::Key_idiaeresis`
- `Qt::Key_eth`
- `Qt::Key_ntilde`
- `Qt::Key_ograve`
- `Qt::Key_oacute`
- `Qt::Key_ocircumflex`
- `Qt::Key_otilde`
- `Qt::Key_odiaeresis`
- `Qt::Key_division`
- `Qt::Key_oslash`
- `Qt::Key_ugrave`
- `Qt::Key_uacute`
- `Qt::Key_ucircumflex`
- `Qt::Key_udiaeresis`
- `Qt::Key_yacute`

- `Qt::Key_thorn`
- `Qt::Key_ydiaeresis`
- `Qt::Key_unknown`
- `Qt::Key_Direction_L` - internal use only
- `Qt::Key_Direction_R` - internal use only

Qt::Modifier

This enum type describes the keyboard modifier keys supported by Qt. The currently defined values are:

- `Qt::SHIFT` - the Shift keys provided on all standard keyboards.
- `Qt::CTRL` - the Ctrl keys.
- `Qt::ALT` - the normal Alt keys, but not e.g. AltGr.
- `Qt::MODIFIER_MASK` - is a mask of Shift, Ctrl and Alt.
- `Qt::UNICODE_ACCEL` - the accelerator is specified as a Unicode code point, not as a Qt Key.

Qt::Orientation

This type is used to signify an object's orientation.

- `Qt::Horizontal`
- `Qt::Vertical`

Orientation is used with QScrollBar for example.

Qt::PaintUnit

- `Qt::PixelUnit`
- `Qt::LoMetricUnit` - *obsolete*
- `Qt::HiMetricUnit` - *obsolete*
- `Qt::LoEnglishUnit` - *obsolete*
- `Qt::HiEnglishUnit` - *obsolete*
- `Qt::TwipsUnit` - *obsolete*

Qt::PenCapStyle

This enum type defines the pen cap styles supported by Qt, i.e. the line end caps that can be drawn using QPainter. The available styles are:

- `Qt::FlatCap` - a square line end that does not cover the end point of the line.
- `Qt::SquareCap` - a square line end that covers the end point and extends beyond it with half the line width.
- `Qt::RoundCap` - a rounded line end.
- `Qt::MPenCapStyle` - mask of the pen cap styles.

Qt::PenJoinStyle

This enum type defines the pen join styles supported by Qt, i.e. which joins between two connected lines can be drawn using QPainter. The available styles are:

- `Qt::MiterJoin` - The outer edges of the lines are extended to meet at an angle, and this area is filled.
- `Qt::BevelJoin` - The triangular notch between the two lines is filled.
- `Qt::RoundJoin` - A circular arc between the two lines is filled.
- `Qt::MPenJoinStyle` - mask of the pen join styles.

Qt::PenStyle

This enum type defines the pen styles that can be drawn using QPainter. The styles are

- `Qt::NoPen` - no line at all. For example, QPainter::drawRect() fills but does not draw any boundary line.
- `Qt::SolidLine` - a simple line.
- `Qt::DashLine` - dashes separated by a few pixels.
- `Qt::DotLine` - dots separated by a few pixels.
- `Qt::DashDotLine` - alternate dots and dashes.
- `Qt::DashDotDotLine` - one dash, two dots, one dash, two dots.
- `Qt::MPenStyle` - mask of the pen styles.

Qt::RasterOp

This enum type is used to describe the way things are written to the paint device. Each bit of the *src* (what you write) interacts with the corresponding bit of the *dst* pixel. The currently defined values are:

- `Qt::CopyROP` - dst = src
- `Qt::OrROP` - dst = src OR dst
- `Qt::XorROP` - dst = src XOR dst
- `Qt::NotAndROP` - dst = (NOT src) AND dst
- `Qt::EraseROP` - an alias for NotAndROP
- `Qt::NotCopyROP` - dst = NOT src
- `Qt::NotOrROP` - dst = (NOT src) OR dst
- `Qt::NotXorROP` - dst = (NOT src) XOR dst
- `Qt::AndROP` - dst = src AND dst
- `Qt::NotEraseROP` - an alias for AndROP
- `Qt::NotROP` - dst = NOT dst
- `Qt::ClearROP` - dst = 0
- `Qt::SetROP` - dst = 1
- `Qt::NopROP` - dst = dst
- `Qt::AndNotROP` - dst = src AND (NOT dst)
- `Qt::OrNotROP` - dst = src OR (NOT dst)
- `Qt::NandROP` - dst = NOT (src AND dst)
- `Qt::NorROP` - dst = NOT (src OR dst)

By far the most useful ones are CopyROP and XorROP.

Qt::StringComparisonMode

This enum type is used to set the string comparison mode when searching for an item. This is implemented in QListBox, QListView and QIconView, for example. We'll refer to the string being searched as the 'target' string.

- `Qt::CaseSensitive` - The strings must match case sensitively.
- `Qt::ExactMatch` - The target and search strings must match exactly.
- `Qt::BeginsWith` - The target string begins with the search string.
- `Qt::EndsWith` - The target string ends with the search string.
- `Qt::Contains` - The target string contains the search string.

If you OR these flags together (excluding CaseSensitive), the search criteria be applied in the following order: ExactMatch, BeginsWith, EndsWith, Contains.

Matching is case-insensitive unless CaseSensitive is set. CaseSensitive may be OR-ed with any combination of the other flags.

Qt::TextFlags

This enum type is used to define some modifier flags. Some of these flags only make sense in the context of printing:

- `Qt::SingleLine` - Treats all whitespace as spaces and prints just one line.
- `Qt::DontClip` - If it's impossible to stay within the given bounds, it prints outside.
- `Qt::ExpandTabs` - Makes the U+0009 (ASCII tab) character move to the next tab stop.
- `Qt::ShowPrefix` - Displays the string "&P" as an underlined P (see QButton for an example). For an ampersand, use "&&".
- `Qt::WordBreak` - Breaks lines at appropriate points, e.g. at word boundaries.
- `Qt::BreakAnywhere` - Breaks lines anywhere, even within words.
- `Qt::NoAccel` - Synonym for ShowPrefix.
- `Qt::DontPrint` - (internal)

You can use as many modifier flags as you want, except that SingleLine and WordBreak cannot be combined.

Flags that are inappropriate for a given use (e.g. ShowPrefix to QGridLayout::addWidget()) are generally ignored.

Qt::TextFormat

This enum is used in widgets that can display both plain text and rich text, e.g. QLabel. It is used for deciding whether a text string should be interpreted as one or the other. This is normally done by passing one of the enum values to a setTextFormat() function.

- `Qt::PlainText` - The text string is interpreted as a plain text string.
- `Qt::RichText` - The text string is interpreted as a rich text string using the current QStyleSheet::defaultSheet().
- `Qt::AutoText` - The text string is interpreted as for RichText if QStyleSheet::mightBeRichText() returns TRUE, otherwise as for PlainText.

Qt::UIEffect

- `Qt::UI_General`
- `Qt::UI_AnimateMenu`
- `Qt::UI_FadeMenu`
- `Qt::UI_AnimateCombo`
- `Qt::UI_AnimateTooltip`
- `Qt::UI_FadeTooltip`

Qt::WidgetFlags

This enum type is used to specify various window-system properties of the widget. They are fairly unusual but necessary in a few cases. Some of these flags depend on whether the underlying window manager supports them.

The main types are

- `Qt::WType_TopLevel` - indicates that this widget is a top-level widget, usually with a window-system frame and so on.
- `Qt::WType_Dialog` - indicates that this widget is a secondary top-level widget. When combined with WShow-Modal, the dialog becomes a modal dialog i.e. prevents any other top-level window in the application from getting any input. WType_Dialog implies WType_TopLevel.
- `Qt::WType_Popup` - indicates that this widget is a popup top-level window, i.e. that it is modal, but has a window system frame appropriate for popup menus. WType_Popup implies WType_TopLevel.
- `Qt::WType_Desktop` - indicates that this widget is the desktop. See also WPaintDesktop [p. 192] below. WType_Desktop [p. 192] implies WType_TopLevel [p. 192].

There are also a number of flags which you can use to customize the appearance of top-level windows. These have no effect on other windows:

- `Qt::WStyle_Customize` - indicates that the `WStyle_*` flags should be used to build the window instead of the default flags.
- `Qt::WStyle_NormalBorder` - gives the window a normal border. Cannot be combined with WStyle_DialogBorder or WStyle_NoBorder.
- `Qt::WStyle_DialogBorder` - gives the window a thin dialog border. Cannot be combined with WStyle_NormalBorder or WStyle_NoBorder.
- `Qt::WStyle_NoBorder` - produces a borderless window. Note that the user cannot move or resize a borderless window via the window system. Cannot be combined with WStyle_NormalBorder or WStyle_DialogBorder. On Windows, the flag works fine. On X11, the result of the flag is dependent on the window manager and its ability to understand MOTIF and/or NETWM hints: most existing modern window managers can handle this. With WX11BypassWM, you can bypass the window manager completely. This results in a borderless window that is not managed at all (i.e. no keyboard input unless you call setActiveWindow() manually).
- `Qt::WStyle_NoBorderEx` - this value is obsolete. It has the same effect as using WStyle_NoBorder.
- `Qt::WStyle_Title` - gives the window a title bar.
- `Qt::WStyle_SysMenu` - adds a window system menu.
- `Qt::WStyle_Minimize` - adds a minimize button. Note that on Windows this has to be combined with WStyle_SysMenu for it to work.
- `Qt::WStyle_Maximize` - adds a maximize button. Note that on Windows this has to be combined with WStyle_SysMenu for it to work.
- `Qt::WStyle_MinMax` - is equal to `WStyle_Minimize|WStyle_Maximize`. Note that on Windows this has to be combined with WStyle_SysMenu to work.
- `Qt::WStyle_ContextHelp` - adds a context help button to dialogs.

- `Qt::WStyle_Tool` - makes the window a tool window. A tool window is often a small window with a smaller than usual title bar and decoration, typically used for collections of tool buttons. It there is a parent, the tool window will always be kept on top of it. If there isn't a parent, you may consider passing WStyle_StaysOnTop as well. If the window system supports it, a tool window can be decorated with a somewhat lighter frame. It can also be combined with WStyle_NoBorder.
- `Qt::WStyle_StaysOnTop` - informs the window system that the window should stay on top of all other windows.
- `Qt::WStyle_Dialog` - indicates that the window is a logical subwindow of its parent (in other words, a dialog). The window will not get its own taskbar entry and will be kept on top of its parent by the window system. Usually it will also be minimized when the parent is minimized. If not customized, the window is decorated with a slightly simpler title bar. This is the flag QDialog uses.

Modifier flags:

- `Qt::WDestructiveClose` - makes Qt delete this object when the object has accepted closeEvent(), or when the widget tried to ignore closeEvent() but could not.
- `Qt::WPaintDesktop` - gives this widget paint events for the desktop.
- `Qt::WPaintUnclipped` - makes all painters operating on this widget unclipped. Children of this widget or other widgets in front of it do not clip the area the painter can paint on.
- `Qt::WPaintClever` - indicates that Qt should *not* try to optimize repainting for the widget, but instead pass on window system repaint events directly. (This tends to produce more events and smaller repaint regions.)
- `Qt::WResizeNoErase` - indicates that resizing the widget should not erase it. This allows smart-repainting to avoid flicker.
- `Qt::WMouseNoMask` - indicates that even if the widget has a mask, it wants mouse events for its entire rectangle.
- `Qt::WStaticContents` - indicates that the widget contents are north-west aligned and static. On resize, such a widget will receive paint events only for the newly visible part of itself.
- `Qt::WRepaintNoErase` - indicates that the widget paints all its pixels. Updating, scrolling and focus changes should therefore not erase the widget. This allows smart-repainting to avoid flicker.
- `Qt::WGroupLeader` - makes this widget or window a group leader. Modality of secondary windows only affects windows within the same group.

Miscellaneous flags

- `Qt::WShowModal` - see WType_Dialog

Internal flags.

- `Qt::WNoMousePropagation`
- `Qt::WStaticContents`
- `Qt::WStyle_Reserved`
- `Qt::WSubWindow`
- `Qt::WType_Modal`
- `Qt::WWinOwnDC`
- `Qt::WX11BypassWM`
- `Qt::WStyle_Mask`
- `Qt::WType_Mask`

Qt::WidgetState

Internal flags.

- `Qt::WState_Created`
- `Qt::WState_Disabled`
- `Qt::WState_Visible`
- `Qt::WState_ForceHide`
- `Qt::WState_OwnCursor`
- `Qt::WState_MouseTracking`
- `Qt::WState_CompressKeys`
- `Qt::WState_BlockUpdates`
- `Qt::WState_InPaintEvent`
- `Qt::WState_Reparented`
- `Qt::WState_ConfigPending`
- `Qt::WState_Resized`
- `Qt::WState_AutoMask`
- `Qt::WState_Polished`
- `Qt::WState_DND`
- `Qt::WState_Reserved0`
- `Qt::WState_Reserved1`
- `Qt::WState_Reserved2`
- `Qt::WState_Reserved3`
- `Qt::WState_Maximized`
- `Qt::WState_Minimized`
- `Qt::WState_ForceDisabled`
- `Qt::WState_Exposed`
- `Qt::WState_HasMouse`

Qt::WindowsVersion

- `Qt::WV_32s`
- `Qt::WV_95`
- `Qt::WV_98`
- `Qt::WV_Me`
- `Qt::WV_DOS_based`
- `Qt::WV_NT`
- `Qt::WV_2000`
- `Qt::WV_XP`
- `Qt::WV_NT_based`

QTableSelection Class Reference

The QTableSelection class provides access to a selected area in a QTable.

This class is part of the **table** module.

```
#include <qtable.h>
```

Public Members

- **QTableSelection** ()
- void **init** (int row, int col)
- void **expandTo** (int row, int col)
- bool **operator==** (const QTableSelection & s) const
- int **topRow** () const
- int **bottomRow** () const
- int **leftCol** () const
- int **rightCol** () const
- int **anchorRow** () const
- int **anchorCol** () const
- bool **isActive** () const

Detailed Description

The QTableSelection class provides access to a selected area in a QTable.

The selection is a rectangular set of cells. One of the rectangle's cells is called the anchor cell; this is the cell that was selected first. The init() function sets the anchor and the selection rectangle to exactly this cell; the expandTo() function expands the selection rectangle to include additional cells.

There are various access functions to find out about the area: anchorRow() and anchorCol() return the anchor's position; leftCol(), rightCol(), topRow() and bottomRow() return the rectangle's four edges. All four are part of the selection.

A newly created QTableSelection is inactive — isActive() returns FALSE. You must use init() and expandTo() to activate it.

See also QTable [Widgets with Qt], QTable::addSelection() [Widgets with Qt], QTable::selection() [Widgets with Qt] and Advanced Widgets.

Member Function Documentation

QTableSelection::QTableSelection ()

Creates an inactive selection. Use init() and expandTo() to activate it.

int QTableSelection::anchorCol () const

Returns the anchor column of the selection.

See also anchorRow() [p. 196] and expandTo() [p. 196].

int QTableSelection::anchorRow () const

Returns the anchor row of the selection.

See also anchorCol() [p. 196] and expandTo() [p. 196].

int QTableSelection::bottomRow () const

Returns the bottom row of the selection.

See also topRow() [p. 197], leftCol() [p. 196] and rightCol() [p. 197].

void QTableSelection::expandTo (int row, int col)

Expands the selection to include cell *row*, *col*. The new selection rectangle is the bounding rectangle of *row*, *col* and the previous selection rectangle. After calling this function the selection is active.

If you haven't called init(), this function does nothing.

See also init() [p. 196] and isActive() [p. 196].

void QTableSelection::init (int row, int col)

Sets the selection anchor to cell *row*, *col* and the selection to contain only this cell.

To extend the selection to include additional cells, call expandTo().

See also isActive() [p. 196].

bool QTableSelection::isActive () const

Returns whether the selection is active or not. A selection is active after init() and expandTo() have been called.

int QTableSelection::leftCol () const

Returns the left column of the selection.

See also topRow() [p. 197], bottomRow() [p. 196] and rightCol() [p. 197].

bool QTableSelection::operator== (const QTableSelection & s) const

Returns TRUE if *s* includes the same cells as the selection; otherwise returns FALSE.

int QTableSelection::rightCol () const

Returns the right column of the selection.

See also topRow() [p. 197], bottomRow() [p. 196] and leftCol() [p. 196].

int QTableSelection::topRow () const

Returns the top row of the selection.

See also bottomRow() [p. 196], leftCol() [p. 196] and rightCol() [p. 197].

QTime Class Reference

The QTime class provides clock time functions.

`#include <qdatetime.h>`

Public Members

- **QTime** ()
- **QTime** (int h, int m, int s = 0, int ms = 0)
- bool **isNull** () const
- bool **isValid** () const
- int **hour** () const
- int **minute** () const
- int **second** () const
- int **msec** () const
- QString **toString** (Qt::DateFormat f = Qt::TextDate) const
- QString **toString** (const QString & format) const
- bool **setHMS** (int h, int m, int s, int ms = 0)
- QTime **addSecs** (int nsecs) const
- int **secsTo** (const QTime & t) const
- QTime **addMSecs** (int ms) const
- int **msecsTo** (const QTime & t) const
- bool **operator==** (const QTime & t) const
- bool **operator!=** (const QTime & t) const
- bool **operator<** (const QTime & t) const
- bool **operator<=** (const QTime & t) const
- bool **operator>** (const QTime & t) const
- bool **operator>=** (const QTime & t) const
- void **start** ()
- int **restart** ()
- int **elapsed** () const

Static Public Members

- QTime **currentTime** ()
- QTime **fromString** (const QString & s, Qt::DateFormat f = Qt::TextDate)
- bool **isValid** (int h, int m, int s, int ms = 0)

Related Functions

- QDataStream & **operator<<** (QDataStream & s, const QTime & t)
- QDataStream & **operator>>** (QDataStream & s, QTime & t)

Detailed Description

The QTime class provides clock time functions.

A QTime object contains a clock time, i.e. the number of hours, minutes, seconds, and milliseconds since midnight. It can read the current time from the system clock and measure a span of elapsed time. It provides functions for comparing times and for manipulating a time by adding a number of (milli)seconds.

QTime operates with 24-hour clock format; it has no concept of AM/PM. It operates in local time; it knows nothing about time zones or daylight savings time.

A QTime object is typically created either by giving the number of hours, minutes, seconds, and milliseconds explicitly, or by using the static function currentTime(), which makes a QTime object that contains the system's clock time. Note that the accuracy depends on the accuracy of the underlying operating system; not all systems provide 1-millisecond accuracy.

The hour(), minute(), second(), and msec() functions provide access to the number of hours, minutes, seconds, and milliseconds of the time. The same information is provided in textual format by the toString() function.

QTime provides a full set of operators to compare two QTime objects. One time is considered smaller than another if it is earlier than the other.

The time a given number of seconds or milliseconds later than a given time can be found using the addSecs() or addMSecs() functions. Correspondingly, the number of (milli)seconds between two times can be found using the secsTo() or msecsTo() functions.

QTime can be used to measure a span of elapsed time using the start(), restart(), and elapsed() functions.

See also QDate [p. 36], QDateTime [p. 44] and Time and Date.

Member Function Documentation

QTime::QTime ()

Constructs the time 0 hours, minutes, seconds and milliseconds, i.e. 00:00:00.000 (midnight). This is a valid time.

See also isValid() [p. 201].

QTime::QTime (int h, int m, int s = 0, int ms = 0)

Constructs a time with hour *h*, minute *m*, seconds *s* and milliseconds *ms*.

h must be in the range 0..23, *m* and *s* must be in the range 0..59, and *ms* must be in the range 0..999.

See also isValid() [p. 201].

QTime QTime::addMSecs (int ms) const

Returns a QTime object containing a time *ms* milliseconds later than the time of this object (or earlier if *ms* is negative).

Note that the time will wrap if it passes midnight. See addSecs() for an example.

See also addSecs() [p. 200] and msecsTo() [p. 201].

QTime QTime::addSecs (int nsecs) const

Returns a QTime object containing a time *nsecs* seconds later than the time of this object (or earlier if *nsecs* is negative).

Note that the time will wrap if it passes midnight.

Example:

```
    QTime n( 14, 0, 0 );                // n == 14:00:00
    QTime t;
    t = n.addSecs( 70 );                // t == 14:01:10
    t = n.addSecs( -70 );               // t == 13:58:50
    t = n.addSecs( 10*60*60 + 5 );      // t == 00:00:05
    t = n.addSecs( -15*60*60 );         // t == 23:00:00
```

See also addMSecs() [p. 199], secsTo() [p. 202] and QDateTime::addSecs() [p. 46].

QTime QTime::currentTime () [static]

Returns the current time as reported by the system clock.

Note that the accuracy depends on the accuracy of the underlying operating system; not all systems provide 1-millisecond accuracy.

Examples: aclock/aclock.cpp, dclock/dclock.cpp, t12/cannon.cpp and tictac/tictac.cpp.

int QTime::elapsed () const

Returns the number of milliseconds that have elapsed since the last time start() or restart() was called.

Note that the counter wraps to zero 24 hours after the last call to start() or restart.

Note that the accuracy depends on the accuracy of the underlying operating system; not all systems provide 1-millisecond accuracy.

Warning: If the system's clock setting has been changed since the last time start() or restart() was called, the result is undefined. This can happen when daylight savings time is turned on or off.

See also start() [p. 203] and restart() [p. 202].

QTime QTime::fromString (const QString & s, Qt::DateFormat f = Qt::TextDate) [static]

Returns the representation *s* as a QTime using the format *f*, or an invalid time if this is not possible.

Note that Qt::LocalDate cannot be used here.

int QTime::hour () const

Returns the hour part (0..23) of the time.

Example: tictac/tictac.cpp.

bool QTime::isNull () const

Returns TRUE if the time is equal to 00:00:00.000; otherwise returns FALSE. A null time is valid.

See also isValid() [p. 201].

bool QTime::isValid () const

Returns TRUE if the time is valid; otherwise returns FALSE. The time 23:30:55.746 is valid, whereas 24:12:30 is invalid.

See also isNull() [p. 201].

bool QTime::isValid (int h, int m, int s, int ms = 0) [static]

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns TRUE if the specified time is valid; otherwise returns FALSE.

The time is valid if *h* is in the range 0..23, *m* and *s* are in the range 0..59, and *ms* is in the range 0..999.

Example:

```
    QTime::isValid(21, 10, 30); // returns TRUE
    QTime::isValid(22, 5,  62); // returns FALSE
```

int QTime::minute () const

Returns the minute part (0..59) of the time.

Examples: aclock/aclock.cpp and tictac/tictac.cpp.

int QTime::msec () const

Returns the millisecond part (0..999) of the time.

int QTime::msecsTo (const QTime & t) const

Returns the number of milliseconds from this time to *t* (which is negative if *t* is earlier than this time).

Because QTime measures time within a day and there are 86400 seconds in a day, the result is between -86400 and 86400s.

See also secsTo() [p. 202].

bool QTime::operator!= (const QTime & t) const

Returns TRUE if this time is different from *t*; otherwise returns FALSE.

bool QTime::operator< (const QTime & t) const

Returns TRUE if this time is earlier than *t*; otherwise returns FALSE.

bool QTime::operator<= (const QTime & t) const

Returns TRUE if this time is earlier than or equal to *t*; otherwise returns FALSE.

bool QTime::operator== (const QTime & t) const

Returns TRUE if this time is equal to *t*; otherwise returns FALSE.

bool QTime::operator> (const QTime & t) const

Returns TRUE if this time is later than *t*; otherwise returns FALSE.

bool QTime::operator>= (const QTime & t) const

Returns TRUE if this time is later than or equal to *t*; otherwise returns FALSE.

int QTime::restart ()

Sets this time to the current time and returns the number of milliseconds that have elapsed since the last time start() or restart() was called.

This function is guaranteed to be atomic and is thus very handy for repeated measurements. Call start() to start the first measurement and then restart() for each later measurement.

Note that the counter wraps to zero 24 hours after the last call to start() or restart().

Warning: If the system's clock setting has been changed since the last time start() or restart() was called, the result is undefined. This can happen when daylight savings time is turned on or off.

See also start() [p. 203], elapsed() [p. 200] and currentTime() [p. 200].

int QTime::second () const

Returns the second part (0..59) of the time.

Example: tictac/tictac.cpp.

int QTime::secsTo (const QTime & t) const

Returns the number of seconds from this time to *t* (which is negative if *t* is earlier than this time).

Because QTime measures time within a day and there are 86400 seconds in a day, the result is between -86400 and 86400.

See also addSecs() [p. 200] and QDateTime::secsTo() [p. 47].

Example: t12/cannon.cpp.

bool QTime::setHMS (int h, int m, int s, int ms = 0)

Sets the time to hour *h*, minute *m*, seconds *s* and milliseconds *ms*.

h must be in the range 0..23, *m* and *s* must be in the range 0..59, and *ms* must be in the range 0..999. Returns TRUE if the set time is valid; otherwise returns FALSE.

See also isValid() [p. 201].

void QTime::start ()

Sets this time to the current time. This is practical for timing:

```
    QTime t;
    t.start();                    // start clock
    ... // some lengthy task
    qDebug( "%d\n", t.elapsed() ); // prints the number of msecs elapsed
```

See also restart() [p. 202], elapsed() [p. 200] and currentTime() [p. 200].

QString QTime::toString (const QString & format) const

Returns the time as a string. The *format* parameter determines the format of the result string.

These expressions may be used:

- *h* - the hour without a leading zero (0..23 or 1..12 if AM/PM display)
- *hh* - the hour with a leading zero (00..23 or 01..12 if AM/PM display)
- *m* - the minute without a leading zero (0..59)
- *mm* - the minute with a leading zero (00..59)
- *s* - the second whithout a leading zero (0..59)
- *ss* - the second whith a leading zero (00..59)
- *z* - the milliseconds without leading zeroes (0..999)
- *zzz* - the milliseconds with leading zeroes (000..999)
- *AP* - switch to AM/PM display. *AP* will be replaced by either "AM" or "PM".
- *ap* - switch to am/pm display. *ap* will be replaced by either "am" or "pm".

All other input characters will be ignored.

Example format Strings (assuming that the QTime is 14:13:09.042)

- "hh:mm:ss.zzz" will result in "14:13:09.042"
- "h:m:s ap" will result in "2:13:9 pm"

See also QDate::toString() [p. 42] and QTime::toString() [p. 203].

QString QTime::toString (Qt::DateFormat f = Qt::TextDate) const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns the time as a string. Milliseconds are not included. The *f* parameter determines the format of the string.

If *f* is Qt::TextDate, the string format is HH:MM:SS; e.g. 1 second before midnight would be "23:59:59".

If *f* is Qt::ISODate, the string format corresponds to the ISO 8601 specification for representations of dates, which is also HH:MM:SS.

If *f* is Qt::LocalDate, the string format depends on the locale settings of the system.

Related Functions

QDataStream & operator<< (QDataStream & s, const QTime & t)

Writes time *t* to the stream *s*.

See also Format of the QDataStream operators [Input/Output and Networking with Qt].

QDataStream & operator>> (QDataStream & s, QTime & t)

Reads a time from the stream *s* into *t*.

See also Format of the QDataStream operators [Input/Output and Networking with Qt].

QTimer Class Reference

The QTimer class provides timer signals and single-shot timers.

`#include <qtimer.h>`

Inherits QObject [p. 123].

Public Members

- **QTimer** (QObject * parent = 0, const char * name = 0)
- **~QTimer** ()
- bool **isActive** () const
- int **start** (int msec, bool sshot = FALSE)
- void **changeInterval** (int msec)
- void **stop** ()

Signals

- void **timeout** ()

Static Public Members

- void **singleShot** (int msec, QObject * receiver, const char * member)

Detailed Description

The QTimer class provides timer signals and single-shot timers.

It uses timer events internally to provide a more versatile timer. QTimer is very easy to use: create a QTimer, call start() to start it and connect its timeout() to the appropriate slots. When the time is up it will emit the timeout() signal.

Note that a QTimer object is destroyed automatically when its parent object is destroyed.

Example:

```
QTimer *timer = new QTimer( myObject );
connect( timer, SIGNAL(timeout()), myObject, SLOT(timerDone()) );
timer->start( 2000, TRUE ); // 2 seconds single-shot timer
```

You can also use the static singleShot() function to create a single shot timer.

As a special case, a QTimer with timeout 0 times out as soon as all the events in the window system's event queue have been processed.

This can be used to do heavy work while providing a snappy user interface:

```
    QTimer *t = new QTimer( myObject );
    connect( t, SIGNAL(timeout()), SLOT(processOneThing()) );
    t->start( 0, FALSE );
```

myObject->processOneThing() will be called repeatedly and should return quickly (typically after processing one data item) so that Qt can deliver events to widgets and stop the timer as soon as it has done all its work. This is the traditional way of implementing heavy work in GUI applications; multi-threading is now becoming available on more and more platforms, and we expect that null events will eventually be replaced by threading.

Note that QTimer's accuracy depends on the underlying operating system and hardware. Most platforms support an accuracy of 20ms; some provide more. If Qt is unable to deliver the requested number of timer clicks, it will silently discard some.

An alternative to using QTimer is to call QObject::startTimer() for your object and reimplement the QObject::timerEvent() event handler in your class (which must, of course, inherit QObject). The disadvantage is that timerEvent() does not support such high-level features as single-shot timers or signals.

Some operating systems limit the number of timers that may be used; Qt tries to work around these limitations.

See also Event Classes and Time and Date.

Member Function Documentation

QTimer::QTimer (QObject * parent = 0, const char * name = 0)

Constructs a timer with a *parent* and a *name*.

Note that the destructor of the parent object will destroy this timer object.

QTimer::~QTimer ()

Destroys the timer.

void QTimer::changeInterval (int msec)

Changes the timeout interval to *msec* milliseconds.

If the timer signal is pending, it will be stopped and restarted; otherwise it will be started.

See also start() [p. 207] and isActive() [p. 206].

bool QTimer::isActive () const

Returns TRUE if the timer is running (pending) or FALSE if the timer is idle.

Example: t11/cannon.cpp.

void QTimer::singleShot (int msec, QObject * receiver, const char * member) [static]

This static function calls a slot after a given time interval.

It is very convenient to use this function because you do not need to bother with a timerEvent or to create a local QTimer object.

Example:

```
#include <qapplication.h>
#include <qtimer.h>

int main( int argc, char **argv )
{
    QApplication a( argc, argv );
    QTimer::singleShot( 10*60*1000, &a, SLOT(quit()) );
        ... // create and show your widgets
    return a.exec();
}
```

This sample program automatically terminates after 10 minutes (i.e. 600000 milliseconds).

The *receiver* is the receiving object and the *member* is the slot. The time interval is *msec*.

int QTimer::start (int msec, bool sshot = FALSE)

Starts the timer with an *msec* milliseconds timeout.

If *sshot* is TRUE, the timer will be activated only once; otherwise it will continue until it is stopped.

Any pending timer will be stopped.

See also singleShot() [p. 207], stop() [p. 207], changeInterval() [p. 206] and isActive() [p. 206].

Examples: aclock/aclock.cpp, dirview/dirview.cpp, forever/forever.cpp, hello/hello.cpp, t11/cannon.cpp and t13/cannon.cpp.

void QTimer::stop ()

Stops the timer.

See also start() [p. 207].

Examples: dirview/dirview.cpp, t11/cannon.cpp, t12/cannon.cpp and t13/cannon.cpp.

void QTimer::timeout () [signal]

This signal is emitted when the timer is activated.

Examples: aclock/aclock.cpp, dirview/dirview.cpp, forever/forever.cpp, hello/hello.cpp and t11/cannon.cpp.

QValidator Class Reference

The QValidator class provides validation of input text.

`#include <qvalidator.h>`

Inherits QObject [p. 123].

Inherited by QIntValidator [p. 113], QDoubleValidator [p. 50] and QRegExpValidator [p. 153].

Public Members

- **QValidator** (QObject * parent, const char * name = 0)
- **~QValidator** ()
- enum **State** { Invalid, Intermediate, Valid = Intermediate, Acceptable }
- virtual State **validate** (QString & input, int & pos) const
- virtual void **fixup** (QString & input) const

Detailed Description

The QValidator class provides validation of input text.

The class itself is abstract. Two subclasses, QIntValidator and QDoubleValidator, provide basic numeric-range checking, and QRegExpValidator provides general checking using a custom regular expression.

If the built-in validators aren't sufficient, you can subclass QValidator. The class has two virtual functions: validate() and fixup().

validate() must be implemented by every subclass. It returns Invalid, Intermediate or Acceptable depending on whether its argument is valid (for the subclass's definition of valid).

These three states require some explanation. An Invalid string is *clearly* invalid. Intermediate is less obvious - the concept of validity is slippery when the string is incomplete (still being edited). QValidator defines Intermediate as the property of a string that is neither clearly invalid nor acceptable as a final result. Acceptable means that the string is acceptable as a final result. One might say that any string that is a plausible intermediate state during entry of an Acceptable string is Intermediate.

Here are some examples:

- For a line edit that accepts integers from 0 to 999 inclusive, 42 and 666 are Acceptable, the empty string and 1114 are Intermediate and asdf is Invalid.
- For an editable combobox that accepts URLs, any well-formed URL is Acceptable, "http://www.trolltech.com/," is Intermediate (it might be a cut-and-paste that accidentally took in a comma at the end), the empty string is valid (the user might select and delete all of the text in preparation of entering a new URL) and "http:///./" is Invalid.

- For a spin box that accepts lengths, "11cm" and "1in" are Acceptable, "11" and the empty string are Intermediate and "http://www.trolltech.com" and "hour" are Invalid.

fixup() is provided for validators that can repair some user errors. The default implementation does nothing. QLineEdit, for example, will call fixup() if the user presses Enter and the content is not currently valid, in case fixup() can do magic. This allows some Invalid strings to be made Acceptable, too.

QValidator is typically used with QLineEdit, QSpinBox and QComboBox.

See also Miscellaneous Classes.

Member Type Documentation

QValidator::State

This enum type defines the states in which a validated string can exist. There are currently three states:

- `QValidator::Invalid` - the string is *clearly* invalid.
- `QValidator::Intermediate` - the string is a plausible intermediate value during editing.
- `QValidator::Acceptable` - the string is acceptable as a final result, i.e. it is valid.

Member Function Documentation

QValidator::QValidator (QObject * parent, const char * name = 0)

Sets up the internal data structures used by the validator. At the moment there aren't any. The *parent* and *name* parameters are passed to the QObject constructor.

QValidator::~QValidator ()

Destroys the validator, freeing any storage and other resources used.

void QValidator::fixup (QString & input) const [virtual]

This function attempts to change *input* to be valid according to this validator's rules. It need not result in a valid string - callers of this function must re-test afterwards; the default does nothing.

Reimplementations of this function can change *input* even if they do not produce a valid string. For example, an ISBN validator might want to delete every character except digits and "-", even if the result is not a valid ISBN; a surname validator might want to remove whitespace from the start and end of the string, even if the resulting string is not in the list of accepted surnames.

State QValidator::validate (QString & input, int & pos) const [virtual]

This pure virtual function returns Invalid if *input* is invalid according to this validator's rules, Intermediate if it is likely that a little more editing will make the input acceptable (e.g. the user types '4' into a widget which accepts integers between 10 and 99) and Acceptable if the input is valid.

The function can change *input* and *pos* (the cursor position) if it wants to.

Reimplemented in QIntValidator, QDoubleValidator and QRegExpValidator.

Index