Dialogs and Windows with Qt

Qt 3.0

The definitive Qt documentation is provided in HTML format supplied with Qt, and available online at http://doc.trolltech.com. This PDF file was generated automatically from the HTML source as a convenience to users, although PDF is not an official Qt documentation format.

Contents

QColorDialog Class Reference

The QColorDialog class provides a dialog widget for specifying colors.

`#include <qcolordialog.h>`

Inherits QDialog [p. 10].

Static Public Members

- QColor **getColor** (const QColor & initial = white, QWidget * parent = 0, const char * name = 0)
- QRgb **getRgba** (QRgb initial, bool * ok = 0, QWidget * parent = 0, const char * name = 0)
- int **customCount** ()
- QRgb **customColor** (int i)
- void **setCustomColor** (int i, QRgb c)

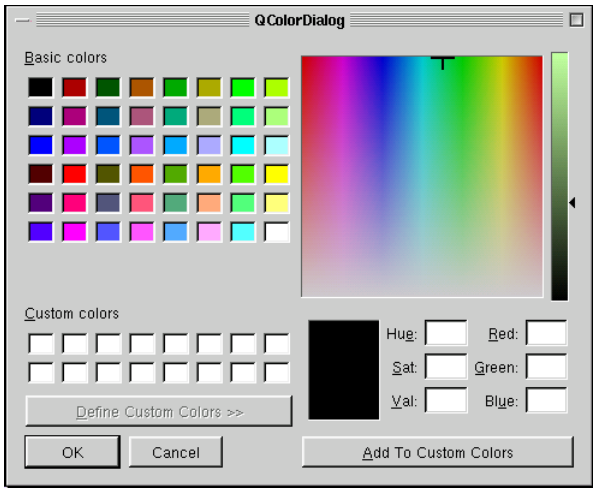
Detailed Description

The QColorDialog class provides a dialog widget for specifying colors.

The color dialog's function is to allow users to choose colors - for instance, you might use this in a drawing program to allow the user to set the brush color.

This version of Qt provides only modal color dialogs. The static getColor() function shows the dialog and allows the user to specify a color, whereas getRgba() does the same but allows the user to specify a color with an alpha channel (transparency) value.

The user can store customCount() different custom colors. The custom colors are shared by all color dialogs, and remembered during the execution of the program. Use setCustomColor() to set the custom colors, and use customColor() to get them.



See also Dialog Classes and Graphics Classes.

Member Function Documentation

QRgb QColorDialog::customColor (int i) [static]

Returns custom color number *i* as a QRgb.

int QColorDialog::customCount () [static]

Returns the number of custom colors supported by QColorDialog. All color dialogs share the same custom colors.

QColor QColorDialog::getColor (const QColor & initial = white, QWidget * parent = 0, const char * name = 0) [static]

Pops up a modal color dialog, lets the user choose a color, and returns that color. The color is initially set to *initial*. The dialog is a child of *parent* and is called *name*. Returns an invalid (see QColor::isValid()) color if the user cancels the dialog. All colors allocated by the dialog will be deallocated before this function returns.

Example: scribble/scribble.cpp.

QRgb QColorDialog::getRgba (QRgb initial, bool * ok = 0, QWidget * parent = 0, const char * name = 0) [static]

Pops up a modal color dialog to allow the user to choose a color and an alpha channel value. The color+alpha is initially set to *initial*. The dialog is a child of *parent* and called *name*.

If *ok* is non-null, `*ok` is set to TRUE if the user clicked OK, and FALSE if the user clicked Cancel.

If the user clicks Cancel, the *initial* value is returned.

void QColorDialog::setCustomColor (int i, QRgb c) [static]

Sets custom color number *i* to the QRgb value *c*.

QCustomMenuItem Class Reference

The QCustomMenuItem class is an abstract base class for custom menu items in popup menus.

`#include <qmenudata.h>`

Inherits Qt [Additional Functionality with Qt].

Public Members

- **QCustomMenuItem** ()
- virtual **~QCustomMenuItem** ()
- virtual bool **fullSpan** () const
- virtual bool **isSeparator** () const
- virtual void **setFont** (const QFont & font)
- virtual void **paint** (QPainter * p, const QColorGroup & cg, bool act, bool enabled, int x, int y, int w, int h)
- virtual QSize **sizeHint** ()

Detailed Description

The QCustomMenuItem class is an abstract base class for custom menu items in popup menus.

A custom menu item is a menu item that is defined by two purely virtual functions, paint() and sizeHint(). The size hint tells the menu how much space it needs to reserve for this item, and paint is called whenever the item needs painting.

This simple mechanism allows you to create all kinds of application specific menu items. Examples are items showing different fonts in a word processor or menus that allow the selection of drawing utilities in a vector drawing program.

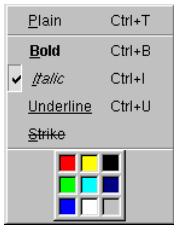
A custom item is inserted into a popup menu with QPopupMenu::insertItem().

By default, a custom item can also have an icon set and a keyboard accelerator. You can reimplement fullSpan() to return TRUE if you want the item to span the entire popup menu width. This is particularly useful for labels.

If you want the custom item to be treated just as a separator, reimplement isSeparator() to return TRUE.

Note that you can insert pixmaps or bitmaps as items into a popup menu without needing to create a QCustomMenuItem. However, custom menu items offer more flexibility, and — especially important with windows style — provide the possibility of drawing the item with a different color when it is highlighted.

menu/menu.cpp shows a simply example how custom menu items can be used.



See also QMenuData [p. 80], QPopupMenu [p. 106] and Miscellaneous Classes.

Member Function Documentation

QCustomMenuItem::QCustomMenuItem ()

Constructs a QCustomMenuItem

QCustomMenuItem::~QCustomMenuItem () [virtual]

Destroys a QCustomMenuItem

bool QCustomMenuItem::fullSpan () const [virtual]

Returns TRUE if this item wants to span the entire popup menu width. The default is FALSE, meaning that the menu may show an icon and an accelerator key for this item as well.

bool QCustomMenuItem::isSeparator () const [virtual]

Returns TRUE if this item is just a separator; otherwise returns FALSE.

void QCustomMenuItem::paint (QPainter * p, const QColorGroup & cg, bool act, bool enabled, int x, int y, int w, int h) [virtual]

Paints this item. When this function is invoked, the painter *p* is set to the right font and the right foreground color suitable for a menu item text using color group *cg*. The item is active if *act* is TRUE and enabled if *enabled* is TRUE. The geometry values *x*, *y*, *w* and *h* specify where to draw the item.

Do not draw any background, this has already been done by the popup menu according to the current GUI style.

void QCustomMenuItem::setFont (const QFont & font) [virtual]

Sets the font of the custom menu item to *font*.

This function is called whenever the font in the popup menu changes. For menu items that show their own individual font entry, you want to ignore this.

QSize QCustomMenuItem::sizeHint () [virtual]

Returns the size hint of this item.

QDesktopWidget Class Reference

The QDesktopWidget class provides access to screen information on multi-head systems.

`#include <qdesktopwidget.h>`

Inherits QWidget [Widgets with Qt].

Public Members

- **QDesktopWidget** ()
- **~QDesktopWidget** ()
- bool **isVirtualDesktop** () const
- int **numScreens** () const
- int **primaryScreen** () const
- int **screenNumber** (QWidget * widget = 0) const
- int **screenNumber** (const QPoint & point) const
- QWidget * **screen** (int screen = -1)
- const QRect & **screenGeometry** (int screen = -1) const

Detailed Description

The QDesktopWidget class provides access to screen information on multi-head systems.

Systems with more than one graphics card and monitor can manage the physical screen space available either as multiple desktops, or as a large virtual desktop, which usually has the size of the bounding rectangle of all the screens (see isVirtualDesktop()). For an application, one of the available screens is the primary screen, i.e. the screen where the main widget resides (see primaryScreen()). All windows opened in the context of the application have to be constrained to the boundaries of the primary screen; for example, it would be inconvenient if a dialog box popped up on a different screen, or split over two screens.

The QDesktopWidget provides information about the geometry of the available screens with screenGeometry(). The number of screens available is returned by numScreens(). The screen number that a particular point or widget is located in is returned by screenNumber().

Widgets provided by Qt use this class, for example, to place tooltips, menus and dialog boxes according to the parent or application widget.

Applications can use this class to save window positions, or to place child widgets on one screen.

See also Advanced Widgets and Environment Classes.

Member Function Documentation

QDesktopWidget::QDesktopWidget ()

Creates the desktop widget.

If the system supports a virtual desktop, this widget will have the size of the virtual desktop; otherwise this widget will have the size of the primary screen.

QDesktopWidget::~QDesktopWidget ()

Destroy the object and free allocated resources.

bool QDesktopWidget::isVirtualDesktop () const

Returns TRUE if the system manages the available screens in a virtual desktop; otherwise returns FALSE.

For virtual desktops, screen() will always return the same widget. The size of the virtual desktop is the size of this desktop widget.

int QDesktopWidget::numScreens () const

Returns the number of available screens.

See also primaryScreen() [p. 8].

int QDesktopWidget::primaryScreen () const

Returns the index of the primary screen.

See also numScreens() [p. 8].

QWidget * QDesktopWidget::screen (int screen = -1)

Returns a widget that represents the screen with index *screen*. This widget can be used to draw directly on the desktop, using an unclipped painter like this:

```
QPainter paint( QApplication::desktop()->screen( 0 ), TRUE );
paint.draw...
...
paint.end();
```

If the system uses a virtual desktop, the returned widget will have the geometry of the entire virtual desktop i.e. bounding every *screen*.

See also primaryScreen() [p. 8], numScreens() [p. 8] and isVirtualDesktop() [p. 8].

const QRect & QDesktopWidget::screenGeometry (int screen = -1) const

Returns the geometry of the screen with index *screen*.

See also screenNumber() [p. 9].

int QDesktopWidget::screenNumber (QWidget * widget = 0) const

Returns the index of the screen that contains the largest part of *widget*, or -1 if the widget not on a screen.

See also primaryScreen() [p. 8].

int QDesktopWidget::screenNumber (const QPoint & point) const

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

Returns the index of the screen that contains *point*, or -1 if no screen contains the point.

See also primaryScreen() [p. 8].

QDialog Class Reference

The QDialog class is the base class of dialog windows.

`#include <qdialog.h>`

Inherits QWidget [Widgets with Qt].

Inherited by QColorDialog [p. 3], QErrorMessage [Additional Functionality with Qt], QFileDialog [p. 31], QFontDialog [Additional Functionality with Qt], QInputDialog [p. 49], QMessageBox [p. 93], QProgressDialog [p. 121], QTabDialog [p. 129] and QWizard [p. 159].

Public Members

- **QDialog** (QWidget * parent = 0, const char * name = 0, bool modal = FALSE, WFlags f = 0)
- **~QDialog** ()
- enum **DialogCode** { Rejected, Accepted }
- int **result** () const
- virtual void **show** ()
- void **setOrientation** (Orientation orientation)
- Orientation **orientation** () const
- void **setExtension** (QWidget * extension)
- QWidget * **extension** () const
- void **setSizeGripEnabled** (bool)
- bool **isSizeGripEnabled** () const

Public Slots

- int **exec** ()

Properties

- bool **sizeGripEnabled** — whether the size grip is enabled

Protected Members

- void **setResult** (int i)

Protected Slots

- virtual void **done** (int r)
- virtual void **accept** ()
- virtual void **reject** ()
- void **showExtension** (bool showIt)

Detailed Description

The QDialog class is the base class of dialog windows.

A dialog window is a top-level window mostly used for short-term tasks and brief communications with the user. QDialogs may be modal or modeless. QDialogs can have default buttons, support extensibility and may provide a return value. QDialogs can have a QSizeGrip in their lower-right corner, using setSizeGripEnabled().

Note that QDialog uses the parent widget slightly differently from other classes in Qt. A dialog is always a top-level widget, but if it has a parent, its default location is centered on top of the parent. It will also share the parent's taskbar entry, for example.

There are three kinds of dialog that are useful:

1. A **modal** dialog is a dialog that blocks input to other visible windows in the same application: users must finish interacting with the dialog and close it before they can access any other window in the application. Modal dialogs have their own local event loop. Dialogs which are used to request a filename from the user or which are used to set application preferences are usually modal. Call exec() to display a modal dialog. When the user closes the dialog, exec() will provide a useful return value, and the flow of control will follow on from the exec() call at this time. Typically we connect a default button, e.g. "OK", to the accept() slot and a "Cancel" button to the reject() slot, to get the dialog to close and return the appropriate value. Alternatively you can connect to the done() slot, passing it Accepted or Rejected.
2. A **modeless** dialog is a dialog that operates independently of other windows in the same application. Find and replace dialogs in word-processors are often modeless to allow the user to interact with both the application's main window and the dialog. Call show() to display a modeless dialog. show() returns immediately so the flow of control will continue in the calling code. In practice you will often call show() and come to the end of the function in which show() is called with control returning to the main event loop.
3. A "**semi-modal**" dialog is a modal dialog that returns control to the caller immediately. Semi-modal dialogs do not have their own event loop, so you will need to call QApplication::processEvents() periodically to give the semi-modal dialog the opportunity to process its events. A progress dialog (e.g. QProgressDialog) is an example, where you only want the user to be able to interact with the progress dialog, e.g. to cancel a long running operation, but need to actually carry out the operation. Semi-modal dialogs are displayed by setting the modal flag to TRUE and calling the show() function.

Default button A dialog's "default" button is the button that's pressed when the user presses Enter or Return. This button is used to signify that the user accepts the dialog's settings and wishes to close the dialog. Use QPushButton::setDefault(), QPushButton::isDefault() and QPushButton::autoDefault() to set and control the dialog's default button.

Extensibility Extensibility is the ability to show the dialog in two ways: a partial dialog that shows the most commonly used options, and a full dialog that shows all the options. Typically an extensible dialog will initially appear as a partial dialog, but with a "More" button. If the user clicks the "More" button, the full dialog will appear. Extensibility is controlled with setExtension(), setOrientation() and showExtension().

Return value (modal dialogs) Modal dialogs are often used in situations where a return value is required; for example to indicate whether the user pressed OK or Cancel. A dialog can be closed by calling the accept() or the reject() slots, and exec() will return Accepted or Rejected as appropriate. After the exec() call has returned the result is available from result().

Examples

A modal dialog.

```
        QFileDialog *dlg = new QFileDialog( workingDirectory,
                QString::null, 0, 0, TRUE );
        dlg->setCaption( QFileDialog::tr( "Open" ) );
        dlg->setMode( QFileDialog::ExistingFile );
        QString result;
        if ( dlg->exec() == QDialog::Accepted ) {
            result = dlg->selectedFile();
            workingDirectory = dlg->url();
        }
        delete dlg;
        return result;
```

A modeless dialog. After the show() call, control returns to the main event loop.

```
    int main( int argc, char **argv )
    {
        QApplication a( argc, argv );

        int scale = 10;

        LifeDialog *life = new LifeDialog( scale );
        a.setMainWidget( life );
        life->setCaption("Qt Example - Life");
        life->show();

        return a.exec();
    }
```

See the QProgressDialog [p. 121] documentation for an example of a semi-modal dialog.

See also QTabDialog [p. 129], QWidget [Widgets with Qt], QProgressDialog [p. 121], GUI Design Handbook: Dialogs, Standard, Abstract Widget Classes and Dialog Classes.

Member Type Documentation

QDialog::DialogCode

The value returned by a modal dialog.

- `QDialog::Accepted`
- `QDialog::Rejected`

Member Function Documentation

QDialog::QDialog (QWidget * parent = 0, const char * name = 0, bool modal = FALSE, WFlags f = 0)

Constructs a dialog called *name*, with parent *parent*.

If *modal* is FALSE (the default), the dialog is modeless and should be displayed with show(). If *modal* is TRUE and the dialog is displayed with exec(), the dialog is modal, i.e. blocks input to other windows. If *modal* is TRUE and the dialog is displayed show(), the dialog is semi-modal.

The widget flags *f* are passed on to the QWidget constructor.

If, for example, you don't want a What's this button in the titlebar of the dialog, pass WStyle_Customize | WStyle_NormalBorder | WStyle_Title | WStyle_SysMenu in *f*.

We recommend that you always pass a non-null parent.

See also QWidget::setWFlags() [Widgets with Qt] and Qt::WidgetFlags [Additional Functionality with Qt].

QDialog::~QDialog ()

Destroys the QDialog, deleting all its children.

void QDialog::accept () [virtual protected slot]

Hides the modal dialog and sets the result code to Accepted.

See also reject() [p. 14] and done() [p. 13].

void QDialog::done (int r) [virtual protected slot]

Hides the modal dialog and sets its result code to *r*. This uses the local event loop to finish, and exec() to return *r*.

If the dialog has the WDestructiveClose flag set, done() also deletes the dialog. If the dialog is the applications's main widget, the application terminates.

See also accept() [p. 13], reject() [p. 14], QApplication::mainWidget() [Additional Functionality with Qt] and QApplication::quit() [Additional Functionality with Qt].

Example: movies/main.cpp.

int QDialog::exec () [slot]

Executes a modal dialog. Control passes to the dialog until the user closes it, at which point the local event loop finishes and the function returns with the DialogCode result. Users will not be able to interact with any other window in the same application until they close this dialog. For a modeless or semi-modal dialog use show().

See also show() [p. 14] and result() [p. 14].

Examples: i18n/main.cpp, network/networkprotocol/view.cpp, qdir/qdir.cpp, showimg/showimg.cpp and wizard/main.cpp.

QWidget * QDialog::extension () const

Returns the dialog's extension or 0 if no extension has been defined.

See also setExtension() [p. 14].

bool QDialog::isSizeGripEnabled () const

Returns TRUE if the size grip is enabled; otherwise returns FALSE. See the "sizeGripEnabled" [p. 15] property for details.

Orientation QDialog::orientation () const

Returns the dialog's extension orientation.

See also setOrientation() [p. 14].

void QDialog::reject () [virtual protected slot]

Hides the modal dialog and sets the result code to Rejected.

See also accept() [p. 13] and done() [p. 13].

int QDialog::result () const

Returns the modal dialog's result code, Accepted or Rejected.

void QDialog::setExtension (QWidget * extension)

Sets the widget, *extension*, to be the dialog's extension, deleting any previous extension. The dialog takes ownership of the extension. Note that if 0 is passed any existing extension will be deleted.

This function must only be called while the dialog is hidden.

See also showExtension() [p. 15], setOrientation() [p. 14] and extension() [p. 13].

void QDialog::setOrientation (Orientation orientation)

If *orientation* is Horizontal, the extension will be displayed to the right of the dialog's main area. If *orientation* is Vertical, the extension will be displayed below the dialog's main area.

See also orientation() [p. 14] and setExtension() [p. 14].

void QDialog::setResult (int i) [protected]

Sets the modal dialog's result code to *i*.

void QDialog::setSizeGripEnabled (bool)

Sets whether the size grip is enabled. See the "sizeGripEnabled" [p. 15] property for details.

void QDialog::show () [virtual]

Shows a modeless or semi-modal dialog. Control returns immediately to the calling code.

The dialog does not have a local event loop so you must call QApplication::processEvents() periodically to give the dialog the opportunity to process its events.

The dialog will be semi-modal if the modal flag was set to TRUE in the constructor.

Warning:

In Qt 2.x, calling show() on a modal dialog enters a local event loop, and works like exec(), but doesn't return the result code exec() returns. Trolltech has always warned that doing this is unwise.

See also exec() [p. 13].

Examples: movies/main.cpp, showimg/showimg.cpp, sql/overview/form1/main.cpp and tabdialog/main.cpp.

Reimplemented from QWidget [Widgets with Qt].

void QDialog::showExtension (bool showIt) [protected slot]

If *showIt* is TRUE, the dialog's extension is shown; otherwise the extension is hidden.

This slot is usually connected to the QButton::toggled() signal of a QPushButton.

If the dialog is not visible, or has no extension, nothing happens.

See also show() [p. 14], setExtension() [p. 14] and setOrientation() [p. 14].

Property Documentation

bool sizeGripEnabled

This property holds whether the size grip is enabled.

A QSizeGrip is placed in the bottom right corner of the dialog when this property is enabled. By default, the size grip is disabled.

Set this property's value with setSizeGripEnabled() and get this property's value with isSizeGripEnabled().

QDockArea Class Reference

The QDockArea class manages and lays out QDockWindows.

`#include <qdockarea.h>`

Inherits QWidget [Widgets with Qt].

Public Members

- enum **HandlePosition** { Normal, Reverse }
- **QDockArea** (Orientation o, HandlePosition h = Normal, QWidget * parent = 0, const char * name = 0)
- **~QDockArea** ()
- void **moveDockWindow** (QDockWindow * w, const QPoint & p, const QRect & r, bool swap)
- void **removeDockWindow** (QDockWindow * w, bool makeFloating, bool swap, bool fixNewLines = TRUE)
- void **moveDockWindow** (QDockWindow * w, int index = -1)
- bool **hasDockWindow** (QDockWindow * w, int * index = 0)
- Orientation **orientation** () const
- HandlePosition **handlePosition** () const
- bool **isEmpty** () const
- int **count** () const
- QPtrList<QDockWindow> **dockWindowList** () const
- bool **isDockWindowAccepted** (QDockWindow * dw)
- void **setAcceptDockWindow** (QDockWindow * dw, bool accept)

Public Slots

- void **lineUp** (bool keepNewLines)

Properties

- int **count** — the number of dock windows in the dock area *(read only)*
- bool **empty** — whether the dock area is empty *(read only)*
- HandlePosition **handlePosition** — where the dock window splitter handle is placed in the dock area *(read only)*
- Orientation **orientation** — the dock area's orientation *(read only)*

Related Functions

- QTextStream & **operator<<** (QTextStream & ts, const QDockArea & dockArea)
- QTextStream & **operator>>** (QTextStream & ts, QDockArea & dockArea)

Detailed Description

The QDockArea class manages and lays out QDockWindows.

A QDockArea is a container which manages a list of QDockWindows which it lays out within its area. In cooperation with the QDockWindows it is responsible for the docking and undocking of QDockWindows and moving them inside the dock area. QDockAreas also handle the wrapping of QDockWindows to fill the available space as compactly as possible. QDockAreas can contain QToolBars since QToolBar is a QDockWindow subclass.

QMainWindow contains four QDockAreas which you can use for your QToolBars and QDockWindows, so in most situations you do not need to use the QDockArea class directly. Although QMainWindow contains support for its own dock areas but isn't convenient for adding new QDockAreas. If you need to create your own dock areas we suggest that you create a subclass of QWidget and add your QDockAreas to your subclass.

Lines. QDockArea uses the concept of lines. A line is a horizontal region which may contain dock windows side-by-side. A dock area may have room for more than one line. When dock windows are docked into a dock area they are usually added at the right hand side of the top-most line that has room (unless manually placed by the user). When users move dock windows they may leave empty lines or gaps in non-empty lines. Dock windows can be lined up to minimize wasted space using the lineUp() function.

The QDockArea class maintains a position list of all its child dock windows. Dock windows are added to a dock area from position 0 onwards. Dock windows are laid out sequentially in position order from left to right, and in the case of multiple lines of dock windows, from top to bottom. If a dock window is floated it still retains its position since this is where the window will return if the user double clicks its caption. A dock window's position can be determined with hasDockWindow(). The position can be changed with moveDockWindow().

To dock or undock a dock window use QDockWindow::dock() and QDockWindow::undock() respectively. If you want to control which dock windows can dock in a dock area use setAcceptDockWindow(). To see if a dock area contains a particular dock window use hasDockWindow(); to see how many dock windows a dock area contains use count().

The streaming operators can write the positions of the dock windows in the dock area to a QTextStream. The positions can be read back later to restore the saved positions.

Save the positions to a QTextStream:

```
ts <> *myDockArea;
```

See also Main Window and Related Classes.

Member Type Documentation

QDockArea::HandlePosition

A dock window has two kinds of handles, the dock window handle used for dragging the dock window, and the splitter handle used to resize the dock window in relation to other dock windows using a splitter. (The splitter handle is only visible for docked windows.)

This enum specifies where the dock window splitter handle is placed in the dock area.

- `QDockArea::Normal` - The splitter handles of dock windows are placed at the right or bottom.
- `QDockArea::Reverse` - The splitter handles of dock windows are placed at the left or top.

Member Function Documentation

QDockArea::QDockArea (Orientation o, HandlePosition h = Normal, QWidget * parent = 0, const char * name = 0)

Constructs a QDockArea with orientation *o*, HandlePosition *h*, parent *parent* and name *name*.

QDockArea::~QDockArea ()

Destroys the dock area and all the dock windows docked in the dock area.

Does not affect any floating dock windows or dock windows in other dock areas, even if they first appeared in this dock area. Floating dock windows are effectively top level windows and are not child windows of the dock area. When a floating dock window is docked (dragged into a dock area) its parent becomes the dock area.

int QDockArea::count () const

Returns the number of dock windows in the dock area. See the "count" [p. 19] property for details.

QPtrList<QDockWindow> QDockArea::dockWindowList () const

Returns a list of the dock windows in the dock area.

HandlePosition QDockArea::handlePosition () const

Returns where the dock window splitter handle is placed in the dock area. See the "handlePosition" [p. 20] property for details.

bool QDockArea::hasDockWindow (QDockWindow * w, int * index = 0)

Returns TRUE if the dock area contains the dock window *w*, otherwise returns FALSE. If a non-null pointer is passed as *index* it will be set as follows: if the dock area contains the dock window *index* is set to *w*'s position; otherwise *index* is set to -1.

bool QDockArea::isDockWindowAccepted (QDockWindow * dw)

Returns TRUE if dock window *dw* could be docked into the dock area; otherwise returns FALSE.

See also setAcceptDockWindow() [p. 19].

bool QDockArea::isEmpty () const

Returns TRUE if the dock area is empty; otherwise returns FALSE. See the "empty" [p. 19] property for details.

void QDockArea::lineUp (bool keepNewLines) [slot]

Lines up the dock windows in this dock area to minimize wasted space. If *keepNewLines* is TRUE, only space within lines is cleaned up. If *keepNewLines* is FALSE the number of lines might be changed.

void QDockArea::moveDockWindow (QDockWindow * w, int index = -1)

Moves the QDockWindow *w* within the dock area. If *w* is not already docked in this area, *w* is docked first. If *index* is -1 or larger than the number of docked widgets, *w* is appended at the end, otherwise it is inserted at the position *index*.

void QDockArea::moveDockWindow (QDockWindow * w, const QPoint & p, const QRect & r, bool swap)

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

Moves the dock window *w* inside the dock area where *p* is the new position (in global screen coordinates), *r* is the suggested rectangle of the dock window and *swap* specifies whether or not the orientation of the docked widget needs to be changed.

This function is used internally by QDockWindow. You shouldn't need to call it yourself.

Orientation QDockArea::orientation () const

Returns the dock area's orientation. See the "orientation" [p. 20] property for details.

void QDockArea::removeDockWindow (QDockWindow * w, bool makeFloating, bool swap, bool fixNewLines = TRUE)

Removes the dock window *w* from the dock area. If *makeFloating* is TRUE, *w* gets floated, and if *swap* is TRUE, the orientation of *w* gets swapped. If *fixNewLines* is TRUE (the default) newlines in the area will be fixed.

You should never need to call this function yourself. Use QDockWindow::dock() and QDockWindow::undock() instead.

void QDockArea::setAcceptDockWindow (QDockWindow * dw, bool accept)

If *accept* is TRUE dock window *dw* can be docked in the dock area. If *accept* is FALSE dock window *dw* cannot be docked in the dock area.

See also isDockWindowAccepted() [p. 18].

Property Documentation

int count

This property holds the number of dock windows in the dock area.

Get this property's value with count().

bool empty

This property holds whether the dock area is empty.

Get this property's value with isEmpty().

HandlePosition handlePosition

This property holds where the dock window splitter handle is placed in the dock area.

The default position is Normal.

Get this property's value with handlePosition().

Orientation orientation

This property holds the dock area's orientation.

There is no default value; the orientation is specified in the constructor.

Get this property's value with orientation().

Related Functions

QTextStream & operator<< (QTextStream & ts, const QDockArea & dockArea)

Writes the layout of the dock windows in dock area *dockArea* to the text stream *ts*.

See also operator>>() [p. 20].

QTextStream & operator>> (QTextStream & ts, QDockArea & dockArea)

Reads the layout description of the dock windows in dock area *dockArea* from the text stream *ts* and restores it. The layout description must have been previously written by the operator<<() function.

See also operator<<() [p. 20].

QDockWindow Class Reference

The QDockWindow class provides a widget which can be docked inside a QDockArea or floated as a top level window on the desktop.

`#include <qdockwindow.h>`

Inherits QFrame [Widgets with Qt].

Inherited by QToolBar [p. 139].

Public Members

- enum **Place** { InDock, OutsideDock }
- enum **CloseMode** { Never = 0, Docked = 1, Undocked = 2, Always = Docked | Undocked }
- **QDockWindow** (Place p = InDock, QWidget * parent = 0, const char * name = 0, WFlags f = 0)
- virtual void **setWidget** (QWidget * w)
- QWidget * **widget** () const
- Place **place** () const
- QDockArea * **area** () const
- virtual void **setCloseMode** (int m)
- bool **isCloseEnabled** () const
- int **closeMode** () const
- virtual void **setResizeEnabled** (bool b)
- virtual void **setMovingEnabled** (bool b)
- bool **isResizeEnabled** () const
- bool **isMovingEnabled** () const
- virtual void **setHorizontallyStretchable** (bool b)
- virtual void **setVerticallyStretchable** (bool b)
- bool **isHorizontallyStretchable** () const
- bool **isVerticallyStretchable** () const
- void setHorizontalStretchable (bool b) *(obsolete)*
- void setVerticalStretchable (bool b) *(obsolete)*
- bool isHorizontalStretchable () const *(obsolete)*
- bool isVerticalStretchable () const *(obsolete)*
- bool **isStretchable** () const
- virtual void **setOffset** (int o)
- int **offset** () const
- virtual void **setFixedExtentWidth** (int w)
- virtual void **setFixedExtentHeight** (int h)
- QSize **fixedExtent** () const
- virtual void **setNewLine** (bool b)
- bool **newLine** () const

- Qt::Orientation **orientation** () const
- QBoxLayout * **boxLayout** ()
- virtual void **setOpaqueMoving** (bool b)
- bool **opaqueMoving** () const

Public Slots

- virtual void **undock** (QWidget * w)
- virtual void **dock** ()
- virtual void **setOrientation** (Orientation o)

Signals

- void **orientationChanged** (Orientation o)
- void **placeChanged** (QDockWindow::Place p)
- void **visibilityChanged** (bool visible)

Properties

- int **closeMode** — the close mode of a dock window
- bool **horizontallyStretchable** — whether the dock window is horizontally stretchable
- bool **movingEnabled** — whether the user can move the dock window within the dock area, move the dock window to another dock area, or float the dock window
- bool **newLine** — whether the dock window prefers to start a new line in the dock area
- int **offset** — the dock window's preferred offset from the dock area's left edge (top edge for vertical dock areas)
- bool **opaqueMoving** — whether the dock window will be shown normally whilst it is being moved
- Place **place** — whether the dock window is in a dock area *(read only)*
- bool **resizeEnabled** — whether the dock window is resizeable
- bool **stretchable** — whether the dock window is stretchable in the current orientation() *(read only)*
- bool **verticallyStretchable** — whether the dock window is vertically stretchable

Detailed Description

The QDockWindow class provides a widget which can be docked inside a QDockArea or floated as a top level window on the desktop.

This class handles moving, resizing, docking and undocking dock windows. QToolBar is a subclass of QDockWindow so the functionality provided for dock windows is available with the same API for toolbars.

If the user drags the dock window into the dock area the dock window will be docked. If the user drags the dock area outside any dock areas the dock window will be undocked (floated) and will become a top level window. Double clicking a floating dock window's titlebar will dock the dock window to the last dock area it was docked in. Double clicking a docked dock window's handle will undock (float) the dock window. Single clicking a docked dock window's handle will minimize the dock window (only its handle will appear, below the menu bar). Single clicking the minimized handle will restore the dock window to the last dock area that it was docked in. If the user clicks the close button (which appears on floating dock windows by default) the dock window will disappear. You can control whether or not a dock window has a close button with setCloseMode().

QMainWindow provides four dock areas (top, left, right and bottom) which can be used by dock windows. For many applications using the dock areas provided by QMainWindow will be sufficient. (See the QDockArea documentation if you want to create your own dock areas.) In QMainWindow a right-click popup menu (the dock window menu) is available which lists dock windows and can be used to show or hide them.

When you construct a dock window you *must* pass it a QDockArea or a QMainWindow as its parent if you want it docked. Pass 0 for the parent if you want it floated.

```
QToolBar *fileTools = new QToolBar( this, "File Actions" );
moveDockWindow( fileTools, Left );
```

In the example above we create a new QToolBar in the constructor of a QMainWindow subclass (so that the *this* pointer points to the QMainWindow). By default the toolbar will be added to the Top dock area, but we've moved it to the Left dock area.

A dock window is often used to contain a single widget. In these cases the widget can be set by calling setWidget(). If you're constructing a dock window that contains multiple widgets, e.g. a toolbar, arrange the widgets within a box layout inside the dock window. To do this use the boxLayout() function to get a pointer to the dock window's box layout, then add widgets to the layout using the box layout's QBoxLayout::addWidget() function. The dock window will dynamically set the orientation of the layout to be vertical or horizontal as necessary, although you can control this yourself with setOrientation().

Although a common use of dock windows is for toolbars, they can be used with any widgets. (See the *Qt Designer* and *Qt Linguist* applications, for example.) When using larger widgets it may make sense for the dock window to be resizable by calling setResizeEnabled(). Resizable dock windows are given splitter-like handles to allow the user to resize them within their dock area. When resizable dock windows are undocked they become top level windows and can be resized like any other top level windows, e.g. by dragging a corner or edge.

Dock windows can be docked and undocked using dock() and undock(). A dock window's orientation can be set with setOrientation(). You can also use QDockArea::moveDockWindow(). If you're using a QMainWindow, QMainWindow::moveDockWindow() and QMainWindow::removeDockWindow() are available.

A dock window can have some preferred settings, for example, you can set a preferred offset from the left edge (or top edge for vertical dock areas) of the dock area using setOffset(). If you'd prefer a dock window to start on a new line when it is docked use setNewLine(). The setFixedExtentWidth() and setFixedExtentHeight() functions can be used to define the dock window's preferred size, and the setHorizontallyStretchable() and setVerticallyStretchable() functions set whether the dock window can be stretched or not. Dock windows can be moved by default, but this can be changed with setMovingEnabled(). When a dock window is moved it is shown as a rectangular outline, but it can be shown normally using setOpaqueMoving().

When a dock window's visibility changes, i.e. it is shown or hidden, the visibilityChanged() signal is emitted. When a dock window is docked or undocked the placeChanged() signal is emitted.

See also Main Window and Related Classes.

Member Type Documentation

QDockWindow::CloseMode

This enum type specifies when (if ever) a dock window has a close button.

- `QDockWindow::Never` - The dock window never has a close button and cannot be closed by the user.
- `QDockWindow::Docked` - The dock window has a close button only when docked.
- `QDockWindow::Undocked` - The dock window has a close button only when floating.
- `QDockWindow::Always` - The dock window always has a close button.

Note that dock windows can always be minimized if the user clicks their dock window handle when they are docked.

QDockWindow::Place

This enum specifies the possible locations for a QDockWindow:

- `QDockWindow::InDock` - Inside a QDockArea.
- `QDockWindow::OutsideDock` - Floating as a top level window on the desktop.

Member Function Documentation

QDockWindow::QDockWindow (Place p = InDock, QWidget * parent = 0, const char * name = 0, WFlags f = 0)

Constructs a QDockWindow with parent *parent*, name *name* and widget flags *f*.

If *p* is InDock, the dock window is docked into a dock area and *parent must* be a QDockArea or a QMainWindow. If the *parent* is a QMainWindow the dock window will be docked in the main window's Top dock area.

If *p* is OutsideDock, the parent *must* be 0 and the dock window is created as a floating window.

We recommend creating the dock area InDock with a QMainWindow as parent then calling QMainWindow::moveDockWindow() to move the dock window where you want it.

QDockArea * QDockWindow::area () const

Returns the dock area in which this dock window is docked, or 0 if the dock window is floating.

QBoxLayout * QDockWindow::boxLayout ()

Returns the layout which is used for adding widgets to the dock window. The layout's orientation is set automatically to match the orientation of the dock window. You can add widgets to the layout using the box layout's QBoxLayout::addWidget() function.

If the dock window only needs to contain a single widget use setWidget() instead.

See also setWidget() [p. 28] and setOrientation() [p. 27].

int QDockWindow::closeMode () const

Returns the close mode of a dock window. See the "closeMode" [p. 28] property for details.

void QDockWindow::dock () [virtual slot]

Docks the dock window into the last dock area in which it was docked.

If the dock window has no last dock area (e.g. it was created as a floating window and has never been docked), or if the last dock area it was docked in does not exist (e.g. the dock area has been deleted), nothing happens.

See also undock() [p. 28].

QSize QDockWindow::fixedExtent () const

Returns the dock window's preferred size (fixed extent).

See also setFixedExtentWidth() [p. 26] and setFixedExtentHeight() [p. 26].

bool QDockWindow::isCloseEnabled () const

Returns TRUE if the dock window has a close button; otherwise returns FALSE. The result depends on the dock window's Place and its CloseMode.

See also closeMode [p. 28].

bool QDockWindow::isHorizontalStretchable () const

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

bool QDockWindow::isHorizontallyStretchable () const

Returns TRUE if the dock window is horizontally stretchable; otherwise returns FALSE. See the "horizontallyStretchable" [p. 28] property for details.

bool QDockWindow::isMovingEnabled () const

Returns TRUE if the user can move the dock window within the dock area, move the dock window to another dock area, or float the dock window; otherwise returns FALSE. See the "movingEnabled" [p. 29] property for details.

bool QDockWindow::isResizeEnabled () const

Returns TRUE if the dock window is resizeable; otherwise returns FALSE. See the "resizeEnabled" [p. 29] property for details.

bool QDockWindow::isStretchable () const

Returns TRUE if the dock window is stretchable in the current orientation(); otherwise returns FALSE. See the "stretchable" [p. 30] property for details.

bool QDockWindow::isVerticalStretchable () const

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

bool QDockWindow::isVerticallyStretchable () const

Returns TRUE if the dock window is vertically stretchable; otherwise returns FALSE. See the "verticallyStretchable" [p. 30] property for details.

bool QDockWindow::newLine () const

Returns TRUE if the dock window prefers to start a new line in the dock area; otherwise returns FALSE. See the "newLine" [p. 29] property for details.

int QDockWindow::offset () const

Returns the dock window's preferred offset from the dock area's left edge (top edge for vertical dock areas). See the "offset" [p. 29] property for details.

bool QDockWindow::opaqueMoving () const

Returns TRUE if the dock window will be shown normally whilst it is being moved; otherwise returns FALSE. See the "opaqueMoving" [p. 29] property for details.

Qt::Orientation QDockWindow::orientation () const

Returns the orientation of the dock window.

See also orientationChanged() [p. 26].

void QDockWindow::orientationChanged (Orientation o) [signal]

This signal is emitted when the orientation of the dock window is changed. The new orientation is *o*.

Place QDockWindow::place () const

Returns TRUE if the dock window is in a dock area; otherwise returns FALSE. See the "place" [p. 29] property for details.

void QDockWindow::placeChanged (QDockWindow::Place p) [signal]

This signal is emitted when the dock window is docked (*p* is InDock) or undocked (*p* is OutsideDock).

See also QDockArea::moveDockWindow() [p. 19], QDockArea::removeDockWindow() [p. 19], QMainWindow::moveDockWindow() [p. 63] and QMainWindow::removeDockWindow() [p. 63].

void QDockWindow::setCloseMode (int m) [virtual]

Sets the close mode of a dock window to *m*. See the "closeMode" [p. 28] property for details.

void QDockWindow::setFixedExtentHeight (int h) [virtual]

Sets the dock window's preferred height for its fixed extent (size) to *h*.

See also setFixedExtentWidth() [p. 26].

void QDockWindow::setFixedExtentWidth (int w) [virtual]

Sets the dock window's preferred width for its fixed extent (size) to *w*.

See also setFixedExtentHeight() [p. 26].

void QDockWindow::setHorizontalStretchable (bool b)

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

void QDockWindow::setHorizontallyStretchable (bool b) [virtual]

Sets whether the dock window is horizontally stretchable to *b*. See the "horizontallyStretchable" [p. 28] property for details.

void QDockWindow::setMovingEnabled (bool b) [virtual]

Sets whether the user can move the dock window within the dock area, move the dock window to another dock area, or float the dock window to *b*. See the "movingEnabled" [p. 29] property for details.

void QDockWindow::setNewLine (bool b) [virtual]

Sets whether the dock window prefers to start a new line in the dock area to *b*. See the "newLine" [p. 29] property for details.

void QDockWindow::setOffset (int o) [virtual]

Sets the dock window's preferred offset from the dock area's left edge (top edge for vertical dock areas) to *o*. See the "offset" [p. 29] property for details.

void QDockWindow::setOpaqueMoving (bool b) [virtual]

Sets whether the dock window will be shown normally whilst it is being moved to *b*. See the "opaqueMoving" [p. 29] property for details.

void QDockWindow::setOrientation (Orientation o) [virtual slot]

Sets the orientation of the dock window to *o*. The orientation is propagated to the layout boxLayout().

void QDockWindow::setResizeEnabled (bool b) [virtual]

Sets whether the dock window is resizeable to *b*. See the "resizeEnabled" [p. 29] property for details.

void QDockWindow::setVerticalStretchable (bool b)

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

void QDockWindow::setVerticallyStretchable (bool b) [virtual]

Sets whether the dock window is vertically stretchable to *b*. See the "verticallyStretchable" [p. 30] property for details.

void QDockWindow::setWidget (QWidget * w) [virtual]

Sets the dock window's main widget to *w*.

See also boxLayout() [p. 24].

void QDockWindow::undock (QWidget * w) [virtual slot]

Undocks the QDockWindow from its current dock area, if it is docked; otherwise does nothing.

Do not pass any *w* paramenter, it is for internal use only.

See also dock() [p. 24], QDockArea::moveDockWindow() [p. 19], QDockArea::removeDockWindow() [p. 19], QMainWindow::moveDockWindow() [p. 63] and QMainWindow::removeDockWindow() [p. 63].

void QDockWindow::visibilityChanged (bool visible) [signal]

This signal is emitted if the visibility of the dock window is changed. If *visible* is TRUE, the QDockWindow is now visible, otherwise it has been hidden.

A dock window can be hidden if it has a close button which the user has clicked. In the case of a QMainWindow a dock window can have its visibility changed (hidden or shown) by clicking its name in the dock window menu that lists the QMainWindow's dock windows.

QWidget * QDockWindow::widget () const

Returns the dock window's main widget.

See also setWidget() [p. 28].

Property Documentation

int closeMode

This property holds the close mode of a dock window.

Defines when (if ever) the dock window has a close button. The choices are Never, Docked (i.e. only when docked), Undocked (only when undocked, i.e. floated) or Always.

The default is Never.

Set this property's value with setCloseMode() and get this property's value with closeMode().

bool horizontallyStretchable

This property holds whether the dock window is horizontally stretchable.

A dock window is horizontally stretchable if you call setHorizontallyStretchable(TRUE) or setResizeEnabled(TRUE).

See also resizeEnabled [p. 29].

Set this property's value with setHorizontallyStretchable() and get this property's value with isHorizontallyStretchable().

bool movingEnabled

This property holds whether the user can move the dock window within the dock area, move the dock window to another dock area, or float the dock window.

This property is TRUE by default.

Set this property's value with setMovingEnabled() and get this property's value with isMovingEnabled().

bool newLine

This property holds whether the dock window prefers to start a new line in the dock area.

The default is FALSE, i.e. the dock window doesn't require a new line in the dock area.

Set this property's value with setNewLine() and get this property's value with newLine().

int offset

This property holds the dock window's preferred offset from the dock area's left edge (top edge for vertical dock areas).

The default is 0.

Set this property's value with setOffset() and get this property's value with offset().

bool opaqueMoving

This property holds whether the dock window will be shown normally whilst it is being moved.

If this property is FALSE, (the default), the dock window will be represented by an outline rectangle whilst it is being moved.

Set this property's value with setOpaqueMoving() and get this property's value with opaqueMoving().

Place place

This property holds whether the dock window is in a dock area.

The place() function returns the current place of the dock window. This is either InDock or OutsideDock.

See also QDockArea::moveDockWindow() [p. 19], QDockArea::removeDockWindow() [p. 19], QMainWindow::moveDockWindow() [p. 63] and QMainWindow::removeDockWindow() [p. 63].

Get this property's value with place().

bool resizeEnabled

This property holds whether the dock window is resizeable.

A resizeable dock window can be resized using splitter-like handles inside a dock area and like every other top level window when floating.

A dock window is both horizontally and vertically stretchable if you call or setResizeEnabled(TRUE).

This property is FALSE by default.

See also verticallyStretchable [p. 30] and horizontallyStretchable [p. 28].

Set this property's value with setResizeEnabled() and get this property's value with isResizeEnabled().

bool stretchable

This property holds whether the dock window is stretchable in the current orientation().

This property can be set using setHorizontallyStretchable() and setVerticallyStretchable(), or with setResizeEnabled().

See also resizeEnabled [p. 29].

Get this property's value with isStretchable().

bool verticallyStretchable

This property holds whether the dock window is vertically stretchable.

A dock window is horizontally stretchable if you call setVerticallyStretchable(TRUE) or setResizeEnabled(TRUE).

See also resizeEnabled [p. 29].

Set this property's value with setVerticallyStretchable() and get this property's value with isVerticallyStretchable().

QFileDialog Class Reference

The QFileDialog class provides dialogs that allow users to select files or directories.

`#include <qfiledialog.h>`

Inherits QDialog [p. 10].

Public Members

- **QFileDialog** (const QString & dirName, const QString & filter = QString::null, QWidget * parent = 0, const char * name = 0, bool modal = FALSE)
- **QFileDialog** (QWidget * parent = 0, const char * name = 0, bool modal = FALSE)
- **~QFileDialog** ()
- QString **selectedFile** () const
- QString **selectedFilter** () const
- virtual void **setSelectedFilter** (const QString & mask)
- virtual void **setSelectedFilter** (int n)
- void **setSelection** (const QString & filename)
- void **selectAll** (bool b)
- QStringList **selectedFiles** () const
- QString **dirPath** () const
- void **setDir** (const QDir & dir)
- const QDir * **dir** () const
- void **setShowHiddenFiles** (bool s)
- bool **showHiddenFiles** () const
- void **rereadDir** ()
- void **resortDir** ()
- enum **Mode** { AnyFile, ExistingFile, Directory, ExistingFiles, DirectoryOnly }
- void **setMode** (Mode)
- Mode **mode** () const
- enum **ViewMode** { Detail, List }
- enum **PreviewMode** { NoPreview, Contents, Info }
- void **setViewMode** (ViewMode m)
- ViewMode **viewMode** () const
- void **setPreviewMode** (PreviewMode m)
- PreviewMode **previewMode** () const
- bool **isInfoPreviewEnabled** () const
- bool **isContentsPreviewEnabled** () const
- void **setInfoPreviewEnabled** (bool)
- void **setContentsPreviewEnabled** (bool)
- void **setInfoPreview** (QWidget * w, QFilePreview * preview)

- void **setContentsPreview** (QWidget * w, QFilePreview * preview)
- QUrl **url** () const
- void **addFilter** (const QString & filter)

Public Slots

- void **setDir** (const QString & pathstr)
- void **setUrl** (const QUrlOperator & url)
- void **setFilter** (const QString & newFilter)
- void **setFilters** (const QString & filters)
- void **setFilters** (const char ** types)
- void **setFilters** (const QStringList &)

Signals

- void **fileHighlighted** (const QString &)
- void **fileSelected** (const QString &)
- void **filesSelected** (const QStringList &)
- void **dirEntered** (const QString &)
- void **filterSelected** (const QString &)

Static Public Members

- QString **getOpenFileName** (const QString & startWith = QString::null, const QString & filter = QString::null, QWidget * parent = 0, const char * name = 0, const QString & caption = QString::null, QString * selectedFilter = 0, bool resolveSymlinks = TRUE)
- QString **getSaveFileName** (const QString & startWith = QString::null, const QString & filter = QString::null, QWidget * parent = 0, const char * name = 0, const QString & caption = QString::null, QString * selectedFilter = 0, bool resolveSymlinks = TRUE)
- QString **getExistingDirectory** (const QString & dir = QString::null, QWidget * parent = 0, const char * name = 0, const QString & caption = QString::null, bool dirOnly = TRUE, bool resolveSymlinks = TRUE)
- QStringList **getOpenFileNames** (const QString & filter = QString::null, const QString & dir = QString::null, QWidget * parent = 0, const char * name = 0, const QString & caption = QString::null, QString * selectedFilter = 0, bool resolveSymlinks = TRUE)
- void **setIconProvider** (QFileIconProvider * provider)
- QFileIconProvider * **iconProvider** ()

Properties

- bool **contentsPreview** — whether the file dialog offers the possibility of previewing the contents of the currently selected file
- QString **dirPath** — the file dialog's working directory *(read only)*
- bool **infoPreview** — whether the file dialog offers the possibility to preview information about the currently selected file
- Mode **mode** — the file dialog's mode
- PreviewMode **previewMode** — the preview mode for the file dialog

- QString **selectedFile** — the name of the selected file *(read only)*
- QStringList **selectedFiles** — a list of selected files *(read only)*
- QString **selectedFilter** — the filter which the user has selected in the file dialog *(read only)*
- bool **showHiddenFiles** — whether hidden files are shown in the file dialog
- ViewMode **viewMode** — the file dialog's view mode

Protected Members

- void **addWidgets** (QLabel * l, QWidget * w, QPushButton * b)
- void **addToolButton** (QButton * b, bool separator = FALSE)
- void **addLeftWidget** (QWidget * w)
- void **addRightWidget** (QWidget * w)

Detailed Description

The QFileDialog class provides dialogs that allow users to select files or directories.

The QFileDialog class enables a user to traverse their file system in order to select one or many files or a directory.

The easiest way to create a QFileDialog is to use the static functions. On Windows, these static functions will call the native Windows file dialog and on Mac OS X, these static function will call the native Mac OS X file dialog.

```
QString s = QFileDialog::getOpenFileName( "/home",
                                          "Images (*.png *.xpm *.jpg)",
                                          this,
                                          "open file dialog"
                                          "Choose a file" );
```

In the above example, a modal QFileDialog is created using a static function. The startup directory is set to "/home". The file filter is set to "Images (*.png *.xpm *.jpg)". The parent of the file dialog is set to *this* and it is given the identification name - "open file dialog". The caption at the top of file dialog is set to "Choose a file".

You can create your own QFileDialog without using the static functions. By calling setMode(), you can set what can be returned by the QFileDialog.

```
QFileDialog* fd = new QFileDialog( this, "file dialog", TRUE );
fd->setMode( QFileDialog::AnyFile );
```

In the above example, the mode of the file dialog is set to AnyFile, meaning that the user can select any file, or even specify a file that doesn't exist. This mode is useful for creating a "File Save As" file dialog. Use ExistingFile if the user must select an existing file or Directory if only a directory must be selected. (See the QFileDialog::Mode enum for the complete list of modes.)

You can retrieve the dialog's mode with mode(). Use setFilter() to set the dialog's file filter, e.g.

```
fd->setFilter( "Images (*.png *.xpm *.jpg)" );
```

In the above example, the filter is set to "Images (*.png *.xpm *.jpg)", this means that only files with the extension `png`, `xpm` or `jpg` files will be visible in the QFileDialog. You can apply several filters by using setFilters() and add additional filters with addFilter(). Use setSelectedFilter() to select one of the filters you've given as the file dialog's default filter. Whenever the user changes the filter the filterSelected() signal is emitted.

The file dialog has two view modes, QFileDialog::List which simply lists file and directory names and QFileDialog::Detail which displays additional information beside each name, e.g. file size, modification date, etc. Set the move with setViewMode().

```
fd->setViewMode( QFileDialog::Detail );
```

The last important function you will need to use when creating your own file dialog is selectedFile().

```
QString fileName;
if ( fd->exec() == QDialog::Accepted )
    fileName = fd->selectedFile();
```

In the above example, a modal file dialog is created and shown. If the user clicked OK, then the file they selected is put in `fileName`.

If you are using the ExistingFiles mode then you will need to use selectedFiles() which will return the selected files in a QStringList.

The dialog's working directory can be set with setDir(). The display of hidden files is controlled with setShowHiddenFiles(). The dialog can be forced to re-read the directory with rereadDir() and re-sort the directory with resortDir(). All the files in the current directory can be selected with selectAll().

Creating and using preview widgets

There are two kinds of preview widgets that can be used with QFileDialogs: *content* preview widgets and *information* preview widgets. They are created and used in the same way except that the function names differ, e.g. setContentsPreview() and setInfoPreview().

A preview widget is a widget that is placed inside a QFileDialog so that the user can see either the contents of the file, or information about the file.

```
class Preview : public QLabel, public QFilePreview
{
public:
    Preview( QWidget *parent=0 ) : QLabel( parent ) {}

    void previewUrl( const QUrl &u )
    {
        QString path = u.path();
        QPixmap pix( path );
        if ( pix.isNull() )
            setText( "This is not a pixmap" );
        else
            setPixmap( pix );
    }
};
```

In the above snippet, we create a preview widget which inherits from QLabel and QFilePreview. File preview widgets *must* inherit from QFilePreview.

Inside the class we reimplement QFilePreview::previewUrl(), this is where we determine what happens when a file is selected. In the case above we only show a preview of the file if it is a valid pixmap. Now we tell a file dialog that we have a preview widget that we want it to use.

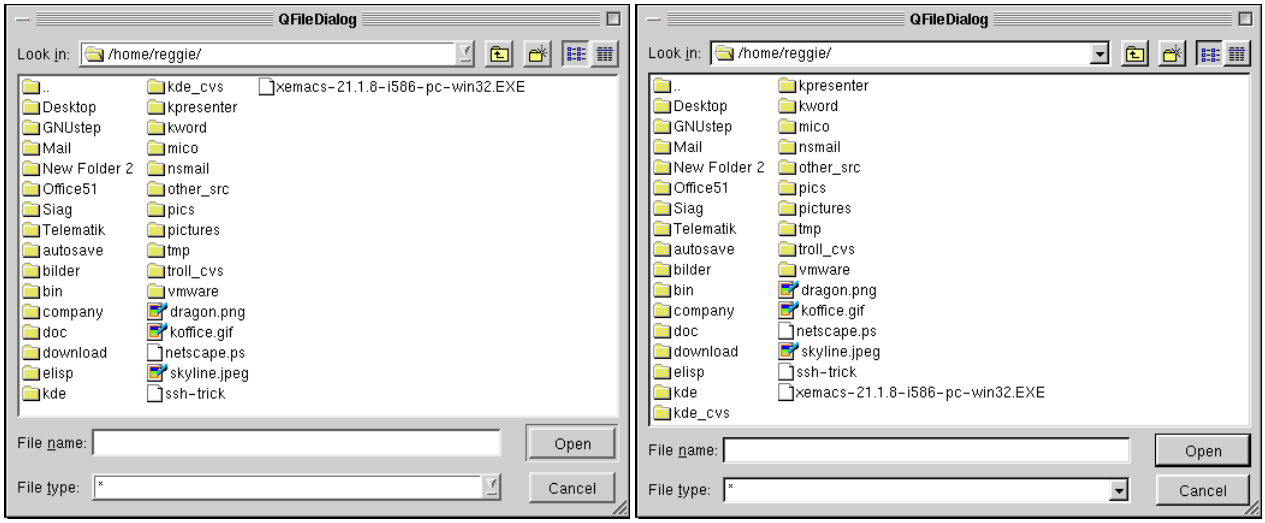
```
Preview* p = new Preview;

QFileDialog* fd = new QFileDialog( this );
fd->setContentsPreviewEnabled( TRUE );
fd->setContentsPreview( p, p );
fd->setPreviewMode( QFileDialog::Contents );
fd->show();
```

The first line creates an instance of our preview widget. We then create our file dialog and call setContentsPreviewEnabled(TRUE), this tell the file dialog to preview the contents of the currently selected file. We then call setContentsPreview() — note that we pass the same preview widget twice. Finally, before showing the file dialog, we call setPreviewMode() setting the mode to *Contents* which will show the contents the file that the user has selected.

If you create another preview widget that is used for displaying information about a file, create it in the same way as we have the contents preview widget and call setInfoPreviewEnabled(), and setInfoPreview(). Then the user will be able to switch between the two preview modes.

For more information about creating a QFilePreview widget see QFilePreview.



See also Dialog Classes.

Member Type Documentation

QFileDialog::Mode

This enum is used to indicate what the user may select in the file dialog, i.e. what the dialog will return if the user clicks OK.

- `QFileDialog::AnyFile` - The name of a file, whether it exists or not.
- `QFileDialog::ExistingFile` - The name of a single existing file.
- `QFileDialog::Directory` - The name of a directory. Both files and directories are displayed.
- `QFileDialog::DirectoryOnly` - The name of a directory. The file dialog will only display directories.
- `QFileDialog::ExistingFiles` - The names of zero or more existing files.

See setMode().

QFileDialog::PreviewMode

This enum describes the preview mode of the file dialog.

- `QFileDialog::NoPreview` - No preview is shown at all.
- `QFileDialog::Contents` - Show a preview of the contents of the current file using the contents preview widget.

- `QFileDialog::Info` - Show information about the current file using the info preview widget.

See setPreviewMode(), setContentsPreview() and setInfoPreview().

QFileDialog::ViewMode

This enum describes the view mode of the file dialog, i.e. what information about each file it will display.

- `QFileDialog::List` - Display file and directory names with icons.
- `QFileDialog::Detail` - Display file and directory names with icons plus additional information, e.g. file size, modification date.

See setViewMode().

Member Function Documentation

QFileDialog::QFileDialog (const QString & dirName, const QString & filter = QString::null, QWidget * parent = 0, const char * name = 0, bool modal = FALSE)

Constructs a file dialog with the parent *parent*, the name *name* If *modal* is TRUE then the file dialog is modal; otherwise it is non-modal.

If *dirName* is specified then it will be used as the dialog's working directory, i.e. it will be the directory that is shown when the dialog appears. If *filter* is specified it will be used as the dialog's file filter.

QFileDialog::QFileDialog (QWidget * parent = 0, const char * name = 0, bool modal = FALSE)

Constructs a file dialog with the parent, *parent*, and the name, *name*. If *modal* is TRUE then the file dialog is modal; otherwise it is non-modal.

QFileDialog::~QFileDialog ()

Destroys the file dialog.

void QFileDialog::addFilter (const QString & filter)

Adds the filter *filter* to the list of filters and makes it the current one.

```
QFileDialog* fd = new QFileDialog( this );
fd->addFilter( "Images (*.png *.jpg *.xpm)" );
fd->show();
```

In the above example, a file dialog is created, and the file filter - "Images (*.png *.jpg *.xpm)" is added and is set as the current filter. The original filter - "All Files (*)" is still available.

See also setFilter() [p. 42] and setFilters() [p. 43].

void QFileDialog::addLeftWidget (QWidget * w) [protected]

Adds the widget *w* to the left-hand side of the file dialog.

See also addRightWidget() [p. 37], addWidgets() [p. 37] and addToolButton() [p. 37].

void QFileDialog::addRightWidget (QWidget * w) [protected]

Adds the widget *w* to the right-hand side of the file dialog.

See also addLeftWidget() [p. 37], addWidgets() [p. 37] and addToolButton() [p. 37].

void QFileDialog::addToolButton (QButton * b, bool separator = FALSE) [protected]

Adds the tool button *b* to the row of tool buttons at the top of the file dialog. The button is appended to the right of this row. If *separator* is TRUE, a small space is inserted between the last button of the row and the new button *b*.

See also addWidgets() [p. 37], addLeftWidget() [p. 37] and addRightWidget() [p. 37].

void QFileDialog::addWidgets (QLabel * l, QWidget * w, QPushButton * b) [protected]

```
MyFileDialog::MyFileDialog( QWidget* parent, const char* name ) :
    QFileDialog( parent, name )
{
    QLabel* label = new QLabel( "Added widgets", this );
    QLineEdit* lineedit = new QLineEdit( this );
    QToolButton* toolbutton = new QToolButton( this );

    addWidgets( label, lineedit, toolbutton );
}
```

Adds the specified widgets to the bottom of the file dialog. The label *l* is placed underneath the "file name" and the "file types" labels. The widget *w* is placed underneath the file types combobox. The button *b* is placed underneath the cancel pushbutton.

If you don't want to have one of the widgets to be added then just pass 0 instead of a label, widget or pushbutton.

Every time you call this function, a new row of widgets will be added to the bottom of the file dialog.

See also addToolButton() [p. 37], addLeftWidget() [p. 37] and addRightWidget() [p. 37].

const QDir * QFileDialog::dir () const

Returns the current directory shown in the file dialog.

See also setDir() [p. 42].

void QFileDialog::dirEntered (const QString &) [signal]

This signal is emitted when the user enters a directory.

See also dir() [p. 37].

QString QFileDialog::dirPath () const

Returns the file dialog's working directory. See the "dirPath" [p. 46] property for details.

void QFileDialog::fileHighlighted (const QString &) [signal]

This signal is emitted when the user highlights a file.

See also fileSelected() [p. 38] and filesSelected() [p. 38].

void QFileDialog::fileSelected (const QString &) [signal]

This signal is emitted when the user selects a file.

See also filesSelected() [p. 38], fileHighlighted() [p. 38] and selectedFile [p. 46].

void QFileDialog::filesSelected (const QStringList &) [signal]

This signal is emitted when the user selects one or more files in *ExistingFiles* mode.

See also fileSelected() [p. 38], fileHighlighted() [p. 38] and selectedFiles [p. 46].

void QFileDialog::filterSelected (const QString &) [signal]

This signal is emitted when the user selects a filter.

See also selectedFilter [p. 47].

QString QFileDialog::getExistingDirectory (const QString & dir = QString::null, QWidget * parent = 0, const char * name = 0, const QString & caption = QString::null, bool dirOnly = TRUE, bool resolveSymlinks = TRUE) [static]

This is a convenience static function that will return an existing directory selected by the user.

```
QString s = QFileDialog::getExistingDirectory(
                "/home",
                this, "get existing directory"
                "Choose a directory", TRUE );
```

This function creates a modal file dialog with parent *parent*, and name *name*. If a parent is specified the dialog will be shown centered over the parent.

The dialog's working directory is set to *dir*, and the caption is set to *caption*. Either of these may be QString::null in which case the current directory and a default caption will be used respectively.

If *dirOnly* is TRUE, then only directories will be shown in the file dialog; otherwise both directories and files will be shown.

Under Unix/X11, the normal behavior of the file dialog is to resolve and follow symlinks. For example, if /usr/tmp is a symlink to /var/tmp, the file dialog will change to /var/tmp after entering /usr/tmp. If *resolveSymlinks* is FALSE, the file dialog will treat symlinks as regular directories.

See also getOpenFileName() [p. 39], getOpenFileNames() [p. 39] and getSaveFileName() [p. 40].

QString QFileDialog::getOpenFileName (const QString & startWith = QString::null, const QString & filter = QString::null, QWidget * parent = 0, const char * name = 0, const QString & caption = QString::null, QString * selectedFilter = 0, bool resolveSymlinks = TRUE) [static]

This is a convenience static function that returns an existing file selected by the user. If the user pressed cancel, it returns a null string.

```
QString s = QFileDialog::getOpenFileName(
                "/home", "Images (*.png *.xpm *.jpg)",
                this, "open file dialog",
                "Choose a file" );
```

The function creates a modal file dialog with parent *parent*, and name *name*. If a parent is specified, then the dialog will be shown centered over the parent.

The file dialog's working directory will be set to *startWith*. If *startWith* includes a file name, the file will be selected. The filter is set to *filter* so that only those files which match the filter are shown. The filter selected is set to *selectedFilter*. The parameters *startWith*, *selectedFilter* and *filter* may be QString::null.

The dialog's caption is set to *caption*. If *caption* is not specified then a default caption will be used.

Under Windows and Mac OS X, this static function will use the native file dialog and not a QFileDialog, unless the style of the application is set to something other than the native style.

Under Unix/X11, the normal behavior of the file dialog is to resolve and follow symlinks. For example, if /usr/tmp is a symlink to /var/tmp, the file dialog will change to /var/tmp after entering /usr/tmp. If *resolveSymlinks* is FALSE, the file dialog will treat symlinks as regular directories.

See also getOpenFileNames() [p. 39], getSaveFileName() [p. 40] and getExistingDirectory() [p. 38].

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

QStringList QFileDialog::getOpenFileNames (const QString & filter = QString::null, const QString & dir = QString::null, QWidget * parent = 0, const char * name = 0, const QString & caption = QString::null, QString * selectedFilter = 0, bool resolveSymlinks = TRUE) [static]

This is a convenience static function that will return one or more existing files as selected by the user.

```
QStringList s = QFileDialog::getOpenFileNames(
                    "Images (*.png *.xpm *.jpg)", "/home",
                    this, "open files dialog"
                    "Select one or more files" );
```

The function creates a modal file dialog with parent *parent*, and name *name*. If a parent is specified, then the dialog will be shown centered over the parent.

The file dialog's working directory will be set to *dir*. If *dir* includes a file name, the file will be selected. The filter is set to *filter* so that only those files which match the filter are shown. The filter selected is set to *selectedFilter*. The parameters *dir*, *selectedFilter* and *filter* may be QString::null.

The dialog's caption is set to *caption*. If *caption* is not specified then a default caption will be used.

Under Windows and Mac OS X, this static function will use the native file dialog and not a QFileDialog, unless the style of the application is set to something other than the native style.

Under Unix/X11, the normal behavior of the file dialog is to resolve and follow symlinks. For example, if /usr/tmp is a symlink to /var/tmp, the file dialog will change to /var/tmp after entering /usr/tmp. If *resolveSymlinks* is FALSE, the file dialog will treat symlinks as regular directories.

See also getOpenFileName() [p. 39], getSaveFileName() [p. 40] and getExistingDirectory() [p. 38].

QString QFileDialog::getSaveFileName (const QString & startWith = QString::null, const QString & filter = QString::null, QWidget * parent = 0, const char * name = 0, const QString & caption = QString::null, QString * selectedFilter = 0, bool resolveSymlinks = TRUE) [static]

This is a convenience static function that will return a file name selected by the user. The file does not have to exist.

It creates a modal file dialog with parent *parent*, and name *name*. If a parent is specified, then the dialog will be shown centered over the parent.

```
QString s = QFileDialog::getSaveFileName(
                "/home", "Images (*.png *.xpm *.jpg)",
                this, "save file dialog"
                "Choose a file" );
```

The file dialog's working directory will be set to *startWith*. If *startWith* includes a file name, the file will be selected. The filter is set to *filter* so that only those files which match the filter are shown. The filter selected is set to *selectedFilter*. The parameters *startWith*, *selectedFilter* and *filter* may be QString::null.

The dialog's caption is set to *caption*. If *caption* is not specified then a default caption will be used.

Under Windows and Mac OS X, this static function will use the native file dialog and not a QFileDialog, unless the style of the application is set to something other than the native style.

Under Unix/X11, the normal behavior of the file dialog is to resolve and follow symlinks. For example, if /usr/tmp is a symlink to /var/tmp, the file dialog will change to /var/tmp after entering /usr/tmp. If *resolveSymlinks* is FALSE, the file dialog will treat symlinks as regular directories.

See also getOpenFileName() [p. 39], getOpenFileNames() [p. 39] and getExistingDirectory() [p. 38].

Examples: action/application.cpp, addressbook/mainwindow.cpp, application/application.cpp, mdi/application.cpp, qmag/qmag.cpp, qwerty/qwerty.cpp and showimg/showimg.cpp.

QFileIconProvider * QFileDialog::iconProvider () [static]

Returns a pointer to the icon provider currently set on the file dialog. By default there is no icon provider, and this function returns 0.

See also setIconProvider() [p. 43] and QFileIconProvider [p. 48].

bool QFileDialog::isContentsPreviewEnabled () const

Returns TRUE if the file dialog offers the possibility of previewing the contents of the currently selected file; otherwise returns FALSE. See the "contentsPreview" [p. 45] property for details.

bool QFileDialog::isInfoPreviewEnabled () const

Returns TRUE if the file dialog offers the possibility to preview information about the currently selected file; otherwise returns FALSE. See the "infoPreview" [p. 46] property for details.

Mode QFileDialog::mode () const

Returns the file dialog's mode. See the "mode" [p. 46] property for details.

PreviewMode QFileDialog::previewMode () const

Returns the preview mode for the file dialog. See the "previewMode" [p. 46] property for details.

void QFileDialog::rereadDir ()

Rereads the current directory shown in the file dialog.

The only time you will need to call this function is if the contents of the directory change and you wish to refresh the file dialog to reflect the change.

See also resortDir() [p. 41].

void QFileDialog::resortDir ()

Re-sorts the displayed directory.

See also rereadDir() [p. 41].

void QFileDialog::selectAll (bool b)

If *b* is TRUE then all the files in the current directory are selected; otherwise, they are deselected.

QString QFileDialog::selectedFile () const

Returns the name of the selected file. See the "selectedFile" [p. 46] property for details.

QStringList QFileDialog::selectedFiles () const

Returns a list of selected files. See the "selectedFiles" [p. 46] property for details.

QString QFileDialog::selectedFilter () const

Returns the filter which the user has selected in the file dialog. See the "selectedFilter" [p. 47] property for details.

void QFileDialog::setContentsPreview (QWidget * w, QFilePreview * preview)

Sets the widget to be used for displaying the contents of the file to the widget *w* and a preview of those contents to the QFilePreview *preview*.

Normally you would create a preview widget that derives from both QWidget and QFilePreview, so you should pass the same widget twice.

```
class Preview : public QLabel, public QFilePreview
{
public:
```

```
        Preview( QWidget *parent=0 ) : QLabel( parent ) {}

        void previewUrl( const QUrl &u )
        {
            QString path = u.path();
            QPixmap pix( path );
            if ( pix.isNull() )
                setText( "This is not a pixmap" );
            else
                setPixmap( pix );
        }
    };

  //...

  int main( int argc, char** argv )
  {
    Preview* p = new Preview;

    QFileDialog* fd = new QFileDialog( this );
    fd->setContentsPreviewEnabled( TRUE );
    fd->setContentsPreview( p, p );
    fd->setPreviewMode( QFileDialog::Contents );
    fd->show();
  }
```

See also contentsPreview [p. 45], setInfoPreview() [p. 43] and previewMode [p. 46].

Example: qdir/qdir.cpp.

void QFileDialog::setContentsPreviewEnabled (bool)

Sets whether the file dialog offers the possibility of previewing the contents of the currently selected file. See the "contentsPreview" [p. 45] property for details.

void QFileDialog::setDir (const QDir & dir)

Sets the file dialog's working directory to *dir*.

See also dir() [p. 37].

void QFileDialog::setDir (const QString & pathstr) [slot]

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

Sets the file dialog's working directory to *pathstr*.

See also dir() [p. 37].

void QFileDialog::setFilter (const QString & newFilter) [slot]

Sets the filter used in the file dialog to *newFilter*.

If *newFilter* contains a pair of parentheses containing one or more of ***anything*something*** separated by spaces or by semi-colons then only the text contained in the parentheses is used as the filter. This means that these calls are

all equivalent:

```
fd->setFilter( "All C++ files (*.cpp *.cc *.C *.cxx *.c++)" );
fd->setFilter( "*.cpp *.cc *.C *.cxx *.c++" );
fd->setFilter( "All C++ files (*.cpp;*.cc;*.C;*.cxx;*.c++)" );
fd->setFilter( "*.cpp;*.cc;*.C;*.cxx;*.c++" );
```

See also setFilters() [p. 43].

void QFileDialog::setFilters (const QString & filters) [slot]

Sets the filters used in the file dialog to *filters*. Each group of filters must be separated by ;;.

```
QString types("*.png;;*.xpm;;*.jpg");
QFileDialog fd = new QFileDialog( this );
fd->setFilters( types );
fd->show();
```

void QFileDialog::setFilters (const char ** types) [slot]

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

types must be a null-terminated list of strings.

void QFileDialog::setFilters (const QStringList &) [slot]

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

void QFileDialog::setIconProvider (QFileIconProvider * provider) [static]

Sets the QFileIconProvider used on the file dialog to the QFileIconProvider specified by *provider*.

The default is that there is no QFileIconProvider and QFileDialog just draws a folder icon next to each directory and nothing next to the files.

See also QFileIconProvider [p. 48] and iconProvider() [p. 40].

Example: showimg/main.cpp.

void QFileDialog::setInfoPreview (QWidget * w, QFilePreview * preview)

Sets the widget to be used for displaying information about the file to the widget *w* and a preview of that information to the QFilePreview *preview*.

Normally you would create a preview widget that derives from both QWidget and QFilePreview, so you should pass the same widget twice.

```
class Preview : public QLabel, public QFilePreview
{
public:
    Preview( QWidget *parent=0 ) : QLabel( parent ) {}

    void previewUrl( const QUrl &u )
```

```
        {
            QString path = u.path();
            QPixmap pix( path );
            if ( pix.isNull() )
                setText( "This is not a pixmap" );
            else
                setText( "This is a pixmap" );
        }
    };

  //...

  int main( int argc, char** argv )
  {
    Preview* p = new Preview;

    QFileDialog* fd = new QFileDialog( this );
    fd->setInfoPreviewEnabled( TRUE );
    fd->setInfoPreview( p, p );
    fd->setPreviewMode( QFileDialog::Info );
    fd->show();
  }
```

See also setContentsPreview() [p. 41], infoPreview [p. 46] and previewMode [p. 46].

void QFileDialog::setInfoPreviewEnabled (bool)

Sets whether the file dialog offers the possibility to preview information about the currently selected file. See the "infoPreview" [p. 46] property for details.

void QFileDialog::setMode (Mode)

Sets the file dialog's mode. See the "mode" [p. 46] property for details.

void QFileDialog::setPreviewMode (PreviewMode m)

Sets the preview mode for the file dialog to *m*. See the "previewMode" [p. 46] property for details.

void QFileDialog::setSelectedFilter (const QString & mask) [virtual]

Sets the current filter selected in the file dialog to the first one that contains the text *mask*.

void QFileDialog::setSelectedFilter (int n) [virtual]

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

Sets the current filter selected in the file dialog to the *n* filter in the filter list.

See also filterSelected() [p. 38], selectedFilter [p. 47], selectedFiles [p. 46] and selectedFile [p. 46].

void QFileDialog::setSelection (const QString & filename)

Sets the default selection to *filename*. If *filename* is absolute, setDir() is also called to set the file dialog's working directory to the filename's directory.

Example: qdir/qdir.cpp.

void QFileDialog::setShowHiddenFiles (bool s)

Sets whether hidden files are shown in the file dialog to *s*. See the "showHiddenFiles" [p. 47] property for details.

void QFileDialog::setUrl (const QUrlOperator & url) [slot]

Sets the file dialog's working directory to the directory specified at *url*.

See also url() [p. 45].

void QFileDialog::setViewMode (ViewMode m)

Sets the file dialog's view mode to *m*. See the "viewMode" [p. 47] property for details.

bool QFileDialog::showHiddenFiles () const

Returns TRUE if hidden files are shown in the file dialog; otherwise returns FALSE. See the "showHiddenFiles" [p. 47] property for details.

QUrl QFileDialog::url () const

Returns the URL of the current working directory in the file dialog.

See also setUrl() [p. 45].

Example: network/networkprotocol/view.cpp.

ViewMode QFileDialog::viewMode () const

Returns the file dialog's view mode. See the "viewMode" [p. 47] property for details.

Property Documentation

bool contentsPreview

This property holds whether the file dialog offers the possibility of previewing the contents of the currently selected file.

The default is FALSE.

See also setContentsPreview() [p. 41] and infoPreview [p. 46].

Set this property's value with setContentsPreviewEnabled() and get this property's value with isContentsPreviewEnabled().

QString dirPath

This property holds the file dialog's working directory.

Get this property's value with dirPath().

See also dir() [p. 37] and setDir() [p. 42].

bool infoPreview

This property holds whether the file dialog offers the possibility to preview information about the currently selected file.

The default is FALSE.

Set this property's value with setInfoPreviewEnabled() and get this property's value with isInfoPreviewEnabled().

Mode mode

This property holds the file dialog's mode.

The default mode is ExistingFile.

Set this property's value with setMode() and get this property's value with mode().

PreviewMode previewMode

This property holds the preview mode for the file dialog.

If you set the mode to be a mode other than *NoPreview*, then use setInfoPreview() or setContentsPreview() to set the dialog's preview widget to your preview widget and enable the preview widget(s) with setInfoPreviewEnabled() or setContentsPreviewEnabled().

See also infoPreview [p. 46], contentsPreview [p. 45] and viewMode [p. 47].

Set this property's value with setPreviewMode() and get this property's value with previewMode().

QString selectedFile

This property holds the name of the selected file.

If a file was selected selectedFile contains the file's name including its absolute path; otherwise selectedFile is empty.

See also QString::isEmpty() [Datastructures and String Handling with Qt], selectedFiles [p. 46] and selectedFilter [p. 47].

Get this property's value with selectedFile().

QStringList selectedFiles

This property holds a list of selected files.

If one or more files were selected selectedFiles contains the names of the selected files including their absolute paths. If no files were selected or the mode wasn't ExistingFiles selectedFiles is an empty list.

It is more convenient to use selectedFile() if the mode is ExistingFile, Directory or DirectoryOnly.

See also selectedFile [p. 46], selectedFilter [p. 47] and QValueList::empty() [Datastructures and String Handling with Qt].

Get this property's value with selectedFiles().

QString selectedFilter

This property holds the filter which the user has selected in the file dialog.

Get this property's value with selectedFilter().

See also filterSelected() [p. 38], selectedFiles [p. 46] and selectedFile [p. 46].

bool showHiddenFiles

This property holds whether hidden files are shown in the file dialog.

The default is FALSE, i.e. don't show hidden files.

Set this property's value with setShowHiddenFiles() and get this property's value with showHiddenFiles().

ViewMode viewMode

This property holds the file dialog's view mode.

If you set the view mode to be *Detail* (the default), then you will see the file's details, such as the size of the file and the date the file was last modified alongside the file.

If you set the view mode to be *List*, then you will just see a list of the files and folders.

See QFileDialog::ViewMode

Set this property's value with setViewMode() and get this property's value with viewMode().

QFileIconProvider Class Reference

The QFileIconProvider class provides icons for QFileDialog to use.

`#include <qfiledialog.h>`

Inherits QObject [Additional Functionality with Qt].

Public Members

- **QFileIconProvider** (QObject * parent = 0, const char * name = 0)
- virtual const QPixmap * **pixmap** (const QFileInfo & info)

Detailed Description

The QFileIconProvider class provides icons for QFileDialog to use.

By default QFileIconProvider is not used, but any application or library can subclass it, reimplement pixmap() to return a suitable icon, and make all QFileDialog objects use it by calling the static function QFileDialog::setIconProvider().

It is advisable to make all the icons that QFileIconProvider returns be the same size or at least the same width. This makes the list view look much better.

See also QFileDialog [p. 31] and Miscellaneous Classes.

Member Function Documentation

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

Constructs an empty file icon provider with the parent *parent* and name *name*.

const QPixmap * QFileIconProvider::pixmap (const QFileInfo & info) [virtual]

Returns a pointer to a pixmap that should be used for visualizing the file with the information *info*.

If pixmap() returns 0, QFileDialog draws the default pixmap.

The default implementation returns particular icons for files, directories, link-files and link-directories. It returns a blank "icon" for other types.

If you return a pixmap here, it should measure 16x16.

QInputDialog Class Reference

The QInputDialog class provides a simple convenience dialog to get a single value from the user.

`#include <qinputdialog.h>`

Inherits QDialog [p. 10].

Static Public Members

- QString **getText** (const QString & caption, const QString & label, QLineEdit::EchoMode mode = QLineEdit::Normal, const QString & text = QString::null, bool * ok = 0, QWidget * parent = 0, const char * name = 0)
- int **getInteger** (const QString & caption, const QString & label, int num = 0, int from = -2147483647, int to = 2147483647, int step = 1, bool * ok = 0, QWidget * parent = 0, const char * name = 0)
- double **getDouble** (const QString & caption, const QString & label, double num = 0, double from = -2147483647, double to = 2147483647, int decimals = 1, bool * ok = 0, QWidget * parent = 0, const char * name = 0)
- QString **getItem** (const QString & caption, const QString & label, const QStringList & list, int current = 0, bool editable = TRUE, bool * ok = 0, QWidget * parent = 0, const char * name = 0)

Detailed Description

The QInputDialog class provides a simple convenience dialog to get a single value from the user.

The QInputDialog is a simple dialog which can be used if you need to get a single input value from the user. The input value can be a string, a number or an item from a list. A label has to be set to tell the user what they should input.

Four static convenience functions are provided: getText(), getInteger(), getDouble() and getItem(). All the functions can be used in a similar way, for example:

```
bool ok = FALSE;
QString text = QInputDialog::getText(
                  tr( "Application name" ),
                  tr( "Please enter your name" ),
                  QLineEdit::Normal, QString::null, &ok, this );
if ( ok && !text.isEmpty() )
    ;// user entered something and pressed OK
else
    ;// user entered nothing or pressed Cancel
```

See also Dialog Classes.

Member Function Documentation

double QInputDialog::getDouble (const QString & caption, const QString & label, double num = 0, double from = -2147483647, double to = 2147483647, int decimals = 1, bool * ok = 0, QWidget * parent = 0, const char * name = 0) [static]

Static convenience function to get a floating point number from the user. *caption* is the text which is displayed in the title bar of the dialog. *label* is the text which is shown to the user (it should mention what they should input), *num* is the default floating point number that the line edit will be set to. *from* and *to* are the minimum and maximum values the user may choose, and *decimals* is the maximum number of decimal places the number may have.

If *ok* is not-null it will be set to TRUE if the user pressed OK and FALSE if the user pressed Cancel. The dialog's parent is *parent*; the dialog is called *name*. The dialog will be modal.

This method returns the floating point number which has been entered by the user.

Use this static method like this:

```
bool ok = FALSE;
double res = QInputDialog::getDouble(
              tr( "Application name" ),
              tr( "Please enter a decimal number" ),
              33.7, 0, 1000, 2, &ok, this );
if ( ok )
    ;// user entered something and pressed OK
else
    ;// user pressed Cancel
```

int QInputDialog::getInteger (const QString & caption, const QString & label, int num = 0, int from = -2147483647, int to = 2147483647, int step = 1, bool * ok = 0, QWidget * parent = 0, const char * name = 0) [static]

Static convenience function to get an integer input from the user. *caption* is the text which is displayed in the title bar of the dialog. *label* is the text which is shown to the user (it should mention what they should input), *num* is the default number which the spinbox will be set to. *from* and *to* are the minimum and maximum values the user may choose, and *step* is the amount by which the values change as the user presses the arrow buttons to increment or decrement the value.

If *ok* is not-null it will be set to TRUE if the user pressed OK and FALSE if the user pressed Cancel. The dialog's parent is *parent*; the dialog is called *name*. The dialog will be modal.

This method returns the number which has been entered by the user.

Use this static method like this:

```
bool ok = FALSE;
int res = QInputDialog::getInteger(
              tr( "Application name" ),
              tr( "Please enter a number" ), 22, 0, 1000, 2, &ok, this );
if ( ok )
    ;// user entered something and pressed OK
else
    ;// user pressed Cancel
```

QString QInputDialog::getItem (const QString & caption, const QString & label, const QStringList & list, int current = 0, bool editable = TRUE, bool * ok = 0, QWidget * parent = 0, const char * name = 0) [static]

Static convenience function to let the user select an item from a string list. *caption* is the text which is displayed in the title bar of the dialog. *label* is the text which is shown to the user (it should mention what they should input). *list* is the string list which is inserted into the combobox, and *current* is the number of the item which should be the current item. If *editable* is TRUE the user can enter their own text; if *editable* is FALSE the user may only select one of the existing items.

If *ok* is not-null it will be set to TRUE if the user pressed OK and FALSE if the user pressed Cancel. The dialog's parent is *parent*; the dialog is called *name*. The dialog will be modal.

This method returns the text of the current item, or if *editable* is TRUE, the current text of the combobox.

Use this static method like this:

```
QStringList lst;
lst << "First" << "Second" << "Third" << "Fourth" << "Fifth";
bool ok = FALSE;
QString res = QInputDialog::getItem(
                  tr( "Application name" ),
                  tr( "Please select an item" ), lst, 1, TRUE, &ok, this );
if ( ok )
    ;// user selected an item and pressed OK
else
    ;// user pressed Cancel
```

QString QInputDialog::getText (const QString & caption, const QString & label, QLineEdit::EchoMode mode = QLineEdit::Normal, const QString & text = QString::null, bool * ok = 0, QWidget * parent = 0, const char * name = 0) [static]

Static convenience function to get a string from the user. *caption* is the text which is displayed in the title bar of the dialog. *label* is the text which is shown to the user (it should mention what they should input), *text* the default text which is placed in the line edit. The *mode* is the echo mode the line edit will use. If *ok* is not-null it will be set to TRUE if the user pressed OK and FALSE if the user pressed Cancel. The dialog's parent is *parent*; the dialog is called *name*. The dialog will be modal.

This method returns the text which has been entered in the line edit.

Use this static method like this:

```
bool ok = FALSE;
QString text = QInputDialog::getText(
                  tr( "Application name" ),
                  tr( "Please enter your name" ),
                  QLineEdit::Normal, QString::null, &ok, this );
if ( ok && !text.isEmpty() )
    ;// user entered something and pressed OK
else
    ;// user entered nothing or pressed Cancel
```

Example: network/ftpclient/ftpmainwindow.cpp.

QMainWindow Class Reference

The QMainWindow class provides a main application window, with a menu bar, dock windows (e.g. for toolbars), and a status bar.

`#include <qmainwindow.h>`

Inherits QWidget [Widgets with Qt].

Public Members

- **QMainWindow** (QWidget * parent = 0, const char * name = 0, WFlags f = WType_TopLevel)
- **~QMainWindow** ()
- QMenuBar * **menuBar** () const
- QStatusBar * **statusBar** () const
- QToolTipGroup * **toolTipGroup** () const
- virtual void **setCentralWidget** (QWidget * w)
- QWidget * **centralWidget** () const
- virtual void **setDockEnabled** (Dock dock, bool enable)
- bool **isDockEnabled** (Dock dock) const
- bool **isDockEnabled** (QDockArea * area) const
- virtual void **setDockEnabled** (QDockWindow * dw, Dock dock, bool enable)
- bool **isDockEnabled** (QDockWindow * tb, Dock dock) const
- bool **isDockEnabled** (QDockWindow * dw, QDockArea * area) const
- virtual void **addDockWindow** (QDockWindow * dockWindow, Dock edge = DockTop, bool newLine = FALSE)
- virtual void **addDockWindow** (QDockWindow * dockWindow, const QString & label, Dock edge = DockTop, bool newLine = FALSE)
- virtual void **moveDockWindow** (QDockWindow * dockWindow, Dock edge = DockTop)
- virtual void **moveDockWindow** (QDockWindow * dockWindow, Dock edge, bool nl, int index, int extraOffset = -1)
- virtual void **removeDockWindow** (QDockWindow * dockWindow)
- bool **rightJustification** () const
- bool **usesBigPixmaps** () const
- bool **usesTextLabel** () const
- bool **dockWindowsMovable** () const
- bool **opaqueMoving** () const
- bool **getLocation** (QDockWindow * dw, Dock & dock, int & index, bool & nl, int & extraOffset) const
- QPtrList<QDockWindow> **dockWindows** (Dock dock) const
- QPtrList<QDockWindow> **dockWindows** () const
- void **lineUpDockWindows** (bool keepNewLines = FALSE)
- bool **isDockMenuEnabled** () const

- bool **hasDockWindow** (QDockWindow * dw)
- void addToolBar (QDockWindow *, Dock = DockTop, bool newLine = FALSE) *(obsolete)*
- void addToolBar (QDockWindow *, const QString & label, Dock = DockTop, bool newLine = FALSE) *(obsolete)*
- void moveToolBar (QDockWindow *, Dock = DockTop) *(obsolete)*
- void moveToolBar (QDockWindow *, Dock, bool nl, int index, int extraOffset = -1) *(obsolete)*
- void removeToolBar (QDockWindow *) *(obsolete)*
- bool toolBarsMovable () const *(obsolete)*
- QPtrList<QToolBar> **toolBars** (Dock dock) const
- void lineUpToolBars (bool keepNewLines = FALSE) *(obsolete)*
- QDockArea * **leftDock** () const
- QDockArea * **rightDock** () const
- QDockArea * **topDock** () const
- QDockArea * **bottomDock** () const
- virtual bool **isCustomizable** () const
- bool **appropriate** (QDockWindow * dw) const
- enum **DockWindows** { OnlyToolBars, NoToolBars, AllDockWindows }
- QPopupMenu * **createDockWindowMenu** (DockWindows dockWindows = AllDockWindows) const

Public Slots

- virtual void **setRightJustification** (bool)
- virtual void **setUsesBigPixmaps** (bool)
- virtual void **setUsesTextLabel** (bool)
- virtual void **setDockWindowsMovable** (bool)
- virtual void **setOpaqueMoving** (bool)
- virtual void **setDockMenuEnabled** (bool b)
- virtual void **whatsThis** ()
- virtual void **setAppropriate** (QDockWindow * dw, bool a)
- virtual void **customize** ()
- void setToolBarsMovable (bool) *(obsolete)*

Signals

- void **pixmapSizeChanged** (bool)
- void **usesTextLabelChanged** (bool)
- void **dockWindowPositionChanged** (QDockWindow * dockWindow)
- void toolBarPositionChanged (QToolBar *) *(obsolete)*

Properties

- bool **dockWindowsMovable** — whether the dock windows are movable
- bool **opaqueMoving** — whether dock windows are moved opaquely
- bool **rightJustification** — whether the main window right-justifies its dock windows
- bool **usesBigPixmaps** — whether big pixmaps are enabled
- bool **usesTextLabel** — whether text labels for toolbar buttons are enabled

Protected Members

- virtual void **childEvent** (QChildEvent * e)

Protected Slots

- virtual void **setUpLayout** ()
- virtual bool **showDockMenu** (const QPoint & globalPos)
- void **menuAboutToShow** ()

Related Functions

- QTextStream & **operator<<** (QTextStream & ts, const QMainWindow & mainWindow)
- QTextStream & **operator>>** (QTextStream & ts, QMainWindow & mainWindow)

Detailed Description

The QMainWindow class provides a main application window, with a menu bar, dock windows (e.g. for toolbars), and a status bar.

Main windows are most often used to provide menus, toolbars and a status bar around a large central widget, such as a text edit or drawing canvas. QMainWindow is usually subclassed since this makes it easier to encapsulate the central widget, menus and toolbars as well as the window's state. Subclassing makes it possible to create the slots that are called when the user clicks menu items or toolbar buttons. You can also create main windows using *Qt Designer*. We'll briefly review adding menu items and toolbar buttons then describe the facilities of QMainWindow itself.

```
QMainWindow *mw = new QMainWindow;
QTextEdit *edit = new QTextEdit( mw, "editor" );
edit->setFocus();
mw->setCaption( "Main Window" );
mw->setCentralWidget( edit );
mw->show();
```

QMainWindows may be created in their own right as shown above. The central widget is set with setCentralWidget(). Popup menus can be added to the default menu bar, widgets can be added to the status bar, toolbars and dock windows can be added to any of the dock areas.

```
ApplicationWindow * mw = new ApplicationWindow();
mw->setCaption( "Qt Example - Application" );
mw->show();
```

In the extract above ApplicationWindow is a subclass of QMainWindow that we must write for ourselves; this is the usual approach to using QMainWindow. (The source for the extracts in this description are taken from application/main.cpp, application/application.cpp, action/main.cpp, and action/application.cpp)

When subclassing we add the menu items and toolbars in the subclass's constructor. If we've created a QMainWindow instance directly we can add menu items and toolbars just as easily by passing the QMainWindow instance as the parent instead of the *this* pointer.

```
QPopupMenu * help = new QPopupMenu( this );
menuBar()->insertItem( "&Help", help );

help->insertItem( "&About", this, SLOT(about()), Key_F1 );
```

Here we've added a new menu with one menu item. The menu has been inserted into the menu bar that QMainWindow provides by default and which is accessible through the menuBar() function. The slot will be called when the menu item is clicked.

```
QToolBar * fileTools = new QToolBar( this, "file operations" );
fileTools->setLabel( "File Operations" );

QToolButton * fileOpen
    = new QToolButton( openIcon, "Open File", QString::null,
                       this, SLOT(choose()), fileTools, "open file" );
```

This extract shows the creation of a toolbar with one toolbar button. QMainWindow supplies four dock areas for toolbars. When a toolbar is created as a child of a QMainWindow (or derived class) instance it will be placed in a dock area (the Top dock area by default). The slot will be called when the toolbar button is clicked. Any dock window can be added to a dock area either using addDockWindow(), or by creating a dock window with the QMainWindow as the parent.

```
e = new QTextEdit( this, "editor" );
e->setFocus();
setCentralWidget( e );
statusBar()->message( "Ready", 2000 );
```

Having created the menus and toolbar we create an instance of the large central widget, give it the focus and set it as the main window's central widget. In the example we've also set the status bar, accessed via the statusBar() function, to an initial message which will be displayed for two seconds. Note that you can add additional widgets to the status bar, for example labels, to show further status information. See the QStatusBar documentation for details, particularly the addWidget() function.

Often we want to synchronize a toolbar button with a menu item. For example, if the user clicks a 'bold' toolbar button we want the 'bold' menu item to be checked. This synchronization can be achieved automatically by creating actions and adding the actions to the toolbar and menu.

```
QAction * fileOpenAction;

fileOpenAction = new QAction( "Open File", QPixmap( fileopen ), "&Open",
                              CTRL+Key_O, this, "open" );
connect( fileOpenAction, SIGNAL( activated() ) , this, SLOT( choose() ) );
```

Here we create an action with an icon which will be used in any menu and toolbar that the action is added to. We've also given the action a menu name, '&Open', and a keyboard shortcut. The connection that we have made will be used when the user clicks either the menu item *or* the toolbar button.

```
QPopupMenu * file = new QPopupMenu( this );
menuBar()->insertItem( "&File", file );

fileOpenAction->addTo( file );
```

The extract above shows the creation of a popup menu. We add the menu to the QMainWindow's menu bar and add our action.

```
QToolBar * fileTools = new QToolBar( this, "file operations" );
fileTools->setLabel( "File Operations" );
fileOpenAction->addTo( fileTools );
```

Here we create a new toolbar as a child of the QMainWindow and add our action to the toolbar.

We'll now explore the functionality offered by QMainWindow.

The main window will take care of the dock areas, and the geometry of the central widget, but all other aspects of the central widget are left to you. QMainWindow automatically detects the creation of a menu bar or status bar if you specify the QMainWindow as parent, or you can use the provided menuBar() and statusBar() functions. The functions menuBar() and statusBar() create a suitable widget if one doesn't exist, and update the window's layout to make space.

QMainWindow provides a QToolTipGroup connected to the status bar. The function toolTipGroup() provides access to the default QToolTipGroup. It isn't possible to set a different tool tip group.

New dock windows and toolbars can be added to a QMainWindow using addDockWindow(). Dock windows can be moved using moveDockWindow() and removed with removeDockWindow(). QMainWindow allows default dock window (toolbar) docking in all its dock areas (top, left, right, bottom). You can use setDockEnabled() to enable and disable docking areas for dock windows. When adding or moving dock windows you can specify their 'edge' (dock area). The currently available edges are: Top, Left, Right, Bottom, Minimized (effectively a 'hidden' dock area) and TornOff (floating). See Qt::Dock for an explanation of these areas. Note that the *ToolBar functions are included for backward compatibility, all new code should use the *DockWindow functions. QToolbar is a subclass of QDockWindow so all functions that work with dock windows work on toolbars in the same way. If the user minimizes a dock window by clicking the dock window's window handle then the dock window is moved to the Minimized dock area. If the user clicks the close button, then the dock window is hidden and can only be shown again by using the dock window menu.

Some functions change the appearance of a QMainWindow globally:

- QDockWindow::setHorizontalStretchable() and QDockWindow::setVerticalStretchable() are used to make specific dock windows or toolbars stretchable.
- setUsesBigPixmaps() is used to set whether tool buttons should draw small or large pixmaps (see QIconSet for more information).
- setUsesTextLabel() is used to set whether tool buttons should display a textual label in addition to pixmaps (see QToolButton for more information).

The user can drag dock windows into any enabled docking area. Dock windows can also be dragged *within* a docking area, for example to rearrange the order of some toolbars. Dock windows can also be dragged outside any docking area (undocked or 'floated'). Being able to drag dock windows can be enabled (the default) and disabled using setDockWindowsMovable(). If the user clicks the close button on a floating dock window then the dock window will disappear. To get the dock window back the user must right click a dock area, to pop up the dock window menu, then click the name of the dock window they want to restore. Visible dock windows have a tick by their name in the dock window menu. The dock window menu is created automatically as required by createDockWindowMenu(). Since it may not always be appropriate for a dock window to appear on this menu the setAppropriate() function is used to inform the main window whether or not the dock window menu should include a particular dock window. Double clicking a dock window handle (usually on the left-hand side of the dock window) undocks (floats) the dock window. Double clicking a floating dock window's titlebar will dock the floating dock window.

The Minimized edge is a hidden dock area. If this dock area is enabled the user can hide (minimize) a dock window or show (restore) a minimized dock window by clicking the dock window handle. If the user hovers the mouse cursor over one of the handles, the caption of the dock window is displayed in a tool tip (see QDockWindow::caption() or QToolBar::label()), so if you enable the Minimized dock area, it is best to specify a meaningful caption or label for each dock window. To minimize a dock window programmatically use moveDockWindow() with an edge of Minimized.

Dock windows are moved transparently by default, i.e. during the drag an outline rectangle is drawn on the screen representing the position of the dock window as it moves. If you want the dock window to be shown normally

whilst it is moved use setOpaqueMoving().

The location of a dock window, i.e. its dock area and position within the dock area, can be determined by calling getLocation(). Movable dock windows can be lined up to minimize wasted space with lineUpDockWindows(). Pointers to the dock areas are available from topDock(), leftDock(), rightDock() and bottomDock(). A customize menu item is added to the pop up dock window menu if isCustomizable() returns TRUE; it returns FALSE by default. Reimplement isCustomizable() and customize() if you want to offer this extra menu item, for example, to allow the user to change settings relating to the main window and its toolbars and dock windows.

The main window's menu bar is fixed (at the top) by default. If you want a movable menu bar, create a QMenuBar as a stretchable widget inside its own movable dock window and restrict this dock window to only live within the Top or Bottom dock:

```
QToolBar *tb = new QToolBar( this );
addDockWindow( tb, tr( "Menubar" ), Top, FALSE );
QMenuBar *mb = new QMenuBar( tb );
mb->setFrameStyle( QFrame::NoFrame );
tb->setStretchableWidget( mb );
setDockEnabled( tb, Left, FALSE );
setDockEnabled( tb, Right, FALSE );
```

An application with multiple dock windows can choose to save the current dock window layout in order to restore it later, e.g. in the next session. You can do this by using the streaming operators for QMainWindow.

To save the layout and positions of all the dock windows do this:

```
QFile f( filename );
if ( f.open( IO_WriteOnly ) ) {
    QTextStream ts( &f );
    ts << *mainWindow;
    f.close();
}
```

To restore the dock window positions and sizes (normally when the application is next started), do following:

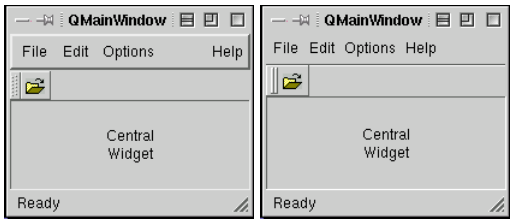
```
QFile f( filename );
if ( f.open( IO_ReadOnly ) ) {
    QTextStream ts( &f );
    ts >> *mainWindow;
    f.close();
}
```

The QSettings class can be used in conjunction with the streaming operators to store the application's settings.

QMainWindow's management of dock windows and toolbars is done transparently behind-the-scenes by QDockArea.

For multi-document interfaces (MDI), use a QWorkspace as the central widget.

Adding dock windows, e.g. toolbars, to QMainWindow's dock areas is straightforward. If the supplied dock areas are not sufficient for your application we suggest that you create a QWidget subclass and add your own dock areas (see QDockArea) to the subclass since QMainWindow provides functionality specific to the standard dock areas it provides.



See also QToolBar [p. 139], QDockWindow [p. 21], QStatusBar [Widgets with Qt], QAction [Events, Actions, Layouts and Styles with Qt], QMenuBar [p. 69], QPopupMenu [p. 106], QToolTipGroup [p. 156], QDialog [p. 10] and Main Window and Related Classes.

Member Type Documentation

QMainWindow::DockWindows

Right-clicking a dock area will pop-up the dock window menu (createDockWindowMenu() is called automatically). When called in code you can specify what items should appear on the menu with this enum.

- `QMainWindow::OnlyToolBars` - The menu will list all the toolbars, but not any other dock windows.
- `QMainWindow::NoToolBars` - The menu will list dock windows but not toolbars.
- `QMainWindow::AllDockWindows` - The menu will list all toolbars and other dock windows. (This is the default.)

Member Function Documentation

QMainWindow::QMainWindow (QWidget * parent = 0, const char * name = 0, WFlags f = WType_TopLevel)

Constructs an empty main window. The *parent*, *name* and widget flags *f*, are passed to the QWidget constructor.

By default, the widget flags are set to WType_TopLevel rather than 0 as it is with QWidget. If you don't want your QMainWindow to be a top level widget then you will need to set *f* to 0.

QMainWindow::~QMainWindow ()

Destroys the object and frees any allocated resources.

void QMainWindow::addDockWindow (QDockWindow * dockWindow, Dock edge = DockTop, bool newLine = FALSE) [virtual]

Adds *dockWindow* to the *edge* dock area.

If *newLine* is FALSE (the default) then the *dockWindow* is added at the end of the *edge*. For vertical edges the end is at the bottom, for horizontal edges (including Minimized) the end is at the right. If *newLine* is TRUE a new line of dock windows is started with *dockWindow* as the first (left-most and top-most) dock window.

If *dockWindow* is managed by another main window, it is first removed from that window.

void QMainWindow::addDockWindow (QDockWindow * dockWindow, const QString & label, Dock edge = DockTop, bool newLine = FALSE) [virtual]

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

Adds *dockWindow* to the dock area with label *label*.

If *newLine* is FALSE (the default) the *dockWindow* is added at the end of the *edge*. For vertical edges the end is at the bottom, for horizontal edges (including Minimized) the end is at the right. If *newLine* is TRUE a new line of dock windows is started with *dockWindow* as the first (left-most and top-most) dock window.

If *dockWindow* is managed by another main window, it is first removed from that window.

void QMainWindow::addToolBar (QDockWindow *, Dock = DockTop, bool newLine = FALSE)

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

void QMainWindow::addToolBar (QDockWindow *, const QString & label, Dock = DockTop, bool newLine = FALSE)

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

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

bool QMainWindow::appropriate (QDockWindow * dw) const

Returns TRUE if it is appropriate to include a menu item listing the *dw* dock window on the dock window menu. Otherwise returns FALSE.

The user is able to change the state (show or hide) a dock window that has a menu item by clicking the item.

Call setAppropriate() to indicate whether or not a particular dock window should appear on the popup menu.

See also setAppropriate() [p. 64].

QDockArea * QMainWindow::bottomDock () const

Returns a pointer the Bottom dock area

See also topDock() [p. 66], leftDock() [p. 62] and rightDock() [p. 64].

QWidget * QMainWindow::centralWidget () const

Returns a pointer to the main window's central widget.

The central widget is surrounded by the left, top, right and bottom dock areas. The menu bar is above the top dock area.

See also setCentralWidget() [p. 64].

Example: qfd/qfd.cpp.

void QMainWindow::childEvent (QChildEvent * e) [virtual protected]

Monitors events, recieved in *e*, to ensure the layout is updated.

Reimplemented from QObject [Additional Functionality with Qt].

QPopupMenu * QMainWindow::createDockWindowMenu (DockWindows dockWindows = AllDockWindows) const

Creates the dock window menu which contains all toolbars (if *dockWindows* is OnlyToolBars), all dock windows (if *dockWindows* is NoToolBars) or all toolbars and dock windows (if *dockWindows* is AllDockWindows - the default).

This function is called internally when necessary, e.g. when the user right clicks a dock area (providing isDockMenuEnabled() returns TRUE). You may reimplement this function if you wish to customize the behaviour.

The menu items representing the toolbars and dock windows are checkable. The visible dock windows are checked and the hidden dock windows are unchecked. The user can click a menu item to change its state (show or hide the dock window).

The list and the state are always kept up-to-date.

Toolbars and dock windows which are not appropriate in the current context (see setAppropriate()) are not listed in the menu.

The menu also has a menu item for lining up the dock windows.

If isCustomizable() returns TRUE, a Customize menu item is added to the menu, which if clicked will call customize(). The isCustomizable() function we provide returns FALSE and customize() does nothing, so they must be reimplemented in a subclass to be useful.

void QMainWindow::customize () [virtual slot]

This function is called when the user clicks the Customize menu item on the dock window menu.

The customize menu item will only appear if isCustomizable() returns TRUE (it returns FALSE by default).

The function is intended, for example, to provide the user a means of telling the application that they wish to customize the main window, dock windows or dock areas.

The default implementation does nothing, but this may change in later Qt versions. In view of this the Customize menu item is not shown on the right-click menu by default. If you want the item to appear then reimplement isCustomizable() to return TRUE.

See also isCustomizable() [p. 61].

void QMainWindow::dockWindowPositionChanged (QDockWindow * dockWindow) [signal]

This signal is emitted when the *dockWindow* has changed its position. A change in position occurs when a dock window is moved within its dock area or moved to another dock area (including the Minimized and `TearOff` dock areas).

See also getLocation() [p. 61].

QPtrList<QDockWindow> QMainWindow::dockWindows (Dock dock) const

Returns a list of all the dock windows which are in the *dock* dock area, regardless of their state.

For example, the TornOff dock area may contain closed dock windows but these are returned along with the visible dock windows.

QPtrList<QDockWindow> QMainWindow::dockWindows () const

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

Returns the list of dock windows which belong to this main window, regardless of which dock area they are in or what their state is, (e.g. irrespective of whether they are visible or not).

bool QMainWindow::dockWindowsMovable () const

Returns TRUE if the dock windows are movable; otherwise returns FALSE. See the "dockWindowsMovable" [p. 67] property for details.

bool QMainWindow::getLocation (QDockWindow * dw, Dock & dock, int & index, bool & nl, int & extraOffset) const

Finds the location of the dock window *dw*.

If the *dw* dock window is found in the main window the function returns TRUE and populates the *dock* variable with the dw's dock area and the *index* with the dw's position within the dock area. It also sets *nl* to TRUE if the *dw* begins a new line (otherwise FALSE), and *extraOffset* with the dw's offset.

If the *dw* dock window is not found then the function returns FALSE and the state of *dock*, *index*, *nl* and *extraOffset* is undefined.

If you want to save and restore dock window positions then use operator>>() and operator<<().

See also operator>>() [p. 68] and operator<<() [p. 68].

bool QMainWindow::hasDockWindow (QDockWindow * dw)

Returns TRUE if *dw* is a dock window known to the main window, otherwise returns FALSE.

bool QMainWindow::isCustomizable () const [virtual]

Returns TRUE if the dock area dock window menu includes the Customize menu item (which calls customize when clicked). Returns FALSE by default, i.e. the popup menu will not contain a Customize menu item. You will need to reimplement this function and set it to return TRUE if you wish the user to be able to see the dock window menu.

See also customize() [p. 60].

bool QMainWindow::isDockEnabled (Dock dock) const

Returns TRUE if the *dock* dock area is enabled, i.e. it can accept user dragged dock windows; otherwise returns FALSE.

See also setDockEnabled() [p. 64].

bool QMainWindow::isDockEnabled (QDockArea * area) const

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

Returns TRUE if *area* is enabled, i.e. it can accept user dragged dock windows; otherwise returns FALSE.

See also setDockEnabled() [p. 64].

bool QMainWindow::isDockEnabled (QDockWindow * tb, Dock dock) const

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

Returns TRUE if *dock* is enabled for the dock window *tb*, otherwise returns FALSE.

See also setDockEnabled() [p. 64].

bool QMainWindow::isDockEnabled (QDockWindow * dw, QDockArea * area) const

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

Returns TRUE if *area* is enabled for the dock window *dw*, otherwise returns FALSE.

See also setDockEnabled() [p. 64].

bool QMainWindow::isDockMenuEnabled () const

Returns TRUE, if the dock window menu is enabled; otherwise returns FALSE.

The menu lists the (appropriate()) dock windows (which may be shown or hidden), and has a "Line Up Dock Windows" menu item. It will also have a "Customize" menu item if isCustomizable() returns TRUE.

See also setDockEnabled() [p. 64], lineUpDockWindows() [p. 62], appropriate() [p. 59] and setAppropriate() [p. 64].

QDockArea * QMainWindow::leftDock () const

Returns the Left dock area

See also rightDock() [p. 64], topDock() [p. 66] and bottomDock() [p. 59].

void QMainWindow::lineUpDockWindows (bool keepNewLines = FALSE)

This function will line up dock windows within the visible dock areas (Top, Left, Right and Bottom) as compactly as possible.

If *keepNewLines* is TRUE, all dock windows stay on their original lines. If *keepNewLines* is FALSE then newlines may be removed to achieve the most compact layout possible.

The method only works if dockWindowsMovable() returns TRUE.

void QMainWindow::lineUpToolBars (bool keepNewLines = FALSE)

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

void QMainWindow::menuAboutToShow () [protected slot]

This slot is called from the aboutToShow() signal of the default dock menu of the mainwindow. The default implementation initializes the menu with all dock windows and toolbars in this slot.

If you want to do small adjustments to the menu, you can do it in this slot. Else reimplement createDockWindowMenu().

QMenuBar * QMainWindow::menuBar () const

Returns the menu bar for this window.

If there isn't one, then menuBar() creates an empty menu bar.

See also statusBar() [p. 66].

void QMainWindow::moveDockWindow (QDockWindow * dockWindow, Dock edge = DockTop) [virtual]

Moves *dockWindow* to the end of the *edge*.

For vertical edges the end is at the bottom, for horizontal edges (including Minimized) the end is at the right.

If *dockWindow* is managed by another main window, it is first removed from that window.

void QMainWindow::moveDockWindow (QDockWindow * dockWindow, Dock edge, bool nl, int index, int extraOffset = -1) [virtual]

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

Moves *dockWindow* to position *index* of *edge*.

Any dock windows with positions *index* or higher have their position number incremented and any of these on the same line are moved right (down for vertical dock areas) to make room.

If *nl* is TRUE, a new dock window line is created below the line in which the moved dock window appears and the moved dock window, with any others with higher positions on the same line, is moved to this new line.

The *extraOffset* is the space to put between the left side of the dock area (top side for vertical dock areas) and the dock window. (This is mostly used for restoring dock windows to the positions the user has dragged them to.)

If *dockWindow* is managed by another main window, it is first removed from that window.

void QMainWindow::moveToolBar (QDockWindow *, Dock = DockTop)

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

void QMainWindow::moveToolBar (QDockWindow *, Dock, bool nl, int index, int extraOffset = -1)

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

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

bool QMainWindow::opaqueMoving () const

Returns TRUE if dock windows are moved opaquely; otherwise returns FALSE. See the "opaqueMoving" [p. 67] property for details.

void QMainWindow::pixmapSizeChanged (bool) [signal]

This signal is called whenever the setUsesBigPixmaps() is called with a value different to the current setting. All widgets that should respond to such changes, e.g. toolbar buttons, must connect to this signal.

void QMainWindow::removeDockWindow (QDockWindow * dockWindow) [virtual]

Removes *dockWindow* from the main window's docking area, provided *dockWindow* is non-null and managed by this main window.

void QMainWindow::removeToolBar (QDockWindow *)

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

QDockArea * QMainWindow::rightDock () const

Returns the Right dock area

See also leftDock() [p. 62], topDock() [p. 66] and bottomDock() [p. 59].

bool QMainWindow::rightJustification () const

Returns TRUE if the main window right-justifies its dock windows; otherwise returns FALSE. See the "rightJustification" [p. 67] property for details.

void QMainWindow::setAppropriate (QDockWindow * dw, bool a) [virtual slot]

Use this function to control whether or not the *dw* dock window's caption should appear as a menu item on the dock window menu that lists the dock windows.

If *a* is TRUE then the *dw* will appear as a menu item on the dock window menu. The user is able to change the state (show or hide) a dock window that has a menu item by clicking the item; depending on the state of your application, this may or may not be appropriate. If *a* is FALSE the *dw* will not appear on the popup menu.

See also showDockMenu() [p. 65], isCustomizable() [p. 61] and customize() [p. 60].

void QMainWindow::setCentralWidget (QWidget * w) [virtual]

Sets the central widget for this window to *w*.

The central widget is surrounded by the left, top, right and bottom dock areas. The menu bar is above the top dock area.

See also centralWidget() [p. 59].

void QMainWindow::setDockEnabled (Dock dock, bool enable) [virtual]

If *enable* is TRUE then users can dock windows in the *dock* area. If *enable* is FALSE users cannot dock windows in the *dock* area.

Users can dock (drag) dock windows into any enabled dock area.

void QMainWindow::setDockEnabled (QDockWindow * dw, Dock dock, bool enable) [virtual]

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

If *enable* is TRUE then users can dock the *dw* dock window in the *dock* area. If *enable* is FALSE users cannot dock the *dw* dock window in the *dock* area.

In general users can dock (drag) dock windows into any enabled dock area. Using this function particular dock areas can be enabled (or disabled) as docking points for particular dock windows.

void QMainWindow::setDockMenuEnabled (bool b) [virtual slot]

If *b* is TRUE then right clicking on a dock window or dock area will pop up the dock window menu. If *b* is FALSE right clicking a dock window or dock area will not pop up the menu.

The menu lists the (appropriate()) dock windows (which may be shown or hidden), and has a line up dock window item. It will also have a Customize menu item if isCustomizable() returns TRUE.

See also lineUpDockWindows() [p. 62] and isDockMenuEnabled() [p. 62].

void QMainWindow::setDockWindowsMovable (bool) [virtual slot]

Sets whether the dock windows are movable. See the "dockWindowsMovable" [p. 67] property for details.

void QMainWindow::setOpaqueMoving (bool) [virtual slot]

Sets whether dock windows are moved opaquely. See the "opaqueMoving" [p. 67] property for details.

void QMainWindow::setRightJustification (bool) [virtual slot]

Sets whether the main window right-justifies its dock windows. See the "rightJustification" [p. 67] property for details.

void QMainWindow::setToolBarsMovable (bool) [slot]

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

void QMainWindow::setUpLayout () [virtual protected slot]

Sets up the geometry management of the window. It is called automatically when needed, so you shouldn't need to call it.

void QMainWindow::setUsesBigPixmaps (bool) [virtual slot]

Sets whether big pixmaps are enabled. See the "usesBigPixmaps" [p. 68] property for details.

void QMainWindow::setUsesTextLabel (bool) [virtual slot]

Sets whether text labels for toolbar buttons are enabled. See the "usesTextLabel" [p. 68] property for details.

bool QMainWindow::showDockMenu (const QPoint & globalPos) [virtual protected slot]

Shows the dock menu at the position *globalPos*. The menu lists the dock windows so that they can be shown (or hidden), lined up, and possibly customized.

The default implementation uses the dock window menu which gets created by createDockWindowMenu(). You can reimplement createDockWindowMenu() if you want to use your own specialized popup menu.

QStatusBar * QMainWindow::statusBar () const

Returns the status bar for this window. If there isn't one, statusBar() creates an empty status bar, and if necessary a tool tip group too.

See also menuBar() [p. 62] and toolTipGroup() [p. 66].

Example: qfd/qfd.cpp.

void QMainWindow::toolBarPositionChanged (QToolBar *) [signal]

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

QPtrList<QToolBar> QMainWindow::toolBars (Dock dock) const

Returns a list of all the toolbars which are in the *dock* dock area, regardless of their state.

For example, the TornOff dock area may contain closed toolbars but these are returned along with the visible toolbars.

See also dockWindows() [p. 60].

bool QMainWindow::toolBarsMovable () const

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

QToolTipGroup * QMainWindow::toolTipGroup () const

Returns the tool tip group for this window. If there isn't one, toolTipGroup() creates an empty tool tip groups.

See also menuBar() [p. 62] and statusBar() [p. 66].

QDockArea * QMainWindow::topDock () const

Returns the Top dock area

See also bottomDock() [p. 59], leftDock() [p. 62] and rightDock() [p. 64].

bool QMainWindow::usesBigPixmaps () const

Returns TRUE if big pixmaps are enabled; otherwise returns FALSE. See the "usesBigPixmaps" [p. 68] property for details.

bool QMainWindow::usesTextLabel () const

Returns TRUE if text labels for toolbar buttons are enabled; otherwise returns FALSE. See the "usesTextLabel" [p. 68] property for details.

void QMainWindow::usesTextLabelChanged (bool) [signal]

This signal is called whenever the setUsesTextLabel() is called with a value different to the current setting. All widgets that should respond to such changes, e.g. toolbar buttons, must connect to this signal.

void QMainWindow::whatsThis () [virtual slot]

Enters 'What's This?' question mode and returns immediately.

This is the same as QWhatsThis::enterWhatsThisMode(), but implemented as a main window object's slot. This way it can easily be used for popup menus, for example:

```
QPopupMenu * help = new QPopupMenu( this );
help->insertItem( "What's &This", this , SLOT(whatsThis()), SHIFT+Key_F1);
```

See also QWhatsThis::enterWhatsThisMode() [Widgets with Qt].

Property Documentation

bool dockWindowsMovable

This property holds whether the dock windows are movable.

If TRUE (the default), the user will be able to move Movable dock windows from one QMainWindow dock area to another, including the `TearOff` area (i.e. where the dock window floats freely as a window in its own right), and the Minimized area (where only the dock window's handle is shown below the menu bar). Moveable dock windows can also be moved within QMainWindow dock areas, i.e. to rearrange them within a dock area.

If FALSE the user will not be able to move any dock windows.

By default dock windows are moved transparently (i.e. only an outline rectangle is shown during the drag), but this setting can be changed with setOpaqueMoving().

See also setDockEnabled() [p. 64] and opaqueMoving [p. 67].

Set this property's value with setDockWindowsMovable() and get this property's value with dockWindowsMovable().

bool opaqueMoving

This property holds whether dock windows are moved opaquely.

If TRUE the dock windows of the main window are shown opaquely (i.e. it shows the toolbar as it looks when docked) when moved. If FALSE (the default) they are shown transparently, (i.e. as an outline rectangle).

Set this property's value with setOpaqueMoving() and get this property's value with opaqueMoving().

bool rightJustification

This property holds whether the main window right-justifies its dock windows.

If disabled (the default), stretchable dock windows are expanded, and non-stretchable dock windows are given the minimum space they need. Since most dock windows are not stretchable, this usually results in a unjustified right edge (or unjustified bottom edge for a vertical dock area). If enabled, the main window will right-justify its dock windows.

See also QDockWindow::setVerticalStretchable() [p. 27] and QDockWindow::setHorizontalStretchable() [p. 27].

Set this property's value with setRightJustification() and get this property's value with rightJustification().

bool usesBigPixmaps

This property holds whether big pixmaps are enabled.

If FALSE (the default), the tool buttons will use small pixmaps; otherwise big pixmaps will be used.

Tool buttons and other widgets that wish to respond to this setting are responsible for reading the correct state on startup, and for connecting to the main window's widget's pixmapSizeChanged() signal.

Set this property's value with setUsesBigPixmaps() and get this property's value with usesBigPixmaps().

bool usesTextLabel

This property holds whether text labels for toolbar buttons are enabled.

If disabled (the default), the tool buttons will not use text labels. If enabled, text labels will be used.

Tool buttons and other widgets that wish to respond to this setting are responsible for reading the correct state on startup, and for connecting to the main window's widget's usesTextLabelChanged() signal.

See also QToolButton::usesTextLabel [p. 149].

Set this property's value with setUsesTextLabel() and get this property's value with usesTextLabel().

Related Functions

QTextStream & operator<< (QTextStream & ts, const QMainWindow & mainWindow)

Writes the layout (sizes and positions) of the dock windows in the dock areas of the QMainWindow *mainWindow*, including Minimized and TornOff dock windows, to the text stream *ts*.

This can be used, for example, in conjunction with QSettings to save the user's layout.

See also operator>>() [p. 68].

QTextStream & operator>> (QTextStream & ts, QMainWindow & mainWindow)

Reads the layout (sizes and positions) of the dock windows in the dock areas of the QMainWindow *mainWindow* from the text stream, *ts*, including Minimized and TornOff dock windows. Restores the dock windows and dock areas to these sizes and positions. The layout information must be in the format produced by operator<<().

This can be used, for example, in conjunction with QSettings to restore the user's layout.

See also operator<<() [p. 68].

QMenuBar Class Reference

The QMenuBar class provides a horizontal menu bar.

`#include <qmenubar.h>`

Inherits QFrame [Widgets with Qt] and QMenuData [p. 80].

Public Members

- **QMenuBar** (QWidget * parent = 0, const char * name = 0)
- **~QMenuBar** ()
- virtual void **show** ()
- virtual void **hide** ()
- virtual int **heightForWidth** (int max_width) const
- enum **Separator** { Never = 0, InWindowsStyle = 1 }
- Separator separator () const *(obsolete)*
- virtual void setSeparator (Separator when) *(obsolete)*
- void **setDefaultUp** (bool)
- bool **isDefaultUp** () const

Signals

- void **activated** (int id)
- void **highlighted** (int id)

Important Inherited Members

- int **insertItem** (const QString & text, const QObject * receiver, const char * member, const QKeySequence & accel = 0, int id = -1, int index = -1)
- int **insertItem** (const QIconSet & icon, const QString & text, const QObject * receiver, const char * member, const QKeySequence & accel = 0, int id = -1, int index = -1)
- int **insertItem** (const QPixmap & pixmap, const QObject * receiver, const char * member, const QKeySequence & accel = 0, int id = -1, int index = -1)
- int **insertItem** (const QIconSet & icon, const QPixmap & pixmap, const QObject * receiver, const char * member, const QKeySequence & accel = 0, int id = -1, int index = -1)
- int **insertItem** (const QString & text, int id = -1, int index = -1)
- int **insertItem** (const QIconSet & icon, const QString & text, int id = -1, int index = -1)
- int **insertItem** (const QString & text, QPopupMenu * popup, int id = -1, int index = -1)
- int **insertItem** (const QIconSet & icon, const QString & text, QPopupMenu * popup, int id = -1, int index = -1)

- int **insertItem** (const QPixmap & pixmap, int id = -1, int index = -1)
- int **insertItem** (const QIconSet & icon, const QPixmap & pixmap, int id = -1, int index = -1)
- int **insertItem** (const QPixmap & pixmap, QPopupMenu * popup, int id = -1, int index = -1)
- int **insertItem** (const QIconSet & icon, const QPixmap & pixmap, QPopupMenu * popup, int id = -1, int index = -1)
- int **insertItem** (QWidget * widget, int id = -1, int index = -1)
- int **insertItem** (const QIconSet & icon, QCustomMenuItem * custom, int id = -1, int index = -1)
- int **insertItem** (QCustomMenuItem * custom, int id = -1, int index = -1)
- int **insertSeparator** (int index = -1)
- void **removeItem** (int id)
- void **clear** ()
- bool **isItemEnabled** (int id) const
- void **setItemEnabled** (int id, bool enable)

Properties

- bool **defaultUp** — the popup orientation
- Separator separator — in which cases a menubar sparator is drawn *(obsolete)*

Protected Members

- virtual void **drawContents** (QPainter * p)
- virtual void **menuContentsChanged** ()
- virtual void **menuStateChanged** ()

Detailed Description

The QMenuBar class provides a horizontal menu bar.

A menu bar consists of a list of pull-down menu items. You add menu items with insertItem(). For example, asuming that `menubar` is a pointer to a QMenuBar and `filemenu` is a pointer to a QPopupMenu, the following statement inserts the menu into the menu bar:

```
menubar->insertItem( "&File", filemenu );
```

The ampersand in the menu item's text sets Alt+F as a shortcut for this menu. (You can use "&&" to get a real ampersand in the menu bar.)

Items are either enabled or disabled. You toggle their state with setItemEnabled().

There is no need to lay out a menu bar. It automatically sets its own geometry to the top of the parent widget and changes it appropriately whenever the parent is resized.

Example of creating a menu bar with menu items (from menu/menu.cpp):

```
QPopupMenu *file = new QPopupMenu( this );

file->insertItem( p1, "&Open",  this, SLOT(open()), CTRL+Key_O );
file->insertItem( p2, "&New", this, SLOT(news()), CTRL+Key_N );

menu = new QMenuBar( this );
```

```
menu->insertItem( "&File", file );
```

In most main window style applications you would use the menuBar() provided in QMainWindow, adding QPopupMenus to the menu bar and adding QActions to the popup menus.

Example (from action/application.cpp):

```
QPopupMenu * file = new QPopupMenu( this );
menuBar()->insertItem( "&File", file );
fileNewAction->addTo( file );
```

Menu items can have text and pixmaps (or iconsets), see the various insertItem() overloads, as well as separators, see insertSeparator(). You can also add custom menu items that are derived from QCustomMenuItem.

Menu items may be removed with removeItem() and enabled or disabled with setItemEnabled().

QMenuBar on Qt/Mac is a wrapper for using the system-wide menubar. However if you have multiple menubars in one dialog the outermost menubar (normally inside a widget with WType_TopLevel) will be used for the global menubar.

See also QPopupMenu [p. 106], QAccel [Events, Actions, Layouts and Styles with Qt], QAction [Events, Actions, Layouts and Styles with Qt], GUI Design Handbook: Menu Bar and Main Window and Related Classes.

Member Type Documentation

QMenuBar::Separator

This enum type is used to decide whether QMenuBar should draw a separator line at its bottom. The possible values are:

- `QMenuBar::Never` - In many applications there is already a separator, and having two looks wrong.
- `QMenuBar::InWindowsStyle` - In some other applications a separator looks good in Windows style, but nowhere else.

Member Function Documentation

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

Constructs a menu bar with a *parent* and a *name*.

QMenuBar::~QMenuBar ()

Destroys the menu bar.

void QMenuBar::activated (int id) [signal]

This signal is emitted when a menu item is selected; *id* is the id of the selected item.

Normally you will connect each menu item to a single slot using QMenuData::insertItem(), but sometimes you will want to connect several items to a single slot (most often if the user selects from an array). This signal is useful in such cases.

See also highlighted() [p. 72] and QMenuData::insertItem() [p. 84].

Example: progress/progress.cpp.

void QMenuData::clear ()

Removes all menu items.

See also removeItem() [p. 90] and removeItemAt() [p. 90].

Examples: mdi/application.cpp and qwerty/qwerty.cpp.

void QMenuBar::drawContents (QPainter * p) [virtual protected]

Called from QFrame::paintEvent(). Draws the menu bar contents using painter *p*.

Reimplemented from QFrame [Widgets with Qt].

int QMenuBar::heightForWidth (int max_width) const [virtual]

Returns the height that the menu would resize itself to if its parent (and hence itself) resized to the given *max_width*. This can be useful for simple layout tasks in which the height of the menu bar is needed after items have been inserted. See showimg/showimg.cpp for an example of the usage.

Example: showimg/showimg.cpp.

Reimplemented from QWidget [Widgets with Qt].

void QMenuBar::hide () [virtual]

Reimplements QWidget::hide() in order to deselect any selected item, and calls setUpLayout() for the main window.

Example: grapher/grapher.cpp.

Reimplemented from QWidget [Widgets with Qt].

void QMenuBar::highlighted (int id) [signal]

This signal is emitted when a menu item is highlighted; *id* is the id of the highlighted item.

Normally, you will connect each menu item to a single slot using QMenuData::insertItem(), but sometimes you will want to connect several items to a single slot (most often if the user selects from an array). This signal is useful in such cases.

See also activated() [p. 71] and QMenuData::insertItem() [p. 84].

int QMenuData::insertItem (const QString & text, const QObject * receiver, const char * member, const QKeySequence & accel = 0, int id = -1, int index = -1)

The family of insertItem() functions inserts menu items into a popup menu or a menu bar.

A menu item is usually either a text string or a pixmap, both with an optional icon or keyboard accelerator. For special cases it is also possible to insert custom items (see QCustomMenuItem) or even widgets into popup menus.

Some insertItem() members take a popup menu as an additional argument. Use this to insert submenus to existing menus or pulldown menus to a menu bar.

The number of insert functions may look confusing, but they are actually quite simple to use.

This default version inserts a menu item with the text *text*, the accelerator key *accel*, an id and an optional index and connects it to the slot *member* in the object *receiver*.

Example:

```
QMenuBar   *mainMenu = new QMenuBar;
QPopupMenu *fileMenu = new QPopupMenu;
fileMenu->insertItem( "New",  myView, SLOT(newFile()), CTRL+Key_N );
fileMenu->insertItem( "Open", myView, SLOT(open()),    CTRL+Key_O );
mainMenu->insertItem( "File", fileMenu );
```

Not all insert functions take an object/slot parameter or an accelerator key. Use connectItem() and setAccel() on these items.

If you need to translate accelerators, use tr() with a string description that use pass to the QKeySequence constructor:

```
fileMenu->insertItem( tr("Open"), myView, SLOT(open()),
                      tr("Ctrl+O") );
```

In the example above, pressing Ctrl+N or selecting "Open" from the menu activates the myView->open() function.

Some insert functions take a QIconSet parameter to specify the little menu item icon. Note that you can always pass a QPixmap object instead.

The *index* specifies the position in the menu. The menu item is appended at the end of the list if *index* is negative.

Note that keyboard accelerators in Qt are not application-global, instead they are bound to a certain top-level window. For example, accelerators in QPopupMenu items only work for menus that are associated with a certain window. This is true for popup menus that live in a menu bar since their accelerators will then be installed in the menu bar itself. This also applies to stand-alone popup menus that have a top-level widget in their parentWidget() chain. The menu will then install its accelerator object on that top-level widget. For all other cases use an independent QAccel object.

Warning: Be careful when passing a literal 0 to insertItem() because some C++ compilers choose the wrong overloaded function. Cast the 0 to what you mean, e.g. `(QObject*)0`.

Returns the allocated menu identifier number (*id* if *id* >= 0).

See also removeItem() [p. 90], changeItem() [p. 82], setAccel() [p. 90], connectItem() [p. 83], QAccel [Events, Actions, Layouts and Styles with Qt] and qnamespace.h.

Examples: addressbook/mainwindow.cpp, mdi/application.cpp, menu/menu.cpp, qwerty/qwerty.cpp, scrollview/scrollview.cpp and showimg/showimg.cpp.

int QMenuData::insertItem (const QIconSet & icon, const QString & text, const QObject * receiver, const char * member, const QKeySequence & accel = 0, int id = -1, int index = -1)

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

Inserts a menu item with icon *icon*, text *text*, accelerator *accel*, optional id *id*, and optional *index*. The menu item is connected it to the *receiver*'s *member* slot. The icon will be displayed to the left of the text in the item.

Returns the allocated menu identifier number (*id* if *id* >= 0).

See also removeItem() [p. 90], changeItem() [p. 82], setAccel() [p. 90], connectItem() [p. 83], QAccel [Events, Actions, Layouts and Styles with Qt] and qnamespace.h.

int QMenuData::insertItem (const QPixmap & pixmap, const QObject * receiver, const char * member, const QKeySequence & accel = 0, int id = -1, int index = -1)

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

Inserts a menu item with pixmap *pixmap*, accelerator *accel*, optional id *id*, and optional *index*. The menu item is connected it to the *receiver*'s *member* slot. The icon will be displayed to the left of the text in the item.

To look best when being highlighted as a menu item, the pixmap should provide a mask (see QPixmap::mask()).

Returns the allocated menu identifier number (*id* if *id* >= 0).

See also removeItem() [p. 90], changeItem() [p. 82], setAccel() [p. 90] and connectItem() [p. 83].

int QMenuData::insertItem (const QIconSet & icon, const QPixmap & pixmap, const QObject * receiver, const char * member, const QKeySequence & accel = 0, int id = -1, int index = -1)

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

Inserts a menu item with icon *icon*, pixmap *pixmap*, accelerator *accel*, optional id *id*, and optional *index*. The icon will be displayed to the left of the pixmap in the item. The item is connected to the *member* slot in the *receiver* object.

To look best when being highlighted as a menu item, the pixmap should provide a mask (see QPixmap::mask()).

Returns the allocated menu identifier number (*id* if *id* >= 0).

See also removeItem() [p. 90], changeItem() [p. 82], setAccel() [p. 90], connectItem() [p. 83], QAccel [Events, Actions, Layouts and Styles with Qt] and qnamespace.h.

int QMenuData::insertItem (const QString & text, int id = -1, int index = -1)

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

Inserts a menu item with text *text*, optional id *id*, and optional *index*.

Returns the allocated menu identifier number (*id* if *id* >= 0).

See also removeItem() [p. 90], changeItem() [p. 82], setAccel() [p. 90] and connectItem() [p. 83].

int QMenuData::insertItem (const QIconSet & icon, const QString & text, int id = -1, int index = -1)

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

Inserts a menu item with icon *icon*, text *text*, optional id *id*, and optional *index*. The icon will be displayed to the left of the text in the item.

Returns the allocated menu identifier number (*id* if *id* >= 0).

See also removeItem() [p. 90], changeItem() [p. 82], setAccel() [p. 90] and connectItem() [p. 83].

int QMenuData::insertItem (const QString & text, QPopupMenu * popup, int id = -1, int index = -1)

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

Inserts a menu item with text *text*, submenu *popup*, optional id *id*, and optional *index*.

The *popup* must be deleted by the programmer or by its parent widget. It is not deleted when this menu item is removed or when the menu is deleted.

Returns the allocated menu identifier number (*id* if *id* >= 0).

See also removeItem() [p. 90], changeItem() [p. 82], setAccel() [p. 90] and connectItem() [p. 83].

int QMenuData::insertItem (const QIconSet & icon, const QString & text, QPopupMenu * popup, int id = -1, int index = -1)

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

Inserts a menu item with icon *icon*, text *text*, submenu *popup*, optional id *id*, and optional *index*. The icon will be displayed to the left of the text in the item.

The *popup* must be deleted by the programmer or by its parent widget. It is not deleted when this menu item is removed or when the menu is deleted.

Returns the allocated menu identifier number (*id* if *id* >= 0).

See also removeItem() [p. 90], changeItem() [p. 82], setAccel() [p. 90] and connectItem() [p. 83].

int QMenuData::insertItem (const QPixmap & pixmap, int id = -1, int index = -1)

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

Inserts a menu item with pixmap *pixmap*, optional id *id*, and optional *index*.

To look best when being highlighted as a menu item, the pixmap should provide a mask (see QPixmap::mask()).

Returns the allocated menu identifier number (*id* if *id* >= 0).

See also removeItem() [p. 90], changeItem() [p. 82], setAccel() [p. 90] and connectItem() [p. 83].

int QMenuData::insertItem (const QIconSet & icon, const QPixmap & pixmap, int id = -1, int index = -1)

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

Inserts a menu item with icon *icon*, pixmap *pixmap*, optional id *id*, and optional *index*. The icon will be displayed to the left of the pixmap in the item.

Returns the allocated menu identifier number (*id* if *id* >= 0).

See also removeItem() [p. 90], changeItem() [p. 82], setAccel() [p. 90] and connectItem() [p. 83].

int QMenuData::insertItem (const QPixmap & pixmap, QPopupMenu * popup, int id = -1, int index = -1)

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

Inserts a menu item with pixmap *pixmap*, submenu *popup*, optional id *id*, and optional *index*.

The *popup* must be deleted by the programmer or by its parent widget. It is not deleted when this menu item is removed or when the menu is deleted.

Returns the allocated menu identifier number (*id* if *id* >= 0).

See also removeItem() [p. 90], changeItem() [p. 82], setAccel() [p. 90] and connectItem() [p. 83].

int QMenuData::insertItem (const QIconSet & icon, const QPixmap & pixmap, QPopupMenu * popup, int id = -1, int index = -1)

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

Inserts a menu item with icon *icon*, pixmap *pixmap* submenu *popup*, optional id *id*, and optional *index*. The icon will be displayed to the left of the pixmap in the item.

The *popup* must be deleted by the programmer or by its parent widget. It is not deleted when this menu item is removed or when the menu is deleted.

Returns the allocated menu identifier number (*id* if *id* >= 0).

See also removeItem() [p. 90], changeItem() [p. 82], setAccel() [p. 90] and connectItem() [p. 83].

int QMenuData::insertItem (QWidget * widget, int id = -1, int index = -1)

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

Inserts a menu item that consists of the widget *widget* with optional id *id*, and optional *index*.

Ownership of *widget* is transferred to the popup menu or to the menu bar.

Theoretically, any widget can be inserted into a popup menu. In practice, this only makes sense with certain widgets.

If a widget is not focus-enabled (see QWidget::isFocusEnabled()), the menu treats it as a separator; this means that the item is not selectable and will never get focus. In this way you can, for example, simply insert a QLabel if you need a popup menu with a title.

If the widget is focus-enabled it will get focus when the user traverses the popup menu with the arrow keys. If the widget does not accept ArrowUp and ArrowDown in its key event handler, the focus will move back to the menu when the respective arrow key is hit one more time. This works with a QLineEdit, for example. If the widget accepts the arrow key itself, it must also provide the possibility to put the focus back on the menu again by calling QWidget::focusNextPrevChild(). Futhermore, if the embedded widget closes the menu when the user made a selection, this can be done safely by calling

```
if ( isVisible() &&
     parentWidget() &&
     parentWidget()->inherits("QPopupMenu") )
    parentWidget()->close();
```

Returns the allocated menu identifier number (*id* if *id* >= 0).

See also removeItem() [p. 90].

int QMenuData::insertItem (const QIconSet & icon, QCustomMenuItem * custom, int id = -1, int index = -1)

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

Inserts a custom menu item *custom* with an *icon* and with optional id *id*, and optional *index*.

This only works with popup menus. It is not supported for menu bars. Ownership of *custom* is transferred to the popup menu.

If you want to connect a custom item to a certain slot, use connectItem().

Returns the allocated menu identifier number (*id* if *id* >= 0).

See also connectItem() [p. 83], removeItem() [p. 90] and QCustomMenuItem [p. 5].

int QMenuData::insertItem (QCustomMenuItem * custom, int id = -1, int index = -1)

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

Inserts a custom menu item *custom* with optional id *id*, and optional *index*.

This only works with popup menus. It is not supported for menu bars. Ownership of *custom* is transferred to the popup menu.

If you want to connect a custom item to a certain slot, use connectItem().

Returns the allocated menu identifier number (*id* if *id* >= 0).

See also connectItem() [p. 83], removeItem() [p. 90] and QCustomMenuItem [p. 5].

int QMenuData::insertSeparator (int index = -1)

Inserts a separator at position *index*. The separator becomes the last menu item if *index* is negative.

In a popup menu a separator is rendered as a horizontal line. In a Motif menu bar a separator is spacing, so the rest of the items (normally just "Help") are drawn right-justified. In a Windows menu bar separators are ignored (to comply with the Windows style guidelines).

Examples: addressbook/mainwindow.cpp, mdi/application.cpp, menu/menu.cpp, progress/progress.cpp, qwerty/qwerty.cpp, scrollview/scrollview.cpp and showimg/showimg.cpp.

bool QMenuBar::isDefaultUp () const

Returns the popup orientation. See the "defaultUp" [p. 78] property for details.

bool QMenuData::isItemEnabled (int id) const

Returns TRUE if the item with identifier *id* is enabled; otherwise returns FALSE

See also setItemEnabled() [p. 91].

void QMenuBar::menuContentsChanged () [virtual protected]

Recomputes the menu bar's display data according to the new contents.

You should never need to call this; it is called automatically by QMenuData whenever it needs to be called.

Reimplemented from QMenuData [p. 89].

void QMenuBar::menuStateChanged () [virtual protected]

Recomputes the menu bar's display data according to the new state.

You should never need to call this; it is called automatically by QMenuData whenever it needs to be called.

Reimplemented from QMenuData [p. 90].

void QMenuData::removeItem (int id)

Removes the menu item that has the identifier *id*.

See also removeItemAt() [p. 90] and clear() [p. 83].

Separator QMenuBar::separator () const

Returns in which cases a menubar sparator is drawn. See the "separator" [p. 79] property for details.

void QMenuBar::setDefaultUp (bool)

Sets the popup orientation. See the "defaultUp" [p. 78] property for details.

void QMenuData::setItemEnabled (int id, bool enable)

If *enable* is TRUE, enables the menu item with identifier *id*; otherwise disables the menu item with identifier *id*.

See also isItemEnabled() [p. 89].

Examples: mdi/application.cpp, menu/menu.cpp, progress/progress.cpp and showimg/showimg.cpp.

void QMenuBar::setSeparator (Separator when) [virtual]

Sets in which cases a menubar sparator is drawn to *when*. See the "separator" [p. 79] property for details.

void QMenuBar::show () [virtual]

Reimplements QWidget::show() in order to set up the correct keyboard accelerators and to raise itself to the top of the widget stack.

Example: grapher/grapher.cpp.

Reimplemented from QWidget [Widgets with Qt].

Property Documentation

bool defaultUp

This property holds the popup orientation.

The default popup orientation. By default, menus pop "down" the screen. By setting the property to TRUE, the menu will pop "up". You might call this for menus that are *below* the document to which they refer.

If the menu would not fit on the screen, the other direction is used rather than the default.

Set this property's value with setDefaultUp() and get this property's value with isDefaultUp().

Separator separator

This property holds in which cases a menubar sparator is drawn.

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

Set this property's value with setSeparator() and get this property's value with separator().

QMenuData Class Reference

The QMenuData class is a base class for QMenuBar and QPopupMenu.

`#include <qmenudata.h>`

Inherited by QMenuBar [p. 69] and QPopupMenu [p. 106].

Public Members

- **QMenuData** ()
- virtual **~QMenuData** ()
- uint **count** () const
- int **insertItem** (const QString & text, const QObject * receiver, const char * member, const QKeySequence & accel = 0, int id = -1, int index = -1)
- int **insertItem** (const QIconSet & icon, const QString & text, const QObject * receiver, const char * member, const QKeySequence & accel = 0, int id = -1, int index = -1)
- int **insertItem** (const QPixmap & pixmap, const QObject * receiver, const char * member, const QKeySequence & accel = 0, int id = -1, int index = -1)
- int **insertItem** (const QIconSet & icon, const QPixmap & pixmap, const QObject * receiver, const char * member, const QKeySequence & accel = 0, int id = -1, int index = -1)
- int **insertItem** (const QString & text, int id = -1, int index = -1)
- int **insertItem** (const QIconSet & icon, const QString & text, int id = -1, int index = -1)
- int **insertItem** (const QString & text, QPopupMenu * popup, int id = -1, int index = -1)
- int **insertItem** (const QIconSet & icon, const QString & text, QPopupMenu * popup, int id = -1, int index = -1)
- int **insertItem** (const QPixmap & pixmap, int id = -1, int index = -1)
- int **insertItem** (const QIconSet & icon, const QPixmap & pixmap, int id = -1, int index = -1)
- int **insertItem** (const QPixmap & pixmap, QPopupMenu * popup, int id = -1, int index = -1)
- int **insertItem** (const QIconSet & icon, const QPixmap & pixmap, QPopupMenu * popup, int id = -1, int index = -1)
- int **insertItem** (QWidget * widget, int id = -1, int index = -1)
- int **insertItem** (const QIconSet & icon, QCustomMenuItem * custom, int id = -1, int index = -1)
- int **insertItem** (QCustomMenuItem * custom, int id = -1, int index = -1)
- int **insertSeparator** (int index = -1)
- void **removeItem** (int id)
- void **removeItemAt** (int index)
- void **clear** ()
- QKeySequence **accel** (int id) const
- void **setAccel** (const QKeySequence & key, int id)
- QIconSet * **iconSet** (int id) const
- QString **text** (int id) const
- QPixmap * **pixmap** (int id) const

- void **setWhatsThis** (int id, const QString & text)
- QString **whatsThis** (int id) const
- void **changeItem** (int id, const QString & text)
- void **changeItem** (int id, const QPixmap & pixmap)
- void **changeItem** (int id, const QIconSet & icon, const QString & text)
- void **changeItem** (int id, const QIconSet & icon, const QPixmap & pixmap)
- void changeItem (const QString & text, int id) *(obsolete)*
- void changeItem (const QPixmap & pixmap, int id) *(obsolete)*
- void changeItem (const QIconSet & icon, const QString & text, int id) *(obsolete)*
- bool **isItemActive** (int id) const
- bool **isItemEnabled** (int id) const
- void **setItemEnabled** (int id, bool enable)
- bool **isItemChecked** (int id) const
- void **setItemChecked** (int id, bool check)
- virtual void **updateItem** (int id)
- int **indexOf** (int id) const
- int **idAt** (int index) const
- virtual void **setId** (int index, int id)
- bool **connectItem** (int id, const QObject * receiver, const char * member)
- bool **disconnectItem** (int id, const QObject * receiver, const char * member)
- bool **setItemParameter** (int id, int param)
- int **itemParameter** (int id) const
- QMenuItem * **findItem** (int id) const
- QMenuItem * **findItem** (int id, QMenuData ** parent) const
- virtual void **activateItemAt** (int index)

Protected Members

- virtual void **menuContentsChanged** ()
- virtual void **menuStateChanged** ()
- virtual void **menuInsPopup** (QPopupMenu *)
- virtual void **menuDelPopup** (QPopupMenu *)

Detailed Description

The QMenuData class is a base class for QMenuBar and QPopupMenu.

QMenuData has an internal list of menu items. A menu item is a text, pixmap or separator, and may also have a popup menu (separators have no popup menus).

The menu item sends out an activated() signal when it is selected and a highlighted() signal when it receives the user input focus.

Menu items are assigned the menu identifier *id* that is passed in insertItem() or an automatically generated identifier if *id* is < 0 (the default). The generated identifiers (negative integers) are guaranteed to be unique within the entire application. The identifier is used to access the menu item in other functions.

Menu items can be removed with removeItem() or changed with changeItem(). Accelerators can be changed or set with setAccel(). Checkable items can be checked or unchecked with setItemChecked(). Items can be enabled or disabled using setItemEnabled() and connected and disconnected with connectItem() and disconnectItem() respectively.

Menu items are stored in a list. Use findItem() to find an item by its list position or by its menu identifier.

See also QAccel [Events, Actions, Layouts and Styles with Qt], QPopupMenu [p. 106], QAction [Events, Actions, Layouts and Styles with Qt] and Miscellaneous Classes.

Member Function Documentation

QMenuData::QMenuData ()

Constructs an empty menu data list.

QMenuData::~QMenuData () [virtual]

Removes all menu items and disconnects any signals that have been connected.

QKeySequence QMenuData::accel (int id) const

Returns the accelerator key that has been defined for the menu item *id*, or 0 if it has no accelerator key.

See also setAccel() [p. 90], QAccel [Events, Actions, Layouts and Styles with Qt] and qnamespace.h.

void QMenuData::activateItemAt (int index) [virtual]

Activates the menu item at position *index*.

If the index is invalid (for example, -1), the object itself is deactivated.

void QMenuData::changeItem (int id, const QString & text)

Changes the text of the menu item *id* to *text*. If the item has an icon, the icon remains unchanged.

See also text() [p. 92].

void QMenuData::changeItem (int id, const QPixmap & pixmap)

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

Changes the pixmap of the menu item *id* to the pixmap *pixmap*. If the item has an icon, the icon is unchanged.

See also pixmap() [p. 90].

void QMenuData::changeItem (int id, const QIconSet & icon, const QString & text)

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

Changes the iconset and text of the menu item *id* to the *icon* and *text* respectively.

See also pixmap() [p. 90].

void QMenuData::changeItem (int id, const QIconSet & icon, const QPixmap & pixmap)

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

Changes the iconset and pixmap of the menu item *id* to *icon* and *pixmap* respectively.

See also pixmap() [p. 90].

void QMenuData::changeItem (const QString & text, int id)

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

Changes the text of the menu item *id*. If the item has an icon, the icon remains unchanged.

See also text() [p. 92].

void QMenuData::changeItem (const QPixmap & pixmap, int id)

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

Changes the pixmap of the menu item *id*. If the item has an icon, the icon remains unchanged.

See also pixmap() [p. 90].

void QMenuData::changeItem (const QIconSet & icon, const QString & text, int id)

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

Changes the icon and text of the menu item *id*.

See also pixmap() [p. 90].

void QMenuData::clear ()

Removes all menu items.

See also removeItem() [p. 90] and removeItemAt() [p. 90].

Examples: mdi/application.cpp and qwerty/qwerty.cpp.

bool QMenuData::connectItem (int id, const QObject * receiver, const char * member)

Connects the menu item with identifier *id* to *receiver*'s *member* slot or signal.

The receiver's slot/signal is activated when the menu item is activated.

See also disconnectItem() [p. 84] and setItemParameter() [p. 91].

Example: menu/menu.cpp.

uint QMenuData::count () const

Returns the number of items in the menu.

bool QMenuData::disconnectItem (int id, const QObject * receiver, const char * member)

Disconnects the *receiver*'s *member* from the menu item with identifier *id*.

All connections are removed when the menu data object is destroyed.

See also connectItem() [p. 83] and setItemParameter() [p. 91].

QMenuItem * QMenuData::findItem (int id) const

Returns a pointer to the menu item with identifier *id*, or 0 if there is no item with this identifier.

See also indexOf() [p. 84].

QMenuItem * QMenuData::findItem (int id, QMenuData ** parent) const

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

Returns a pointer to the menu item with identifier *id*, or 0 if there is no item with this identifier. Changes **parent* to point to the parent of the return value.

See also indexOf() [p. 84].

QIconSet * QMenuData::iconSet (int id) const

Returns the icon set that has been set for menu item *id*, or 0 if no icon set has been set.

See also changeItem() [p. 82], text() [p. 92] and pixmap() [p. 90].

int QMenuData::idAt (int index) const

Returns the identifier of the menu item at position *index* in the internal list, or -1 if *index* is out of range.

See also setId() [p. 91] and indexOf() [p. 84].

int QMenuData::indexOf (int id) const

Returns the index of the menu item with identifier *id*, or -1 if there is no item with this identifier.

See also idAt() [p. 84] and findItem() [p. 84].

Example: scrollview/scrollview.cpp.

int QMenuData::insertItem (const QString & text, const QObject * receiver, const char * member, const QKeySequence & accel = 0, int id = -1, int index = -1)

The family of insertItem() functions inserts menu items into a popup menu or a menu bar.

A menu item is usually either a text string or a pixmap, both with an optional icon or keyboard accelerator. For special cases it is also possible to insert custom items (see QCustomMenuItem) or even widgets into popup menus.

Some insertItem() members take a popup menu as an additional argument. Use this to insert submenus to existing menus or pulldown menus to a menu bar.

The number of insert functions may look confusing, but they are actually quite simple to use.

This default version inserts a menu item with the text *text*, the accelerator key *accel*, an id and an optional index and connects it to the slot *member* in the object *receiver*.

Example:

```
QMenuBar   *mainMenu = new QMenuBar;
QPopupMenu *fileMenu = new QPopupMenu;
fileMenu->insertItem( "New",  myView, SLOT(newFile()), CTRL+Key_N );
fileMenu->insertItem( "Open", myView, SLOT(open()),    CTRL+Key_O );
mainMenu->insertItem( "File", fileMenu );
```

Not all insert functions take an object/slot parameter or an accelerator key. Use connectItem() and setAccel() on these items.

If you need to translate accelerators, use tr() with a string description that use pass to the QKeySequence constructor:

```
fileMenu->insertItem( tr("Open"), myView, SLOT(open()),
                      tr("Ctrl+O") );
```

In the example above, pressing Ctrl+N or selecting "Open" from the menu activates the myView->open() function.

Some insert functions take a QIconSet parameter to specify the little menu item icon. Note that you can always pass a QPixmap object instead.

The *index* specifies the position in the menu. The menu item is appended at the end of the list if *index* is negative.

Note that keyboard accelerators in Qt are not application-global, instead they are bound to a certain top-level window. For example, accelerators in QPopupMenu items only work for menus that are associated with a certain window. This is true for popup menus that live in a menu bar since their accelerators will then be installed in the menu bar itself. This also applies to stand-alone popup menus that have a top-level widget in their parentWidget() chain. The menu will then install its accelerator object on that top-level widget. For all other cases use an independent QAccel object.

Warning: Be careful when passing a literal 0 to insertItem() because some C++ compilers choose the wrong overloaded function. Cast the 0 to what you mean, e.g. `(QObject*)0`.

Returns the allocated menu identifier number (*id* if *id* >= 0).

See also removeItem() [p. 90], changeItem() [p. 82], setAccel() [p. 90], connectItem() [p. 83], QAccel [Events, Actions, Layouts and Styles with Qt] and qnamespace.h.

Examples: addressbook/mainwindow.cpp, mdi/application.cpp, menu/menu.cpp, qwerty/qwerty.cpp, scrollview/scrollview.cpp and showimg/showimg.cpp.

int QMenuData::insertItem (const QIconSet & icon, const QString & text, const QObject * receiver, const char * member, const QKeySequence & accel = 0, int id = -1, int index = -1)

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

Inserts a menu item with icon *icon*, text *text*, accelerator *accel*, optional id *id*, and optional *index*. The menu item is connected it to the *receiver*'s *member* slot. The icon will be displayed to the left of the text in the item.

Returns the allocated menu identifier number (*id* if *id* >= 0).

See also removeItem() [p. 90], changeItem() [p. 82], setAccel() [p. 90], connectItem() [p. 83], QAccel [Events, Actions, Layouts and Styles with Qt] and qnamespace.h.

int QMenuData::insertItem (const QPixmap & pixmap, const QObject * receiver, const char * member, const QKeySequence & accel = 0, int id = -1, int index = -1)

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

Inserts a menu item with pixmap *pixmap*, accelerator *accel*, optional id *id*, and optional *index*. The menu item is connected it to the *receiver*'s *member* slot. The icon will be displayed to the left of the text in the item.

To look best when being highlighted as a menu item, the pixmap should provide a mask (see QPixmap::mask()).

Returns the allocated menu identifier number (*id* if *id* >= 0).

See also removeItem() [p. 90], changeItem() [p. 82], setAccel() [p. 90] and connectItem() [p. 83].

int QMenuData::insertItem (const QIconSet & icon, const QPixmap & pixmap, const QObject * receiver, const char * member, const QKeySequence & accel = 0, int id = -1, int index = -1)

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

Inserts a menu item with icon *icon*, pixmap *pixmap*, accelerator *accel*, optional id *id*, and optional *index*. The icon will be displayed to the left of the pixmap in the item. The item is connected to the *member* slot in the *receiver* object.

To look best when being highlighted as a menu item, the pixmap should provide a mask (see QPixmap::mask()).

Returns the allocated menu identifier number (*id* if *id* >= 0).

See also removeItem() [p. 90], changeItem() [p. 82], setAccel() [p. 90], connectItem() [p. 83], QAccel [Events, Actions, Layouts and Styles with Qt] and qnamespace.h.

int QMenuData::insertItem (const QString & text, int id = -1, int index = -1)

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

Inserts a menu item with text *text*, optional id *id*, and optional *index*.

Returns the allocated menu identifier number (*id* if *id* >= 0).

See also removeItem() [p. 90], changeItem() [p. 82], setAccel() [p. 90] and connectItem() [p. 83].

int QMenuData::insertItem (const QIconSet & icon, const QString & text, int id = -1, int index = -1)

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

Inserts a menu item with icon *icon*, text *text*, optional id *id*, and optional *index*. The icon will be displayed to the left of the text in the item.

Returns the allocated menu identifier number (*id* if *id* >= 0).

See also removeItem() [p. 90], changeItem() [p. 82], setAccel() [p. 90] and connectItem() [p. 83].

int QMenuData::insertItem (const QString & text, QPopupMenu * popup, int id = -1, int index = -1)

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

Inserts a menu item with text *text*, submenu *popup*, optional id *id*, and optional *index*.

The *popup* must be deleted by the programmer or by its parent widget. It is not deleted when this menu item is removed or when the menu is deleted.

Returns the allocated menu identifier number (*id* if *id* >= 0).

See also removeItem() [p. 90], changeItem() [p. 82], setAccel() [p. 90] and connectItem() [p. 83].

int QMenuData::insertItem (const QIconSet & icon, const QString & text, QPopupMenu * popup, int id = -1, int index = -1)

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

Inserts a menu item with icon *icon*, text *text*, submenu *popup*, optional id *id*, and optional *index*. The icon will be displayed to the left of the text in the item.

The *popup* must be deleted by the programmer or by its parent widget. It is not deleted when this menu item is removed or when the menu is deleted.

Returns the allocated menu identifier number (*id* if *id* >= 0).

See also removeItem() [p. 90], changeItem() [p. 82], setAccel() [p. 90] and connectItem() [p. 83].

int QMenuData::insertItem (const QPixmap & pixmap, int id = -1, int index = -1)

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

Inserts a menu item with pixmap *pixmap*, optional id *id*, and optional *index*.

To look best when being highlighted as a menu item, the pixmap should provide a mask (see QPixmap::mask()).

Returns the allocated menu identifier number (*id* if *id* >= 0).

See also removeItem() [p. 90], changeItem() [p. 82], setAccel() [p. 90] and connectItem() [p. 83].

int QMenuData::insertItem (const QIconSet & icon, const QPixmap & pixmap, int id = -1, int index = -1)

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

Inserts a menu item with icon *icon*, pixmap *pixmap*, optional id *id*, and optional *index*. The icon will be displayed to the left of the pixmap in the item.

Returns the allocated menu identifier number (*id* if *id* >= 0).

See also removeItem() [p. 90], changeItem() [p. 82], setAccel() [p. 90] and connectItem() [p. 83].

int QMenuData::insertItem (const QPixmap & pixmap, QPopupMenu * popup, int id = -1, int index = -1)

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

Inserts a menu item with pixmap *pixmap*, submenu *popup*, optional id *id*, and optional *index*.

The *popup* must be deleted by the programmer or by its parent widget. It is not deleted when this menu item is removed or when the menu is deleted.

Returns the allocated menu identifier number (*id* if *id* >= 0).

See also removeItem() [p. 90], changeItem() [p. 82], setAccel() [p. 90] and connectItem() [p. 83].

int QMenuData::insertItem (const QIconSet & icon, const QPixmap & pixmap, QPopupMenu * popup, int id = -1, int index = -1)

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

Inserts a menu item with icon *icon*, pixmap *pixmap* submenu *popup*, optional id *id*, and optional *index*. The icon will be displayed to the left of the pixmap in the item.

The *popup* must be deleted by the programmer or by its parent widget. It is not deleted when this menu item is removed or when the menu is deleted.

Returns the allocated menu identifier number (*id* if *id* >= 0).

See also removeItem() [p. 90], changeItem() [p. 82], setAccel() [p. 90] and connectItem() [p. 83].

int QMenuData::insertItem (QWidget * widget, int id = -1, int index = -1)

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

Inserts a menu item that consists of the widget *widget* with optional id *id*, and optional *index*.

Ownership of *widget* is transferred to the popup menu or to the menu bar.

Theoretically, any widget can be inserted into a popup menu. In practice, this only makes sense with certain widgets.

If a widget is not focus-enabled (see QWidget::isFocusEnabled()), the menu treats it as a separator; this means that the item is not selectable and will never get focus. In this way you can, for example, simply insert a QLabel if you need a popup menu with a title.

If the widget is focus-enabled it will get focus when the user traverses the popup menu with the arrow keys. If the widget does not accept ArrowUp and ArrowDown in its key event handler, the focus will move back to the menu when the respective arrow key is hit one more time. This works with a QLineEdit, for example. If the widget accepts the arrow key itself, it must also provide the possibility to put the focus back on the menu again by calling QWidget::focusNextPrevChild(). Futhermore, if the embedded widget closes the menu when the user made a selection, this can be done safely by calling

```
if ( isVisible() &&
     parentWidget() &&
     parentWidget()->inherits("QPopupMenu") )
    parentWidget()->close();
```

Returns the allocated menu identifier number (*id* if *id* >= 0).

See also removeItem() [p. 90].

int QMenuData::insertItem (const QIconSet & icon, QCustomMenuItem * custom, int id = -1, int index = -1)

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

Inserts a custom menu item *custom* with an *icon* and with optional id *id*, and optional *index*.

This only works with popup menus. It is not supported for menu bars. Ownership of *custom* is transferred to the popup menu.

If you want to connect a custom item to a certain slot, use connectItem().

Returns the allocated menu identifier number (*id* if *id* >= 0).

See also connectItem() [p. 83], removeItem() [p. 90] and QCustomMenuItem [p. 5].

int QMenuData::insertItem (QCustomMenuItem * custom, int id = -1, int index = -1)

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

Inserts a custom menu item *custom* with optional id *id*, and optional *index*.

This only works with popup menus. It is not supported for menu bars. Ownership of *custom* is transferred to the popup menu.

If you want to connect a custom item to a certain slot, use connectItem().

Returns the allocated menu identifier number (*id* if *id* >= 0).

See also connectItem() [p. 83], removeItem() [p. 90] and QCustomMenuItem [p. 5].

int QMenuData::insertSeparator (int index = -1)

Inserts a separator at position *index*. The separator becomes the last menu item if *index* is negative.

In a popup menu a separator is rendered as a horizontal line. In a Motif menu bar a separator is spacing, so the rest of the items (normally just "Help") are drawn right-justified. In a Windows menu bar separators are ignored (to comply with the Windows style guidelines).

Examples: addressbook/mainwindow.cpp, mdi/application.cpp, menu/menu.cpp, progress/progress.cpp, qwerty/qwerty.cpp, scrollview/scrollview.cpp and showimg/showimg.cpp.

bool QMenuData::isItemActive (int id) const

Returns TRUE if the menu item with the id *id* is currently active; otherwise returns FALSE.

bool QMenuData::isItemChecked (int id) const

Returns TRUE if the menu item with the id *id* has been checked; otherwise returns FALSE.

See also setItemChecked() [p. 91].

Examples: progress/progress.cpp and showimg/showimg.cpp.

bool QMenuData::isItemEnabled (int id) const

Returns TRUE if the item with identifier *id* is enabled; otherwise returns FALSE

See also setItemEnabled() [p. 91].

int QMenuData::itemParameter (int id) const

Returns the parameter of the activation signal of item *id*.

If no parameter has been specified for this item with setItemParameter(), the value defaults to *id*.

See also connectItem() [p. 83], disconnectItem() [p. 84] and setItemParameter() [p. 91].

void QMenuData::menuContentsChanged () [virtual protected]

Virtual function; notifies subclasses that one or more items have been inserted or removed.

Reimplemented in QMenuBar.

void QMenuData::menuDelPopup (QPopupMenu *) [virtual protected]

Virtual function; notifies subclasses that a popup menu item has been removed.

void QMenuData::menuInsPopup (QPopupMenu *) [virtual protected]

Virtual function; notifies subclasses that a popup menu item has been inserted.

void QMenuData::menuStateChanged () [virtual protected]

Virtual function; notifies subclasses that one or more items have changed state (enabled/disabled or checked/unchecked).

Reimplemented in QMenuBar.

QPixmap * QMenuData::pixmap (int id) const

Returns the pixmap that has been set for menu item *id*, or 0 if no pixmap has been set.

See also changeItem() [p. 82], text() [p. 92] and iconSet() [p. 84].

void QMenuData::removeItem (int id)

Removes the menu item that has the identifier *id*.

See also removeItemAt() [p. 90] and clear() [p. 83].

void QMenuData::removeItemAt (int index)

Removes the menu item at position *index*.

See also removeItem() [p. 90] and clear() [p. 83].

void QMenuData::setAccel (const QKeySequence & key, int id)

Sets the accelerator key for the menu item *id* to *key*.

An accelerator key consists of a key code and a combination of the modifiers `SHIFT`, `CTRL`, `ALT` or `UNICODE_ACCEL` (OR'ed or added). The header file qnamespace.h contains a list of key codes.

Defining an accelerator key produces a text that is added to the menu item; for instance, `CTRL + Key_O` produces "Ctrl+O". The text is formatted differently for different platforms.

Note that keyboard accelerators in Qt are not application-global, instead they are bound to a certain top-level window. For example, accelerators in QPopupMenu items only work for menus that are associated with a certain window. This is true for popup menus that live in a menu bar since their accelerators will then be installed in the menu bar itself. This also applies to stand-alone popup menus that have a top-level widget in their parentWidget() chain. The menu will then install its accelerator object on that top-level widget. For all other cases use an independent QAccel object.

Example:

```
QMenuBar   *mainMenu = new QMenuBar;
QPopupMenu *fileMenu = new QPopupMenu;       // file sub menu
```

```
fileMenu->insertItem( "Open Document", 67 ); // add "Open" item
fileMenu->setAccel( CTRL + Key_O, 67 );      // Control and O to open
fileMenu->insertItem( "Quit", 69 );          // add "Quit" item
fileMenu->setAccel( CTRL + ALT + Key_Delete, 69 );
mainMenu->insertItem( "File", fileMenu );    // add the file menu
```

If you need to translate accelerators, use QAccel::stringToKey():

```
fileMenu->setAccel( QAccel::stringToKey(tr("Ctrl+O")), 67 );
```

You can also specify the accelerator in the insertItem() function. You may prefer to use QAction to associate accelerators with menu items.

See also accel() [p. 82], insertItem() [p. 84], QAccel [Events, Actions, Layouts and Styles with Qt], qnamespace.h and QAction [Events, Actions, Layouts and Styles with Qt].

Example: menu/menu.cpp.

void QMenuData::setId (int index, int id) [virtual]

Sets the menu identifier of the item at *index* to *id*.

If index is out of range, the operation is ignored.

See also idAt() [p. 84].

void QMenuData::setItemChecked (int id, bool check)

If *check* is TRUE, checks the menu item with id *id*; otherwise unchecks the menu item with id *id*. Calls QPopupMenu::setCheckable(TRUE) if necessary.

See also isItemChecked() [p. 89].

Examples: grapher/grapher.cpp, mdi/application.cpp, menu/menu.cpp, progress/progress.cpp, scrollview/scrollview.cpp and showimg/showimg.cpp.

void QMenuData::setItemEnabled (int id, bool enable)

If *enable* is TRUE, enables the menu item with identifier *id*; otherwise disables the menu item with identifier *id*.

See also isItemEnabled() [p. 89].

Examples: mdi/application.cpp, menu/menu.cpp, progress/progress.cpp and showimg/showimg.cpp.

bool QMenuData::setItemParameter (int id, int param)

Sets the parameter of the activation signal of item *id* to *param*.

If any receiver takes an integer parameter, this value is passed.

See also connectItem() [p. 83], disconnectItem() [p. 84] and itemParameter() [p. 89].

Example: mdi/application.cpp.

void QMenuData::setWhatsThis (int id, const QString & text)

Sets *text* as What's This help for the menu item with identifier *id*.

See also whatsThis() [p. 92].

Examples: application/application.cpp and mdi/application.cpp.

QString QMenuData::text (int id) const

Returns the text that has been set for menu item *id*, or a null string if no text has been set.

See also changeItem() [p. 82], pixmap() [p. 90] and iconSet() [p. 84].

Examples: qdir/qdir.cpp and showimg/showimg.cpp.

void QMenuData::updateItem (int id) [virtual]

Virtual function; notifies subclasses about an item with *id* that has been changed.

Reimplemented in QPopupMenu.

QString QMenuData::whatsThis (int id) const

Returns the What's This help text for the item with identifier *id* or QString::null if no text has yet been defined.

See also setWhatsThis() [p. 91].

QMessageBox Class Reference

The QMessageBox class provides a modal dialog with a short message, an icon, and some buttons.

`#include <qmessagebox.h>`

Inherits QDialog [p. 10].

Public Members

- enum **Icon** { NoIcon = 0, Information = 1, Warning = 2, Critical = 3 }
- **QMessageBox** (QWidget * parent = 0, const char * name = 0)
- **QMessageBox** (const QString & caption, const QString & text, Icon icon, int button0, int button1, int button2, QWidget * parent = 0, const char * name = 0, bool modal = TRUE, WFlags f = WStyle_DialogBorder)
- **~QMessageBox** ()
- QString **text** () const
- void **setText** (const QString &)
- Icon **icon** () const
- void **setIcon** (Icon)
- const QPixmap * **iconPixmap** () const
- void **setIconPixmap** (const QPixmap &)
- QString **buttonText** (int button) const
- void **setButtonText** (int button, const QString & text)
- virtual void **adjustSize** ()
- TextFormat **textFormat** () const
- void **setTextFormat** (TextFormat)

Static Public Members

- int **information** (QWidget * parent, const QString & caption, const QString & text, int button0, int button1 = 0, int button2 = 0)
- int **information** (QWidget * parent, const QString & caption, const QString & text, const QString & button0Text = QString::null, const QString & button1Text = QString::null, const QString & button2Text = QString::null, int defaultButtonNumber = 0, int escapeButtonNumber = -1)
- int **warning** (QWidget * parent, const QString & caption, const QString & text, int button0, int button1, int button2 = 0)
- int **warning** (QWidget * parent, const QString & caption, const QString & text, const QString & button0Text = QString::null, const QString & button1Text = QString::null, const QString & button2Text = QString::null, int defaultButtonNumber = 0, int escapeButtonNumber = -1)
- int **critical** (QWidget * parent, const QString & caption, const QString & text, int button0, int button1, int button2 = 0)

- int **critical** (QWidget * parent, const QString & caption, const QString & text, const QString & button0Text = QString::null, const QString & button1Text = QString::null, const QString & button2Text = QString::null, int defaultButtonNumber = 0, int escapeButtonNumber = -1)
- void **about** (QWidget * parent, const QString & caption, const QString & text)
- void **aboutQt** (QWidget * parent, const QString & caption = QString::null)
- int message (const QString & caption, const QString & text, const QString & buttonText = QString::null, QWidget * parent = 0, const char * = 0) *(obsolete)*
- bool query (const QString & caption, const QString & text, const QString & yesButtonText = QString::null, const QString & noButtonText = QString::null, QWidget * parent = 0, const char * = 0) *(obsolete)*
- QPixmap standardIcon (Icon icon, GUIStyle style) *(obsolete)*
- QPixmap **standardIcon** (Icon icon)

Properties

- Icon **icon** — the messagebox icon
- QPixmap **iconPixmap** — the current icon
- QString **text** — the message box text to be displayed
- TextFormat **textFormat** — the format of the text displayed by the message box

Detailed Description

The QMessageBox class provides a modal dialog with a short message, an icon, and some buttons.

A message box is a modal dialog that displays an icon, some text and up to three push buttons. It's used for simple messages and questions.

QMessageBox provides a range of different messages, arranged roughly along two axes: severity and complexity.

Severity is

- Information - for message boxes that are part of normal operation
- Warning - for message boxes that tell the user about unusual errors
- Critical - as Warning, but for critical errors

The message box has a different icon for each of the severity levels.

Complexity is one button (OK) for a simple messages, or two or even three buttons for questions.

There are static functions for common cases. For example:

If a program is unable to find a supporting file, but can do perfectly well without it:

```
QMessageBox::information( this, "Application name",
    "Unable to find the user preferences file.\n"
    "The factory default will be used instead." );
```

warning() can be used to tell the user about unusual errors, or errors which can't be easily fixed:

```
switch( QMessageBox::warning( this, "Application name",
        "Could not connect to the  server.\n"
        "This program can't function correctly "
        "without the server.\n\n",
        "Retry",
```

```
        "Quit", 0, 0, 1 ) )
    case 0: // The user clicked the Retry again button or pressed Enter
        // try again
        break;
    case 1: // The user clicked the Quit or pressed Escape
        // exit
        break;
}
```

The text part of all message box messages can be either rich text or plain text. If you specify a rich text formatted string, it will be rendered using the default stylesheet. See QStyleSheet::defaultSheet() for details. With certain strings that contain XML meta characters, the auto-rich text detection may fail, interpreting plain text incorrectly as rich text. In these rare cases, use QStyleSheet::convertFromPlainText() to convert your plain text string to a visually equivalent rich text string or set the text format explicitly with setTextFormat().

Below are some examples of how to use the static member functions. After these examples you will find an overview of the non-static member functions.

If a program is unable to find a supporting file, it may do the following:

```
QMessageBox::information( this, "Application name here",
    "Unable to find the file \"index.html\".\n"
    "The factory default will be used instead." );
```

The Microsoft Windows User Interface Guidelines strongly recommend using the application name as the window's caption. The message box has just one button, OK, and its text tells the user both what happened and what the program will do about it. Because the application is able to make do, the message box is just information, not a warning or a critical error.

Exiting a program is part of its normal operation. If there is unsaved data the user probably should be asked if they want to save the data. For example:

```
switch( QMessageBox::information( this, "Application name here",
        "The document contains unsaved changes\n"
        "Do you want to save the changes before exiting?",
        "&Save", "&Discard", "Cancel",
        0,      // Enter == button 0
        2 ) ) { // Escape == button 2
    case 0: // Save clicked or Alt+S pressed or Enter pressed.
        // save
        break;
    case 1: // Discard clicked or Alt+D pressed
        // don't save but exit
        break;
    case 2: // Cancel clicked or Alt+C pressed or Escape pressed
        // don't exit
        break;
}
```

The application name is used as the window caption in accordance with the Microsoft recommendation. The Escape button cancels the entire exit operation, and pressing Enter causes the changes to be saved before the exit occurs.

Disk full errors are unusual (in a perfect world, they are) and they certainly can be hard to correct. This example uses predefined buttons instead of hard-coded button texts:

```
switch( QMessageBox::warning( this, "Application name here",
        "Could not save the user preferences,\n"
        "because the disk is full.  You can delete\n"
```

```
        "some files and press Retry, or you can\n"
        "abort the Save Preferences operation.",
        QMessageBox::Retry | QMessageBox::Default,
        QMessageBox::Abort | QMessageBox::Escape )) {
    case QMessageBox::Retry: // Retry clicked or Enter pressed
        // try again
        break;
    case QMessageBox::Abort: // Abort clicked or Escape pressed
        // abort
        break;
}
```

The critical() function should be reserved for critical errors. In this example errorDetails is a QString or const char*, and QString is used to concatenate several strings:

```
QMessageBox::critical( 0, "Application name here",
        QString("An internal error occurred. Please ") +
        "call technical support at 123456789 and report\n"+
        "these numbers:\n\n" + errorDetails +
        "\n\n will now exit." );
```

In this example an OK button is displayed.

QMessageBox provides a very simple About box, which displays an appropriate icon and the string you provide:

```
QMessageBox::about( this, "About ",
        " is a \n\n"
        "Copyright 1951-2002 Such-and-such.  "
        "\n\n"
        "For technical support, call 123456789 or see\n"
        "http://www.such-and-such.com/Application/\n" );
```

See about() for more information.

Finally, you can create a QMessageBox from scratch and with custom button texts:

```
QMessageBox mb( "Application name here",
        "Saving the file will overwrite the original file on the disk.\n"
        "Do you really want to save?",
        QMessageBox::Information,
        QMessageBox::Yes | QMessageBox::Default,
        QMessageBox::No,
        QMessageBox::Cancel | QMessageBox::Escape );
mb.setButtonText( QMessageBox::Yes, "Save" );
mb.setButtonText( QMessageBox::No, "Discard" );
switch( mb.exec() ) {
    case QMessageBox::Yes:
        // save and exit
        break;
    case QMessageBox::No:
        // exit without saving
        break;
    case QMessageBox::Cancel:
        // don't save and don't exit
        break;
}
```

QMessageBox defines two enum types: Icon and an unnamed button type. Icon defines the Information, Warning, and Critical icons for each GUI style. It is used by the constructor and by the static member functions information(), warning() and critical(). A function called standardIcon() gives you access to the various icons.

The button types are:

- Ok - the default for single-button message boxes
- Cancel - note that this is *not* automatically Escape
- Yes
- No
- Abort
- Retry
- Ignore

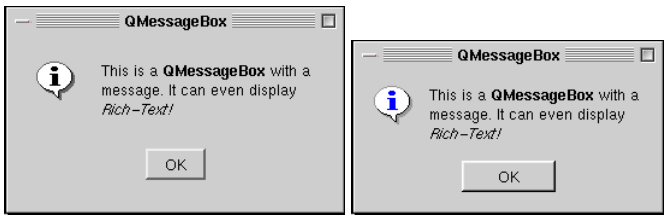
Button types can be combined with two modifiers by using OR, '|':

- Default - makes pressing Enter equivalent to clicking this button. Normally used with Ok, Yes or similar.
- Escape - makes pressing Escape equivalent to clicking this button. Normally used with Abort, Cancel or similar.

The text(), icon() and iconPixmap() functions provide access to the current text and pixmap of the message box. The setText(), setIcon() and setIconPixmap() let you change it. The difference between setIcon() and setIconPixmap() is that the former accepts a QMessageBox::Icon and can be used to set standard icons, whereas the latter accepts a QPixmap and can be used to set custom icons.

setButtonText() and buttonText() provide access to the buttons.

QMessageBox has no signals or slots.



See also QDialog [p. 10], Isys on error messages, GUI Design Handbook: Message Box and Dialog Classes.

Member Type Documentation

QMessageBox::Icon

This enum includes provides the following values:

- `QMessageBox::NoIcon` - the message box does not have any icon.
- `QMessageBox::Information` - an icon indicating that the message is nothing out of the ordinary.
- `QMessageBox::Warning` - an icon indicating that the message is a warning, but can be dealt with.
- `QMessageBox::Critical` - an icon indicating that the message represents a critical problem.

Member Function Documentation

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

Constructs a message box with no text and a button with the label "OK".

If *parent* is 0, the message box becomes an application-global modal dialog box. If *parent* is a widget, the message box becomes modal relative to *parent*.

The *parent* and *name* arguments are passed to the QDialog constructor.

QMessageBox::QMessageBox (const QString & caption, const QString & text, Icon icon, int button0, int button1, int button2, QWidget * parent = 0, const char * name = 0, bool modal = TRUE, WFlags f = WStyle_DialogBorder)

Constructs a message box with a *caption*, a *text*, an *icon*, and up to three buttons.

The *icon* must be one of the following:

- QMessageBox::NoIcon
- QMessageBox::Information
- QMessageBox::Warning
- QMessageBox::Critical

Each button, *button0*, *button1* and *button2*, can have one of the following values:

- QMessageBox::NoButton
- QMessageBox::Ok
- QMessageBox::Cancel
- QMessageBox::Yes
- QMessageBox::No
- QMessageBox::Abort
- QMessageBox::Retry
- QMessageBox::Ignore

Use QMessageBox::NoButton for the later parameters to have fewer than three buttons in your message box.

One of the buttons can be OR-ed with the `QMessageBox::Default` flag to make it the default button (clicked when Enter is pressed).

One of the buttons can be OR-ed with the `QMessageBox::Escape` flag to make it the cancel or close button (clicked when Escape is pressed).

Example:

```
QMessageBox mb( "Application Name",
        "Hardware failure.\n\nDisk error detected\nDo you want to stop?",
        QMessageBox::NoIcon,
        QMessageBox::Yes | QMessageBox::Default,
        QMessageBox::No  | QMessageBox::Escape );
if ( mb.exec() == QMessageBox::No )
    // try again
```

If *parent* is 0, the message box becomes an application-global modal dialog box. If *parent* is a widget, the message box becomes modal relative to *parent*.

If *modal* is TRUE the message becomes modal; otherwise it becomes modeless.

The *parent*, *name*, *modal*, and *f* arguments are passed to the QDialog constructor.

See also caption [Widgets with Qt], text [p. 104] and icon [p. 104].

QMessageBox::~QMessageBox ()

Destroys the message box.

void QMessageBox::about (QWidget * parent, const QString & caption, const QString & text) [static]

Displays a simple about box with caption *caption* and text *text*. The about box's parent is *parent*.

about() looks for a suitable icon in four locations:

1. It prefers parent->icon() if that exists.
2. If not, it tries the top-level widget containing *parent*.
3. If that fails, it tries the main widget.
4. As a last resort it uses the Information icon.

The about box has a single button labelled OK.

See also QWidget::icon [Widgets with Qt] and QApplication::mainWidget() [Additional Functionality with Qt].

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

void QMessageBox::aboutQt (QWidget * parent, const QString & caption = QString::null) [static]

Displays a simple message box about Qt, with caption *caption* and optionally centered over *parent*. The message includes the version number of Qt being used by the application.

This is useful for inclusion in the Help menu. See the examples/menu/menu.cpp example.

Examples: action/application.cpp, application/application.cpp, helpviewer/helpwindow.cpp, mdi/application.cpp, menu/menu.cpp, themes/themes.cpp and trivial/trivial.cpp.

void QMessageBox::adjustSize () [virtual]

Adjusts the size of the message box to fit the contents just before QDialog::exec() or QDialog::show() is called.

This function will not be called if the message box has been explicitly resized before showing it.

Reimplemented from QWidget [Widgets with Qt].

QString QMessageBox::buttonText (int button) const

Returns the text of the messagebox button *button*, or null if the message box does not contain the button.

See also setButtonText() [p. 102].

int QMessageBox::critical (QWidget * parent, const QString & caption, const QString & text, int button0, int button1, int button2 = 0) [static]

Opens a critical message box with the caption *caption* and the text *text*. The dialog may have up to three buttons. Each of the button parameters, *button0*, *button1* and *button2* may be set to one of the following values:

- QMessageBox::NoButton
- QMessageBox::Ok
- QMessageBox::Cancel
- QMessageBox::Yes
- QMessageBox::No
- QMessageBox::Abort
- QMessageBox::Retry
- QMessageBox::Ignore

If you don't want all three buttons, set the last button, or last two buttons to QMessageBox::NoButton.

Returns the index of the button that was clicked.

If *parent* is 0, the message box becomes an application-global modal dialog box. If *parent* is a widget, the message box becomes modal relative to *parent*.

See also information() [p. 101] and warning() [p. 103].

Examples: network/ftpclient/ftpmainwindow.cpp, process/process.cpp and xml/outliner/outlinetree.cpp.

int QMessageBox::critical (QWidget * parent, const QString & caption, const QString & text, const QString & button0Text = QString::null, const QString & button1Text = QString::null, const QString & button2Text = QString::null, int defaultButtonNumber = 0, int escapeButtonNumber = -1) [static]

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

Displays a critical error message box with a caption, a text, and 1..3 buttons. Returns the number of the button that was clicked (0, 1 or 2).

button0Text is the text of the first button and is optional. If *button0Text* is not supplied, "OK" (translated) will be used. *button1Text* is the text of the second button and is optional, and *button2Text* is the text of the third button and is optional. *defaultButtonNumber* (0..2) is the index of the default button; pressing Return or Enter is the same as clicking the default button. It defaults to 0 (the first button). *escapeButtonNumber* is the index of the Escape button; pressing Escape is the same as clicking this button. It defaults to -1 (pressing Escape does nothing); supply 0, 1, or 2 to make pressing Escape equivalent to clicking the relevant button.

If *parent* is 0, the message box becomes an application-global modal dialog box. If *parent* is a widget, the message box becomes modal relative to *parent*.

See also information() [p. 101] and warning() [p. 103].

Icon QMessageBox::icon () const

Returns the messagebox icon. See the "icon" [p. 104] property for details.

const QPixmap * QMessageBox::iconPixmap () const

Returns the current icon. See the "iconPixmap" [p. 104] property for details.

int QMessageBox::information (QWidget * parent, const QString & caption, const QString & text, int button0, int button1 = 0, int button2 = 0) [static]

Opens an information message box with the caption *caption* and the text *text*. The dialog may have up to three buttons. Each of the buttons, *button0*, *button1* and *button2* may be set to one of the following values:

- QMessageBox::NoButton
- QMessageBox::Ok
- QMessageBox::Cancel
- QMessageBox::Yes
- QMessageBox::No
- QMessageBox::Abort
- QMessageBox::Retry
- QMessageBox::Ignore

If you don't want all three buttons, set the last button, or last two buttons to QMessageBox::NoButton.

Returns the index of the button that was clicked.

If *parent* is 0, the message box becomes an application-global modal dialog box. If *parent* is a widget, the message box becomes modal relative to *parent*.

See also warning() [p. 103] and critical() [p. 100].

Examples: action/application.cpp, application/application.cpp, dirview/dirview.cpp, fileiconview/qfileiconview.cpp, picture/picture.cpp, qwerty/qwerty.cpp and sql/sqltable/main.cpp.

int QMessageBox::information (QWidget * parent, const QString & caption, const QString & text, const QString & button0Text = QString::null, const QString & button1Text = QString::null, const QString & button2Text = QString::null, int defaultButtonNumber = 0, int escapeButtonNumber = -1) [static]

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

Displays an information message box with caption *caption*, text *text* and one, two or three buttons. Returns the index of the button that was clicked (0, 1 or 2).

button0Text is the text of the first button and is optional. If *button0Text* is not supplied, "OK" (translated) will be used. *button1Text* is the text of the second button and is optional. *button2Text* is the text of the third button and is optional. *defaultButtonNumber* (0..2) is the index of the default button; pressing Return or Enter is the same as clicking the default button. It defaults to 0 (the first button). *escapeButtonNumber* is the index of the Escape button; pressing Escape is the same as clicking this button. It defaults to -1 (pressing Escape does nothing); supply 0, 1 or 2 to make pressing Escape equivalent to clicking the relevant button.

If *parent* is 0, the message box becomes an application-global modal dialog box. If *parent* is a widget, the message box becomes modal relative to *parent*.

See also warning() [p. 103] and critical() [p. 100].

int QMessageBox::message (const QString & caption, const QString & text, const QString & buttonText = QString::null, QWidget * parent = 0, const char * = 0) [static]

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

Opens a modal message box directly using the specified parameters.

Please use information(), warning() or critical() instead.

Example: grapher/grapher.cpp.

bool QMessageBox::query (const QString & caption, const QString & text, const QString & yesButtonText = QString::null, const QString & noButtonText = QString::null, QWidget * parent = 0, const char * = 0) [static]

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

Queries the user using a modal message box with two buttons. Note that *caption* is not always shown, it depends on the window manager.

Please use information(), warning() or critical() instead.

void QMessageBox::setButtonText (int button, const QString & text)

Sets the text of the message box button *button* to *text*. Setting the text of a button that is not in the message box is silently ignored.

See also buttonText() [p. 99].

void QMessageBox::setIcon (Icon)

Sets the messagebox icon. See the "icon" [p. 104] property for details.

void QMessageBox::setIconPixmap (const QPixmap &)

Sets the current icon. See the "iconPixmap" [p. 104] property for details.

void QMessageBox::setText (const QString &)

Sets the message box text to be displayed. See the "text" [p. 104] property for details.

void QMessageBox::setTextFormat (TextFormat)

Sets the format of the text displayed by the message box. See the "textFormat" [p. 104] property for details.

QPixmap QMessageBox::standardIcon (Icon icon) [static]

Returns the pixmap used for a standard icon. This allows the pixmaps to be used in more complex message boxes. *icon* specifies the required icon, e.g. QMessageBox::Information, QMessageBox::Warning or QMessageBox::Critical.

QPixmap QMessageBox::standardIcon (Icon icon, GUIStyle style) [static]

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

Returns the pixmap used for a standard icon. This allows the pixmaps to be used in more complex message boxes. *icon* specifies the required icon, e.g. QMessageBox::Information, QMessageBox::Warning or QMessageBox::Critical.

style is unused.

QString QMessageBox::text () const

Returns the message box text to be displayed. See the "text" [p. 104] property for details.

TextFormat QMessageBox::textFormat () const

Returns the format of the text displayed by the message box. See the "textFormat" [p. 104] property for details.

int QMessageBox::warning (QWidget * parent, const QString & caption, const QString & text, int button0, int button1, int button2 = 0) [static]

Opens a warning message box with the caption *caption* and the text *text*. The dialog may have up to three buttons. Each of the button parameters, *button0*, *button1* and *button2* may be set to one of the following values:

- QMessageBox::NoButton
- QMessageBox::Ok
- QMessageBox::Cancel
- QMessageBox::Yes
- QMessageBox::No
- QMessageBox::Abort
- QMessageBox::Retry
- QMessageBox::Ignore

If you don't want all three buttons, set the last button, or last two buttons to QMessageBox::NoButton.

Returns the index of the button that was clicked.

If *parent* is 0, the message box becomes an application-global modal dialog box. If *parent* is a widget, the message box becomes modal relative to *parent*.

See also information() [p. 101] and critical() [p. 100].

Examples: i18n/main.cpp, movies/main.cpp, network/mail/smtp.cpp, qwerty/qwerty.cpp and showimg/showimg.cpp.

int QMessageBox::warning (QWidget * parent, const QString & caption, const QString & text, const QString & button0Text = QString::null, const QString & button1Text = QString::null, const QString & button2Text = QString::null, int defaultButtonNumber = 0, int escapeButtonNumber = -1) [static]

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

Displays a warning message box with a caption, a text, and 1..3 buttons. Returns the number of the button that was clicked (0, 1, or 2).

button0Text is the text of the first button and is optional. If *button0Text* is not supplied, "OK" (translated) will be used. *button1Text* is the text of the second button and is optional, and *button2Text* is the text of the third button and is optional. *defaultButtonNumber* (0..2) is the index of the default button; pressing Return or Enter is the same

as clicking the default button. It defaults to 0 (the first button). *escapeButtonNumber* is the index of the Escape button; pressing Escape is the same as clicking this button. It defaults to -1 (pressing Escape does nothing); supply 0, 1, or 2 to make pressing Escape equivalent to clicking the relevant button.

If *parent* is 0, the message box becomes an application-global modal dialog box. If *parent* is a widget, the message box becomes modal relative to *parent*.

See also information() [p. 101] and critical() [p. 100].

Property Documentation

Icon icon

This property holds the messagebox icon.

The icon of the message box can be one of the following predefined icons:

- QMessageBox::NoIcon
- QMessageBox::Information
- QMessageBox::Warning
- QMessageBox::Critical

The actual pixmap used for displaying the icon depends on the current GUI style. You can also set a custom pixmap icon using the QMessageBox::iconPixmap property. The default icon is QMessageBox::NoIcon.

See also iconPixmap [p. 104].

Set this property's value with setIcon() and get this property's value with icon().

QPixmap iconPixmap

This property holds the current icon.

The icon currently used by the message box. Note that it's often hard to draw one pixmap that looks appropriate in both Motif and Windows GUI styles; you may want to draw two pixmaps.

See also icon [p. 104].

Set this property's value with setIconPixmap() and get this property's value with iconPixmap().

QString text

This property holds the message box text to be displayed.

The text will be interpreted either as a plain text or as a rich text, depending on the text format setting (QMessageBox::textFormat). The default setting is AutoText, i.e. the message box will try to auto-detect the format of the text.

The default value of the property is QString::null.

See also textFormat [p. 104].

Set this property's value with setText() and get this property's value with text().

TextFormat textFormat

This property holds the format of the text displayed by the message box.

The current text format used by the message box. See the Qt::TextFormat [Additional Functionality with Qt] enum for an explanation of the possible options.

The default format is AutoText.

See also text [p. 104].

Set this property's value with setTextFormat() and get this property's value with textFormat().

QPopupMenu Class Reference

The QPopupMenu class provides a popup menu widget.

`#include <qpopupmenu.h>`

Inherits QFrame [Widgets with Qt] and QMenuData [p. 80].

Public Members

- **QPopupMenu** (QWidget * parent = 0, const char * name = 0)
- **~QPopupMenu** ()
- void **popup** (const QPoint & pos, int indexAtPoint = 0)
- virtual void **updateItem** (int id)
- virtual void **setCheckable** (bool)
- bool **isCheckable** () const
- int **exec** ()
- int **exec** (const QPoint & pos, int indexAtPoint = 0)
- virtual void **setActiveItem** (int i)
- int **idAt** (int index) const
- int **idAt** (const QPoint & pos) const
- int **insertTearOffHandle** (int id = -1, int index = -1)

Signals

- void **activated** (int id)
- void **highlighted** (int id)
- void **aboutToShow** ()
- void **aboutToHide** ()

Important Inherited Members

- int **insertItem** (const QString & text, const QObject * receiver, const char * member, const QKeySequence & accel = 0, int id = -1, int index = -1)
- int **insertItem** (const QIconSet & icon, const QString & text, const QObject * receiver, const char * member, const QKeySequence & accel = 0, int id = -1, int index = -1)
- int **insertItem** (const QPixmap & pixmap, const QObject * receiver, const char * member, const QKeySequence & accel = 0, int id = -1, int index = -1)
- int **insertItem** (const QIconSet & icon, const QPixmap & pixmap, const QObject * receiver, const char * member, const QKeySequence & accel = 0, int id = -1, int index = -1)

- int **insertItem** (const QString & text, int id = -1, int index = -1)
- int **insertItem** (const QIconSet & icon, const QString & text, int id = -1, int index = -1)
- int **insertItem** (const QString & text, QPopupMenu * popup, int id = -1, int index = -1)
- int **insertItem** (const QIconSet & icon, const QString & text, QPopupMenu * popup, int id = -1, int index = -1)
- int **insertItem** (const QPixmap & pixmap, int id = -1, int index = -1)
- int **insertItem** (const QIconSet & icon, const QPixmap & pixmap, int id = -1, int index = -1)
- int **insertItem** (const QPixmap & pixmap, QPopupMenu * popup, int id = -1, int index = -1)
- int **insertItem** (const QIconSet & icon, const QPixmap & pixmap, QPopupMenu * popup, int id = -1, int index = -1)
- int **insertItem** (QWidget * widget, int id = -1, int index = -1)
- int **insertItem** (const QIconSet & icon, QCustomMenuItem * custom, int id = -1, int index = -1)
- int **insertItem** (QCustomMenuItem * custom, int id = -1, int index = -1)
- int **insertSeparator** (int index = -1)
- void **removeItem** (int id)
- void **removeItemAt** (int index)
- void **clear** ()
- QKeySequence **accel** (int id) const
- void **setAccel** (const QKeySequence & key, int id)
- QIconSet * **iconSet** (int id) const
- QString **text** (int id) const
- QPixmap * **pixmap** (int id) const
- void **setWhatsThis** (int id, const QString & text)
- QString **whatsThis** (int id) const
- void **changeItem** (int id, const QString & text)
- void **changeItem** (int id, const QPixmap & pixmap)
- void **changeItem** (int id, const QIconSet & icon, const QString & text)
- void **changeItem** (int id, const QIconSet & icon, const QPixmap & pixmap)
- bool **isItemEnabled** (int id) const
- void **setItemEnabled** (int id, bool enable)
- bool **isItemChecked** (int id) const
- void **setItemChecked** (int id, bool check)
- bool **connectItem** (int id, const QObject * receiver, const char * member)
- bool **disconnectItem** (int id, const QObject * receiver, const char * member)
- bool **setItemParameter** (int id, int param)
- int **itemParameter** (int id) const

Properties

- bool **checkable** — whether the display of check marks on menu items is enabled

Protected Members

- int **itemHeight** (int row) const
- int **itemHeight** (QMenuItem * mi) const
- void **drawItem** (QPainter * p, int tab_, QMenuItem * mi, bool act, int x, int y, int w, int h)
- virtual void **drawContents** (QPainter * p)
- int **columns** () const

Detailed Description

The QPopupMenu class provides a popup menu widget.

A popup menu widget is a selection menu. It can be either a pull-down menu in a menu bar or a standalone context (popup) menu. Pull-down menus are shown by the menu bar when the user clicks on the respective item or hits the specified shortcut key. Use QMenuBar::insertItem() to insert a popup menu into a menu bar. Show a context menu either asynchronously with popup() or synchronously with exec().

Technically, a popup menu consists of a list of menu items. You add items with insertItem(). An item is either a string, a pixmap or a custom item that provides its own drawing function (see QCustomMenuItem). In addition, items can have an optional icon drawn on the very left side and an accelerator key such as "Ctrl+X". The accelerator can also be changed at run-time by holding the left mouse button over an item and pressing the new accelerator.

There are three kinds of menu items: separators, menu items that perform an action and menu items that show a submenu. Separators are inserted with insertSeparator(). For submenus, you pass a pointer to a QPopupMenu in your call to insertItem(). All other items are considered action items.

When inserting action items you usually specify a receiver and a slot. The receiver will be notifed whenever the item is selected. In addition, QPopupMenu provides two signals, activated() and highlighted(), which signal the identifier of the respective menu item. It is sometimes practical to connect several items to one slot. To distinguish between them, specify a slot that takes an integer argument and use setItemParameter() to associate a unique value with each item.

You clear a popup menu with clear() and remove single items with removeItem() or removeItemAt().

A popup menu can display check marks for certain items when enabled with setCheckable(TRUE). You check or uncheck items with setItemChecked().

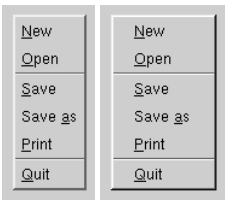
Items are either enabled or disabled. You toggle their state with setItemEnabled(). Just before a popup menu becomes visible, it emits the aboutToShow() signal. You can use this signal to set the correct enabled/disabled states of all menu items before the user sees it. The corresponding aboutToHide() signal is emitted when the menu hides again.

You can provide What's This? help for single menu items with setWhatsThis(). See QWhatsThis for general information about this kind of lightweight online help.

For ultimate flexibility, you can also add entire widgets as items into a popup menu (for example, a color selector).

A QPopupMenu can also provide a tear-off menu. A tear-off menu is a top-level window that contains a copy of the menu. This makes it possible for the user to "tear off" frequently used menus and position them in a convenient place on the screen. If you want that functionality for a certain menu, insert a tear-off handle with insertTearOffHandle(). When using tear-off menus, bear in mind that the concept isn't typically used on Microsoft Windows so users may not be familiar with it. Consider using a QToolBar instead.

menu/menu.cpp is a typical example of QMenuBar and QPopupMenu use.



See also QMenuBar [p. 69], GUI Design Handbook: Menu, Drop-Down and Pop-Up, Main Window and Related Classes and Basic Widgets.

Member Function Documentation

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

Constructs a popup menu with a the *parent* called *name*.

Although a popup menu is always a top-level widget, if a parent is passed the popup menu will be deleted when that parent is destroyed (as with any other QObject).

QPopupMenu::~QPopupMenu ()

Destroys the popup menu.

void QPopupMenu::aboutToHide () [signal]

This signal is emitted just before the popup menu is hidden after it has been displayed.

See also aboutToShow() [p. 109], setItemEnabled() [p. 91], setItemChecked() [p. 91], insertItem() [p. 84] and removeItem() [p. 90].

void QPopupMenu::aboutToShow () [signal]

This signal is emitted just before the popup menu is displayed. You can connect it to any slot that sets up the menu contents (e.g. to ensure that the right items are enabled).

See also aboutToHide() [p. 109], setItemEnabled() [p. 91], setItemChecked() [p. 91], insertItem() [p. 84] and removeItem() [p. 90].

Example: mdi/application.cpp.

QKeySequence QMenuData::accel (int id) const

Returns the accelerator key that has been defined for the menu item *id*, or 0 if it has no accelerator key.

See also setAccel() [p. 90], QAccel [Events, Actions, Layouts and Styles with Qt] and qnamespace.h.

void QPopupMenu::activated (int id) [signal]

This signal is emitted when a menu item is selected; *id* is the id of the selected item.

Normally, you connect each menu item to a single slot using QMenuData::insertItem(), but sometimes you will want to connect several items to a single slot (most often if the user selects from an array). This signal is useful in such cases.

See also highlighted() [p. 112] and QMenuData::insertItem() [p. 84].

Examples: grapher/grapher.cpp, helpviewer/helpwindow.cpp, qdir/qdir.cpp, qwerty/qwerty.cpp, scrollview/scrollview.cpp and showimg/showimg.cpp.

void QMenuData::changeItem (int id, const QString & text)

Changes the text of the menu item *id* to *text*. If the item has an icon, the icon remains unchanged.

See also text() [p. 92].

void QMenuData::changeItem (int id, const QPixmap & pixmap)

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

Changes the pixmap of the menu item *id* to the pixmap *pixmap*. If the item has an icon, the icon is unchanged.

See also pixmap() [p. 90].

void QMenuData::changeItem (int id, const QIconSet & icon, const QString & text)

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

Changes the iconset and text of the menu item *id* to the *icon* and *text* respectively.

See also pixmap() [p. 90].

void QMenuData::changeItem (int id, const QIconSet & icon, const QPixmap & pixmap)

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

Changes the iconset and pixmap of the menu item *id* to *icon* and *pixmap* respectively.

See also pixmap() [p. 90].

void QMenuData::clear ()

Removes all menu items.

See also removeItem() [p. 90] and removeItemAt() [p. 90].

Examples: mdi/application.cpp and qwerty/qwerty.cpp.

int QPopupMenu::columns () const [protected]

If a popup menu does not fit on the screen it lays itself out in multiple columns until it does fit.

This functions returns the number of columns necessary.

See also sizeHint [Widgets with Qt].

bool QMenuData::connectItem (int id, const QObject * receiver, const char * member)

Connects the menu item with identifier *id* to *receiver*'s *member* slot or signal.

The receiver's slot/signal is activated when the menu item is activated.

See also disconnectItem() [p. 84] and setItemParameter() [p. 91].

Example: menu/menu.cpp.

bool QMenuData::disconnectItem (int id, const QObject * receiver, const char * member)

Disconnects the *receiver*'s *member* from the menu item with identifier *id*.

All connections are removed when the menu data object is destroyed.

See also connectItem() [p. 83] and setItemParameter() [p. 91].

void QPopupMenu::drawContents (QPainter * p) [virtual protected]

Draws all menu items using painter *p*.

Reimplemented from QFrame [Widgets with Qt].

void QPopupMenu::drawItem (QPainter * p, int tab_, QMenuItem * mi, bool act, int x, int y, int w, int h) [protected]

Draws menu item *mi* in the area *x*, *y*, *w*, *h*, using painter *p*. The item is drawn active if *act* is TRUE or drawn inactive if *act* is FALSE. The rightmost *tab_* pixels are used for accelerator text.

See also QStyle::drawControl() [Events, Actions, Layouts and Styles with Qt].

int QPopupMenu::exec ()

Executes this popup synchronously.

This is equivalent to `exec(mapToGlobal(QPoint(0,0)))`. In most situations you'll want to specify the position yourself, for example at the current mouse position:

```
exec(QCursor::pos());
```

or aligned to a widget:

```
exec(somewidget.mapToGlobal(QPoint(0,0)));
```

Examples: fileiconview/qfileiconview.cpp and scribble/scribble.cpp.

int QPopupMenu::exec (const QPoint & pos, int indexAtPoint = 0)

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

Executes this popup synchronously.

Opens the popup menu so that the item number *indexAtPoint* will be at the specified *global* position *pos*. To translate a widget's local coordinates into global coordinates, use QWidget::mapToGlobal().

The return code is the id of the selected item in either the popup menu or one of its submenus, or -1 if no item is selected (normally because the user presses Escape).

Note that all signals are emitted as usual. If you connect a menu item to a slot and call the menu's exec(), you get the result both via the signal-slot connection and in the return value of exec().

Common usage is to position the popup at the current mouse position:

```
exec(QCursor::pos());
```

or aligned to a widget:

```
exec(somewidget.mapToGlobal(QPoint(0,0)));
```

When positioning a popup with exec() or popup(), bear in mind that you cannot rely on the popup menu's current size(). For performance reasons, the popup adapts its size only when necessary. So in many cases, the size before and after the show is different. Instead, use sizeHint(). It calculates the proper size depending on the menu's current contents.

See also popup() [p. 118] and sizeHint [Widgets with Qt].

void QPopupMenu::highlighted (int id) [signal]

This signal is emitted when a menu item is highlighted; *id* is the id of the highlighted item.

Normally, you connect each menu item to a single slot using QMenuData::insertItem(), but sometimes you will want to connect several items to a single slot (most often if the user selects from an array). This signal is useful in such cases.

See also activated() [p. 109] and QMenuData::insertItem() [p. 84].

QIconSet * QMenuData::iconSet (int id) const

Returns the icon set that has been set for menu item *id*, or 0 if no icon set has been set.

See also changeItem() [p. 82], text() [p. 92] and pixmap() [p. 90].

int QPopupMenu::idAt (int index) const

Returns the identifier of the menu item at position *index* in the internal list, or -1 if *index* is out of range.

See also QMenuData::setId() [p. 91] and QMenuData::indexOf() [p. 84].

Example: scrollview/scrollview.cpp.

int QPopupMenu::idAt (const QPoint & pos) const

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

Returns the id of the item at *pos*, or -1 if there is no item there or if it is a separator item.

int QMenuData::insertItem (const QString & text, const QObject * receiver, const char * member, const QKeySequence & accel = 0, int id = -1, int index = -1)

The family of insertItem() functions inserts menu items into a popup menu or a menu bar.

A menu item is usually either a text string or a pixmap, both with an optional icon or keyboard accelerator. For special cases it is also possible to insert custom items (see QCustomMenuItem) or even widgets into popup menus.

Some insertItem() members take a popup menu as an additional argument. Use this to insert submenus to existing menus or pulldown menus to a menu bar.

The number of insert functions may look confusing, but they are actually quite simple to use.

This default version inserts a menu item with the text *text*, the accelerator key *accel*, an id and an optional index and connects it to the slot *member* in the object *receiver*.

Example:

```
QMenuBar   *mainMenu = new QMenuBar;
QPopupMenu *fileMenu = new QPopupMenu;
fileMenu->insertItem( "New",  myView, SLOT(newFile()), CTRL+Key_N );
fileMenu->insertItem( "Open", myView, SLOT(open()),    CTRL+Key_O );
mainMenu->insertItem( "File", fileMenu );
```

Not all insert functions take an object/slot parameter or an accelerator key. Use connectItem() and setAccel() on these items.

If you need to translate accelerators, use tr() with a string description that use pass to the QKeySequence constructor:

```
fileMenu->insertItem( tr("Open"), myView, SLOT(open()),
                      tr("Ctrl+O") );
```

In the example above, pressing Ctrl+N or selecting "Open" from the menu activates the myView->open() function.

Some insert functions take a QIconSet parameter to specify the little menu item icon. Note that you can always pass a QPixmap object instead.

The *index* specifies the position in the menu. The menu item is appended at the end of the list if *index* is negative.

Note that keyboard accelerators in Qt are not application-global, instead they are bound to a certain top-level window. For example, accelerators in QPopupMenu items only work for menus that are associated with a certain window. This is true for popup menus that live in a menu bar since their accelerators will then be installed in the menu bar itself. This also applies to stand-alone popup menus that have a top-level widget in their parentWidget() chain. The menu will then install its accelerator object on that top-level widget. For all other cases use an independent QAccel object.

Warning: Be careful when passing a literal 0 to insertItem() because some C++ compilers choose the wrong overloaded function. Cast the 0 to what you mean, e.g. `(QObject*)0`.

Returns the allocated menu identifier number (*id* if *id* >= 0).

See also removeItem() [p. 90], changeItem() [p. 82], setAccel() [p. 90], connectItem() [p. 83], QAccel [Events, Actions, Layouts and Styles with Qt] and qnamespace.h.

Examples: addressbook/mainwindow.cpp, mdi/application.cpp, menu/menu.cpp, qwerty/qwerty.cpp, scrollview/scrollview.cpp and showimg/showimg.cpp.

int QMenuData::insertItem (const QIconSet & icon, const QString & text, const QObject * receiver, const char * member, const QKeySequence & accel = 0, int id = -1, int index = -1)

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

Inserts a menu item with icon *icon*, text *text*, accelerator *accel*, optional id *id*, and optional *index*. The menu item is connected it to the *receiver*'s *member* slot. The icon will be displayed to the left of the text in the item.

Returns the allocated menu identifier number (*id* if *id* >= 0).

See also removeItem() [p. 90], changeItem() [p. 82], setAccel() [p. 90], connectItem() [p. 83], QAccel [Events, Actions, Layouts and Styles with Qt] and qnamespace.h.

int QMenuData::insertItem (const QPixmap & pixmap, const QObject * receiver, const char * member, const QKeySequence & accel = 0, int id = -1, int index = -1)

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

Inserts a menu item with pixmap *pixmap*, accelerator *accel*, optional id *id*, and optional *index*. The menu item is connected it to the *receiver*'s *member* slot. The icon will be displayed to the left of the text in the item.

To look best when being highlighted as a menu item, the pixmap should provide a mask (see QPixmap::mask()).

Returns the allocated menu identifier number (*id* if *id* >= 0).

See also removeItem() [p. 90], changeItem() [p. 82], setAccel() [p. 90] and connectItem() [p. 83].

int QMenuData::insertItem (const QIconSet & icon, const QPixmap & pixmap, const QObject * receiver, const char * member, const QKeySequence & accel = 0, int id = -1, int index = -1)

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

Inserts a menu item with icon *icon*, pixmap *pixmap*, accelerator *accel*, optional id *id*, and optional *index*. The icon will be displayed to the left of the pixmap in the item. The item is connected to the *member* slot in the *receiver* object.

To look best when being highlighted as a menu item, the pixmap should provide a mask (see QPixmap::mask()).

Returns the allocated menu identifier number (*id* if *id* >= 0).

See also removeItem() [p. 90], changeItem() [p. 82], setAccel() [p. 90], connectItem() [p. 83], QAccel [Events, Actions, Layouts and Styles with Qt] and qnamespace.h.

int QMenuData::insertItem (const QString & text, int id = -1, int index = -1)

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

Inserts a menu item with text *text*, optional id *id*, and optional *index*.

Returns the allocated menu identifier number (*id* if *id* >= 0).

See also removeItem() [p. 90], changeItem() [p. 82], setAccel() [p. 90] and connectItem() [p. 83].

int QMenuData::insertItem (const QIconSet & icon, const QString & text, int id = -1, int index = -1)

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

Inserts a menu item with icon *icon*, text *text*, optional id *id*, and optional *index*. The icon will be displayed to the left of the text in the item.

Returns the allocated menu identifier number (*id* if *id* >= 0).

See also removeItem() [p. 90], changeItem() [p. 82], setAccel() [p. 90] and connectItem() [p. 83].

int QMenuData::insertItem (const QString & text, QPopupMenu * popup, int id = -1, int index = -1)

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

Inserts a menu item with text *text*, submenu *popup*, optional id *id*, and optional *index*.

The *popup* must be deleted by the programmer or by its parent widget. It is not deleted when this menu item is removed or when the menu is deleted.

Returns the allocated menu identifier number (*id* if *id* >= 0).

See also removeItem() [p. 90], changeItem() [p. 82], setAccel() [p. 90] and connectItem() [p. 83].

int QMenuData::insertItem (const QIconSet & icon, const QString & text, QPopupMenu * popup, int id = -1, int index = -1)

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

Inserts a menu item with icon *icon*, text *text*, submenu *popup*, optional id *id*, and optional *index*. The icon will be displayed to the left of the text in the item.

The *popup* must be deleted by the programmer or by its parent widget. It is not deleted when this menu item is removed or when the menu is deleted.

Returns the allocated menu identifier number (*id* if *id* >= 0).

See also removeItem() [p. 90], changeItem() [p. 82], setAccel() [p. 90] and connectItem() [p. 83].

int QMenuData::insertItem (const QPixmap & pixmap, int id = -1, int index = -1)

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

Inserts a menu item with pixmap *pixmap*, optional id *id*, and optional *index*.

To look best when being highlighted as a menu item, the pixmap should provide a mask (see QPixmap::mask()).

Returns the allocated menu identifier number (*id* if *id* >= 0).

See also removeItem() [p. 90], changeItem() [p. 82], setAccel() [p. 90] and connectItem() [p. 83].

int QMenuData::insertItem (const QIconSet & icon, const QPixmap & pixmap, int id = -1, int index = -1)

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

Inserts a menu item with icon *icon*, pixmap *pixmap*, optional id *id*, and optional *index*. The icon will be displayed to the left of the pixmap in the item.

Returns the allocated menu identifier number (*id* if *id* >= 0).

See also removeItem() [p. 90], changeItem() [p. 82], setAccel() [p. 90] and connectItem() [p. 83].

int QMenuData::insertItem (const QPixmap & pixmap, QPopupMenu * popup, int id = -1, int index = -1)

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

Inserts a menu item with pixmap *pixmap*, submenu *popup*, optional id *id*, and optional *index*.

The *popup* must be deleted by the programmer or by its parent widget. It is not deleted when this menu item is removed or when the menu is deleted.

Returns the allocated menu identifier number (*id* if *id* >= 0).

See also removeItem() [p. 90], changeItem() [p. 82], setAccel() [p. 90] and connectItem() [p. 83].

int QMenuData::insertItem (const QIconSet & icon, const QPixmap & pixmap, QPopupMenu * popup, int id = -1, int index = -1)

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

Inserts a menu item with icon *icon*, pixmap *pixmap* submenu *popup*, optional id *id*, and optional *index*. The icon will be displayed to the left of the pixmap in the item.

The *popup* must be deleted by the programmer or by its parent widget. It is not deleted when this menu item is removed or when the menu is deleted.

Returns the allocated menu identifier number (*id* if *id* >= 0).

See also removeItem() [p. 90], changeItem() [p. 82], setAccel() [p. 90] and connectItem() [p. 83].

int QMenuData::insertItem (QWidget * widget, int id = -1, int index = -1)

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

Inserts a menu item that consists of the widget *widget* with optional id *id*, and optional *index*.

Ownership of *widget* is transferred to the popup menu or to the menu bar.

Theoretically, any widget can be inserted into a popup menu. In practice, this only makes sense with certain widgets.

If a widget is not focus-enabled (see QWidget::isFocusEnabled()), the menu treats it as a separator; this means that the item is not selectable and will never get focus. In this way you can, for example, simply insert a QLabel if you need a popup menu with a title.

If the widget is focus-enabled it will get focus when the user traverses the popup menu with the arrow keys. If the widget does not accept ArrowUp and ArrowDown in its key event handler, the focus will move back to the menu when the respective arrow key is hit one more time. This works with a QLineEdit, for example. If the widget accepts the arrow key itself, it must also provide the possibility to put the focus back on the menu again by calling QWidget::focusNextPrevChild(). Futhermore, if the embedded widget closes the menu when the user made a selection, this can be done safely by calling

```
if ( isVisible() &&
     parentWidget() &&
     parentWidget()->inherits("QPopupMenu") )
    parentWidget()->close();
```

Returns the allocated menu identifier number (*id* if *id* >= 0).

See also removeItem() [p. 90].

int QMenuData::insertItem (const QIconSet & icon, QCustomMenuItem * custom, int id = -1, int index = -1)

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

Inserts a custom menu item *custom* with an *icon* and with optional id *id*, and optional *index*.

This only works with popup menus. It is not supported for menu bars. Ownership of *custom* is transferred to the popup menu.

If you want to connect a custom item to a certain slot, use connectItem().

Returns the allocated menu identifier number (*id* if *id* >= 0).

See also connectItem() [p. 83], removeItem() [p. 90] and QCustomMenuItem [p. 5].

int QMenuData::insertItem (QCustomMenuItem * custom, int id = -1, int index = -1)

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

Inserts a custom menu item *custom* with optional id *id*, and optional *index*.

This only works with popup menus. It is not supported for menu bars. Ownership of *custom* is transferred to the popup menu.

If you want to connect a custom item to a certain slot, use connectItem().

Returns the allocated menu identifier number (*id* if *id* >= 0).

See also connectItem() [p. 83], removeItem() [p. 90] and QCustomMenuItem [p. 5].

int QMenuData::insertSeparator (int index = -1)

Inserts a separator at position *index*. The separator becomes the last menu item if *index* is negative.

In a popup menu a separator is rendered as a horizontal line. In a Motif menu bar a separator is spacing, so the rest of the items (normally just "Help") are drawn right-justified. In a Windows menu bar separators are ignored (to comply with the Windows style guidelines).

Examples: addressbook/mainwindow.cpp, mdi/application.cpp, menu/menu.cpp, progress/progress.cpp, qwerty/qwerty.cpp, scrollview/scrollview.cpp and showimg/showimg.cpp.

int QPopupMenu::insertTearOffHandle (int id = -1, int index = -1)

Inserts a tear-off handle into the menu. A tear-off handle is a special menu item that creates a copy of the menu when the menu is selected. This "torn-off" copy lives in a separate window. It contains the same menu items as the original menu, with the exception of the tear-off handle.

The handle item is assigned the identifier *id* or an automatically generated identifier if *id* is < 0. The generated identifiers (negative integers) are guaranteed to be unique within the entire application.

The *index* specifies the position in the menu. The tear-off handle is appended at the end of the list if *index* is negative.

Example: menu/menu.cpp.

bool QPopupMenu::isCheckable () const

Returns TRUE if the display of check marks on menu items is enabled; otherwise returns FALSE. See the "checkable" [p. 120] property for details.

bool QMenuData::isItemChecked (int id) const

Returns TRUE if the menu item with the id *id* has been checked; otherwise returns FALSE.

See also setItemChecked() [p. 91].

Examples: progress/progress.cpp and showimg/showimg.cpp.

bool QMenuData::isItemEnabled (int id) const

Returns TRUE if the item with identifier *id* is enabled; otherwise returns FALSE

See also setItemEnabled() [p. 91].

int QPopupMenu::itemHeight (int row) const [protected]

Calculates the height in pixels of the item in row *row*.

int QPopupMenu::itemHeight (QMenuItem * mi) const [protected]

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

Calculates the height in pixels of the menu item *mi*.

int QMenuData::itemParameter (int id) const

Returns the parameter of the activation signal of item *id*.

If no parameter has been specified for this item with setItemParameter(), the value defaults to *id*.

See also connectItem() [p. 83], disconnectItem() [p. 84] and setItemParameter() [p. 91].

QPixmap * QMenuData::pixmap (int id) const

Returns the pixmap that has been set for menu item *id*, or 0 if no pixmap has been set.

See also changeItem() [p. 82], text() [p. 92] and iconSet() [p. 84].

void QPopupMenu::popup (const QPoint & pos, int indexAtPoint = 0)

Displays the popup menu so that the item number *indexAtPoint* will be at the specified *global* position *pos*. To translate a widget's local coordinates into global coordinates, use QWidget::mapToGlobal().

When positioning a popup with exec() or popup(), bear in mind that you cannot rely on the popup menu's current size(). For performance reasons, the popup adapts its size only when necessary, so in many cases, the size before and after the show is different. Instead, use sizeHint(). It calculates the proper size depending on the menu's current contents.

Examples: listviews/listviews.cpp and qtimage/qtimage.cpp.

void QMenuData::removeItem (int id)

Removes the menu item that has the identifier *id*.

See also removeItemAt() [p. 90] and clear() [p. 83].

void QMenuData::removeItemAt (int index)

Removes the menu item at position *index*.

See also removeItem() [p. 90] and clear() [p. 83].

void QMenuData::setAccel (const QKeySequence & key, int id)

Sets the accelerator key for the menu item *id* to *key*.

An accelerator key consists of a key code and a combination of the modifiers SHIFT, CTRL, ALT or UNICODE_ACCEL (OR'ed or added). The header file qnamespace.h contains a list of key codes.

Defining an accelerator key produces a text that is added to the menu item; for instance, CTRL + Key_O produces "Ctrl+O". The text is formatted differently for different platforms.

Note that keyboard accelerators in Qt are not application-global, instead they are bound to a certain top-level window. For example, accelerators in QPopupMenu items only work for menus that are associated with a certain window. This is true for popup menus that live in a menu bar since their accelerators will then be installed in the

menu bar itself. This also applies to stand-alone popup menus that have a top-level widget in their parentWidget() chain. The menu will then install its accelerator object on that top-level widget. For all other cases use an independent QAccel object.

Example:

```
QMenuBar   *mainMenu = new QMenuBar;
QPopupMenu *fileMenu = new QPopupMenu;       // file sub menu
fileMenu->insertItem( "Open Document", 67 ); // add "Open" item
fileMenu->setAccel( CTRL + Key_O, 67 );      // Control and O to open
fileMenu->insertItem( "Quit", 69 );          // add "Quit" item
fileMenu->setAccel( CTRL + ALT + Key_Delete, 69 );
mainMenu->insertItem( "File", fileMenu );    // add the file menu
```

If you need to translate accelerators, use QAccel::stringToKey():

```
fileMenu->setAccel( QAccel::stringToKey(tr("Ctrl+O")), 67 );
```

You can also specify the accelerator in the insertItem() function. You may prefer to use QAction to associate accelerators with menu items.

See also accel() [p. 82], insertItem() [p. 84], QAccel [Events, Actions, Layouts and Styles with Qt], qnamespace.h and QAction [Events, Actions, Layouts and Styles with Qt].

Example: menu/menu.cpp.

void QPopupMenu::setActiveItem (int i) [virtual]

Sets the currently active item to *i* and repaints as necessary.

void QPopupMenu::setCheckable (bool) [virtual]

Sets whether the display of check marks on menu items is enabled. See the "checkable" [p. 120] property for details.

void QMenuData::setItemChecked (int id, bool check)

If *check* is TRUE, checks the menu item with id *id*; otherwise unchecks the menu item with id *id*. Calls QPopupMenu::setCheckable(TRUE) if necessary.

See also isItemChecked() [p. 89].

Examples: grapher/grapher.cpp, mdi/application.cpp, menu/menu.cpp, progress/progress.cpp, scrollview/scrollview.cpp and showimg/showimg.cpp.

void QMenuData::setItemEnabled (int id, bool enable)

If *enable* is TRUE, enables the menu item with identifier *id*; otherwise disables the menu item with identifier *id*.

See also isItemEnabled() [p. 89].

Examples: mdi/application.cpp, menu/menu.cpp, progress/progress.cpp and showimg/showimg.cpp.

bool QMenuData::setItemParameter (int id, int param)

Sets the parameter of the activation signal of item *id* to *param*.

If any receiver takes an integer parameter, this value is passed.

See also connectItem() [p. 83], disconnectItem() [p. 84] and itemParameter() [p. 89].

Example: mdi/application.cpp.

void QMenuData::setWhatsThis (int id, const QString & text)

Sets *text* as What's This help for the menu item with identifier *id*.

See also whatsThis() [p. 92].

Examples: application/application.cpp and mdi/application.cpp.

QString QMenuData::text (int id) const

Returns the text that has been set for menu item *id*, or a null string if no text has been set.

See also changeItem() [p. 82], pixmap() [p. 90] and iconSet() [p. 84].

Examples: qdir/qdir.cpp and showimg/showimg.cpp.

void QPopupMenu::updateItem (int id) [virtual]

Updates the item with identity *id*.

Reimplemented from QMenuData [p. 92].

QString QMenuData::whatsThis (int id) const

Returns the What's This help text for the item with identifier *id* or QString::null if no text has yet been defined.

See also setWhatsThis() [p. 91].

Property Documentation

bool checkable

This property holds whether the display of check marks on menu items is enabled.

When TRUE, the display of check marks on menu items is enabled. Checking is always enabled when in Windows-style.

See also QMenuData::setItemChecked() [p. 91].

Set this property's value with setCheckable() and get this property's value with isCheckable().

QProgressDialog Class Reference

The QProgressDialog class provides feedback on the progress of a slow operation.

`#include <qprogressdialog.h>`

Inherits QDialog [p. 10].

Public Members

- **QProgressDialog** (QWidget * creator = 0, const char * name = 0, bool modal = FALSE, WFlags f = 0)
- **QProgressDialog** (const QString & labelText, const QString & cancelButtonText, int totalSteps, QWidget * creator = 0, const char * name = 0, bool modal = FALSE, WFlags f = 0)
- **~QProgressDialog** ()
- void **setLabel** (QLabel * label)
- void **setCancelButton** (QPushButton * cancelButton)
- void **setBar** (QProgressBar * bar)
- bool **wasCancelled** () const
- int **totalSteps** () const
- int **progress** () const
- virtual QSize **sizeHint** () const
- QString **labelText** () const
- void **setAutoReset** (bool b)
- bool **autoReset** () const
- void **setAutoClose** (bool b)
- bool **autoClose** () const
- int **minimumDuration** () const

Public Slots

- void **cancel** ()
- void **reset** ()
- void **setTotalSteps** (int totalSteps)
- void **setProgress** (int progress)
- void **setLabelText** (const QString &)
- void **setCancelButtonText** (const QString & cancelButtonText)
- void **setMinimumDuration** (int ms)

Signals

- void **cancelled** ()

Properties

- bool **autoClose** — whether the dialog gets hidden by reset()
- bool **autoReset** — whether the progress dialog calls reset() as soon as progress() equals totalSteps()
- QString **labelText** — the label's text
- int **minimumDuration** — the time that the progress should run for before the dialog opens
- int **progress** — the current amount of progress made
- int **totalSteps** — the total number of steps
- bool **wasCancelled** — whether the dialog was cancelled *(read only)*

Protected Slots

- void **forceShow** ()

Detailed Description

The QProgressDialog class provides feedback on the progress of a slow operation.

A progress dialog is used to give the user an indication of how long an operation is going to take to perform, and to indicate that the application has not frozen. It can also gives the user an opportunity to abort the operation.

A common problem with progress dialogs is that it is difficult to know when to use them; operations take different amounts of time on different computer hardware. QProgressDialog offers a solution to this problem: it estimates the time the operation will take (based on time for steps), and only shows itself if that estimate is beyond minimumDuration() (4 seconds by default).

Use setTotalSteps() (or the constructor) to set the number of "steps" in the operation and call setProgress() as the operation progresses. The step value can be chosen arbitrarily. It can be the number of files copied, the number of bytes received, the number of iterations through the main loop of your algorithm, or some other suitable unit. Progress starts at 0, and the progress dialog shows that the operation has completeed when you call setProgress() with totalSteps() as argument.

The dialog automatically resets and hides itself at the end of the operation. Use setAutoReset() and setAutoClose() to change this behavior.

There are two ways of using QProgressDialog: modal and non-modal.

Using a modal QProgressDialog is simpler for the programmer, but you have to call qApp->processEvents() to keep the event loop running to ensure that the application doesn't freeze. Do the operation in a loop, call setProgress() at intervals, and check for cancellation with wasCancelled(). For example:

```
QProgressDialog progress( "Copying files...", "Abort Copy", numFiles,
                          this, "progress", TRUE );
for ( int i = 0; i processEvents();

    if ( progress.wasCancelled() )
        break;
    //... copy one file
}
progress.setProgress( numFiles );
```

A non-modal progress dialog is suitable for operations that take place in the background, where the user is able to interact with the application. Such operations are typically based on QTimer (or QObject::timerEvent()), QSocketNotifier, or QUrlOperator; or performed in a separate thread. A QProgressBar in the status bar of your main window is often an alternative to a non-modal progress dialog.

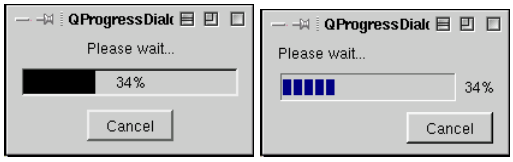
You need an event loop to be running. Connect the cancelled() signal to a slot that stops the operation, and call setProgress() at intervals. For example:

```
Operation::Operation( QObject *parent = 0 )
    : QObject( parent ), steps( 0 )
{
    pd = new QProgressDialog( "Operation in progress.", "Cancel", 100 );
    connect( pd, SIGNAL(cancelled()), this, SLOT(cancel()) );
    t = new QTimer( this );
    connect( t, SIGNAL(timeout()), this, SLOT(perform()) );
    t->start( 0 );
}

void Operation::perform()
{
    pd->setProgress( steps );
    //... perform one percent of the operation
    steps++;
    if ( steps > pd->totalSteps() )
        t->stop();
}

void Operation::cancel()
{
    t->stop();
    //... cleanup
}
```

In both modes the progress dialog may be customized by replacing the child widgets with custom widgets by using setLabel(), setBar(), and setCancelButton(). The functions setLabelText() and setCancelButtonText() set the texts shown.



See also QDialog [p. 10], QProgressBar [Widgets with Qt], GUI Design Handbook: Progress Indicator and Dialog Classes.

Member Function Documentation

QProgressDialog::QProgressDialog (QWidget * creator = 0, const char * name = 0, bool modal = FALSE, WFlags f = 0)

Constructs a progress dialog.

Default settings:

- The label text is empty.
- The cancel button text is "Cancel".
- The total number of steps is 100.

The top level parent of the *creator* widget becomes the parent of the dialog. The *name*, *modal*, and the widget flags, *f*, are passed to the QDialog::QDialog() constructor. Note that if *modal* is FALSE (the default), you must have an

event loop proceeding for any redrawing of the dialog to occur. If *modal* is TRUE, the dialog ensures that events are processed when needed.

See also labelText [p. 127], setLabel() [p. 126], setCancelButtonText() [p. 126], setCancelButton() [p. 125] and totalSteps [p. 127].

QProgressDialog::QProgressDialog (const QString & labelText, const QString & cancelButtonText, int totalSteps, QWidget * creator = 0, const char * name = 0, bool modal = FALSE, WFlags f = 0)

Constructs a progress dialog.

The *labelText* is text used to remind the user what is progressing.

The *cancelButtonText* is the text to display on the cancel button, or 0 if no cancel button is to be shown.

The *totalSteps* is the total number of steps in the operation of which this progress dialog shows the progress. For example, if the operation is to examine 50 files, this value would be 50. Before examining the first file, call setProgress(0). As each file is processed call setProgress(1), setProgress(2), etc., finally calling setProgress(50) after examining the last file.

The *name*, *modal*, and widget flags, *f*, are passed to the QDialog::QDialog() constructor. Note that if *modal* is FALSE (the default), you will need to have an event loop proceeding for any redrawing of the dialog to occur. If *modal* is TRUE, the dialog ensures that events are processed when needed.

The *creator* argument is the widget to use as the dialog's parent. If *creator* is not a top level widget the argument passed on to the QDialog constructor will be 0.

See also labelText [p. 127], setLabel() [p. 126], setCancelButtonText() [p. 126], setCancelButton() [p. 125] and totalSteps [p. 127].

QProgressDialog::~QProgressDialog ()

Destroys the progress dialog.

bool QProgressDialog::autoClose () const

Returns TRUE if the dialog gets hidden by reset(); otherwise returns FALSE. See the "autoClose" [p. 127] property for details.

bool QProgressDialog::autoReset () const

Returns TRUE if the progress dialog calls reset() as soon as progress() equals totalSteps(); otherwise returns FALSE. See the "autoReset" [p. 127] property for details.

void QProgressDialog::cancel () [slot]

Resets the progress dialog. wasCancelled() becomes TRUE until the progress dialog is reset. The progress dialog becomes hidden.

void QProgressDialog::cancelled () [signal]

This signal is emitted when the cancel button is clicked. It is connected to the cancel() slot by default.

See also wasCancelled [p. 128].

Example: progress/progress.cpp.

void QProgressDialog::forceShow () [protected slot]

Shows the dialog if it is still hidden after the algorithm has been started and the minimumDuration is over.

See also minimumDuration [p. 127].

QString QProgressDialog::labelText () const

Returns the label's text. See the "labelText" [p. 127] property for details.

int QProgressDialog::minimumDuration () const

Returns the time that the progress should run for before the dialog opens. See the "minimumDuration" [p. 127] property for details.

int QProgressDialog::progress () const

Returns the current amount of progress made. See the "progress" [p. 127] property for details.

void QProgressDialog::reset () [slot]

Resets the progress dialog. The progress dialog becomes hidden if autoClose() is TRUE.

See also autoClose [p. 127] and autoReset [p. 127].

void QProgressDialog::setAutoClose (bool b)

Sets whether the dialog gets hidden by reset() to *b*. See the "autoClose" [p. 127] property for details.

void QProgressDialog::setAutoReset (bool b)

Sets whether the progress dialog calls reset() as soon as progress() equals totalSteps() to *b*. See the "autoReset" [p. 127] property for details.

void QProgressDialog::setBar (QProgressBar * bar)

Sets the progress bar widget to *bar*. The progress dialog resizes to fit. The progress dialog takes ownership of the progress *bar* which will be deleted when necessary.

void QProgressDialog::setCancelButton (QPushButton * cancelButton)

Sets the cancel button to the push button, *cancelButton*. The progress dialog takes ownership of this button which will be deleted when necessary, so do not pass the address of an object that is on the stack, i.e. use new() to create the button.

See also setCancelButtonText() [p. 126].

void QProgressDialog::setCancelButtonText (const QString & cancelButtonText) [slot]

Sets the cancel button's text to *cancelButtonText*.

See also setCancelButton() [p. 125].

void QProgressDialog::setLabel (QLabel * label)

Sets the label to *label*. The progress dialog resizes to fit. The label becomes owned by the progress dialog and will be deleted when necessary, so do not pass the address of an object on the stack.

See also labelText [p. 127].

Example: progress/progress.cpp.

void QProgressDialog::setLabelText (const QString &) [slot]

Sets the label's text. See the "labelText" [p. 127] property for details.

void QProgressDialog::setMinimumDuration (int ms) [slot]

Sets the time that the progress should run for before the dialog opens to *ms*. See the "minimumDuration" [p. 127] property for details.

void QProgressDialog::setProgress (int progress) [slot]

Sets the current amount of progress made to *progress*. See the "progress" [p. 127] property for details.

void QProgressDialog::setTotalSteps (int totalSteps) [slot]

Sets the total number of steps to *totalSteps*. See the "totalSteps" [p. 127] property for details.

QSize QProgressDialog::sizeHint () const [virtual]

Returns a size that fits the contents of the progress dialog. The progress dialog resizes itself as required, so you should not need to call this yourself.

int QProgressDialog::totalSteps () const

Returns the total number of steps. See the "totalSteps" [p. 127] property for details.

bool QProgressDialog::wasCancelled () const

Returns TRUE if the dialog was cancelled; otherwise returns FALSE. See the "wasCancelled" [p. 128] property for details.

Property Documentation

bool autoClose

This property holds whether the dialog gets hidden by reset().

The default is TRUE.

See also autoReset [p. 127].

Set this property's value with setAutoClose() and get this property's value with autoClose().

bool autoReset

This property holds whether the progress dialog calls reset() as soon as progress() equals totalSteps().

The default is TRUE.

See also autoClose [p. 127].

Set this property's value with setAutoReset() and get this property's value with autoReset().

QString labelText

This property holds the label's text.

The default text is QString::null.

Set this property's value with setLabelText() and get this property's value with labelText().

int minimumDuration

This property holds the time that the progress should run for before the dialog opens.

The dialog will not appear if the anticipated duration of the progressing task is less than the minimum duration.

If set to 0, the dialog is always shown as soon as any progress is set. The default is 4000.

Set this property's value with setMinimumDuration() and get this property's value with minimumDuration().

int progress

This property holds the current amount of progress made.

For the progress dialog to work as expected, you should initially set this property to 0 and finally set it to QProgressDialog::totalSteps(); you can call setProgress() any number of times in-between.

Warning: If the progress dialog is modal (see QProgressDialog::QProgressDialog()), this function calls QApplication::processEvents(), so take care that this does not cause undesirable re-entrancy in your code. For example, don't use a QProgressDialog inside a paintEvent()!

See also totalSteps [p. 127].

Set this property's value with setProgress() and get this property's value with progress().

int totalSteps

This property holds the total number of steps.

The default is 0.

Set this property's value with setTotalSteps() and get this property's value with totalSteps().

bool wasCancelled

This property holds whether the dialog was cancelled.

Get this property's value with wasCancelled().

See also progress [p. 127].

QTabDialog Class Reference

The QTabDialog class provides a stack of tabbed widgets.

`#include <qtabdialog.h>`

Inherits QDialog [p. 10].

Public Members

- **QTabDialog** (QWidget * parent = 0, const char * name = 0, bool modal = FALSE, WFlags f = 0)
- **~QTabDialog** ()
- virtual void **setFont** (const QFont & font)
- void **addTab** (QWidget * child, const QString & label)
- void **addTab** (QWidget * child, const QIconSet & iconset, const QString & label)
- void **addTab** (QWidget * child, QTab * tab)
- void **insertTab** (QWidget * child, const QString & label, int index = -1)
- void **insertTab** (QWidget * child, const QIconSet & iconset, const QString & label, int index = -1)
- void **insertTab** (QWidget * child, QTab * tab, int index = -1)
- void **changeTab** (QWidget * w, const QString & label)
- void **changeTab** (QWidget * w, const QIconSet & iconset, const QString & label)
- bool **isTabEnabled** (QWidget * w) const
- void **setTabEnabled** (QWidget * w, bool enable)
- bool isTabEnabled (const char * name) const *(obsolete)*
- void setTabEnabled (const char * name, bool enable) *(obsolete)*
- void **showPage** (QWidget * w)
- void **removePage** (QWidget * w)
- QString **tabLabel** (QWidget * w)
- QWidget * **currentPage** () const
- void **setDefaultButton** (const QString & text)
- void **setDefaultButton** ()
- bool **hasDefaultButton** () const
- void **setHelpButton** (const QString & text)
- void **setHelpButton** ()
- bool **hasHelpButton** () const
- void **setCancelButton** (const QString & text)
- void **setCancelButton** ()
- bool **hasCancelButton** () const
- void **setApplyButton** (const QString & text)
- void **setApplyButton** ()
- bool **hasApplyButton** () const
- void **setOkButton** (const QString & text)
- void **setOkButton** ()
- bool **hasOkButton** () const

Signals

- void **aboutToShow** ()
- void **applyButtonPressed** ()
- void **cancelButtonPressed** ()
- void **defaultButtonPressed** ()
- void **helpButtonPressed** ()
- void **currentChanged** (QWidget *)
- void selected (const QString &) *(obsolete)*

Protected Members

- void **setTabBar** (QTabBar * tb)
- QTabBar * **tabBar** () const

Detailed Description

The QTabDialog class provides a stack of tabbed widgets.

A tabbed dialog is one in which several "pages" are available. The user selects which page to see and use by clicking on its tab or by pressing the indicated Alt+*letter* key combination.

QTabDialog provides a tab bar consisting of single row of tabs at the top; each tab has an associated widget which is that tab's "page". In addition, QTabDialog provides an OK button and the following optional buttons: Apply, Cancel, Defaults and Help.

QTabDialog doesn't provide for tabs on the sides or bottom, nor can you set or retrieve the visible page. If you need more functionality than QTabDialog provides, consider creating a QDialog and using a QTabBar with QTabWidgets.

The normal way to use QTabDialog is to do the following in the constructor:

1. Create a QTabDialog.
2. Create a QWidget for each of the pages in the tab dialog, insert children into it, set up geometry management for it, and use addTab() (or insertTab()) to set up a tab and keyboard accelerator for it.
3. Set up the buttons for the tab dialog using setOkButton(), setApplyButton(), setDefaultsButton(), setCancelButton() and setHelpButton().
4. Connect to the signals and slots.

If you don't call addTab() the page you have created will not be visible. Don't confuse the object name you supply to the QWidget constructor and the tab label you supply to addTab(); addTab() takes a name that indicates an accelerator and is meaningful and descriptive to the user, whereas the widget name is used primarily for debugging.

Almost all applications have to connect the applyButtonPressed() signal to something. applyButtonPressed() is emitted when either OK or Apply is clicked, and your slot must copy the dialog's state into the application.

There are also several other signals which may be useful:

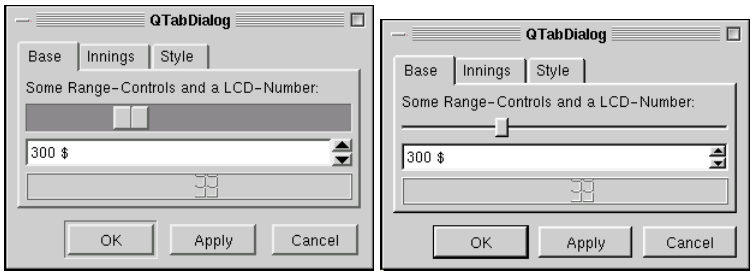
- cancelButtonPressed() is emitted when the user clicks Cancel.
- defaultButtonPressed() is emitted when the user clicks Defaults; the slot it is connected to should reset the state of the dialog to the application defaults.
- helpButtonPressed() is emitted when the user clicks Help.
- aboutToShow() is emitted at the start of show(); if there is any chance that the state of the application may change between the creation of the tab dialog and the time show() is called, you must connect this signal to a slot that resets the state of the dialog.

- currentChanged() is emitted when the user selects some page.

Each tab is either enabled or disabled at any given time (see setTabEnabled()). If a tab is enabled the tab text is drawn in black and the user can select that tab. If it is disabled the tab is drawn in a different way and the user cannot select that tab. Note that even if a tab is disabled, the page can still be visible; for example, if all of the tabs happen to be disabled.

You can change a tab's label and iconset using changeTab(). A tab page can be removed with removePage() and shown with showPage(). The current page is given by currentPage().

Most of the functionality in QTabDialog is provided by a QTabWidget.



See also QDialog [p. 10] and Dialog Classes.

Member Function Documentation

QTabDialog::QTabDialog (QWidget * parent = 0, const char * name = 0, bool modal = FALSE, WFlags f = 0)

Constructs a QTabDialog with only an OK button. The *parent*, *name*, *modal* and widget flag, *f*, arguments are passed on to the QDialog constructor.

QTabDialog::~QTabDialog ()

Destroys the tab dialog.

void QTabDialog::aboutToShow () [signal]

This signal is emitted by show() when it is time to set the state of the dialog's contents. The dialog should reflect the current state of the application when it appears; if there is any possibility that the state of the application may change between the time you call QTabDialog::QTabDialog() and QTabDialog::show(), you should set the dialog's state in a slot and connect this signal to it.

This applies mainly to QTabDialog objects that are kept around hidden, rather than being created, shown, and deleted afterwards.

See also applyButtonPressed() [p. 132], show() [p. 14] and cancelButtonPressed() [p. 132].

void QTabDialog::addTab (QWidget * child, const QString & label)

Adds another tab and page to the tab view.

The new page is *child*; the tab's label is *label*. Note the difference between the widget name (which you supply to widget constructors and to setTabEnabled(), for example) and the tab label. The name is internal to the program and invariant, whereas the label is shown on-screen and may vary according to language and other factors.

If the tab's *label* contains an ampersand, the letter following the ampersand is used as an accelerator for the tab, e.g. if the label is "Bro&wse" then Alt+W becomes an accelerator which will move the focus to this tab.

If you call addTab() after show() the screen will flicker and the user may be confused.

See also insertTab() [p. 134].

void QTabDialog::addTab (QWidget * child, const QIconSet & iconset, const QString & label)

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

This version of the function shows the *iconset* as well as the *label* on the tab of *child*.

void QTabDialog::addTab (QWidget * child, QTab * tab)

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

This is a lower-level method for adding tabs, similar to the other addTab() method. It is useful if you are using setTabBar() to set a QTabBar subclass with an overridden QTabBar::paint() routine for a subclass of QTab.

The *child* is the widget to be placed on the new tab page. The *tab* is the tab to display on the tab page — normally this shows a label or an icon that identifies the tab page.

void QTabDialog::applyButtonPressed () [signal]

This signal is emitted when the Apply or OK button is clicked.

It should be connected to a slot (or several slots) that change the application's state according to the state of the dialog.

See also cancelButtonPressed() [p. 132], defaultButtonPressed() [p. 133] and setApplyButton() [p. 135].

void QTabDialog::cancelButtonPressed () [signal]

This signal is emitted when the Cancel button is clicked. It is automatically connected to QDialog::reject(), which will hide the dialog.

The Cancel button should not change the application's state at all, so you should generally not need to connect it to any slot.

See also applyButtonPressed() [p. 132], defaultButtonPressed() [p. 133] and setCancelButton() [p. 135].

void QTabDialog::changeTab (QWidget * w, const QIconSet & iconset, const QString & label)

Changes tab page *w*'s iconset to *iconset* and label to *label*.

void QTabDialog::changeTab (QWidget * w, const QString & label)

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

Defines a new *label* for the tab of page *w*

void QTabDialog::currentChanged (QWidget *) [signal]

This signal is emitted whenever the current page changes.

See also currentPage() [p. 133], showPage() [p. 137] and tabLabel() [p. 138].

QWidget * QTabDialog::currentPage () const

Returns a pointer to the page currently being displayed by the tab dialog. The tab dialog does its best to make sure that this value is never 0 (but if you try hard enough, it can be).

void QTabDialog::defaultButtonPressed () [signal]

This signal is emitted when the Defaults button is pressed. It should reset the dialog (but not the application) to the "factory defaults".

The application's state should not be changed until the user clicks Apply or OK.

See also applyButtonPressed() [p. 132], cancelButtonPressed() [p. 132] and setDefaultButton() [p. 136].

bool QTabDialog::hasApplyButton () const

Returns TRUE if the tab dialog has an Apply button; otherwise returns FALSE.

See also setApplyButton() [p. 135], applyButtonPressed() [p. 132], hasCancelButton() [p. 133] and hasDefaultButton() [p. 133].

bool QTabDialog::hasCancelButton () const

Returns TRUE if the tab dialog has a Cancel button; otherwise returns FALSE.

See also setCancelButton() [p. 135], cancelButtonPressed() [p. 132], hasApplyButton() [p. 133] and hasDefaultButton() [p. 133].

bool QTabDialog::hasDefaultButton () const

Returns TRUE if the tab dialog has a Defaults button; otherwise returns FALSE.

See also setDefaultButton() [p. 136], defaultButtonPressed() [p. 133], hasApplyButton() [p. 133] and hasCancelButton() [p. 133].

bool QTabDialog::hasHelpButton () const

Returns TRUE if the tab dialog has a Help button; otherwise returns FALSE.

See also setHelpButton() [p. 136], helpButtonPressed() [p. 134], hasApplyButton() [p. 133] and hasCancelButton() [p. 133].

bool QTabDialog::hasOkButton () const

Returns TRUE if the tab dialog has an OK button; otherwise returns FALSE.

See also setOkButton() [p. 136], hasApplyButton() [p. 133], hasCancelButton() [p. 133] and hasDefaultButton() [p. 133].

void QTabDialog::helpButtonPressed () [signal]

This signal is emitted when the Help button is pressed. It could be used to present information about how to use the dialog.

See also applyButtonPressed() [p. 132], cancelButtonPressed() [p. 132] and setHelpButton() [p. 136].

void QTabDialog::insertTab (QWidget * child, const QString & label, int index = -1)

Inserts another tab and page to the tab view.

The new page is *child*; the tab's label is *label*. Note the difference between the widget name (which you supply to widget constructors and to setTabEnabled(), for example) and the tab label. The name is internal to the program and invariant, whereas the label is shown on-screen and may vary according to language and other factors.

If the tab's *label* contains an ampersand, the letter following the ampersand is used as an accelerator for the tab, e.g. if the label is "Bro&wse" then Alt+W becomes an accelerator which will move the focus to this tab.

If *index* is not specified, the tab is simply added. Otherwise it is inserted at the specified position.

If you call insertTab() after show(), the screen will flicker and the user may be confused.

See also addTab() [p. 131].

void QTabDialog::insertTab (QWidget * child, const QIconSet & iconset, const QString & label, int index = -1)

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

This version of the function shows the *iconset* as well as the *label* on the tab of *child*.

void QTabDialog::insertTab (QWidget * child, QTab * tab, int index = -1)

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

This is a lower-level method for inserting tabs, similar to the other insertTab() method. It is useful if you are using setTabBar() to set a QTabBar subclass with an overridden QTabBar::paint() routine for a subclass of QTab.

The *child* is the widget to be placed on the new tab page. The *tab* is the tab to display on the tab page — normally this shows a label or an icon that identifies the tab page. The *index* is the position where this tab page should be inserted.

bool QTabDialog::isTabEnabled (QWidget * w) const

Returns TRUE if the page *w* is enabled; otherwise returns FALSE.

See also setTabEnabled() [p. 137] and QWidget::enabled [Widgets with Qt].

bool QTabDialog::isTabEnabled (const char * name) 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 page with object name *name* is enabled and FALSE if it is disabled.

If *name* is 0 or not the name of any of the pages, isTabEnabled() returns FALSE.

See also setTabEnabled() [p. 137] and QWidget::enabled [Widgets with Qt].

void QTabDialog::removePage (QWidget * w)

Removes page *w* from this stack of widgets. Does not delete *w*.

See also showPage() [p. 137], QTabWidget::removePage() [Widgets with Qt] and QWidgetStack::removeWidget() [Widgets with Qt].

void QTabDialog::selected (const QString &) [signal]

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

This signal is emitted whenever a tab is selected (raised), including during the first show().

See also raise() [Widgets with Qt].

void QTabDialog::setApplyButton (const QString & text)

Adds an Apply button to the dialog. The button's text is set to *text*.

The Apply button should apply the current settings in the dialog box to the application while keeping the dialog visible.

When Apply is clicked, the applyButtonPressed() signal is emitted.

If *text* is a null string, no button is shown.

See also setCancelButton() [p. 135], setDefaultButton() [p. 136] and applyButtonPressed() [p. 132].

void QTabDialog::setApplyButton ()

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

Adds an Apply button to the dialog. The button's text is set to a localizable "Apply".

void QTabDialog::setCancelButton (const QString & text)

Adds a Cancel button to the dialog. The button's text is set to *text*.

The cancel button should always return the application to the state it was in before the tab view popped up, or if the user has clicked Apply, back to the state immediately after the last Apply.

When Cancel is clicked, the cancelButtonPressed() signal is emitted. The dialog is closed at the same time.

If *text* is a null string, no button is shown.

See also setApplyButton() [p. 135], setDefaultButton() [p. 136] and cancelButtonPressed() [p. 132].

void QTabDialog::setCancelButton ()

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

Adds a Cancel button to the dialog. The button's text is set to a localizable "Cancel".

void QTabDialog::setDefaultButton (const QString & text)

Adds a Defaults button to the dialog. The button's text is set to *text*.

The Defaults button should set the dialog (but not the application) back to the application defaults.

When Defaults is clicked, the defaultButtonPressed() signal is emitted.

If *text* is a null string, no button is shown.

See also setApplyButton() [p. 135], setCancelButton() [p. 135] and defaultButtonPressed() [p. 133].

void QTabDialog::setDefaultButton ()

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

Adds a Defaults button to the dialog. The button's text is set to a localizable "Defaults".

void QTabDialog::setFont (const QFont & font) [virtual]

Sets the font for the tabs to *font*.

If the widget is visible, the display is updated with the new font immediately. There may be some geometry changes, depending on the size of the old and new fonts.

Reimplemented from QWidget [Widgets with Qt].

void QTabDialog::setHelpButton (const QString & text)

Adds a Help button to the dialog. The button's text is set to *text*.

When Help is clicked, the helpButtonPressed() signal is emitted.

If *text* is a null string, no button is shown.

See also setApplyButton() [p. 135], setCancelButton() [p. 135] and helpButtonPressed() [p. 134].

void QTabDialog::setHelpButton ()

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

Adds a Help button to the dialog. The button's text is set to a localizable "Help".

void QTabDialog::setOkButton (const QString & text)

Adds an OK button to the dialog and sets the button's text to *text*.

When the OK button is clicked, the applyButtonPressed() signal is emitted, and the current settings in the dialog box should be applied to the application. The dialog then closes.

If *text* is a null string, no button is shown.

See also setCancelButton() [p. 135], setDefaultButton() [p. 136] and applyButtonPressed() [p. 132].

void QTabDialog::setOkButton ()

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

Adds an OK button to the dialog. The button's text is set to a localizable "OK".

void QTabDialog::setTabBar (QTabBar * tb) [protected]

Replaces the QTabBar heading the dialog by the given tab bar, *tb*. Note that this must be called *before* any tabs have been added, or the behavior is undefined.

See also tabBar() [p. 137].

void QTabDialog::setTabEnabled (QWidget * w, bool enable)

If *enable* is TRUE the page *w* is enabled; otherwise *w* is disabled. The page's tab is redrawn appropriately.

QTabWidget uses QWidget::setEnabled() internally, rather than keeping a separate flag.

Note that even a disabled tab/page may be visible. If the page is already visible QTabWidget will not hide it; if all the pages are disabled QTabWidget will show one of them.

See also isTabEnabled() [p. 134] and QWidget::enabled [Widgets with Qt].

void QTabDialog::setTabEnabled (const char * name, bool enable)

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

Finds the page with object name *name*, enables/disables it according to the value of *enable* and redraws the page's tab appropriately.

QTabDialog uses QWidget::setEnabled() internally, rather than keeping a separate flag.

Note that even a disabled tab/page may be visible. If the page is already visible QTabDialog will not hide it; if all the pages are disabled QTabDialog will show one of them.

The object name is used (rather than the tab label) because the tab text may not be invariant in multi-language applications.

See also isTabEnabled() [p. 134] and QWidget::enabled [Widgets with Qt].

void QTabDialog::showPage (QWidget * w)

Ensures that widget *w* is shown. This is useful mainly for accelerators.

Warning: If used carelessly, this function can easily surprise or confuse the user.

See also QTabBar::currentTab [Widgets with Qt].

QTabBar * QTabDialog::tabBar () const [protected]

Returns the currently set QTabBar.

See also setTabBar() [p. 137].

QString QTabDialog::tabLabel (QWidget * w)

Returns the text in the tab for page *w*.

QToolBar Class Reference

The QToolBar class provides a movable panel containing widgets such as tool buttons.

`#include <qtoolbar.h>`

Inherits QDockWindow [p. 21].

Public Members

- QToolBar (const QString & label, QMainWindow *, ToolBarDock = DockTop, bool newLine = FALSE, const char * name = 0) *(obsolete)*
- **QToolBar** (const QString & label, QMainWindow * mainWindow, QWidget * parent, bool newLine = FALSE, const char * name = 0, WFlags f = 0)
- **QToolBar** (QMainWindow * parent = 0, const char * name = 0)
- void **addSeparator** ()
- QMainWindow * **mainWindow** () const
- virtual void **setStretchableWidget** (QWidget * w)
- virtual void **setLabel** (const QString &)
- QString **label** () const
- virtual void **clear** ()

Properties

- QString **label** — the label of the toolbar

Detailed Description

The QToolBar class provides a movable panel containing widgets such as tool buttons.

A toolbar is a panel that contains a set of controls, usually represented by small icons. It's purpose is to provide quick access to frequently used commands or options. Within a QMainWindow the user can drag toolbars within and between the dock areas. Toolbars can also be dragged out of any dock area to float freely as top level windows.

QToolBar is a specialization of QDockWindow, and so provides all the functionality of a QDockWindow.

To use QToolBar you simply create a QToolBar as a child of a QMainWindow, create a number of QToolButton widgets (or other widgets) in left to right (or top to bottom) order and call addSeparator() when you want a separator. When a toolbar is floated the caption used is the label given in the constructor call. This can be changed with setLabel().

```
QToolBar * fileTools = new QToolBar( this, "file operations" );
fileTools->setLabel( "File Operations" );
```

```
fileOpenAction->addTo( fileTools );
fileSaveAction->addTo( fileTools );
```

This extract from the application/application.cpp example shows the creation of a new toolbar as a child of a QMainWindow and adding two QActions.

You may use most widgets within a toolbar, with QToolButton and QComboBox being the most common.

QToolBars, like QDockWindows, are located in QDockAreas or float as top level windows. QMainWindow provides four QDockAreas (top, left, right and bottom). When you create a new toolbar (as in the example above) as a child of a QMainWindow the toolbar will be added to the top dock area. You can move it to another dock area (or float it) by calling QMainWindow::moveDockWindow(). QDock areas lay out their windows in Lines.

If the main window is resized so that the area occupied by the toolbar is too small to show all its widgets a little arrow button (which looks like a right-pointing chevron, 'ż') will appear at the right or bottom of the toolbar depending on its orientation. Clicking this button pops up a menu that shows the 'overflowing' items.

Usually a toolbar will get precisely the space it needs. However, with setHorizontalStretchable(), setVerticalStretchable() or setStretchableWidget() you can tell the main window to expand the toolbar to fill all available space in the specified orientation.

The toolbar arranges its buttons either horizontally or vertically (see orientation() for details). Generally, QDockArea will set the orientation correctly for you, but you can set it yourself with setOrientation() and track any changes by connecting to the orientationChanged() signal.

You can use the clear() method to remove all items from a toolbar.

See also QToolButton [p. 142], QMainWindow [p. 52], Parts of Isys on Visual Design, GUI Design Handbook: Tool Bar and Main Window and Related Classes.

Member Function Documentation

QToolBar::QToolBar (const QString & label, QMainWindow *, ToolBarDock = DockTop, bool newLine = FALSE, const char * name = 0)

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

QToolBar::QToolBar (const QString & label, QMainWindow * mainWindow, QWidget * parent, bool newLine = FALSE, const char * name = 0, WFlags f = 0)

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

Constructs an empty horizontal toolbar.

The toolbar is a child of *parent* and is managed by *mainWindow*. The *label* and *newLine* parameters are passed straight to QMainWindow::addDockWindow(). *name* is the object name and *f* is the widget flags.

Use this constructor if you want to create torn-off (undocked, floating) toolbars or toolbars in the status bar.

QToolBar::QToolBar (QMainWindow * parent = 0, const char * name = 0)

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

Constructs an empty toolbar, with parent *parent* and name *name*, in its *parent*'s top dock area, without any label and without requiring a newline.

void QToolBar::addSeparator ()

Adds a separator to the right/bottom of the toolbar.

Examples: fileiconview/mainwindow.cpp, helpviewer/helpwindow.cpp, qfd/fontdisplayer.cpp and scribble/scribble.cpp.

void QToolBar::clear () [virtual]

Deletes all the toolbar's child widgets.

QString QToolBar::label () const

Returns the label of the toolbar. See the "label" [p. 141] property for details.

QMainWindow * QToolBar::mainWindow () const

Returns a pointer to the QMainWindow which manages this toolbar.

void QToolBar::setLabel (const QString &) [virtual]

Sets the label of the toolbar. See the "label" [p. 141] property for details.

void QToolBar::setStretchableWidget (QWidget * w) [virtual]

Sets the widget *w* to be expanded if this toolbar is requested to stretch.

The request to stretch might occur because QMainWindow right-justifies the dock it's in, or because this toolbar's isVerticalStretchable() or isHorizontalStretchable() is set to TRUE.

If you call this function and the toolbar is not yet stretchable, setStretchable() is called.

See also QMainWindow::rightJustification [p. 67], setVerticalStretchable() [p. 27] and setHorizontalStretchable() [p. 27].

Examples: fileiconview/mainwindow.cpp and helpviewer/helpwindow.cpp.

Property Documentation

QString label

This property holds the label of the toolbar.

If the toolbar is floated the label becomes the toolbar window's caption. There is no default label text.

Set this property's value with setLabel() and get this property's value with label().

QToolButton Class Reference

The QToolButton class provides a quick-access button to commands or options, usually used inside a QToolBar.

`#include <qtoolbutton.h>`

Inherits QButton [Widgets with Qt].

Public Members

- **QToolButton** (QWidget * parent, const char * name = 0)
- **QToolButton** (const QIconSet & iconSet, const QString & textLabel, const QString & grouptext, QObject * receiver, const char * slot, QToolBar * parent, const char * name = 0)
- **QToolButton** (ArrowType type, QWidget * parent, const char * name = 0)
- **~QToolButton** ()
- void setOnIconSet (const QIconSet &) *(obsolete)*
- void setOffIconSet (const QIconSet &) *(obsolete)*
- void setIconSet (const QIconSet & set, bool on) *(obsolete)*
- QIconSet onIconSet () const *(obsolete)*
- QIconSet offIconSet () const *(obsolete)*
- QIconSet iconSet (bool on) const *(obsolete)*
- virtual void **setIconSet** (const QIconSet &)
- QIconSet **iconSet** () const
- bool **usesBigPixmap** () const
- bool **usesTextLabel** () const
- QString **textLabel** () const
- void **setPopup** (QPopupMenu * popup)
- QPopupMenu * **popup** () const
- void **setPopupDelay** (int delay)
- int **popupDelay** () const
- void **openPopup** ()
- void **setAutoRaise** (bool enable)
- bool **autoRaise** () const

Public Slots

- virtual void **setUsesBigPixmap** (bool enable)
- virtual void **setUsesTextLabel** (bool enable)
- virtual void **setTextLabel** (const QString & newLabel, bool tipToo)
- virtual void **setToggleButton** (bool enable)
- virtual void **setOn** (bool enable)
- void **toggle** ()
- void **setTextLabel** (const QString &)

Properties

- bool **autoRaise** — whether auto-raising is enabled
- QIconSet **iconSet** — the icon set providing the icon shown on the button
- QIconSet offIconSet — the icon set that is used when the button is in an "off" state *(obsolete)*
- bool **on** — whether this tool button is on
- QIconSet onIconSet — the icon set that is used when the button is in an "on" state *(obsolete)*
- int **popupDelay** — the time delay between pressing the button and the appearance of the associated popup menu in milliseconds
- QString **textLabel** — the label of this button
- bool **toggleButton** — whether this tool button is a toggle button
- bool **usesBigPixmap** — whether this toolbutton uses big pixmaps
- bool **usesTextLabel** — whether the toolbutton displays a text label below the button pixmap

Protected Members

- bool **uses3D** () const

Detailed Description

The QToolButton class provides a quick-access button to commands or options, usually used inside a QToolBar.

A tool button is a special button that provides quick-access to specific commands or options. As opposed to a normal command button, a tool button usually doesn't normally show a text label, but an icon. Its classic usage is to select tools, for example the "pen" tool in a drawing program. This would be implemented with a QToolButton as toggle button (see setToggleButton()).

QToolButton supports auto-raising. In auto-raise mode, the button draws a 3D frame only when the mouse points at it. The feature is automatically turned on when a button is used inside a QToolBar. Change it with setAutoRaise().

A tool button's icon is set as QIconSet. This makes it possible to specify different pixmaps for the disabled and active state. The disabled pixmap is used when the button's functionality is not available. The active pixmap is displayed when the button is auto-raised because the user is pointing at it.

The button's look and dimension is adjustable with setUsesBigPixmap() and setUsesTextLabel(). When used inside a QToolBar, the button automatically adjusts to QMainWindow's settings (see QMainWindow::setUsesTextLabel() and QMainWindow::setUsesBigPixmaps()).

A tool button can offer additional choices in a popup menu. The feature is sometimes used with the "Back" button in a web browser. After pressing the button down for awhile, a menu pops up showing all possible pages to browse back. With QToolButton you can set a popup menu using setPopup(). The default delay is 600ms; you may adjust it with setPopupDelay().

See also QPushButton [Widgets with Qt], QToolBar [p. 139], QMainWindow [p. 52], GUI Design Handbook: Push Button and Basic Widgets.

Member Function Documentation

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

Constructs an empty tool button with parent *parent* and name *name*.

QToolButton::QToolButton (const QIconSet & iconSet, const QString & textLabel, const QString & grouptext, QObject * receiver, const char * slot, QToolBar * parent, const char * name = 0)

Constructs a tool button that is a child of *parent* (which must be a QToolBar) and named *name*.

The tool button will display *iconSet*, with its text label and tool tip set to *textLabel* and its status bar message set to *grouptext*. It will be connected to the *slot* in object *receiver*.

QToolButton::QToolButton (ArrowType type, QWidget * parent, const char * name = 0)

Constructs a tool button as an arrow button. The ArrowType *type* defines the arrow direction. Possible values are LeftArrow, RightArrow, UpArrow and DownArrow.

An arrow button has auto-repeat turned on by default.

The *parent* and *name* arguments are sent to the QWidget constructor.

QToolButton::~QToolButton ()

Destroys the object and frees any allocated resources.

bool QToolButton::autoRaise () const

Returns TRUE if auto-raising is enabled; otherwise returns FALSE. See the "autoRaise" [p. 147] property for details.

QIconSet QToolButton::iconSet () const

Returns the icon set providing the icon shown on the button. See the "iconSet" [p. 147] property for details.

QIconSet QToolButton::iconSet (bool on) 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.

Since Qt 3.0, QIconSet contains both the On and Off icons.

For ease of porting, this function ignores the *on* parameter and returns the iconSet property. If you relied on the *on* parameter, you probably want to update your code to use the QIconSet On/Off mechanism.

QIconSet QToolButton::offIconSet () const

Returns the icon set that is used when the button is in an "off" state. See the "offIconSet" [p. 147] property for details.

QIconSet QToolButton::onIconSet () const

Returns the icon set that is used when the button is in an "on" state. See the "onIconSet" [p. 148] property for details.

void QToolButton::openPopup ()

Opens (pops up) the associated popup menu. If there is no such menu, this function does nothing. This function does not return until the popup menu has been closed by the user.

QPopupMenu * QToolButton::popup () const

Returns the associated popup menu, or 0 if no popup menu has been defined.

See also setPopup() [p. 146].

int QToolButton::popupDelay () const

Returns the time delay between pressing the button and the appearance of the associated popup menu in milliseconds. See the "popupDelay" [p. 148] property for details.

void QToolButton::setAutoRaise (bool enable)

Sets whether auto-raising is enabled to *enable*. See the "autoRaise" [p. 147] property for details.

void QToolButton::setIconSet (const QIconSet &) [virtual]

Sets the icon set providing the icon shown on the button. See the "iconSet" [p. 147] property for details.

void QToolButton::setIconSet (const QIconSet & set, bool on)

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.

Since Qt 3.0, QIconSet contains both the On and Off icons.

For ease of porting, this function ignores the *on* parameter and sets the iconSet property. If you relied on the *on* parameter, you probably want to update your code to use the QIconSet On/Off mechanism.

See also iconSet [p. 147] and QIconSet::State [Graphics with Qt].

void QToolButton::setOffIconSet (const QIconSet &)

Sets the icon set that is used when the button is in an "off" state. See the "offIconSet" [p. 147] property for details.

void QToolButton::setOn (bool enable) [virtual slot]

Sets whether this tool button is on to *enable*. See the "on" [p. 148] property for details.

void QToolButton::setOnIconSet (const QIconSet &)

Sets the icon set that is used when the button is in an "on" state. See the "onIconSet" [p. 148] property for details.

void QToolButton::setPopup (QPopupMenu * popup)

Associates the popup menu *popup* with this tool button.

The popup will be shown each time the tool button has been pressed down for a certain amount of time. A typical application example is the "back" button in some web browsers's tool bars. If the user clicks it, the browser simply browses back to the previous page. If the user holds the button down for a while, the tool button shows a menu containing the current history list.

Ownership of the popup menu is not transferred to the tool button.

See also popup() [p. 145].

void QToolButton::setPopupDelay (int delay)

Sets the time delay between pressing the button and the appearance of the associated popup menu in milliseconds to *delay*. See the "popupDelay" [p. 148] property for details.

void QToolButton::setTextLabel (const QString &) [slot]

Sets the label of this button. See the "textLabel" [p. 148] property for details.

void QToolButton::setTextLabel (const QString & newLabel, bool tipToo) [virtual slot]

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

Sets the label of this button to *newLabel* and automatically sets it as tool tip too if *tipToo* is TRUE.

void QToolButton::setToggleButton (bool enable) [virtual slot]

Sets whether this tool button is a toggle button to *enable*. See the "toggleButton" [p. 148] property for details.

void QToolButton::setUsesBigPixmap (bool enable) [virtual slot]

Sets whether this toolbutton uses big pixmaps to *enable*. See the "usesBigPixmap" [p. 149] property for details.

void QToolButton::setUsesTextLabel (bool enable) [virtual slot]

Sets whether the toolbutton displays a text label below the button pixmap to *enable*. See the "usesTextLabel" [p. 149] property for details.

QString QToolButton::textLabel () const

Returns the label of this button. See the "textLabel" [p. 148] property for details.

void QToolButton::toggle () [slot]

Toggles the state of this tool button.

This function has no effect on non-toggling buttons.

See also toggleButton [p. 148] and toggled() [Widgets with Qt].

bool QToolButton::uses3D () const [protected]

Returns TRUE if this button should be drawn using raised edges; otherwise returns FALSE.

See also drawButton() [Widgets with Qt].

bool QToolButton::usesBigPixmap () const

Returns TRUE if this toolbutton uses big pixmaps; otherwise returns FALSE. See the "usesBigPixmap" [p. 149] property for details.

bool QToolButton::usesTextLabel () const

Returns TRUE if the toolbutton displays a text label below the button pixmap; otherwise returns FALSE. See the "usesTextLabel" [p. 149] property for details.

Property Documentation

bool autoRaise

This property holds whether auto-raising is enabled.

The default is disabled (i.e. FALSE).

Set this property's value with setAutoRaise() and get this property's value with autoRaise().

QIconSet iconSet

This property holds the icon set providing the icon shown on the button.

Setting this property sets QToolButton::pixmap to a null pixmap. There is no default iconset.

See also pixmap [Widgets with Qt], toggleButton [p. 148] and on [p. 148].

Set this property's value with setIconSet() and get this property's value with iconSet().

QIconSet offIconSet

This property holds the icon set that is used when the button is in an "off" state.

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

Since Qt 3.0, QIconSet contains both the On and Off icons. There is now an QToolButton::iconSet property that replaces both QToolButton::onIconSet and QToolButton::offIconSet.

For ease of porting, this property is a synonym for QToolButton::iconSet. You probably want to go over your application code and use the QIconSet On/Off mechanism.

See also iconSet [p. 147] and QIconSet::State [Graphics with Qt].

Set this property's value with setOffIconSet() and get this property's value with offIconSet().

bool on

This property holds whether this tool button is on.

This property has no effect on non-toggling buttons. The default is FALSE (i.e. off).

See also toggleButton [p. 148] and toggle() [p. 146].

Set this property's value with setOn().

QIconSet onIconSet

This property holds the icon set that is used when the button is in an "on" state.

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

Since Qt 3.0, QIconSet contains both the On and Off icons. There is now an QToolButton::iconSet property that replaces both QToolButton::onIconSet and QToolButton::offIconSet.

For ease of porting, this property is a synonym for QToolButton::iconSet. You probably want to go over your application code and use the QIconSet On/Off mechanism.

See also iconSet [p. 147] and QIconSet::State [Graphics with Qt].

Set this property's value with setOnIconSet() and get this property's value with onIconSet().

int popupDelay

This property holds the time delay between pressing the button and the appearance of the associated popup menu in milliseconds.

Usually this is around half a second. A value of 0 will add a special section to the toolbutton that can be used to open the popupmenu.

See also setPopup() [p. 146].

Set this property's value with setPopupDelay() and get this property's value with popupDelay().

QString textLabel

This property holds the label of this button.

Setting this property automatically sets the text as tool tip too. There is no default text.

Set this property's value with setTextLabel() and get this property's value with textLabel().

bool toggleButton

This property holds whether this tool button is a toggle button.

Toggle buttons have an on/off state similar to check boxes. A tool button is not a toggle button by default.

See also on [p. 148] and toggle() [p. 146].

Set this property's value with setToggleButton().

bool usesBigPixmap

This property holds whether this toolbutton uses big pixmaps.

QToolButton automatically connects this property to the relevant signal in the QMainWindow in which it resides. We strongly recommend that you use QMainWindow::setUsesBigPixmaps() instead.

This property's default is TRUE.

Warning: If you set some buttons (in a QMainWindow) to have big pixmaps and others to have small pixmaps, QMainWindow may not get the geometry right.

Set this property's value with setUsesBigPixmap() and get this property's value with usesBigPixmap().

bool usesTextLabel

This property holds whether the toolbutton displays a text label below the button pixmap.

The default is FALSE.

QToolButton automatically connects this slot to the relevant signal in the QMainWindow in which is resides.

Set this property's value with setUsesTextLabel() and get this property's value with usesTextLabel().

QToolTip Class Reference

The QToolTip class provides tool tips (balloon help) for any widget or rectangular part of a widget.

`#include <qtooltip.h>`

Inherits Qt [Additional Functionality with Qt].

Public Members

- **QToolTip** (QWidget * widget, QToolTipGroup * group = 0)
- QWidget * **parentWidget** () const
- QToolTipGroup * **group** () const

Static Public Members

- void **add** (QWidget * widget, const QString & text)
- void **add** (QWidget * widget, const QString & text, QToolTipGroup * group, const QString & longText)
- void **remove** (QWidget * widget)
- void **add** (QWidget * widget, const QRect & rect, const QString & text)
- void **add** (QWidget * widget, const QRect & rect, const QString & text, QToolTipGroup * group, const QString & groupText)
- void **remove** (QWidget * widget, const QRect & rect)
- QString **textFor** (QWidget * widget, const QPoint & pos = QPoint ( ))
- void **hide** ()
- QFont **font** ()
- void **setFont** (const QFont & font)
- QPalette **palette** ()
- void **setPalette** (const QPalette & palette)
- void setEnabled (bool enable) *(obsolete)*
- bool enabled () *(obsolete)*
- void **setGloballyEnabled** (bool enable)
- bool **isGloballyEnabled** ()

Protected Members

- virtual void **maybeTip** (const QPoint & p)
- void **tip** (const QRect & rect, const QString & text)
- void **tip** (const QRect & rect, const QString & text, const QString & groupText)
- void **clear** ()

Detailed Description

The QToolTip class provides tool tips (balloon help) for any widget or rectangular part of a widget.

The tip is a short, single line of text reminding the user of the widget's or rectangle's function. It is drawn immediately below the region in a distinctive black-on-yellow combination.

QToolTipGroup provides a way for tool tips to display another text elsewhere (most often in a status bar).

At any point in time, QToolTip is either dormant or active. In dormant mode the tips are not shown and in active mode they are. The mode is global, not particular to any one widget.

QToolTip switches from dormant to active mode when the user hovers the mouse on a tip-equipped region for a second or so and remains active until the user either clicks a mouse button, presses a key, lets the mouse hover for five seconds or moves the mouse outside *all* tip-equipped regions for at least a second.

The QToolTip class can be used in three different ways:

1. Adding a tip to an entire widget.
2. Adding a tip to a fixed rectangle within a widget.
3. Adding a tip to a dynamic rectangle within a widget.

To add a tip to a widget, call the *static* function QToolTip::add() with the widget and tip as arguments:

```
QToolTip::add( quitButton, "Leave the application" );
```

This is the simplest and most common use of QToolTip. The tip will be deleted automatically when *quitButton* is deleted, but you can remove it yourself, too:

```
QToolTip::remove( quitButton );
```

You can also display another text (typically in a status bar), courtesy of QToolTipGroup. This example assumes that *g* is a `QToolTipGroup *` and is already connected to the appropriate status bar:

```
QToolTip::add( quitButton, "Leave the application", g,
               "Leave the application, prompting to save if necessary" );
QToolTip::add( closeButton, "Close this window", g,
               "Close this window, prompting to save if necessary" );
```

To add a tip to a fixed rectangle within a widget, call the static function QToolTip::add() with the widget, rectangle and tip as arguments. (See the tooltip/tooltip.cpp example.) Again, you can supply a `QToolTipGroup *` and another text if you want.

Both of these are one-liners and cover the vast majority of cases. The third and most general way to use QToolTip uses a pure virtual function to decide whether to pop up a tool tip. The tooltip/tooltip.cpp example demonstrates this, too. This mode can be used to implement tips for text that can move as the user scrolls, for example.

To use QToolTip like this, you need to subclass QToolTip and reimplement maybeTip(). QToolTip calls maybeTip() when a tip should pop up, and maybeTip decides whether to show a tip.

Tool tips can be globally disabled using QToolTip::setGloballyEnabled() or disabled in groups with QToolTipGroup::setEnabled().

You can retreive the text of a tooltip for a given position within a widget using textFor().

The global tooltip font and palette can be set with the static setFont() and setPalette() functions respectively.

See also QStatusBar [Widgets with Qt], QWhatsThis [Widgets with Qt], QToolTipGroup [p. 156], GUI Design Handbook: Tool Tip and Help System.

Member Function Documentation

QToolTip::QToolTip (QWidget * widget, QToolTipGroup * group = 0)

Constructs a tool tip object. This is necessary only if you need tool tips on regions that can move within the widget (most often because the widget's contents can scroll).

widget is the widget you want to add dynamic tool tips to and *group* (optional) is the tool tip group they should belong to.

Warning: QToolTip is not a subclass of QObject, so the instance of QToolTip is not deleted when *widget* is deleted.

See also maybeTip() [p. 153].

void QToolTip::add (QWidget * widget, const QString & text) [static]

Adds a tool tip to *widget*. *text* is the text to be shown in the tool tip.

This is the most common entry point to the QToolTip class; it is suitable for adding tool tips to buttons, check boxes, combo boxes and so on.

Examples: qdir/qdir.cpp, scribble/scribble.cpp and tooltip/tooltip.cpp.

void QToolTip::add (QWidget * widget, const QString & text, QToolTipGroup * group, const QString & longText) [static]

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

Adds a tool tip to *widget* and to tool tip group *group*.

text is the text shown in the tool tip and *longText* is the text emitted from *group*.

Normally, *longText* is shown in a status bar or similar.

void QToolTip::add (QWidget * widget, const QRect & rect, const QString & text) [static]

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

Adds a tool tip to a fixed rectangle, *rect*, within *widget*. *text* is the text shown in the tool tip.

void QToolTip::add (QWidget * widget, const QRect & rect, const QString & text, QToolTipGroup * group, const QString & groupText) [static]

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

Adds a tool tip to an entire *widget* and to tool tip group *group*. The tooltip will disappear when the mouse leaves the *rect*.

text is the text shown in the tool tip and *groupText* is the text emitted from *group*.

Normally, *groupText* is shown in a status bar or similar.

void QToolTip::clear () [protected]

Immediately removes all tool tips for this tooltip's parent widget.

bool QToolTip::enabled () [static]

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

QFont QToolTip::font () [static]

Returns the font common to all tool tips.

See also setFont() [p. 154].

QToolTipGroup * QToolTip::group () const

Returns the tool tip group this QToolTip is a member of or 0 if it isn't a member of any group.

The tool tip group is the object responsible for maintaining contact between tool tips and a status bar or something else which can show the longer help text.

See also parentWidget() [p. 154] and QToolTipGroup [p. 156].

void QToolTip::hide () [static]

Hides any tip that is currently being shown.

Normally, there is no need to call this function; QToolTip takes care of showing and hiding the tips as the user moves the mouse.

bool QToolTip::isGloballyEnabled () [static]

Returns whether tool tips are enabled globally.

See also setGloballyEnabled() [p. 154].

void QToolTip::maybeTip (const QPoint & p) [virtual protected]

This pure virtual function is half of the most versatile interface QToolTip offers.

It is called when there is a possibility that a tool tip should be shown and must decide whether there is a tool tip for the point *p* in the widget that this QToolTip object relates to. If so, maybeTip() must call tip() with the rectangle the tip applies to, the tip's text and optionally the QToolTipGroup details.

p is given in that widget's local coordinates. Most maybeTip() implementations will be of the form:

```
if (  ) {
    tip( ,  );
}
```

The first argument to tip() (a rectangle) must encompass *p*, i.e. the tip must apply to the current mouse position; otherwise QToolTip's operation is undefined.

Note that the tip will disappear once the mouse moves outside the rectangle you give to tip(), and will not reappear if the mouse moves back in - maybeTip() is called again instead.

See also tip() [p. 155].

Example: tooltip/tooltip.cpp.

QPalette QToolTip::palette () [static]

Returns the palette common to all tool tips.

See also setPalette() [p. 154].

QWidget * QToolTip::parentWidget () const

Returns the widget this QToolTip applies to.

The tool tip is destroyed automatically when the parent widget is destroyed.

See also group() [p. 153].

void QToolTip::remove (QWidget * widget) [static]

Removes the tool tip from *widget*.

If there is more than one tool tip on *widget*, only the one covering the entire widget is removed.

void QToolTip::remove (QWidget * widget, const QRect & rect) [static]

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

Removes the tool tip for *rect* from *widget*.

If there is more than one tool tip on *widget*, only the one covering rectangle *rect* is removed.

void QToolTip::setEnabled (bool enable) [static]

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

void QToolTip::setFont (const QFont & font) [static]

Sets the font for all tool tips to *font*.

See also font() [p. 153].

void QToolTip::setGloballyEnabled (bool enable) [static]

If *enable* is TRUE sets all tool tips to be enabled (shown when needed); if *enable* is FALSE sets all tool tips to be disabled (never shown).

By default, tool tips are enabled. Note that this function affects all tool tips in the entire application.

See also QToolTipGroup::enabled [p. 158].

void QToolTip::setPalette (const QPalette & palette) [static]

Sets the palette for all tool tips to *palette*.

See also palette() [p. 154].

QString QToolTip::textFor (QWidget * widget, const QPoint & pos = QPoint ( )) [static]

Returns the text for *widget* at position *pos*, or a null string if there is no tool tip for the *widget*.

void QToolTip::tip (const QRect & rect, const QString & text) [protected]

Immediately pops up a tip saying *text* and removes the tip once the cursor moves out of rectangle *rect* (which is given in the coordinate system of the widget this QToolTip relates to).

The tip will not reappear if the cursor moves back; your maybeTip() has to reinstate it each time.

void QToolTip::tip (const QRect & rect, const QString & text, const QString & groupText) [protected]

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

Immediately pops up a tip saying *text* and removes that tip once the cursor moves out of rectangle *rect*. *groupText* is the text emitted from the group.

The tip will not reappear if the cursor moves back; your maybeTip() has to reinstate it each time.

QToolTipGroup Class Reference

The QToolTipGroup class collects tool tips into related groups.

`#include <qtooltip.h>`

Inherits QObject [Additional Functionality with Qt].

Public Members

- **QToolTipGroup** (QObject * parent, const char * name = 0)
- **~QToolTipGroup** ()
- bool **delay** () const
- bool **enabled** () const

Public Slots

- void **setDelay** (bool)
- void **setEnabled** (bool)

Signals

- void **showTip** (const QString & longText)
- void **removeTip** ()

Properties

- bool **delay** — whether the display of the group text is delayed
- bool **enabled** — whether tool tips in the group are enabled

Detailed Description

The QToolTipGroup class collects tool tips into related groups.

Tool tips can display *two* texts: one in the tip and (optionally) one that is typically in a status bar. QToolTipGroup provides a way to link tool tips to this status bar.

QToolTipGroup has practically no API; it is used only as an argument to QToolTip's member functions, for example like this:

```
QToolTipGroup * g = new QToolTipGroup( this, "tool tip relay" );
connect( g, SIGNAL(showTip(const QString&)),
         myLabel, SLOT(setText(const QString&)) );
connect( g, SIGNAL(removeTip()),
         myLabel, SLOT(clear()) );
QToolTip::add( giraffeButton, "feed giraffe",
               g, "Give the giraffe a meal" );
QToolTip::add( gorillaButton, "feed gorilla",
               g, "Give the gorilla a meal" );
```

This example makes the object myLabel (which you have to supply) display (one assumes, though you can make myLabel do anything, of course) the strings "Give the giraffe a meal" and "Give the gorilla a meal" while the relevant tool tips are being displayed.

Deleting a tool tip group removes the tool tips in it.

See also Help System.

Member Function Documentation

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

Constructs a tool tip group with parent *parent* and name *name*.

QToolTipGroup::~QToolTipGroup ()

Destroys this tool tip group and all tool tips in it.

bool QToolTipGroup::delay () const

Returns TRUE if the display of the group text is delayed; otherwise returns FALSE. See the "delay" [p. 158] property for details.

bool QToolTipGroup::enabled () const

Returns TRUE if tool tips in the group are enabled; otherwise returns FALSE. See the "enabled" [p. 158] property for details.

void QToolTipGroup::removeTip () [signal]

This signal is emitted when a tool tip in this group is hidden. See the QToolTipGroup documentation for an example of use.

See also showTip() [p. 158].

void QToolTipGroup::setDelay (bool) [slot]

Sets whether the display of the group text is delayed. See the "delay" [p. 158] property for details.

void QToolTipGroup::setEnabled (bool) [slot]

Sets whether tool tips in the group are enabled. See the "enabled" [p. 158] property for details.

void QToolTipGroup::showTip (const QString & longText) [signal]

This signal is emitted when one of the tool tips in the group is displayed. *longText* is the extra text for the displayed tool tip.

See also removeTip() [p. 157].

Property Documentation

bool delay

This property holds whether the display of the group text is delayed.

This property's default is TRUE.

Set this property's value with setDelay() and get this property's value with delay().

bool enabled

This property holds whether tool tips in the group are enabled.

This property's default is TRUE.

Set this property's value with setEnabled() and get this property's value with enabled().

QWizard Class Reference

The QWizard class provides a framework for wizard dialogs.

`#include <qwizard.h>`

Inherits QDialog [p. 10].

Public Members

- **QWizard** (QWidget * parent = 0, const char * name = 0, bool modal = FALSE, WFlags f = 0)
- **~QWizard** ()
- virtual void **addPage** (QWidget * page, const QString & title)
- virtual void **insertPage** (QWidget * page, const QString & title, int index)
- virtual void **removePage** (QWidget * page)
- QString **title** (QWidget * page) const
- void **setTitle** (QWidget * page, const QString & title)
- QFont **titleFont** () const
- void **setTitleFont** (const QFont &)
- virtual void **showPage** (QWidget * page)
- QWidget * **currentPage** () const
- QWidget * **page** (int index) const
- int **pageCount** () const
- int **indexOf** (QWidget * page) const
- virtual bool **appropriate** (QWidget * page) const
- virtual void **setAppropriate** (QWidget * page, bool appropriate)
- QPushButton * **backButton** () const
- QPushButton * **nextButton** () const
- QPushButton * **finishButton** () const
- QPushButton * **cancelButton** () const
- QPushButton * **helpButton** () const

Public Slots

- virtual void **setBackEnabled** (QWidget * page, bool enable)
- virtual void **setNextEnabled** (QWidget * page, bool enable)
- virtual void **setFinishEnabled** (QWidget * page, bool enable)
- virtual void **setHelpEnabled** (QWidget * page, bool enable)
- virtual void setFinish (QWidget *, bool) *(obsolete)*

Signals

- void **helpClicked** ()
- void **selected** (const QString &)

Properties

- QFont **titleFont** — the font used for page titles

Protected Members

- virtual void **layOutButtonRow** (QHBoxLayout * layout)
- virtual void **layOutTitleRow** (QHBoxLayout * layout, const QString & title)

Protected Slots

- virtual void **back** ()
- virtual void **next** ()
- virtual void **help** ()

Detailed Description

The QWizard class provides a framework for wizard dialogs.

A wizard is a special type of input dialog that consists of a sequence of dialog pages. A wizard's purpose is to assist a user by automating a task by walking the user through the process step by step. Wizards are useful for complex or infrequently occurring tasks that people may find difficult to learn or do.

QWizard provides page titles and displays Next, Back, Finish, Cancel, and Help push buttons, as appropriate to the current position in the page sequence.

Create and populate dialog pages that inherit from QWidget and add them to the wizard using addPage(). Use insertPage() to add a dialog page at a certain position in the page sequence. Use removePage() to remove a page from the page sequence.

Use currentPage() to retrieve a pointer to the currently displayed page. page() returns a pointer to the page at a certain position in the page sequence.

Use pageCount() to retrieve the total number of pages in the page sequence. indexOf() will return the index of a page in the page sequence.

QWizard provides functionality to mark pages as appropriate (or not) in the current context with setAppropriate(). The idea is that a page may be irrelevant and should be skipped depending on the data entered by the user on a preceding page.

It is generally considered good design to provide a greater number of simple pages with fewer choices rather than a smaller number of complex pages.

Example code is available here: wizard/wizard.cpp wizard/wizard.h

See also Abstract Widget Classes, Dialog Classes and Organizers.

Member Function Documentation

QWizard::QWizard (QWidget * parent = 0, const char * name = 0, bool modal = FALSE, WFlags f = 0)

Constructs an empty wizard dialog. The *parent*, *name*, *modal* and *f* arguments are passed to the QDialog constructor.

QWizard::~QWizard ()

Destroys the object and frees any allocated resources, including all pages and controllers.

void QWizard::addPage (QWidget * page, const QString & title) [virtual]

Adds *page* to the end of the page sequence, with the title, *title*.

bool QWizard::appropriate (QWidget * page) const [virtual]

Called when the Next button is clicked; this virtual function returns TRUE if *page* is relevant for display in the current context; otherwise it is ignored by QWizard and returns FALSE. The default implementation returns the value set using setAppropriate(). The ultimate default is TRUE.

Warning: The last page of the wizard will be displayed if no page is relevant in the current context.

void QWizard::back () [virtual protected slot]

Called when the user clicks the Back button; this function shows the preceding relevant page in the sequence.

See also appropriate() [p. 161].

QPushButton * QWizard::backButton () const

Returns a pointer to the Back button of the dialog.

By default, this button is connected to the back() slot, which is virtual so you can reimplement it in a QWizard subclass.

QPushButton * QWizard::cancelButton () const

Returns a pointer to the Cancel button of the dialog.

By default, this button is connected to the QDialog::reject() slot, which is virtual so you can reimplement it in a QWizard subclass.

QWidget * QWizard::currentPage () const

Returns a pointer to the current page in the sequence. Although the wizard does its best to make sure that this value is never 0, it can be if you try hard enough.

QPushButton * QWizard::finishButton () const

Returns a pointer to the Finish button of the dialog.

By default, this button is connected to the QDialog::accept() slot, which is virtual so you can reimplement it in a QWizard subclass.

void QWizard::help () [virtual protected slot]

Called when the user clicks the Help button, this function emits the helpClicked() signal.

QPushButton * QWizard::helpButton () const

Returns a pointer to the Help button of the dialog.

By default, this button is connected to the help() slot, which is virtual so you can reimplement it in a QWizard subclass.

void QWizard::helpClicked () [signal]

This signal is emitted when the user clicks on the Help button.

int QWizard::indexOf (QWidget * page) const

Returns the sequence index of page *page*. If the page is not part of the wizard -1 is returned.

void QWizard::insertPage (QWidget * page, const QString & title, int index) [virtual]

Inserts *page* at index *index* into the page sequence, with title *title*. If index is -1, the page will be appended to the end of the wizard's page sequence.

void QWizard::layOutButtonRow (QHBoxLayout * layout) [virtual protected]

This virtual function is responsible for adding the bottom divider and buttons below it.

layout is the vertical layout of the entire wizard.

void QWizard::layOutTitleRow (QHBoxLayout * layout, const QString & title) [virtual protected]

This virtual function is responsible for laying out the title row and adding the vertical divider between the title and the wizard page. *layout* is the vertical layout for the wizard, and *title* is the title for this page. This function is called every time *title* changes.

void QWizard::next () [virtual protected slot]

Called when the user clicks the Next button, this function shows the next relevant page in the sequence.

See also appropriate() [p. 161].

QPushButton * QWizard::nextButton () const

Returns a pointer to the Next button of the dialog.

By default, this button is connected to the next() slot, which is virtual so you can reimplement it in a QWizard subclass.

QWidget * QWizard::page (int index) const

Returns a pointer to the page at position *index* in the sequence, or 0 if *index* is out of range. The first page has index 0.

int QWizard::pageCount () const

Returns the number of pages in the wizard.

void QWizard::removePage (QWidget * page) [virtual]

Removes *page* from the page sequence but does not delete the page. If *page* is currently being displayed, QWizard will display the page before it in the wizard, or the first page if this was the first page.

void QWizard::selected (const QString &) [signal]

This signal is emitted when the current page changes. The parameter contains the title of the page.

void QWizard::setAppropriate (QWidget * page, bool appropriate) [virtual]

If *appropriate* is TRUE then page *page* is considered relevant in the current context and should be displayed in the page sequence; otherwise *page* should not be displayed in the page sequence.

See also appropriate() [p. 161].

void QWizard::setBackEnabled (QWidget * page, bool enable) [virtual slot]

If *enable* is TRUE, page *page* has a Back button; otherwise *page* has no Back button. By default all pages have this button.

void QWizard::setFinish (QWidget *, bool) [virtual slot]

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

Use setFinishEnabled instead

void QWizard::setFinishEnabled (QWidget * page, bool enable) [virtual slot]

If *enable* is TRUE, page *page* has a Finish button; otherwise *page* has no Finish button. By default *no* pages have this button.

void QWizard::setHelpEnabled (QWidget * page, bool enable) [virtual slot]

If *enable* is TRUE, page *page* has a Help button; otherwise *page* has no Help button. By default all pages have this button.

void QWizard::setNextEnabled (QWidget * page, bool enable) [virtual slot]

If *enable* is TRUE, page *page* has a Next button; otherwise *page* has no Next button. By default all pages have this button.

void QWizard::setTitle (QWidget * page, const QString & title)

Sets the title for page *page* to *title*.

void QWizard::setTitleFont (const QFont &)

Sets the font used for page titles. See the "titleFont" [p. 164] property for details.

void QWizard::showPage (QWidget * page) [virtual]

Makes *page* the current page and emits the selected() signal.

Example: wizard/wizard.cpp.

QString QWizard::title (QWidget * page) const

Returns the title of page *page*.

QFont QWizard::titleFont () const

Returns the font used for page titles. See the "titleFont" [p. 164] property for details.

Property Documentation

QFont titleFont

This property holds the font used for page titles.

The default is QApplication::font().

Set this property's value with setTitleFont() and get this property's value with titleFont().

QWorkspace Class Reference

The QWorkspace widget provides a workspace window that can contain decorated windows, e.g. for MDI.

This class is part of the **workspace** module.

`#include <qworkspace.h>`

Inherits QWidget [Widgets with Qt].

Public Members

- **QWorkspace** (QWidget * parent = 0, const char * name = 0)
- **~QWorkspace** ()
- QWidget * **activeWindow** () const
- QWidgetList **windowList** () const
- bool **scrollBarsEnabled** () const
- void **setScrollBarsEnabled** (bool enable)

Public Slots

- void **cascade** ()
- void **tile** ()

Signals

- void **windowActivated** (QWidget * w)

Properties

- bool **scrollBarsEnabled** — whether the workspace provides scrollbars

Detailed Description

The QWorkspace widget provides a workspace window that can contain decorated windows, e.g. for MDI.

An MDI (multiple document interface) application has one main window with a menu bar. The central widget of this window is a workspace. The workspace itself contains zero, one or more document windows, each of which displays a document.

The workspace itself is an ordinary Qt widget. It has a standard constructor that takes a parent widget and an object name. The parent window is usually a QMainWindow, but it need not be.

Document windows (i.e. MDI windows) are also ordinary Qt widgets which have the workspace as parent widget. When you call show(), hide(), showMaximized(), setCaption(), etc. on a document window, it is shown, hidden, etc. with a frame, caption, icon and icon text, just as you'd expect. You can provide widget flags which will be used for the layout of the decoration or the behaviour of the widget itself.

To change the geometry of the MDI windows it is necessary to make the function calls to the parentWidget() of the widget, as this will move or resize the decorated window.

A document window becomes active when it gets the keyboard focus. You can activate it using setFocus(), and the user can activate it by moving focus in the normal ways. The workspace emits a signal windowActivated() when it detects the activation change, and the function activeWindow() always returns a pointer to the active document window.

The convenience function windowList() returns a list of all document windows. This is useful to create a popup menu "Windows" on the fly, for example.

QWorkspace provides two built-in layout strategies for child windows: cascade() and tile(). Both are slots so you can easily connect menu entries to them.

If you want your users to be able to work with document windows larger than the actual workspace, set the scrollBarsEnabled property to TRUE.

If the top-level window contains a menu bar and a document window is maximised, QWorkspace moves the document window's minimize, restore and close buttons from the document window's frame to the workspace window's menu bar. It then inserts a window operations menu at the extreme left of the menu bar.

See also Main Window and Related Classes and Organizers.

Member Function Documentation

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

Constructs a workspace with a *parent* and a *name*.

QWorkspace::~QWorkspace ()

Destroys the workspace and frees any allocated resources.

QWidget * QWorkspace::activeWindow () const

Returns the active window, or 0 if no window is active.

Example: mdi/application.cpp.

void QWorkspace::cascade () [slot]

Arranges all child windows in a cascade pattern.

See also tile() [p. 167].

Example: mdi/application.cpp.

bool QWorkspace::scrollBarsEnabled () const

Returns TRUE if the workspace provides scrollbars; otherwise returns FALSE. See the "scrollBarsEnabled" [p. 167] property for details.

void QWorkspace::setScrollBarsEnabled (bool enable)

Sets whether the workspace provides scrollbars to *enable*. See the "scrollBarsEnabled" [p. 167] property for details.

void QWorkspace::tile () [slot]

Arranges all child windows in a tile pattern.

See also cascade() [p. 166].

Example: mdi/application.cpp.

void QWorkspace::windowActivated (QWidget * w) [signal]

This signal is emitted when the window widget *w* becomes active. Note that *w* can be null, and that more than one signal may be fired for one activation event.

See also activeWindow() [p. 166] and windowList() [p. 167].

QWidgetList QWorkspace::windowList () const

Returns a list of all windows.

Example: mdi/application.cpp.

Property Documentation

bool scrollBarsEnabled

This property holds whether the workspace provides scrollbars.

If this property is set to TRUE, it is possible to resize child windows over the right or the bottom edge out of the visible area of the workspace. The workspace shows scrollbars to make it possible for the user to access those windows. If this property is set to FALSE (the default), resizing windows out of the visible area of the workspace is not permitted.

Set this property's value with setScrollBarsEnabled() and get this property's value with scrollBarsEnabled().

Index