Troll Tech AS

Qt 1.41

Tutorial and Reference

Preliminary Version, October 2, 1998

http://www.troll.no/
sales@troll.no / qt-bugs@troll.no
fax: +47 22646949

Introduction

This manual consists of two sections, the tutorial (chapters 1-14) and the class reference (the rest). Both are generated automatically from HTML documents and do lose some content when transformed to paper, links mostly. The relevant URL is printed at the end of each chapter.

The tutorial gives an introduction to GUI programming using the Qt toolkit. It introduces some of the most important features, but the emphasis is on the programming philosophy in Qt. It starts with a five-line hello-world and builds gradually, introducing one or a few concepts at a time, into a simple, 650-line game which consists of several reusable components.

To compile and run the tutorial programs, you need the Qt toolkit, a C++ compiler, and X11R4 or later. You do not need libg++ or libstdc++.

You also need a basic knowledge of C++ and makefiles. O'Reilly Associates (for example) publishes good books on both:

> Andrew Oram and Steve Talbott: Managing Projects with make, second edition
> ISBN 0-937175-90-0
> 152 pages
> http://www.oreilly.com/catalog/make2/
>
> Steve Oualline: Practical C++ Programming
> ISBN 1-56592-139-9
> 584 pages
> http://www.oreilly.com/catalog/cplus/

The class reference documents each class in Qt in strictly alphabetical order and lists the example program which uses each class. Some information in the on-line documentation has been left out as we felt it was not worth the paper (several hundred pages).

We appreciate any comments regarding the formatting or content of this manual. Please send comments to qt-bugs@troll.no.

1. Chapter 1: Hello, World!

This first program is a simple hello-world example. It contains only the bare minimum you need to get a Qt application up and running. The picture above is a snapshot of this program.

```
/****************************************************************
**
** Qt tutorial 1
**
****************************************************************/
#include <qapplication.h>
#include <qpushbutton.h>
int main( int argc, char **argv )
{
    QApplication a( argc, argv );
    QPushButton hello( "Hello world!" );
    hello.resize( 100, 30 );
    a.setMainWidget( &hello );
    hello.show();
    return a.exec();
}
```

1.1. Line by Line Walk-Through

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

This line includes the QApplication class definition. There has to be exactly one QApplication object in every application that uses Qt. QApplication manages various application-wide resources, such as the default font and cursor.

```
#include <qpushbutton.h>
```

This line includes the QPushButton class definition. The reference documentation for each class mentions at the top which file needs to be included to use that class.

QPushButton is a classical GUI push button, which the user can press and release. It manages its own look and feel, like every other widget. A widget is a user interface object which can process user input and draw graphics. The programmer can change both the overall look and feel and many minor properties of it such as color, as well as the widget's content. A QPushButton can show either a text or a pixmap.

```
int main( int argc, char **argv )
{
```

The `main()` function is the entry point to the program. Almost always when using Qt, main() only needs to perform some kind of initialization before passing the control to the Qt library, which then tells the program about the user's actions via events.

`argc` is the number of command-line arguments and `argv` is the array of command-line arguments. This is a C/C++ feature. It is not Qt specific, however, Qt needs to process these arguments (see below).

```
    QApplication a( argc, argv );
```

`a` is this program's QApplication. Here it is created and processes some of the command-line arguments (such as -display under X11). Note that all command-line arguments recognized by Qt are removed from `argv` (and `argc` is decremented accordingly). See the QApplication::argv() documentation for details.

Note: It is essential that the QApplication object is created before any window-system parts of Qt are used.

```
QPushButton hello( "Hello world!" );
```

Here, *after* the QApplication, comes the first window-system code: A push button is created.

The button is set up to display the text "Hello world!" and be a window of its own (since the constructor does not specify another window which the button should be part of).

```
hello.resize( 100, 30 );
```

The button is set up to be 100 pixels wide and 30 pixels high (plus the window system frame). In this case we don't care about the button's position and accept the default value.

```
a.setMainWidget( &hello );
```

The push button is chosen as the main widget for the application. If the user closes a main widget, the application exits.

You don't have to have a main widget, but most programs have one.

```
hello.show();
```

A widget is never visible when you create it. You must call show() to make it visible.

```
return a.exec();
```

This is where `main()` passes control to Qt. exec() will return when the application exits.

In exec(), Qt receives and processes user and system events and passes these on to the appropriate widgets.

```
}
```

1.2. Behavior

You should now try to compile and run this program.

When you run it, you will see a small window filled with a single button, and on it you can read the famous words, Hello World!

1.3. Excercises

Try to resize the window. Press the button. If you're running X11, try running the program with the -geometry option (e.g. -geometry 100x200+10+20)

1.3.1. URL

Generated from http://www.troll.no/qt/t1.html.

2. Chapter 2: Calling it Quits

Having created a window in chapter one, we will now go on to make the application quit properly when the user tells it to.

We will also use a font that is more exciting than the default one.

```
/****************************************************************
**
** Qt tutorial 2
**
****************************************************************/
#include <qapplication.h>
#include <qpushbutton.h>
#include <qfont.h>
int main( int argc, char **argv )
{
    QApplication a( argc, argv );
    QPushButton quit( "Quit" );
    quit.resize( 75, 30 );
    quit.setFont( QFont( "Times", 18, QFont::Bold ) );
    QObject::connect( &quit, SIGNAL(clicked()), &a, SLOT(quit()) );
    a.setMainWidget( &quit );
    quit.show();
    return a.exec();
}
```

2.1. Line by Line Walk-Through

```
#include <qfont.h>
```

Since this program uses QFont, it needs to include qfont.h. Qt's font abstraction is rather different from the horror provided by X, and loading and using fonts has been highly optimized.

```
    QPushButton quit( "Quit" );
```

This time, the button says "Quit" and that's exactly what the program will do when the user clicks the button. This is not a coincidence.

```
    quit.resize( 75, 30 );
```

We've chosen another size for the button since the text is a bit shorter than "Hello World!". We could also have used QPushButton::setAutoResize(), or even use QFontMetrics to set right size.

```
    quit.setFont( QFont( "Times", 18, QFont::Bold ) );
```

Here we choose a new font for the button, an 18-point bold font from the Times family. Note that we create the font on the spot.

It is also possible to change the default font for the whole application.

```
    QObject::connect( &quit, SIGNAL(clicked()), &a, SLOT(quit()) );
```

connect() is perhaps *the* most central feature of Qt. Note that connect() is a static function in QObject. Do not confuse it with the connect() function in the socket library.

This line establishes a one-way connection between two Qt objects (objects that inherit QObject, directly or indirectly). Every Qt object can have both `signals` (to send messages) and `slots` (to receive messages). All widgets are Qt objects. They inherit QWidget which in turn inherits QObject.

Here, the *clicked()* signal of *quit* is connected to the *quit()* slot of *a,* so that when the button is clicked, the application quits.

The Signals and Slots documentation describes this topic in detail.

2.2. Behavior

When you run this program, you will see an even smaller window than in chapter one, filled with an even smaller button.

2.3. Excercises

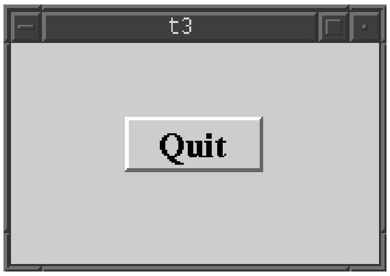
Try to resize the window. Press the button. Oops! That connect() would seem to make some difference :)

Are there any other signals in QPushButton you can connect to quit? Hint: The QPushButton inherits most of its behavior from QButton.

2.3.1. URL

Generated from http://www.troll.no/qt/t2.html.

3. Chapter 3: Family Values



This example shows how to create mother and child widgets.

We'll keep it simple and use just a single mother (uh, family values?) and a lone child.

```
/****************************************************************
**
** Qt tutorial 3
**
****************************************************************/
#include <qapplication.h>
#include <qpushbutton.h>
#include <qfont.h>
int main( int argc, char **argv )
{
    QApplication a( argc, argv );
    QWidget w;
    w.resize( 200, 120 );
    QPushButton quit( "Quit", &w );
    quit.move( 62, 40 );
    quit.resize( 75, 30 );
    quit.setFont( QFont( "Times", 18, QFont::Bold ) );
    QObject::connect( &quit, SIGNAL(clicked()), &a, SLOT(quit()) );
    a.setMainWidget( &w );
    w.show();
    return a.exec();
}
```

3.1. Line by Line Walk-Through

```
QWidget w;
```

Here we simply create a widget. Note that we did not have to include `qwidget.h`, because we have included qpushbutton.h and QPushButton is a subclass of QWidget (see Class Hierarchy).

A QWidget does not handle any events, it simply displays itself using its background color and caption.

```
w.resize( 200, 120 );
```

We set the width to 200 pixels and the height to 120 pixels.

```
QPushButton quit( "Quit", &w );
```

A child is born.

This QPushButton is created with both a text, "Quit", and a mother, w. A child widget is always on top of its mother. When displayed, it is clipped by its mother's bounds.

```
quit.move( 62, 40 );
```

The child is moved to a position of 62,40, relative to its mother's top left corner (excluding the window frame). The coordinate is as usual for a visual display, x increases to the right and y increases downwards.

```
w.show();
```

When a widget is shown, it will call show for all its children (except those you have done an explicit hide() on).

3.2. Behavior

Note that if you resize the widget, the button is not resized, unlike the programs in the first two chapters. That's because the top level widget is a QWidget, which does not react to resizing.

If you're using X11, you will see the same effect if you invoke the program with -geometry.

3.3. Excercises

Try changing the arguments to quit.move() and/or quit.resize(). What happens if quit is partly "outside" w?

For X11 users:

Also try moving the w.resize() to after the quit.move/resize, and to after a.setMainWidget(). Does -geometry still work?

3.3.1. URL

Generated from http://www.troll.no/qt/t3.html.

4. Chapter 4: Let There Be Widgets

This example shows how to create your own widget, how to control the minimum and maximum sizes of a widget, and introduces widget names.

```
/****************************************************************
**
** Qt tutorial 4
**
****************************************************************/
#include <qapplication.h>
#include <qpushbutton.h>
#include <qfont.h>
class MyWidget : public QWidget
{
public:
    MyWidget( QWidget *parent=0, const char *name=0 );
};
MyWidget::MyWidget( QWidget *parent, const char *name )
        : QWidget( parent, name )
{
    setMinimumSize( 200, 120 );
    setMaximumSize( 200, 120 );
    QPushButton *quit = new QPushButton( "Quit", this, "quit" );
    quit->setGeometry( 62, 40, 75, 30 );
    quit->setFont( QFont( "Times", 18, QFont::Bold ) );
    connect( quit, SIGNAL(clicked()), qApp, SLOT(quit()) );
}
int main( int argc, char **argv )
{
    QApplication a( argc, argv );
    MyWidget w;
    w.setGeometry( 100, 100, 200, 120 );
    a.setMainWidget( &w );
    w.show();
    return a.exec();
}
```

4.1. Line by Line Walk-Through

```
class MyWidget : public QWidget
{
public:
    MyWidget( QWidget *parent=0, const char *name=0 );
};
```

Here we create a new class. Since this class inherits from QWidget, the new class is a widget, and may be a top level window or a child widget (like the push button in chapter three).

This class has only one member, a constructor (in addition to the members it inherits from QWidget). The constructor is a standard Qt widget constructor; you should always include a similar constructor when you create widgets.

The first argument is its parent widget. To create a top level window you specify a null pointer as the parent. As you can see, the widget defaults to be a top level window.

The second argument is the widget's name. This is *not* the text that appears in the title bar or in the button. It is a name associated with a widget to make it possible to lookup this widget later. Widget names will become really useful when Qt's GUI designer is finished.

```
MyWidget::MyWidget( QWidget *parent, const char *name )
        : QWidget( parent, name )
```

The implementation of the constructor starts here. Like all widgets, it just passes on the `parent` and `name` to the QWidget constructor.

```
{
    setMinimumSize( 200, 120 );
    setMaximumSize( 200, 120 );
```

Since this widget doesn't know how to handle resizing, we fix its size by setting the minimum and maximum to be equal. In the next chapter, we will show how a widget can respond to resize event from the user.

```
    QPushButton *quit = new QPushButton( "Quit", this, "quit" );
    quit->setGeometry( 62, 40, 75, 30 );
    quit->setFont( QFont( "Times", 18, QFont::Bold ) );
```

Here we create and set up a child widget of this widget (the new widget's parent is `this)` which has the widget name "quit". The widget name has nothing to do with the button text, they just happen to be similar in this case.

Note that `quit` is a local variable in the constructor. MyWidget does not keep track of it, but Qt does, and will by default delete it when MyWidget is deleted. This is why MyWidget doesn't need a destructor. On the other hand, there is no harm in deleting a child when you choose to, the child will automatically tell Qt about its imminent death.

The setGeometry() call does the same as move() and resize() did in the previous chapters.

```
    connect( quit, SIGNAL(clicked()), qApp, SLOT(quit()) );
}
```

Since the MyWidget class doesn't know about the application object, it has to connect to Qt's pointer to it, `qApp.`

A widget is a software component and should know as little as possible about its environment in order to be as general and reusable as possible.

Knowing the name of the application object would break this principle, so Qt offers an alias, qApp, for the rare cases where a component such as MyWidget needs to talk to the application object.

```
int main( int argc, char **argv )
{
    QApplication a( argc, argv );
    MyWidget w;
    w.setGeometry( 100, 100, 200, 120 );
    a.setMainWidget( &w );
    w.show();
    return a.exec();
}
```

Here we instantiate our new baby, set it to be the main widget, and execute the application.

4.2. Behavior

This program is very similar in behavior to the previous one. The difference lies in the way we have implemented it. It does behave slightly different though. Just try to resize it to see.

4.3. Excercises

Try to create another MyWidget object in main(). What happens?

Try to add more buttons, or put in widgets other than QPushButton.

The background color can be changed.

4.3.1. URL

Generated from http://www.troll.no/qt/t4.html.

5. Chapter 5: Building Blocks

This example shows how to create and connect together several widgets using signals and slots, and how to handle resize events.

```
/****************************************************************
**
** Qt tutorial 5
**
****************************************************************/
#include <qapplication.h>
#include <qpushbutton.h>
#include <qscrollbar.h>
#include <qlcdnumber.h>
#include <qfont.h>
class MyWidget : public QWidget
{
public:
    MyWidget( QWidget *parent=0, const char *name=0 );
protected:
    void resizeEvent( QResizeEvent * );
private:
    QPushButton *quit;
    QScrollBar  *sBar;
    QLCDNumber  *lcd;
};
MyWidget::MyWidget( QWidget *parent, const char *name )
        : QWidget( parent, name )
{
    setMinimumSize( 200, 200 );
    quit = new QPushButton( "Quit", this, "quit" );
    quit->setGeometry( 10, 10, 75, 30 );
    quit->setFont( QFont( "Times", 18, QFont::Bold ) );
    connect( quit, SIGNAL(clicked()), qApp, SLOT(quit()) );
    lcd  = new QLCDNumber( 2, this, "lcd" );
    lcd->move( 10, quit->y() + quit->height() + 10 );
    sBar = new QScrollBar( 0, 99,                       // range
                           1, 10,                       // line/page steps
                           0,                           // inital value
                           QScrollBar::Horizontal,      // orientation
                           this, "scrollbar" );
    connect( sBar, SIGNAL(valueChanged(int)), lcd, SLOT(display(int)) );
}
void MyWidget::resizeEvent( QResizeEvent * )
{
    sBar->setGeometry( 10, height() - 10 - 16, width() - 20, 16 );
```

```
    lcd->resize( sBar->width(), sBar->y() - lcd->y() - 5 );
}
int main( int argc, char **argv )
{
    QApplication a( argc, argv );
    MyWidget w;
    w.setGeometry( 100, 100, 200, 200 );
    a.setMainWidget( &w );
    w.show();
    return a.exec();
}
```

5.1. Line by Line Walk-Through

```
#include <qapplication.h>
#include <qpushbutton.h>
#include <qscrollbar.h>
#include <qlcdnumber.h>
#include <qfont.h>
```

Two new include files here, qscrollbar.h and qlcdnumber.h, because we use two new classes; QScrollBar and QLCDNumber.

```
class MyWidget : public QWidget
{
public:
    MyWidget( QWidget *parent=0, const char *name=0 );
```

Nothing new compared to the previous chapter.

```
protected:
    void resizeEvent( QResizeEvent * );
```

MyWidget now gains the ability to handle resize event.

resizeEvent() is one of many virtual functions in QWidget which are used to pass events of various kinds to widgets. These event functions are one of the the central control flow mechanisms in Qt, the other is the signal/slot mechanism.

This one is called, unsurprisingly, whenever the user or a part of the program resizes the widget.

```
private:
    QPushButton *quit;
    QScrollBar  *sBar;
    QLCDNumber  *lcd;
};
```

In chapter three, MyWidget didn’t remember its child widgets. Now it does (so it can resize them).

```
MyWidget::MyWidget( QWidget *parent, const char *name )
        : QWidget( parent, name )
{
    setMinimumSize( 200, 200 );
```

Since we want MyWidget to handle all sizes greater than 200 by 200 pixels, we only set a minimum size.

```
    lcd  = new QLCDNumber( 2, this, "lcd" );
    lcd->move( 10, quit->y() + quit->height() + 10 );
```

`lcd` is a QLCDNumber, a widget which displays numbers in an LCD-like fashion. This instance is set up to display two digits, be a child of *this* and is named "lcd".

It is located at x position 10 and a y position giving 10 pixels between the quit button and the LCD number. It is a good idea to take a look at The Coordinate System. Note that its size is not set here, resizeEvent() will do that when MyWidget’s own size is set.

Hardcoding positions like this is tedious when you have many widgets, but Qt does not include a nice designer yet. Stay tuned.

```
        sBar = new QScrollBar( 0, 99,                       // range
                               1, 10,                       // line/page steps
                               0,                           // inital value
                               QScrollBar::Horizontal,      // orientation
                               this, "scrollbar" );
```

QScrollBar is a classical window system scroll bar. This is one of several possible constructors, there are other constructors that take fewer arguments but then we'd have to set up the same state using several function calls.

We don't set up the scroll bar's geometry, the resize event will handle it.

```
        connect( sBar, SIGNAL(valueChanged(int)), lcd, SLOT(display(int)) );
```

Here we use the signal/slot mechanism to connect the scroll bar's valueChanged() signal to the LCD number's display() slot.

Whenever the scroll bar's value changes, it broadcasts the new value by emitting the valueChanged() signal. Since that signal is connected to the LCD number's display() slot, the slot is called when the signal is broadcast. Neither of the objects know about the other. This is essential in component programming.

Slots are otherwise normal C++ member functions and follow the normal C++ access rules.

```
    void MyWidget::resizeEvent( QResizeEvent * )
    {
        sBar->setGeometry( 10, height() - 10 - 16, width() - 20, 16 );
        lcd->resize( sBar->width(), sBar->y() - lcd->y() - 5 );
    }
```

Here we want to fit the child widgets nicely inside our widget when it is resized. We want the LCD number to be above the scroll bar. First we position the scroll bar at the bottom of MyWidget.

The scroll bar's geometry is set so there is a 10-pixel border to the left, right and below it, and its height is fixed at 16 pixels, which is recommended by most style guides.

Remember that the order of arguments to setGeometry() is left, top, width, height.

The x position is easy; 10 pixels from the left edge. The y position is calculated from the total height of the widget. We subtract 10 pixels for the border and 16 pixels for the scroll bar's height. The width of the scroll bar should be the entire width of MyWidget less a 10 pixel borders on each side. As mentioned above, we give the scroll bar a height of 16 pixels.

Finally, the LCD number is resized. The width is easy. We give the same width as the scroll bar. The height is the distance from the top of the scroll bar to the top of the LCD number less a 5 pixel border. Remember from the constructor that we have positioned the LCD number below the quit button.

Auto-placement of widgets like we do here is tedious. Qt includes geometry management classes that make this job easier. The classes are QBoxLayout and QGridLayout. Later versions of this tutorial will include chapters on geometry management. Stay tuned!

5.2. Behavior

The LCD number reflects everything you do to the scroll bar, and the widget handles resizing well.

5.3. Excercises

Try changing the LCD number to add more digits or change mode. You can even add four push buttons to set mode.

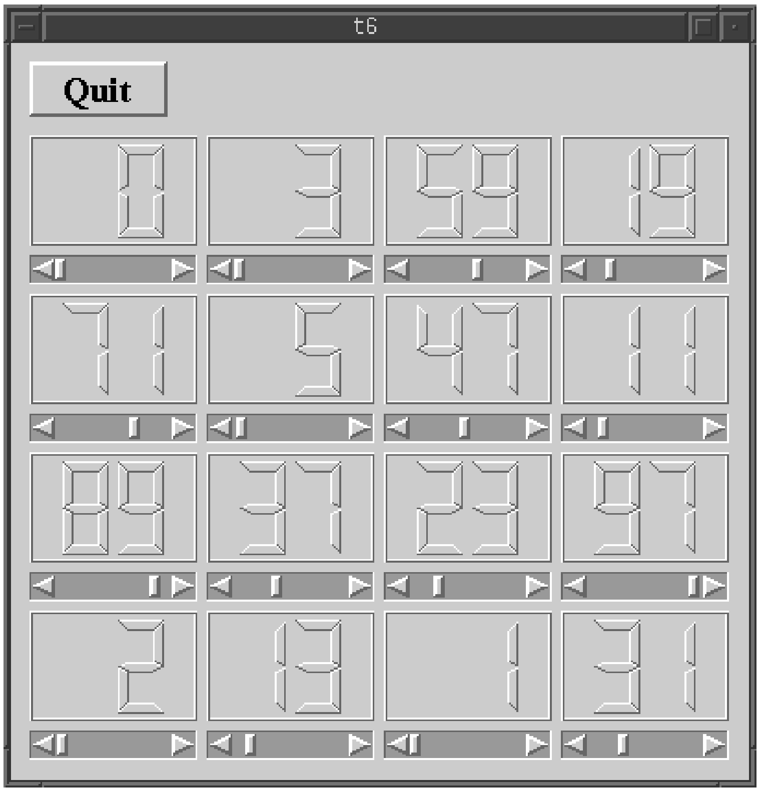
You can also change the scroll bar's range.

Try to make the application quit when the LCD number overflows.

5.3.1. URL

Generated from http://www.troll.no/qt/t5.html.

6. Chapter 6: Building Blocks Galore!



This example shows how to encapsulate two widgets into a new component and how easy it is to use many widgets. For the first time, we use a custom widget as a child widget.

```
/****************************************************************
**
** Qt tutorial 6
**
****************************************************************/
#include <qapplication.h>
#include <qpushbutton.h>
#include <qscrollbar.h>
#include <qlcdnumber.h>
#include <qfont.h>
class LCDRange : public QWidget
{
public:
    LCDRange( QWidget *parent=0, const char *name=0 );
protected:
    void resizeEvent( QResizeEvent * );
private:
    QScrollBar  *sBar;
    QLCDNumber  *lcd;
};
LCDRange::LCDRange( QWidget *parent, const char *name )
        : QWidget( parent, name )
{
    lcd  = new QLCDNumber( 2, this, "lcd"  );
    lcd->move( 0, 0 );
    sBar = new QScrollBar( 0, 99,                       // range
                           1, 10,                       // line/page steps
```

```
                           0,                            // inital value
                           QScrollBar::Horizontal,       // orientation
                           this, "scrollbar" );
    connect( sBar, SIGNAL(valueChanged(int)), lcd, SLOT(display(int)) );
}
void LCDRange::resizeEvent( QResizeEvent * )
{
    sBar->setGeometry( 0, height() - 16, width(), 16 );
    lcd->resize( width(), sBar->y() - 5 );
}
class MyWidget : public QWidget
{
public:
    MyWidget( QWidget *parent=0, const char *name=0 );
protected:
    void resizeEvent( QResizeEvent * );
private:
    QPushButton *quit;
    LCDRange *value[16];
};
MyWidget::MyWidget( QWidget *parent, const char *name )
        : QWidget( parent, name )
{
    setMinimumSize( 200, 300 );
    quit = new QPushButton( "Quit", this, "quit" );
    quit->setGeometry( 10, 10, 75, 30 );
    quit->setFont( QFont( "Times", 18, QFont::Bold ) );
    connect( quit, SIGNAL(clicked()), qApp, SLOT(quit()) );
    for( int i = 0 ; i < 16 ; i++ )
        value[i] = new LCDRange( this );
}
void MyWidget::resizeEvent( QResizeEvent * )
{
    int startx      = 10;
    int starty      = quit->y() + quit->height() + 10;
    int valueWidth  = (width() - startx - 10 - 3*5)/4;
    int valueHeight = (height() - starty - 10 - 3*5)/4;
    for( int i = 0 ; i < 16 ; i++ )
        value[i]->setGeometry( startx + (i%4)*(5+valueWidth),
                               starty + (i/4)*(5+valueHeight),
                               valueWidth, valueHeight );
}
int main( int argc, char **argv )
{
    QApplication a( argc, argv );
    MyWidget w;
    w.setGeometry( 100, 100, 400, 400 );
    a.setMainWidget( &w );
    w.show();
    return a.exec();
}
```

6.1. Line by Line Walk-Through

```
class LCDRange : public QWidget
{
public:
    LCDRange( QWidget *parent=0, const char *name=0 );
protected:
    void resizeEvent( QResizeEvent * );
private:
    QScrollBar  *sBar;
    QLCDNumber  *lcd;
};
```

The LCDRange widget is an encapsulation of a QScrollBar and a QLCDNumber, connected together.

```
LCDRange::LCDRange( QWidget *parent, const char *name )
        : QWidget( parent, name )
{
    lcd  = new QLCDNumber( 2, this, "lcd"  );
    lcd->move( 0, 0 );
    sBar = new QScrollBar( 0, 99,                          // range
                           1, 10,                          // line/page steps
                           0,                              // inital value
                           QScrollBar::Horizontal,         // orientation
                           this, "scrollbar" );
    connect( sBar, SIGNAL(valueChanged(int)), lcd, SLOT(display(int)) );
}
```

This is essentially lifted straight from the MyWidget constructor in chapter five. The only difference is that `lcd` is now placed at coordinate 0,0 (the top left corner of the LCDRange widget) since we're making a new widget and don't want a border. If the widget's user wants a border, he/she can make a border around the LCDRange.

```
void LCDRange::resizeEvent( QResizeEvent * )
{
    sBar->setGeometry( 0, height() - 16, width(), 16 );
    lcd->resize( width(), sBar->y() - 5 );
}
```

Since LCDRange doesn't have either a border or quit button, the resize event is simpler than the one in MyWidget.

```
private:
    QPushButton *quit;
    LCDRange *value[16];
};
```

MyWidget now contains the familiar quit button and an array of 16 LCDRange pointers.

```
    for( int i = 0 ; i < 16 ; i++ )
        value[i] = new LCDRange( this );
```

In MyWidget's constructor, we create 16 LCDRanges, all with `this` as parent. Remember that these will be deleted by Qt when MyWidget is destroyed.

```
void MyWidget::resizeEvent( QResizeEvent * )
{
    int startx      = 10;
    int starty      = quit->y() + quit->height() + 10;
    int valueWidth  = (width() - startx - 10 - 3*5)/4;
    int valueHeight = (height() - starty - 10 - 3*5)/4;
    for( int i = 0 ; i < 16 ; i++ )
        value[i]->setGeometry( startx + (i%4)*(5+valueWidth),
                               starty + (i/4)*(5+valueHeight),
                               valueWidth, valueHeight );
}
```

In the resize event, we calculate and set the geometry of all 16 LCDRanges in a simple for loop.

First we calculate the top left position of the top left LCDRange. We give it a 10 pixel border to the left and a 10 pixel border up to the quit button (see The Coordinate System)

We then calculate the size of each LCDRange. We want a grid of 4x4 widgets with a 10 pixel border to the right and 5 pixels between each one (i.e. 3 borders of 5 pixels in each direction). Note that since we divide by the integer 4, we will get small roundoff errors.

Next, we set each individual geometry in a simple loop. We calculate the position using integer modulo and divide and remember to add 5 for the border between each one.

6.2. Behavior

This program shows how easy it is to use many widgets at a time. Each single one behaves like the scroll bar and LCD number in the previous chapter. Again, the difference lies in the implementation.

6.3. Excercises

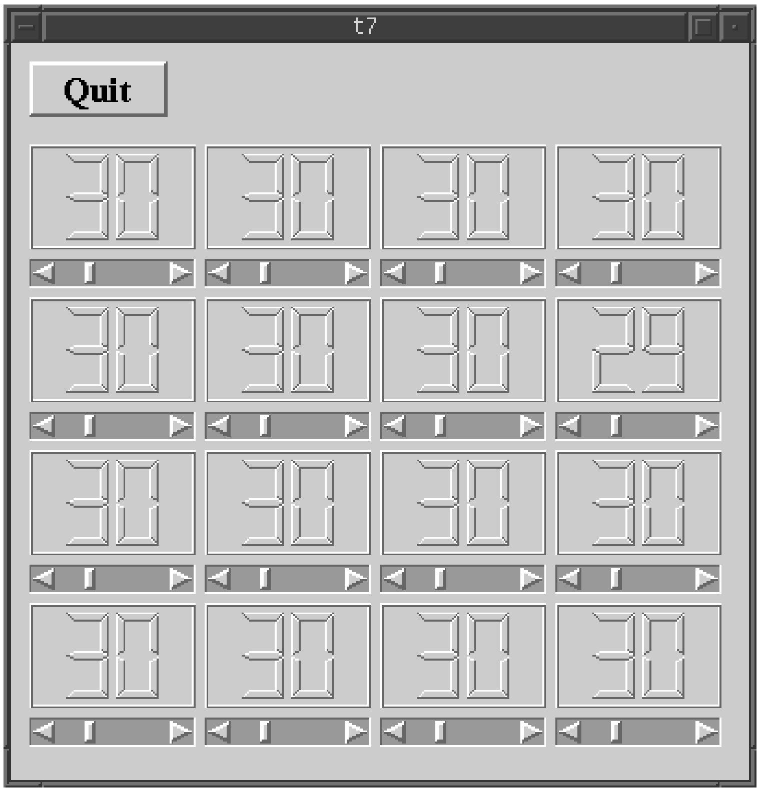
Change MyWidget so each LCDRange is guaranteed to be square.

Initialize each scroll bar with a different/random value on startup.

6.3.1. URL

Generated from http://www.troll.no/qt/t6.html.

7. Chapter 7: One Thing Leads to Another



This example shows how to create custom widgets with signals and slots, and how to connect them together in more complex ways. For the first time, the source is split among several files.

- lcdrange.h contains the LCDRange class definition
- lcdrange.cpp contains the LCDRange implementation
- main.cpp contains MyWidget and main.
- Makefile contains some rules for generating the meta object information necessary for signal/slot creation.

7.1. Line by Line Walk-Through

lcdrange.h

This file is mainly lifted from main.cpp in chapter 6 and only the changes are noted here.

```
#ifndef LCDRANGE_H
#define LCDRANGE_H
```

This is the classical C construction to avoid errors if a header file happens to be included more than once. If you don't use it already: It is a very good habit. The #ifndef should enclose *all* of the header file.

```
#include <qwidget.h>
class QScrollBar;
class QLCDNumber;
```

`qwidget.h` is included. LCDRange inherits QWidget, and the header file of a parent class must always be included. Until now, `qwidget.h` has been included indirectly via other header files like `qpushbutton.h.`

Since the class declaration only uses pointers to QScrollBar and QLCDNumber it does not need their definitions, so we merely declare their names in the header files. This makes the job a little easier for the compiler.

```
class LCDRange : public QWidget
{
    Q_OBJECT
public:
    LCDRange( QWidget *parent=0, const char *name=0 );
```

Note the Q_OBJECT. This macro must be included in *all* classes that contain signals and/or slots. For the curious, it defines the functions that are implemented in the meta object file.

```
    int value() const;
public slots:
    void setValue( int );
signals:
    void valueChanged( int );
```

These three members make up an interface between this widget and other components in a program. Until now, LCDRange didn't really have an interface at all.

value() is a public function for accessing the value of the LCDRange. setValue() is our first custom slot and valueChanged() is our first custom signal.

Slots must be implemented in the normal way (remember, a slot is also a C++ member function). Signals are automatically implemented in the meta object file. Signals follow the access rules of protected C++ functions, i.e. they can only be emitted by the class they are defined in or by classes inheriting from it.

lcdrange.cpp

This file is mainly lifted from t6/main.cpp and only the changes are noted here.

```
        connect( sBar, SIGNAL(valueChanged(int)), lcd, SLOT(display(int)) );
        connect( sBar, SIGNAL(valueChanged(int)), SIGNAL(valueChanged(int)) );
```

This code is from the LCDRange constructor.

The first connect is the same you've seen in the previous chapter. The second is new: It connects sBar's valueChanged() signal to this object's valueChanged *signal.* connect() with 3 arguments always connect to signals or slots in `this` object.

Yes, that's right. Signals can be connected to other signals. When the first is emitted, the second signal is also emitted.

Let's look at what happens when the user operates the scroll bar: The scroll bar sees that its value has changed, and emits the valueChanged() signal. That signal is connected both to the display() slot of the QLCDNumber and to the valueChanged() signal of the LCDRange.

Thus, when the signal is emitted, LCDRange emits its own valueChanged() signal. In addition, QLCDNumber::display() is called and shows the new number.

Note that you're not guaranteed any particular order of execution.

```
int LCDRange::value() const
{
    return sBar->value();
}
```

The implementation of value() is straight forward, it simply returns the scroll bar's value.

```
void LCDRange::setValue( int value )
{
    sBar->setValue( value );
}
```

The implementation of setValue() is equally straight forward. Note that since the scroll bar and LCD number is connected, setting the scroll bar's value automatically updates the LCD number as well. In addition, the scroll bar will automatically adjust the value if it is outside the scroll bar's legal range.

main.cpp

```
for( int i = 0 ; i < 16 ; i++ ) {
    value[i] = new LCDRange( this );
    if ( i > 0 )
        connect( value[i], SIGNAL(valueChanged(int)),
                 value[i - 1], SLOT(setValue(int)) );
}
```

Apart from moving LCDRange into two separate files, these lines are the only changes to the main.cpp file. As in the previous chapter, we create 16 LCDRange objects. In addition, we now connect them together using the signal/slot mechanism. Each one has its valueChanged() signal connected to the setValue() slot in the previous one. Since LCDRange emits the signal valueChanged() when its value changes (surprise!), we are here creating a "chain" of signals and slots.

7.2. Behavior

On startup, the program's appearance is identical to the previous one. Try operating the scroll bar to the bottom right...

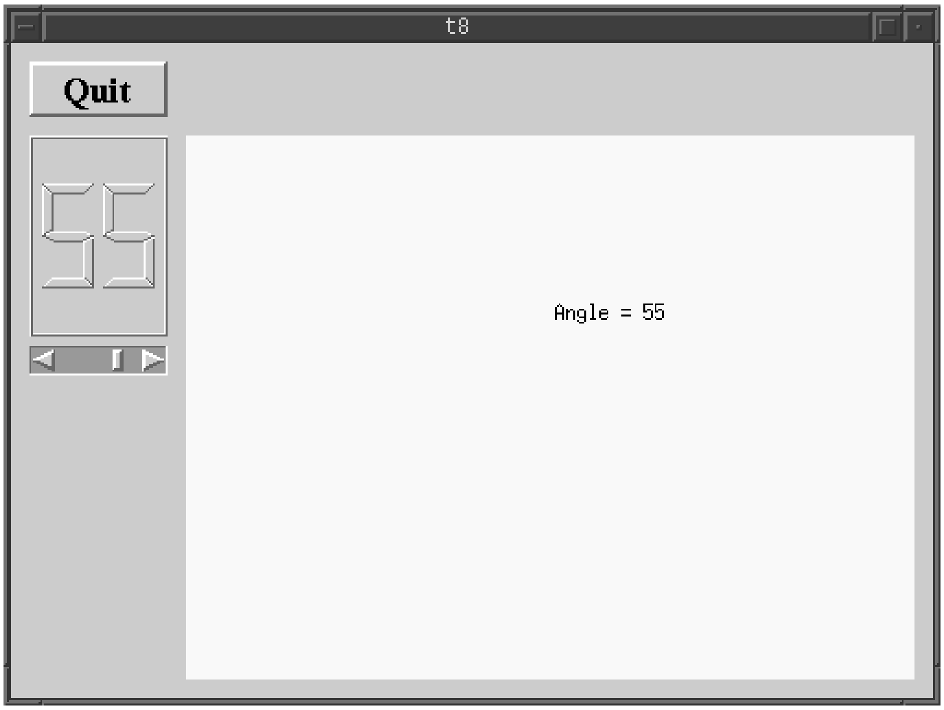
7.3. Excercises

Use the bottom right scroll bar to set all LCDs to 30. Then set the top half to 29 by using the rightmost scroll bar on the 2nd row. Now, use the one to the left of the last one operated to set the first seven LCDs back to 30. Click on the left arrow on the bottom right scroll bar. What happens? Why is this the correct behavior?

7.3.1. URL

Generated from http://www.troll.no/qt/t7.html.

8. Chapter 8: Preparing for Battle



In this example, we introduce the first custom widget that can paint itself.

- lcdrange.h contains the LCDRange class definition
- lcdrange.cpp contains the LCDRange implementation
- cannon.h contains the CannonField class definition
- cannon.cpp contains the CannonField implementation
- main.cpp contains MyWidget and main.
- Makefile contains some rules for generating the meta object information necessary for signal/slot creation.

8.1. Line by Line Walk-Through

lcdrange.h

This file is very similar to the lcdrange.h in chapter 7. We have added one slot, setRange().

```
void setRange( int minVal, int maxVal );
```

We now add the possibility of setting the range of the LCDRange. Until now, it has been fixed at 0..99.

lcdrange.cpp

```
void LCDRange::setRange( int minVal, int maxVal )
{
    if ( minVal < 0 || maxVal > 99 || minVal > maxVal ) {
        warning( "LCDRange::setRange(%d,%d)\n"
                 "\tRange must be 0..99\n"
                 "\tand minVal must not be greater than maxVal",
                 minVal, maxVal );
        return;
    }
    sBar->setRange( minVal, maxVal );
}
```

setRange() sets the range of the scroll bar in the LCDRange. Since we have set up the QLCDNumber to always display two digits, we want to limit the possible range of `minVal` and `maxVal` to 0..99 to avoid overflow of the QLCDNumber. We could have allowed values in -9..99 but choose not to. If the arguments are illegal, we use Qt's warning() function to issue a warning to the user and return immediately. warning() is a printf-like function that by default sends its output to `stderr.` You can install your own handler function if you want.

cannon.h

CannonField is a new custom widget that knows how to display itself.

```
class CannonField : public QWidget
{
    Q_OBJECT
public:
    CannonField( QWidget *parent=0, const char *name=0 );
```

CannonField inherits QWidget and we use the same idiom as for LCDRange.

```
    int angle() const { return ang; }
public slots:
    void setAngle( int degrees );
signals:
    void angleChanged( int );
```

For the time being, CannonField only contains an angle value for which we provide an interface using the same idiom as for value in LCDRange.

```
protected:
    void paintEvent( QPaintEvent * );
```

This is the second of the many event handlers in QWidget that we encounter. This virtual function is called by Qt whenever a widget needs to update itself (i.e. paint the widget's surface).

cannon.cpp

```
CannonField::CannonField( QWidget *parent, const char *name )
        : QWidget( parent, name )
{
```

Again, we use the same idiom as for LCDRange in the previous chapter.

```
    ang = 45;
}
```

The constructor simply initializes the angle value to 45 degrees.

```
void CannonField::setAngle( int degrees )
{
    if ( degrees < 5 )
```

```
        degrees = 5;
    if ( degrees > 70 )
        degrees = 70;
    if ( ang == degrees )
        return;
    ang = degrees;
    repaint();
    emit angleChanged( ang );
}
```

This function sets the angle value. We have chosen a legal range 5..70 and adjust the given number of degrees accordingly. We have chosen not to issue a warning if the new angle is out of range.

If the new angle equals the old one, we return immediately. It is impotrant to only emit the signal angleChanged() when the angle *really* has changed.

Then we set the new angle value and repaint our widget. The repaint() function will clear the widget (i.e. fill it with its background color) and send a paint event to the widget. This will immediately call the paint event function of the widget. If you want Qt to send the event later (when it regains control) use the update() function.

Finally, we emit the angleChanged() signal to tell the outside world that the angle has changed. The `emit` keyword is unique to Qt and not regular C++ syntax. In fact, it is a macro.

```
void CannonField::paintEvent( QPaintEvent * )
{
    QString s;
    s.sprintf( "Angle = %i", ang );
    drawText( 200, 100, s );
}
```

This is our first attempt to write a paint event handler. The event argument contains a description of the paint event. QPaintEvent contains the rectangular area in the widget that must be updated. For the time being, we will be lazy and always update the entire widget.

Our code displays the angle value in the widget at a fixed position. First we create a QString object. QString is a Qt's string class (see the documentation for details). Then we set the string using the QString::sprintf() function, which is similar to sprintf(). Finally, we draw the text at position 200,100 (relative to the baseline of the text) in the widget using the QWidget::drawText() function. Normally, you will use a QPainter to draw in a widget, but drawText() is a convenience function for drawing text. In the next chapter, you'll see how QPainter works.

main.cpp

```
#include "cannon.h"
```

We include our new class.

```
class MyWidget : public QWidget
{
public:
    MyWidget( QWidget *parent=0, const char *name=0 );
protected:
    void resizeEvent( QResizeEvent * );
private:
    QPushButton *quit;
    LCDRange    *angle;
    CannonField *cannonField;
};
```

This time, we include a single LCDRange and a CannonField in our top level widget.

```
    angle  = new LCDRange( this, "angle" );
    angle->setRange( 5, 70 );
```

```
        angle->setGeometry( 10, quit->y() + quit->height() + 10, 75, 130 );
```

In the contructor, we create and set up our LCDRange with a range 5..70 and put it 10 pixels below the quit button and fix its size to 75x130.

```
        cannonField = new CannonField( this, "cannonField" );
        cannonField->move( angle->x() + angle->width() + 10, angle->y() );
        cannonField->setBackgroundColor( QColor( 250, 250, 200) );
```

We create and set up our CannonField. Its position is 10 pixels to the right of the LCDRange and the same y position as LCDRange. The size will be set by the resize event.

Then we set the background color of the CannonField. QColor is one of Qt's color classes. Here we hard code an RGB value red=250, green=250 and blue=200 (which gives an easter-like yellow pastel color). The range of the RGB values is 0..255. You can also set up a QColor using the HSV color model.

Another color class, QPalette, provides a complete array of colors, so you can change the colors without messing up the 3D effects.

```
        connect( angle,SIGNAL(valueChanged(int)), cannonField,SLOT(setAngle(int)));
        connect( cannonField,SIGNAL(angleChanged(int)), angle,SLOT(setValue(int)));
```

Here we connect the valueChanged() signal of the LCDRange to the setAngle() slot of the CannonField. This will update CannonField's angle value whenever the user operates the LCDRange. We also make the reverse connection, so that changing the angle in the CannonField will update the LCDRange value. In our example, we never change the angle of the CannonField directly, but by doing the last connect(), we ensure that no future changes will disrupt the synchronization between those two values.

This illustrates the power of component programming and proper encapsulation.

Notice how important it is to only emit the angleChanged() signal when the angle actually changes. If both the LCDRange and the CannonField had had this design flaw, the program would have entered an infinite loop upon the first change of one of the values.

```
        angle->setValue( 60 );
```

Finally, we set an initial angle value. Note that this will trigger the connection from LCDRange to CannonField.

```
    void MyWidget::resizeEvent( QResizeEvent * )
    {
        cannonField->resize( width()  - cannonField->x() - 10,
                             height() - cannonField->y() - 10 );
    }
```

We give the CannonField all the space we can, except for the 10 pixel border to the bottom and to the right.

8.2. Behavior

When the scroll bar is operated, the CannonField displays the new angle value. Upon resize, CannonField is gives as much space as possible.

On Windows machines with an 8-bit display, the new background color is dithered to death. The next chapter works around this.

8.3. Excercises

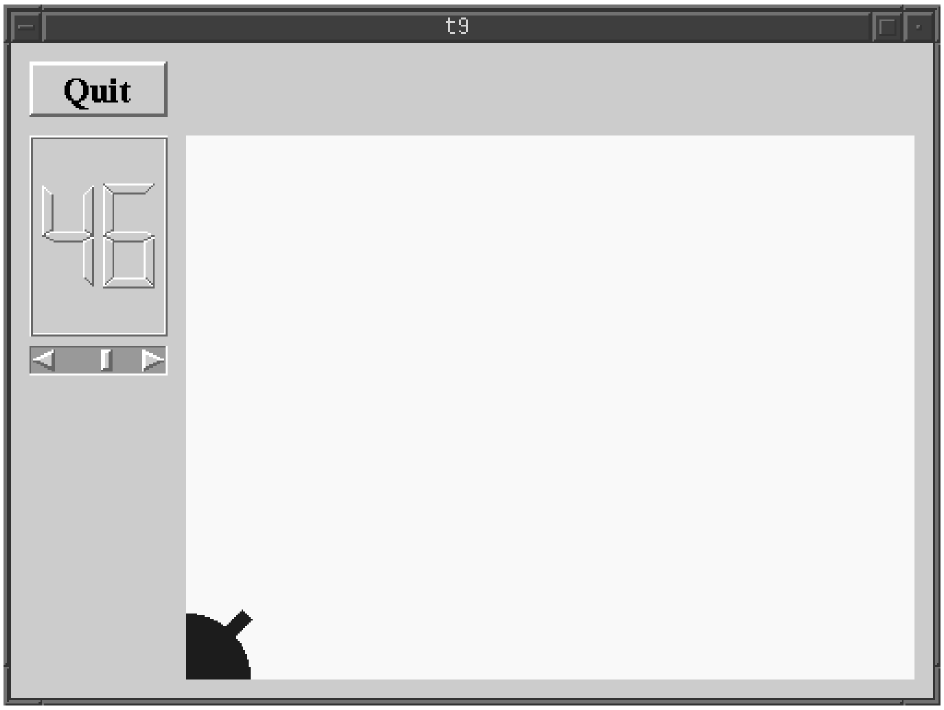
Make the position of the printed text dependent on the angle value.

Change the resize event to give maximum space to the LCDRange instead of the CannonField.

8.3.1. URL

Generated from http://www.troll.no/qt/t8.html.

9. Chapter 9: With Cannon You Can



In this example, we use a QPainter for the first time. It is used to draw a cute little blue cannon. Only cannon.cpp differs from the previous chapter.

- lcdrange.h contains the LCDRange class definition
- lcdrange.cpp contains the LCDRange implementation
- cannon.h contains the CannonField class definition
- cannon.cpp contains the CannonField implementation
- main.cpp contains MyWidget and main.
- Makefile contains some rules for generating the meta object information necessary for signal/slot creation.

9.1. Line by Line Walk-Through

cannon.cpp

```
void CannonField::paintEvent( QPaintEvent * )
{
    QPainter p;
    QBrush   brush( blue );
    QPen     pen( NoPen );
```

In order to paint our widget, we use a QPainter. QPainter is Qt's painting engine. It can render 2D graphics on several types of paint devices. QWidget is a paint device. See the QPainter documentation for details.

QPainter uses a QPen and a QBrush. The pen specifies a line style and a text color while the brush specifies a fill color and/or a fill pattern. In our paintEvent, we want to use a blue brush and an invisible pen.

```
    p.begin( this );
```

Calling begin() tells the painter to start painting on a paint device. Here we start painting on the CannonField widget.

```
        p.setBrush( brush );
        p.setPen( pen );
```

Here we set the brush and pen created above.

```
        p.translate( 0, rect().bottom() );
```

The translate() function translates the coordinate system of the QPainter, i.e. moves it by an offset. Here we set the (0,0) point to the bottom left corner of the widget. The x and y directions remain unchanged, i.e. all the y coordinates inside the widget are now negative (see The Coordinate System).

```
        p.drawPie( QRect(-35, -35, 70, 70), 0, 90*16 );
```

The drawPie() function draws a pie shape inside the specified rectangle using a start angle and an arc length. The angles are specified in 1/16th of a degree. Zero degrees is at the 3 o'clock position. The drawing direction is counter-clockwise. Here we draw a quarter of a circle in the bottom left corner of the widget. The pie is filled with blue and has no outline.

```
        p.rotate( -ang );
```

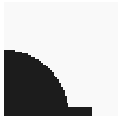
The rotate() function rotates the coordinate system of the QPainter around the point (0,0). The rotation argument is a `float` given in degrees (not given in 1/16th of a degree as above) and clockwise. Here we rotate the coordinate system `ang` degrees counter-clockwise.

```
        p.drawRect( QRect(33, -4, 15, 8) );
```

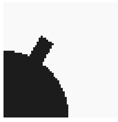
The drawRect() function draws the specified rectangle. Here we draw the barrel of the cannon.

It can often be difficult to envision the resulting drawing when the coordinate system has been transformed (translated, rotated, scaled or sheared) as above.

In this case, the coordinate system is first translated, then rotated. If the rectangle QRect(33, -4, 15, 8) had been drawn in the translated coordinate system, it would have looked like this:



Note that the rectangle is clipped by the border of the CannonField widget. When we rotate the coordinate system, for instance 60 degrees, the rectangle will be rotated around (0,0), which is the bottom left corner, since we have translated the coordinate system. The result looks like this:



```
        p.end();
```

Calling end() tells the painter to end painting on a paint device. We're done, except that we haven't explained why Windows didn't dither this time.

```
    int main( int argc, char **argv )    {
        QApplication::setColorSpec( QApplication::CustomColor );
```

We tell Qt that we want a different color allocation strategy for this program. There is no single correct color allocation strategy. Some strategies look good for some programs, some look bad.

`CustomColor` gives Qt applications better colors on Windows. It has no effect under X11. There are several options. You can read about them in the documentation.

9.2. Behavior

When the scroll bar is operated, the angle of the drawn cannon changes accordingly.

You may notice that the cannon flickers annoyingly, especially on a slow machine. We'll fix this in the next chapter.

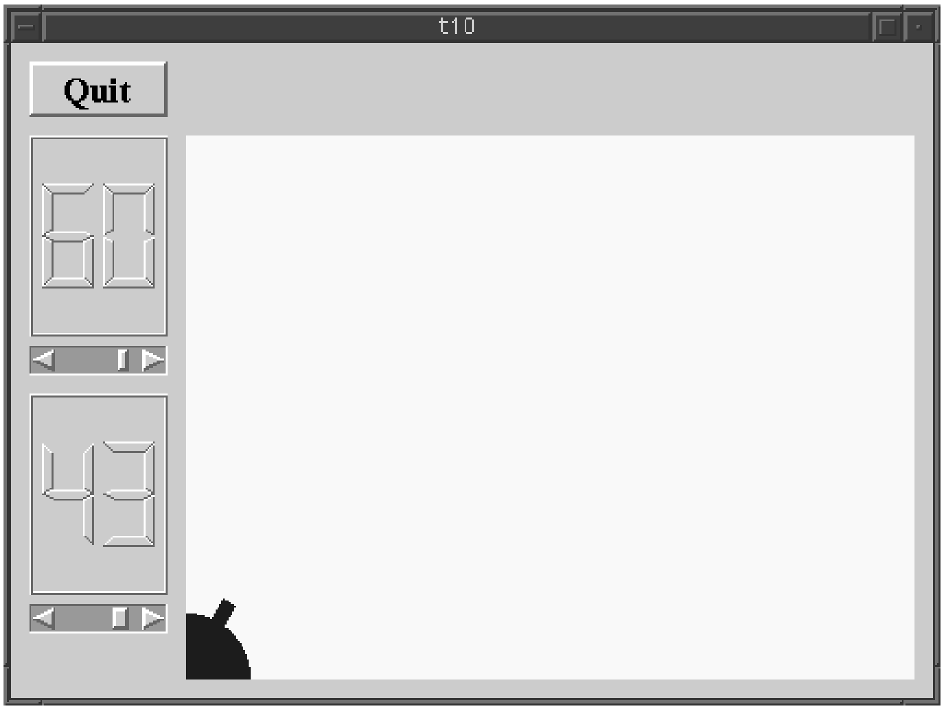
9.3. Excercises

Set a background pixmap for the CannonField widget.

9.3.1. URL

Generated from http://www.troll.no/qt/t9.html.

10. Chapter 10: Smooth as Silk



In this example, we introduce painting in a pixmap to remove flickering. We also add a force control.

- lcdrange.h contains the LCDRange class definition
- lcdrange.cpp contains the LCDRange implementation
- cannon.h contains the CannonField class definition
- cannon.cpp contains the CannonField implementation
- main.cpp contains MyWidget and main.
- Makefile contains some rules for generating the meta object information necessary for signal/slot creation.

10.1. Line by Line Walk-Through

cannon.h

The CannonField now has a force value in addition to the angle.

```
        int   angle() const { return ang; }
        int   force() const { return f; }
    public slots:
        void  setAngle( int degrees );
        void  setForce( int newton );
    signals:
        void  angleChanged( int );
        void  forceChanged( int );
```

The interface to the force follows the same practice as for angle.

```
    private:
        void  paintCannon( QPainter * );
        QRect cannonRect() const;
```

We have moved the painting of the cannon to a separate function and have also put the defintion of the cannon's enclosing rectangle in a separate function.

```
    int ang;
    int f;
};
```

The force is stored in the integer f.

cannon.cpp

```
#include <qpixmap.h>
```

We include the QPixmap class definition.

```
CannonField::CannonField( QWidget *parent, const char *name )
        : QWidget( parent, name )
{
    ang = 45;
    f   = 0;
}
```

The force (f) is initialized to zero.

```
void CannonField::setAngle( int degrees )
{
    if ( degrees < 5 )
        degrees = 5;
    if ( degrees > 70 )
        degrees = 70;
    if ( ang == degrees )
        return;
    ang = degrees;
    repaint( cannonRect(), FALSE );
    emit angleChanged( ang );
}
```

We have made a slight change in the setAngle() function. It repaints only the portion of the widget that contains the cannon. The FALSE argument indicates that the specified rectangle should not be erased before a paint event is sent to the widget.

```
void CannonField::setForce( int newton )
{
    if ( newton < 0 )
        newton = 0;
    if ( f == newton )
        return;
    f = newton;
    emit forceChanged( f );
}
```

The implementation of setForce() is quite similar to that of setAngle(). The only difference is that we don't show the force value, hence we don't need to repaint the widget.

```
void CannonField::paintEvent( QPaintEvent *e )
{
    QRect updateR = e->rect();
    QPainter p;
    p.begin( this );
    if ( updateR.intersects( cannonRect() ) )
        paintCannon( &p );
    p.end();
}
```

We have now optimized the paint event to repaint only the parts of the widget that need updating. First we fetch the update rectangle parameter from the QPaintEvent. Then we begin painting on this widget. We check if the update rectangle intersects with the cannon rectangle. If it does, we paint the cannon. Note that we pass a pointer to the painter to the paintCannon() function.

```
    const QRect barrel_rect(33, -4, 15, 8);
```

We have chosen to define the barrel rectangle as a constant.

```
    void CannonField::paintCannon( QPainter *p )
    {
        QRect    cr = cannonRect();
```

First we fetch the cannon's enclosing rectangle and store it in `cr`.

```
        QPixmap  pix( cr.size() );
```

Here we encounter a new class; the QPixmap. QPixmap is an off-screen paint device which is handy for flicker-free painting. Here we create a pixmap with the same size as the cannon. We want to draw the cannon in this pixmap and then move the pixmap to the screen in one operation. This technique is called double-buffering and virtually removes flickering.

```
        QPainter tmp;
```

We create a temporary painter `tmp` for drawing in the pixmap.

```
        pix.fill( this, cr.topLeft() );
```

Upon creation, a pixmap is filled with arbitrary pixels, thus we first fill it with the widget's background. We have to specify the position where the pixmap will be placed in the widget, so that we can cope with any type of background (for instance background pixmaps).

```
        tmp.begin( &pix );
        tmp.setBrush( blue );
        tmp.setPen( NoPen );
```

Here we start painting on the pixmap. We set a blue brush and an invisible pen, as in the previous chapter. This time we use a shortcut. setPen() and setBrush() are overloaded functions in QPainter.

```
        tmp.translate( 0, pix.height() - 1 );
        tmp.drawPie( QRect( -35,-35, 70, 70 ), 0, 90*16 );
        tmp.rotate( -ang );
        tmp.drawRect( barrel_rect );
        tmp.end();
```

The code that draws the cannon is very similar to the code in the previous example. The only difference is the translation which now is relative to the pixmap height and that we have moved the barrel rectangle into a constant.

```
        p->drawPixmap( cr.topLeft(), pix );
```

Finally, we call drawPixmap() to zip the pixmap onto the screen in one lightning fast operation.

```
    QRect CannonField::cannonRect() const
    {
        QRect r( 0, 0, 50, 50 );
        r.moveBottomLeft( rect().bottomLeft() );
        return r;
    }
```

This function returns the rectangle enclosing the cannon in widget coordinates. First we create a rectangle with the size 50x50, then we set its bottom left corner equal to the widget's bottom left corner.

The rect() function returns the widget's enclosing rectangle in widget coordinates (i.e. the top left corner is 0,0).

main.cpp

```
class MyWidget : public QWidget
{
public:
    MyWidget( QWidget *parent=0, const char *name=0 );
protected:
    void resizeEvent( QResizeEvent * );
private:
    QPushButton *quit;
    LCDRange    *angle;
    LCDRange    *force;
    CannonField *cannonField;
};
```

We add a second LCDRange which will be used to set the force.

```
    force  = new LCDRange( this, "force" );
    force->setRange( 10, 50 );
    force->setGeometry( 10, angle->y() + angle->height() + 10, 75, 130 );
```

We place the `force` LCDRange widget below the `angle` widget and specify the range to 10..50.

```
    connect( force,SIGNAL(valueChanged(int)), cannonField,SLOT(setForce(int)));
    connect( cannonField,SIGNAL(forceChanged(int)), force,SLOT(setValue(int)));
```

We connect the `force` widget and the `cannonField` widget just like we did for the `angle` widget.

```
    force->setValue( 25 );
```

We initialize the force value to 25.

10.2. Behavior

The flicker has gone and we have a force control.

10.3. Excercises

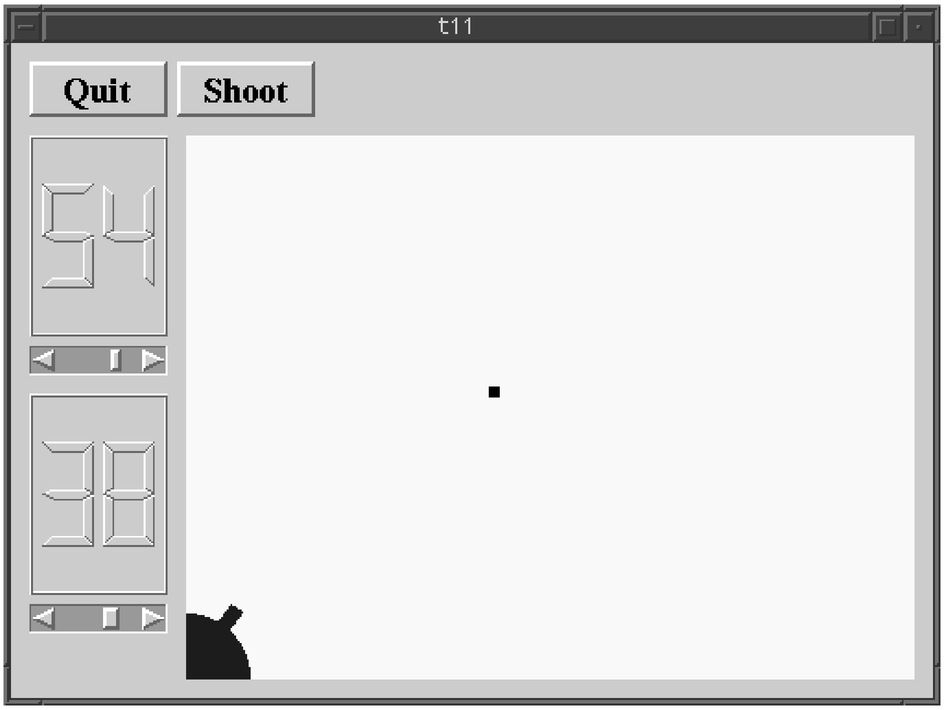
Make the size of the cannon barrel be dependent on the force.

Put the cannon in the bottom right corner.

10.3.1. URL

Generated from http://www.troll.no/qt/t10.html.

11. Chapter 11: Giving It a Shot



In this example, we introduce timer events to implement shooting.

- lcdrange.h contains the LCDRange class definition
- lcdrange.cpp contains the LCDRange implementation
- cannon.h contains the CannonField class definition
- cannon.cpp contains the CannonField implementation
- main.cpp contains MyWidget and main.
- Makefile contains some rules for generating the meta object information necessary for signal/slot creation.

11.1. Line by Line Walk-Through

cannon.h

The CannonField now has a shooting capabilities.

```
        bool  isShooting() const { return shooting; }
```

Returns TRUE if a shot is underway.

```
        void  shoot();
```

Calling this slot will make the cannon shoot if a shot is not in the air.

```
    protected:
        void  timerEvent( QTimerEvent * );
```

The timer event is the third type of widget event we encounter. You can make Qt call this event handler at regular intervals.

```
    private:
        void  stopShooting();
```

This private function stops a shot in mid-air.

```
        void  paintShot( QPainter * );
```

This private function paints the shot.

```
        QRect shotRect() const;
```

This private function returns the shot's enclosing rectangle if one is in the air, otherwise the returned rectangle is undefined.

```
        bool  shooting;
```

This private variable is TRUE when there's a shot in the air.

```
        int   timerCount;
        float shoot_ang;
        float shoot_f;
    };
```

These private variables contain information that describes the shot. The `timerCount` keeps track of the time passed since the shot was fired. The `shoot_ang` is the cannon angle and `shoot_f` is the cannon force when the shot was fired.

cannon.cpp

```
    #include <math.h>
```

We include the math library because we need the sin() and cos() functions.

```
    CannonField::CannonField( QWidget *parent, const char *name )
            : QWidget( parent, name )
    {
        ang           = 45;
        f             = 0;
        shooting      = FALSE;
        timerCount    = 0;
        shoot_ang     = 0;
        shoot_f       = 0;
    }
```

We initialize our new private variables.

```
    void CannonField::shoot()
    {
        if ( shooting )
            return;
        timerCount = 0;
        shoot_ang  = ang;
        shoot_f    = f;
        shooting   = TRUE;
        startTimer( 50 );
    }
```

This function shoots a shot unless a shot is in the air. The `timerCount` is reset to zero. The `shoot_ang` and `shoot_f` are set to the current cannon angle and force. `shooting` is set to TRUE to indicate that a shot is in the air. Finally we start timer events.

Calling startTimer() will make Qt send us timer events at regular intervals. In this case, we want timer events every 50 milliseconds.

```
    void CannonField::timerEvent( QTimerEvent * )
    {
        erase( shotRect() );
        timerCount++;
        QRect shotR = shotRect();
        if ( shotR.x() > width() || shotR.y() > height() ) {
            stopShooting();
            return;
        }
```

```
        repaint( shotR, FALSE );
    }
```

The timerEvent() function is an event handler that receives timer events from Qt. In our case, it is called every 50 milliseconds when a shot is in the air.

First we erase the old shot using the QWidget::erase() function. Note that this is not strictly necessary the first time the timerEvent() is called after a shot has been fired (but it does no harm). The `shotRect()` function (explained below) returns the enclosing rectangle of the shot at the current position.

Then we increment the `timerCount,` which has the effect of moving the shot one step along its trajectory.

Next, we fetch the new shot rectangle.

If the shot has moved beyond the right or bottom edge of the widget, we stop shooting and return.

If not, we paint the new shot by repainting the portion of the widget that contains the shot. The FALSE argument indicates that the specified rectangle should not be erased before a paint event is sent to the widget.

```
    void CannonField::paintEvent( QPaintEvent *e )
    {
        QRect updateR = e->rect();
        QPainter p;
        p.begin( this );
        if ( updateR.intersects( cannonRect() ) )
            paintCannon( &p );
        if ( shooting &&  updateR.intersects( shotRect() ) )
            paintShot( &p );
        p.end();
    }
```

This paint event function is basically the same as the one in the previous chapter. The only difference is that we also paint the shot if necessary.

```
    void CannonField::stopShooting()
    {
        shooting = FALSE;
        killTimers();
    }
```

This private function stops a shot in mid-air. First it resets the `shooting` variable, then it kills all timer events for this widget. It is also possible to kill a single timer.

```
    void CannonField::paintShot( QPainter *p )
    {
        p->setBrush( black );
        p->setPen( NoPen );
        p->drawRect( shotRect() );
    }
```

This private function paints the shot by drawing a black filled rectangle.

```
    QRect CannonField::shotRect() const
    {
        const double gravity = 4;
        double time      = timerCount / 4.0;
        double velocity  = shoot_f;
        double radians   = shoot_ang*3.14159265/180;
        double velx      = velocity*cos( radians );
        double vely      = velocity*sin( radians );
        double x0        = ( barrel_rect.right()  + 5 )*cos(radians);
        double y0        = ( barrel_rect.right()  + 5 )*sin(radians);
        double x         = x0 + velx*time;
        double y         = y0 + vely*time - 0.5*gravity*time*time;
        QRect r = QRect( 0, 0, 6, 6 );
```

```
        r.moveCenter( QPoint( qRound(x), height() - 1 - qRound(y) ) );
        return r;
    }
```

This private function calculates the center point of the shot and returns the enclosing rectangle of the shot. It uses the initial cannon force and angle in addition to `timerCount,` which increases as time passes.

The formula used is the classical Newtonian formula for frictionless movement in a gravity field. For simplicity, we've chosen to disregard any Einsteinian effects.

We calculate the center point in a coordinate system where y coordinates increase upwards. After we have calculated the center point, we construct a QRect with size 6x6 and move its center point to the point calculated above. In the same operation, we convert the point into the widget's coodinate system (see The Coordinate System).

The qRound() function is an inline function defined in qglobal.h (included by all other Qt header files). qRound() rounds a double to the closest integer.

main.cpp

```
class MyWidget : public QWidget
{
public:
    MyWidget( QWidget *parent=0, const char *name=0 );
protected:
    void resizeEvent( QResizeEvent * );
private:
    QPushButton *quit;
    QPushButton *shoot;
    LCDRange    *angle;
    LCDRange    *force;
    CannonField *cannonField;
};
```

The only addition is the shoot button.

```
    shoot = new QPushButton( "Shoot", this, "shoot" );
    shoot->setGeometry( 90, 10, 75, 30 );
    shoot->setFont( QFont( "Times", 18, QFont::Bold ) );
```

In the constructor we create and set up the shoot button exactly like we did with the quit button. Note that the first argument to the constructor is the button text and the third is the widget's name.

```
    connect( shoot, SIGNAL(clicked()), cannonField, SLOT(shoot()) );
```

Connects the clicked() signal of the shoot button to the shoot() slot of the CannonField.

11.2. Behavior

The cannon can shoot, but there's nothing to shoot at.

11.3. Excercises

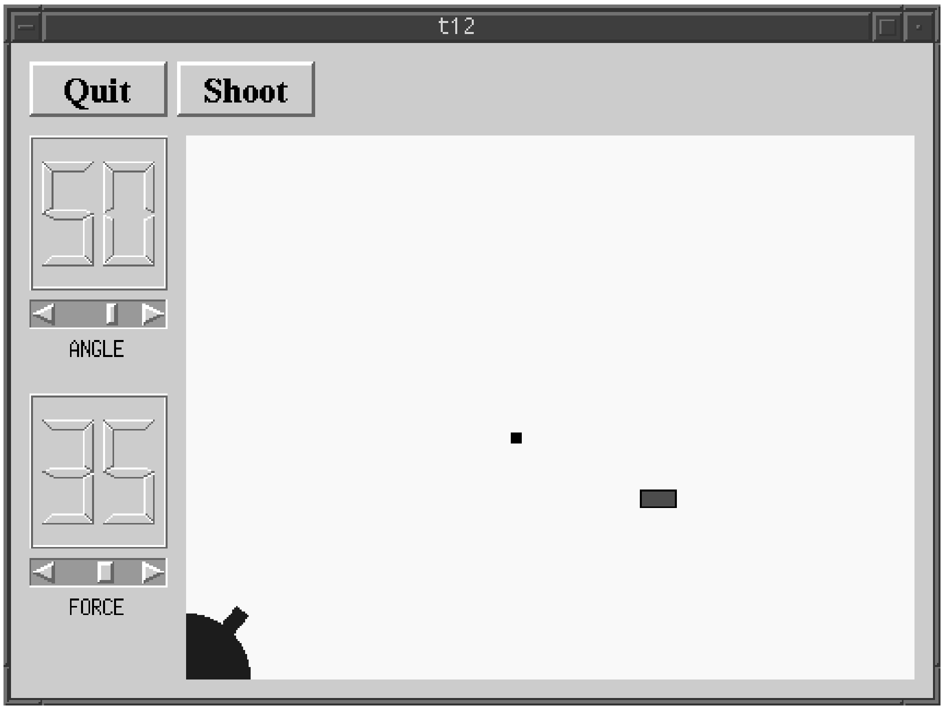
Make the shot a filled circle (hint: QPainter::drawEllipse()).

Change the color of the cannon when a shot is in the air.

11.3.1. URL

Generated from http://www.troll.no/qt/t11.html.

12. Chapter 12: Hanging in the Air the Way Bricks Don't



In this example, we extend our LCDRange class to include a text label. We also provide something to shoot at.

- lcdrange.h contains the LCDRange class definition
- lcdrange.cpp contains the LCDRange implementation
- cannon.h contains the CannonField class definition
- cannon.cpp contains the CannonField implementation
- main.cpp contains MyWidget and main.
- Makefile contains some rules for generating the meta object information necessary for signal/slot creation.

12.1. Line by Line Walk-Through

lcdrange.h

The LCDRange now has a text label.

```
class QLabel;
```

We name declare QLabel since we want to use a pointer to it in the class definition.

```
class LCDRange : public QWidget
{
    Q_OBJECT
public:
    LCDRange( QWidget *parent=0, const char *name=0 );
    LCDRange( const char *s, QWidget *parent=0, const char *name=0 );
```

We have added a new constructor that sets the label text in addition to the parent and name.

```
    const char *text()  const;
```

This function returns the label text.

```
        void setText( const char * );
```

This slot sets the label text.

```
    private:
        void init();
```

Since we now have two constructors, we have chosen to put the common initialization in the private init() function.

```
        QLabel      *label;
```

We also have a new private variable; a QLabel. QLabel is one of Qt's standard widgets and can show a text or a pixmap with or without a frame.

lcdrange.cpp

```
    #include <qlabel.h>
```

Here we include the QLabel class definition.

```
    LCDRange::LCDRange( QWidget *parent, const char *name )
            : QWidget( parent, name )
    {
        init();
    }
```

This constructor calls the init() function, which contains the common initialization code.

```
    LCDRange::LCDRange( const char *s, QWidget *parent, const char *name )
            : QWidget( parent, name )
    {
        init();
        setText( s );
    }
```

This constructor first calls init(), then sets the label text.

```
    void LCDRange::init()
    {
        lcd  = new QLCDNumber( 2, this, "lcd"  );
        lcd->move( 0, 0 );
        sBar = new QScrollBar( 0, 99,                          // range
                               1, 10,                          // line/page steps
                               0,                              // inital value
                               QScrollBar::Horizontal,         // orientation
                               this, "scrollbar" );
        label  = new QLabel( this, "label"  );
        label->setAlignment( AlignCenter );
        connect( sBar, SIGNAL(valueChanged(int)), lcd, SLOT(display(int)) );
        connect( sBar, SIGNAL(valueChanged(int)), SIGNAL(valueChanged(int)) );
    }
```

The set up of `lcd` and `sBar` is the same as in the previous chapter. Next, we create a QLabel and tell it to align the contents centered (both vertically and horizontally). The connect() statements have also been taken from the previous chapter.

```
    const char *LCDRange::text() const
    {
        return label->text();
    }
```

This function returns the label text.

```
    void LCDRange::setText( const char *s )
    {
        label->setText( s );
    }
```

This function sets the label text.

```
void LCDRange::resizeEvent( QResizeEvent * )
{
    lcd->resize( width(), height() - 41 - 5 );
    sBar->setGeometry( 0, lcd->height() + 5, width(), 16 );
    label->setGeometry( 0, lcd->height() + 16 + 5, width(), 20 );
}
```

This event handler is similar to the one in the previous chapter, except that we also must make room for the label. The number 41 in the lcd resize statment is the height of the scroll bar plus the height of the label plus the spacing between them (16+20+5).

cannon.h

The CannonField now has two new signals: hit() and missed(). In addition it contains a target.

```
    void  newTarget();
```

This slot creates a target at a new position.

```
signals:
    void  hit();
    void  missed();
```

The hit() signal is emitted when a shot hits the target. The missed() signal is emitted when the shot moves beyond the right or bottom edge of the widget (i.e. it has not hit the target).

```
    void  paintTarget( QPainter * );
```

This private function paints the target.

```
    QRect targetRect() const;
```

This private function returns the enclosing rectangle of the target.

```
    QPoint target;
```

This private variable contains the center point of the target.

cannon.cpp

```
#include <qdatetime.h>
```

We include the QDate, QTime and QDateTime class definitions.

```
#include <stdlib.h>
```

We include the stdlib library because we need the rand() function.

```
    newTarget();
```

This line has been added to the constructor. It creates a "random" position for the target. In fact, the newTarget() function will try to paint the target. Since we are in a constructor, the CannonField widget is invisible. Qt guarantees that no harm is done when painting a hidden widget.

```
void  CannonField::newTarget()
{
    static bool first_time = TRUE;
    if ( first_time ) {
        first_time = FALSE;
        QTime midnight( 0, 0, 0 );
        srand( midnight.secsTo(QTime::currentTime()) );
    }
    erase( targetRect() );
    target = QPoint( 200 + rand() % 190,
                     10  + rand() % 255 );
    repaint( targetRect() );
}
```

This private function creates a target center point at a new "random" position.

We use the rand() function to fetch random integers. The rand() function normally returns the same series of numbers each time you run a program. This would make the target appear at the same position every time. To avoid this, we must set a random seed the first time this function is called. The random seed must also be random in order to avoid equal random number series. The solution is to use the number of seconds that have passed since midnight as a pseudo-random value.

First we create a static bool local variable. A static variable like this one is guaranteed to keep its value between calls to the function.

The `if` test will only succeed the first time this function is called, because we set `first_time` to FALSE inside the `if` block.

Then we create the QTime object `midnight` which represents the time 00:00:00. Next, we fetch the number of seconds from midnight until now and use it as a random seed. See the documentation of QDate, QTime and QDateTime for more information.

Finally, we calculate the target's center point. We keep it withing the rectangle (x=200,y=35,width=190,height=255), (i.e. the possible x and y values are x = 200..390 and y = 35..290) in a coordinate system where we put y position 0 at the bottom edge of the widget and let y values increase upwards. x is as normal, with 0 at the left edge and with x values increasing to the right.

By experimentation, we have found this to always be in reach of the shot.

Note that rand() return a random integer >= 0.

```
void CannonField::timerEvent( QTimerEvent * )
{
    erase( shotRect() );
    timerCount++;
    QRect shotR = shotRect();
```

This part of the timer event has not changed from the previous chapter.

```
    if ( shotR.intersects( targetRect() ) ) {
        stopShooting();
        emit hit();
        return;
    }
```

This `if` statement checks if the shot rectangle intersects the target rectangle. If it does, the shot has hit the target (ouchh!). We stop shooting and emit the hit() signal to tell the outside world that a target was destroyed and return.

Note that we could have created a new target on the spot, but since the CannonField is a component, we leave such decisions to the user of the component.

```
    if ( shotR.x() > width() || shotR.y() > height() ) {
        stopShooting();
        emit missed();
        return;
    }
```

This `if` statement is the same as in the previous chapter, except that it now emits the missed() signal to tell the outside world about the failure.

```
    repaint( shotR, FALSE );
}
```

No changes here.

Modifications in CannonField::paintEvent():

```
    if ( updateR.intersects( targetRect() ) )
        paintTarget( &p );
```

These two lines have been added to the paintEvent() to paint the target if necessary.

```
void CannonField::paintTarget( QPainter *p )
{
    p->setBrush( red );
    p->setPen( black );
    p->drawRect( targetRect() );
}
```

This private function paints the target; a rectangle filled with red and with a black outline.

```
QRect CannonField::targetRect() const
{
    QRect r( 0, 0, 20, 10 );
    r.moveCenter( QPoint(target.x(),height() - 1 - target.y()) );
    return r;
}
```

This private function returns the enclosing rectangle of the target. Remember from newTarget() that the `target` point uses y coordinate 0 at the bottom of the widget. We calculate the point in widget coordinates before we call moveCenter().

The reason we have chosen this coordinate mapping is to fix the distance between the target and the bottom of the widget. Remember that the widget can be resized by the user or the program at any time.

main.cpp

There are no new members in the MyWidget class, but we have slightly changed the constructor to set the new LCDRange text labels.

```
    angle  = new LCDRange( "ANGLE", this, "angle" );
```

We set the angle text label to "ANGLE".

```
    force  = new LCDRange( "FORCE", this, "force" );
```

We set the force text label to "FORCE".

Note that we have not changed the position or the size of the LCDRanges. The QLCDNumber in LCDRange will become a bit smaller to make room for the label. MyWidget doesn’t care about that, because it uses LCDRange as a component and trust that it will show its value in the best way possible.

12.2. Behavior

The cannon can shoot at a target and a new target is automatically created when one has been hit.

12.3. Excercises

Make a cheat button that when pressed makes the CannonField display the shot trajectory for 5 seconds.

Make a moving target.

Make it possible to have several shots in the air at the same time. Hint: make a Shot object.

12.3.1. URL

Generated from http://www.troll.no/qt/t12.html.

13. Chapter 13: Game Over

In this example we start to approach a real playable game. We give MyWidget a new name; GameBoard and add some slots.

We put the definition in gamebrd.h and the implementation in gamebrd.cpp.

The CannonField now has a game over state.

- lcdrange.h contains the LCDRange class definition
- lcdrange.cpp contains the LCDRange implementation
- cannon.h contains the CannonField class definition
- cannon.cpp contains the CannonField implementation
- gamebrd.h contains the GameBoard class definition
- gamebrd.cpp contains the GameBoard implementation
- main.cpp contains MyWidget and main.
- Makefile contains some rules for generating the meta object information necessary for signal/slot creation.

13.1. Line by Line Walk-Through

cannon.h

The CannonField now has a game over state and a few new functions.

```
bool  gameOver()   const { return gameEnded; }
```

This function returns TRUE if the game is over, or FALSE if a game is going on.

```
void  setGameOver();
void  restartGame();
```

Here are two new slots; setGameOver() and restartGame().

```
        bool gameEnded;
```

This private variable contains the game state. TRUE means that the game is over, and FALSE means that a game is going on.

cannon.cpp

```
    #include <qfont.h>
```

We include the QFont class definition.

```
        gameEnded     = FALSE;
```

This line has been added to the constructor. Initially, the game is not over.

```
    void CannonField::setGameOver()
    {
        if ( gameEnded )
            return;
        if ( shooting )
            stopShooting();
        gameEnded = TRUE;
        repaint();
    }
```

This slot ends the game. It must be called from outside CannonField, because this widget does not know when to end the game. This is is an important design principle in component programming. We choose to make the component as flexible as possible to make it usable for different games with different rules.

If the game has already been ended, we return immediately. If a game is going on, we stop the shot, set the game over flag and repaint the entire widget.

```
    void CannonField::restartGame()
    {
        if ( shooting )
            stopShooting();
        gameEnded = FALSE;
        repaint();
    }
```

This slot starts a new game. If a shot is in the air, we stop shooting. We then reset the `gameEnded` variable and repaint the widget.

Modifications in CannonField::paintEvent():

```
        if ( gameEnded ) {
            p.setPen( black );
            p.setFont( QFont( "Courier", 48, QFont::Bold ) );
            p.drawText( rect(), AlignCenter, "Game Over" );
        } else {
```

The paint event has been enhanced to display the text "Game Over" if the game is over, i.e. `gameEnded` is TRUE. We don't bother to check the update rectangle here, because speed is not critical when the game is over.

To draw the text, we first set a black pen. The pen color is used when drawing text. Next, we choose a 48 point bold font from the Courier family. Finally, we draw the text centered in the widget's rectangle. Unfortunately, on some systems (especially some X servers) it can take a while to load such a large font. Since Qt caches fonts, you will only notice this the first time the font is used.

```
            if ( shooting &&  updateR.intersects( shotRect() ) )
                paintShot( &p );
            if ( updateR.intersects( targetRect() ) )
                paintTarget( &p );
        }
```

We only draw the shot or the target when the game is not over.

gamebrd.h

This file is new. It contains the definition of the GameBoard class, which was last seen as MyWidget.

```
class GameBoard : public QWidget
{
    Q_OBJECT
public:
    GameBoard( QWidget *parent=0, const char *name=0 );
protected:
    void  resizeEvent( QResizeEvent * );
protected slots:
    void  fire();
    void  hit();
    void  missed();
    void  newGame();
private:
    QPushButton *quit;
    QPushButton *shoot;
    QPushButton *restart;
    LCDRange    *angle;
    LCDRange    *force;
    QLCDNumber  *hits;
    QLCDNumber  *shotsLeft;
    CannonField *cannonField;
};
```

We have now added four slots. These are protected and are used internally. We have also added two QLCDNumbers: `hits` and `shotsLeft,` which display the game status. The `restart` button is also new.

gamebrd.cpp

This file is new. It contains the definition of the GameBoard class, which was last seen as MyWidget.

We have made some changes in the GameBoard constructor.

```
    connect( cannonField, SIGNAL(hit()),SLOT(hit()) );
    connect( cannonField, SIGNAL(missed()),SLOT(missed()) );
```

This time we want to do something when the shot has hit or missed the target. Thus we connect the hit() and missed() signals of the CannonField to two protected slots with the same names in this class.

```
    connect( shoot, SIGNAL(clicked()), SLOT(fire()) );
```

Previously, we connected the shoot button's clicked() signal directly to the CannonField's shoot() slot. This time we want to keep track of the number of shots fired, so we connect it to a protected slot in this class instead.

Notice how easy it is to change the behavior of a program when you are working with self-contained components.

```
    restart = new QPushButton( "New Game", this, "newgame" );
    restart->setFont( QFont( "Times", 18, QFont::Bold ) );
    connect( restart, SIGNAL(clicked()), SLOT(newGame()) );
```

We create, set up and connect the restart button like we have done with the other buttons. Clicking this button will activate the newGame() slot in this widget.

```
    hits              = new QLCDNumber( 2, this, "hits" );
    shotsLeft         = new QLCDNumber( 2, this, "shotsleft" );
```

```
        QLabel *hitsL      = new QLabel( "HITS", this, "hitsLabel" );
        QLabel *shotsLeftL = new QLabel( "SHOTS LEFT", this, "shotsleftLabel" );
```

We create four new widgets. Note that we don't bother to keep the pointers to the QLabel widgets in the GameBoard class since they live their own life undisturbed. As mentioned before, Qt will delete them when the GameBoard widget is destroyed.

```
        quit->setGeometry( 10, 10, 75, 30 );
        angle->setGeometry( 10, quit->y() + quit->height() + 10, 75, 130 );
        force->setGeometry( 10, angle->y() + angle->height() + 10, 75, 130 );
        cannonField->move( angle->x() + angle->width() + 10, angle->y() );
        shoot->setGeometry( 10, 315, 75, 30 );
        restart->setGeometry( 380, 10, 110, 30 );
        hits->setGeometry( 130, 10, 40, 30 );
        hitsL->setGeometry( hits->x() + hits->width() + 5, 10, 60, 30 );
        shotsLeft->setGeometry( 240, 10, 40, 30 );
        shotsLeftL->setGeometry( shotsLeft->x()+shotsLeft->width()+5, 10, 60, 30 );
```

Since we have a large number of widgets, we want to put the geometry setting in one place. This will make it more convenient for us to later change the geometry.

```
        newGame();
```

To start the show, we call the newGame() slot.

```
    void GameBoard::fire()
    {
        if ( cannonField->gameOver() || cannonField->isShooting() )
            return;
        shotsLeft->display( shotsLeft->intValue() - 1 );
        cannonField->shoot();
    }
```

This function fires a shot. If the game is over or there is a shot in the air, we return immediately. We decrement number of shots left and tell the cannon to shoot.

```
    void GameBoard::hit()
    {
        hits->display( hits->intValue() + 1 );
        if ( shotsLeft->intValue() == 0 )
            cannonField->setGameOver();
        else
            cannonField->newTarget();
    }
```

This slot is activated when a shot has hit the target. We increment the number of hits. If there are no shots left, the game is over. Otherwise, we make the CannonField generate a new target.

```
    void GameBoard::missed()
    {
        if ( shotsLeft->intValue() == 0 )
            cannonField->setGameOver();
    }
```

This slot is activated when a shot has missed the target. If there are no shots left, the game is over.

```
    void GameBoard::newGame()
    {
        shotsLeft->display( 15 );
        hits->display( 0 );
        cannonField->restartGame();
        cannonField->newTarget();
    }
```

This slot is activated when the user clicks the restart button. It is also called from the constructor. First, it sets the number of shots to 15. Note that this is the only place in the program that we set the number of shots. Change it to whatever you like to change the game rules. Next, we reset the number of hits, restart the game and generate a new target.

main.cpp

This file has just been on a diet. MyWidget is gone and the only thing left is the main() function.

13.2. Behavior

Hits and shots left are displayed and the program keeps track of them. The game can end and there's a button to start a new game.

13.3. Excercises

Add a random wind factor and show it to the user.

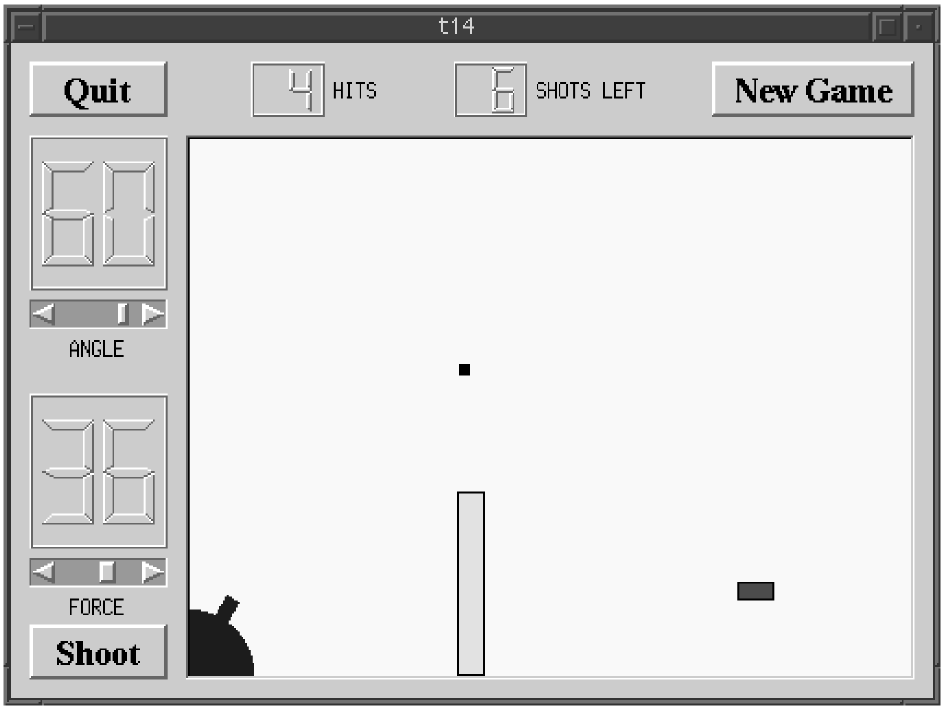
Make some splatter effects when the shot hits the target.

Implement multiple targets.

13.3.1. URL

Generated from http://www.troll.no/qt/t13.html.

14. Chapter 14: Facing the Wall



This is the final example, a complete game.

We add keyboard accelerators, introduce mouse events to CannonField. We put a frame around the CannonField and add a barrier (wall) to make the game more challenging.

- lcdrange.h contains the LCDRange class definition
- lcdrange.cpp contains the LCDRange implementation
- cannon.h contains the CannonField class definition
- cannon.cpp contains the CannonField implementation
- gamebrd.h contains the GameBoard class definition
- gamebrd.cpp contains the GameBoard implementation
- main.cpp contains MyWidget and main.
- Makefile contains some rules for generating the meta object information necessary for signal/slot creation.

14.1. Line by Line Walk-Through

cannon.h

The CannonField can now receive mouse events to make the user aim the barrel by clicking on it and dragging. CannonField also has a barrier wall.

```
protected:
    void  timerEvent( QTimerEvent * );
    void  paintEvent( QPaintEvent * );
    void  mousePressEvent( QMouseEvent * );
    void  mouseMoveEvent( QMouseEvent * );
    void  mouseReleaseEvent( QMouseEvent * );
```

In addition to the familiar event handlers, CannonField implements three mouse event handlers. The names say it all.

```
void  paintBarrier( QPainter * );
```

This private function paints the barrier wall.

```
QRect barrierRect() const;
```

This private function returns the enclosing rectangle of the barrier.

```
bool  barrelHit( const QPoint & ) const;
```

This private function checks if a point is inside the barrel of the cannon.

```
bool barrelPressed;
```

This private variable is TRUE if the user has pressed the mouse on the barrel and not released it.

cannon.cpp

```
barrelPressed = FALSE;
```

This line has been added to the constructor. Initially, the barrel is not pressed.

```
if ( (shotR.x() > width() || shotR.y() > height()) ||
     shotR.intersects(barrierRect()) ) {
    stopShooting();
    emit missed();
    return;
}
```

In the timerEvent() we now have to test if the shot has hit the barrier, in addition to checking if the shot has moved beyond the right or bottom edge of the widget.

```
if ( updateR.intersects( barrierRect() ) )
    paintBarrier( &p );
```

In the paint event, we have added painting of the barrier.

```
void CannonField::mousePressEvent( QMouseEvent *e )
{
    if ( e->button() != LeftButton )
        return;
    if ( barrelHit( e->pos() ) )
        barrelPressed = TRUE;
}
```

This is a Qt event handler. It is called when the user presses a mouse button when the mouse cursor is over the widget.

If the event was not generated by the left mouse button, we return immediately. Otherwise, we check if the position of the mouse cursor is within the cannon's barrel. If it is, we set `barrelPressed` to TRUE.

Notice that the pos() function returns a point in the widget's coordinate system.

```
void CannonField::mouseMoveEvent( QMouseEvent *e )
{
    if ( !barrelPressed )
        return;
    QPoint pnt = e->pos();
    if ( pnt.x() <= 0 )
        pnt.setX( 1 );
    if ( pnt.y() >= height() )
        pnt.setY( height() - 1 );
    double rad = atan( ((double) rect().bottom() - pnt.y()) /  pnt.x() );
    setAngle( qRound ( rad*180/3.14159265 ) );
}
```

This is another Qt event handler. It is called when the user already has pressed the mouse button inside this widget and then moves/drags the mouse. You can make Qt send mouse move events even when no buttons are pressed, see QWidget::setMouseTracking().

This handler repositions the cannon's barrel according to the position of the mouse cursor.

First, if the barrel is not pressed, we return. Next, we fetch the mouse cursor's position. If the mouse cursor is to the left or below the widget, we adjust the point to be inside the widget.

Then we calculate the angle between the bottom edge of the widget and the imaginary line between the bottom left corner of the widget and the cursor position. Finally, we set the cannon's angle to the new value converted to degrees.

Remember that setAngle() redraws the cannon.

```
void CannonField::mouseReleaseEvent( QMouseEvent *e )
{
    if ( e->button() == LeftButton )
        barrelPressed = FALSE;
}
```

This Qt event handler is called whenever the user releases a mouse button after it has been pressed inside this widget.

If the left button is released, we can be sure that the barrel is no longer pressed.

```
void CannonField::paintBarrier( QPainter *p )
{
    p->setBrush( yellow );
    p->setPen( black );
    p->drawRect( barrierRect() );
}
```

This private function paints the barrier as a rectangle filled with yellow and with a black outline.

```
QRect CannonField::barrierRect() const
{
    return QRect( 145, height() - 100, 15, 100 );
}
```

This private function returns the rectangle of the barrier. We fix the bottom edge of the barrier to the bottom edge of the widget.

```
bool CannonField::barrelHit( const QPoint &p ) const
{
    QWMatrix mtx;
    mtx.translate( 0, height() - 1 );
    mtx.rotate( -ang );
    mtx = mtx.invert();
    return barrel_rect.contains( mtx.map(p) );
}
```

This function returns TRUE if the point is in the barrel, otherwise FALSE.

Here we use the class QWMatrix. It is defined in the header file qwmatrix.h, which is included by qpainter.h.

QWMatrix defines a coordinate system mapping. It can perform the same transformations as the QPainter.

Here we perform the same transformation steps as we do when drawing the barrel in the paintCannon() function. First we translate the coordinate system, then we rotate it.

Now we need to check if the point `p` (in widget coordinates) lies inside the barrel. To do this, we invert the transformation matrix. The inverted matrix performs the inverse transformation that we used when drawing the barrel. We map the point `p` using the inverted matrix and return TRUE if it is inside the original barrel rectangle.

gamebrd.h

The only change is that a frame has been added.

```
class QFrame;
```

We name declare QFrame...

```
    QFrame      *frame;
```

and add a QFrame to this widget.

gamebrd.cpp

```
#include <qaccel.h>
```

We include the class definition of QAccel.

```
    frame = new QFrame( this, "cannonFrame" );
    frame->setFrameStyle( QFrame::WinPanel | QFrame::Sunken );
```

We create and set up a frame. We want this frame around the CannonField widget. Since widgets cannot be transparent, we have to put the frame behind the CannonField. We create the frame before we create the CannonField. Qt guarantees that the last created widget always will be on top of the others.

The frame style is set to a sunken WinPanel. See the QFrame documentation for more information.

```
    QAccel *accel = new QAccel( this );
    accel->connectItem( accel->insertItem( Key_Space), this, SLOT(fire()) );
    accel->connectItem( accel->insertItem( Key_Q), qApp, SLOT(quit()) );
```

Here we create and set up an accelerator. An accelerator is an object that intercepts keyboard events to an application and calls slots if certain keys are pressed. This mechanism is also called shortcut keys. Note that an accelerator can be a child of a widget and will be destroyed when the parent is destroyed. QAccel is *not* a widget and has no visible effect on its parent.

We define two shortcut keys. We want the slot fire() to be called when the user presses space, and we want the application to quit when key "Q" is pressed. The `Key_Space` and `Key_Q` constants are defined in the header file qkeycode.h (included by qaccel.h). Also note that `Key_Q` signifies the Q key on the keyboard. The slot will be called when typing both "q" and "Q".

```
    frame->move( angle->x() + angle->width() + 10, angle->y() );
    cannonField->move( frame->x() + 2, frame->y() + 2 );
```

We set the initial position of the frame to the same position we previously used for the CannonField. Next, we set the position of the CannonField relative to the frame.

```
void GameBoard::resizeEvent( QResizeEvent * )
{
    frame->resize( width()  - frame->x() - 10,
                   height() - frame->y() - 10 );
    cannonField->resize( frame->width() - 4, frame->height() - 4 );
}
```

We must now incorporate the frame widget in our resize handler. First we resize the frame exactly as we did with the CannonField in the previous chapter. Then we simply resize the CannonField relative to the frame.

14.2. Behavior

The cannon now shoots when you press space. You can also position the cannon's angle using the mouse. The barrier makes it a little more challenging to play the game. We also have a nice looking frame around the CannonField.

14.3. Excercises

Write a space invaders game and give it to the KDE project.

14.3.1. URL

Generated from http://www.troll.no/qt/t14.html.

15. QAccel Class Reference

`#include <qaccel.h>`

Inherits QObject.

15.1. Detailed Description

The QAccel class handles keyboard accelerator keys.

A QAccel contains a list of accelerator items. Each accelerator item consists of an identifier and a keyboard code combined with modifiers (`SHIFT, CTRL, ALT` or `ASCII_ACCEL).`

For example, `CTRL + Key_P` could be a shortcut for printing a document. The key codes are listed in qkeycode.h.

When pressed, an accelerator key sends out the signal activated() with a number that identifies this particular accelerator item. Accelerator items can also be individually connected, so that two different keys will activate two different slots (see connectItem()).

A QAccel object handles key events to the top level window containing *parent,* and hence to any child widgets of that window. Note that the accelerator will be deleted only when the *parent* is deleted, and will consume relevant key events until then.

Example:

```
QAccel *a = new QAccel( myWindow );         // create accels for myWindow
a->connectItem( a->insertItem(Key_P+CTRL), // adds Ctrl+P accelerator
                myWindow,                   // connected to myWindow's
                SLOT(printDoc()) );         // printDoc() slot
```

See also: QKeyEvent, QWidget::keyPressEvent(), QMenuData::setAccel(), QButton::setAccel() and GUI Design Handbook: Keyboard Shortcuts,

Examples:

application/application.cpp

15.2. Member Function Documentation

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

Creates a QAccel object with a parent widget and a name.

QAccel::~QAccel ()

Destroys the accelerator object.

void QAccel::activated (int id) `[signal]`

This signal is emitted when an accelerator key is pressed. *id* is a number that identifies this particular accelerator item.

void QAccel::clear ()

Removes all accelerator items.

bool QAccel::connectItem (int id, const QObject * receiver, const char * member)

Connects an accelerator item to a slot/signal in another object.

Arguments:

- *id* is the accelerator item id.
- *receiver* is the object to receive a signal.
- *member* is a slot or signal function in the receiver.

```
a->connectItem( 201, mainView, SLOT(quit()) );
```

See also: disconnectItem().

Examples:
application/application.cpp

uint QAccel::count () const

Returns the number of accelerator items.

bool QAccel::disconnectItem (int id, const QObject * receiver, const char * member)

Disconnects an accelerator item from a function in another object.

See also: connectItem().

bool QAccel::eventFilter (QObject *, QEvent * e) `[virtual protected]`

Processes accelerator events intended for the top level widget.

Reimplemented from QObject.

int QAccel::findKey (int key) const

Returns the identifier of the accelerator item with the key code *key,* or -1 if the item cannot be found.

int QAccel::insertItem (int key, int id=-1)

Inserts an accelerator item and returns the item's identifier.

Arguments:

- *key* is a key code plus a combination of SHIFT, CTRL and ALT.
- *id* is the accelerator item id. If *id* is negative, then the item will be assigned a unique identifier.

```
QAccel *a = new QAccel( myWindow );         // create accels for myWindow
a->insertItem( Key_P + CTRL, 200 );         // Ctrl+P to print document
a->insertItem( Key_X + ALT , 201 );         // Alt+X  to quit
a->insertItem( ASCII_ACCEL + 'q', 202 );    // ASCII 'q' to quit
a->insertItem( Key_D );                     // gets id 3
a->insertItem( Key_P + CTRL + SHIFT );      // gets id 4
```

Examples:
application/application.cpp

bool QAccel::isEnabled () const

Returns TRUE if the accelerator is enabled, or FALSE if it is disabled.

See also: setEnabled() and isItemEnabled().

bool QAccel::isItemEnabled (int id) const

Returns TRUE if the accelerator item with the identifier *id* is enabled. Returns FALSE if the item is disabled or cannot be found.

See also: setItemEnabled() and isEnabled().

int QAccel::key (int id)

Returns the key code of the accelerator item with the identifier *id,* or zero if the id cannot be found.

void QAccel::removeItem (int id)

Removes the accelerator item with the identifier *id.*

void QAccel::repairEventFilter ()

Make sure that the accelerator is watching the correct event filter. Used by QWidget::recreate().

void QAccel::setEnabled (bool enable)

Enables the accelerator if *enable* is TRUE, or disables it if *enable* is FALSE.

Individual keys can also be enabled or disabled.

See also: isEnabled() and setItemEnabled().

void QAccel::setItemEnabled (int id, bool enable)

Enables or disables an accelerator item.

Arguments:

- *id* is the item identifier.
- *enable* specifies whether the item should be enabled or disabled.

See also: isItemEnabled() and isEnabled().

15.2.1. URL

Generated from http://www.troll.no/qt/qaccel.html.

16. QApplication Class Reference

`#include <qapplication.h>`

Inherits QObject.

Inherited by QXtApplication.

16.1. Detailed Description

The QApplication class manages the application event queue.

The QApplication class is central to Qt. It receives events from the underlying window system and sends them to the destination widgets. An application object must be created before any widgets can be created!

Only one single QApplication object should be created. In fact Qt complains if you create more than one, and this is normally done in the main() function. Once a QApplication object has been created, `qApp` (defined as `extern QApplication *qApp`) refers to this object.

Example (a complete Qt application):

```
#include <qapplication.h>                            // defines QApplication
#include <qpushbutton.h>                     // defines QPushButton
int main( int argc, char **argv )
{
    QApplication app( argc, argv );          // create application object
    QPushButton  hello( "Hello, world!" );   // create a push button
    app.setMainWidget( &hello );             // define as main widget
    connect( &hello, SIGNAL(clicked()),      // clicking the button
             &app, SLOT(quit()) );           //   quits the application
    hello.show();                            // show button
    return app.exec();                       // run main event loop
}
```

Important

Notice that the QApplication object must be created before any window-system functionality of Qt is used, this includes widgets, colors, fonts etc.

Note also that for X11, setMainWidget() may change the main widget according to the *-geometry* option. To preserve this functionality, you must set your defaults before setMainWidget() and any overrides after.

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

Examples:

forever/forever.cpp
desktop/desktop.cpp
connect/connect.cpp
drawdemo/drawdemo.cpp
movies/main.cpp
picture/picture.cpp
xform/xform.cpp
menu/menu.cpp
progress/progress.cpp
cursor/cursor.cpp
layout/layout.cpp
qmag/qmag.cpp
widgets/widgets.cpp

16.2. Member Function Documentation

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

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

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

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

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

Qt debugging options:

- `-nograb,` tells Qt to never grab the mouse or the keyboard.
- `-sync` (only under X11), switches to synchronous mode for debugging.

See Debugging Techniques for a more detailed explanation.

The X11 version of Qt supports a few more command line options:

- `-display` *display,* sets the X display (default is $DISPLAY).
- `-geometry` *geometry,* sets the client geometry of the main widget.
- `-fn` or `-font` *font,* defines the application font.
- `-bg` or `-background` *color,* sets the default background color and an application palette (light and dark shades are calculated).
- `-fg` or `-foreground` *color,* sets the default foreground color.
- `-name` *name,* sets the application name.
- `-title` *title,* sets the application title (caption).
- `-style=` *style,* sets the application GUI style. Possible values are `motif` and `windows`
- `-visual TrueColor,` forces the application to use a TrueColor visual on an 8-bit display.
- `-ncols` *count,* limits the number of colors allocated in the color cube on a 8-bit display, if the application is using the `QApplication::ManyColor` color specification. If *count* is 216 then a 6x6x6 color cube is used (ie. 6 levels of red, 6 of green, and 6 of blue); for 108, a 3x3x3, and for other values, a cube approximately proportional to a 2x3x1 cube is used.
- `-cmap,` causes the application to install a private color map on an 8-bit display.

See also: argc() and argv().

QApplication::~QApplication () `[virtual]`

Deletes all remaining widgets and cleans up any window system resources that were allocated by this application. Sets the global variable `qApp` to null.

QWidget * QApplication::activeModalWidget () `[static]`

Returns the active modal widget.

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

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

See also: currentPopupWidget() and topLevelWidgets().

QWidget * QApplication::activePopupWidget () [static]

Returns the active popup widget.

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

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

See also: currentModalWidget() and topLevelWidgets().

QWidgetList * QApplication::allWidgets () [static]

Returns a list of all the widgets in the application.

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

The list is empty if there are no widgets.

Note that some of the widgets may be hidden.

Example:

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

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

Warning: Delete the list away as soon you have finished using it. You can get in serious trouble if you for instance try to access a widget that has been deleted.

See also: topLevelWidgets(), QWidget::isVisible() and QList::isEmpty(),.

int QApplication::argc () const

Returns the number of command line arguments.

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

See also: argv() and QApplication::QApplication().

char ** QApplication::argv () const

Returns the command line argument vector.

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

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

Example:

```
// showargs.cpp - displays program arguments in a list box
#include <qapplication.h>
#include <qlistbox.h>
int main( int argc, char **argv )
{
    QApplication a( argc, argv );
    QListBox b;
    a.setMainWidget( &b );
    for ( int i=0; i<a.argc(); i++ )         // a.argc() == argc
        b.insertItem( a.argv()[i] );         // a.argv()[i] == argv[i]
    b.show();
    return a.exec();
}
```

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

See also: argc() and QApplication::QApplication().

void QApplication::beep () `[static]`

Sounds the bell, using the default volume and sound.

QClipboard * QApplication::clipboard () `[static]`

Returns a pointer to the application global clipboard.

bool QApplication::closingDown () `[static]`

Returns TRUE if the application objects are being destroyed.

See also: startingUp().

int QApplication::colorSpec () `[static]`

Returns the color specification.

See also: QApplication::setColorSpec().

Examples:
showimg/showimg.cpp

QWidget * QApplication::desktop () `[static]`

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

The desktop widget is useful for obtaining the size of the screen. It can also be used to draw on the desktop.

```
QWidget *d = QApplication::desktop();
int w=d->width();                    // returns screen width
int h=d->height();                   // returns screen height
d->setBackgroundColor( red );        // makes desktop red
```

Examples:
qmag/qmag.cpp

int QApplication::doubleClickInterval () `[static]`

Returns the maximum duration for a double click.

See also: setDoubleClickInterval().

int QApplication::enter_loop ()

This function enters the main event loop (recursively). Do not call it unless you are an expert.

See also: exit_loop().

int QApplication::exec ()

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

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

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

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

See also: quit(), exit(), processEvents(), setMainWidget() and QTimer.

Examples:

forever/forever.cpp
connect/connect.cpp
mainlyQt/editor.cpp
drawdemo/drawdemo.cpp
movies/main.cpp
picture/picture.cpp
xform/xform.cpp
menu/menu.cpp
progress/progress.cpp
cursor/cursor.cpp
layout/layout.cpp
qmag/qmag.cpp
widgets/widgets.cpp

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

Tells the application to exit with a return code.

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

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

Note that unlike the C library exit function, this function *does* returns to the caller - it is event processing that stops.

See also: quit() and exec().

Examples:
picture/picture.cpp

void QApplication::exit_loop ()

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

See also: enter_loop().

void QApplication::flushX () `[static]`

Flushes the X event queue in the X11 implementation. Does nothing on other platforms.

See also: syncX().

QWidget * QApplication::focusWidget () const

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

See also: QWidget::setFocus() and QWidget::hasFocus().

QFont * QApplication::font () `[static]`

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

See also: setFont(), fontMetrics() and QWidget::font().

QFontMetrics QApplication::fontMetrics () `[static]`

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

See also: font(), setFont(), QWidget::fontMetrics() and QPainter::fontMetrics().

bool QApplication::hasGlobalMouseTracking () `[static]`

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

See also: setGlobalMouseTracking().

void QApplication::lastWindowClosed () `[signal]`

This signal is emitted when the user has closed a top level widget and there are no more visible top level widgets left.

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

See also: mainWidget(), topLevelWidgets() and QWidget::isTopLevel().

QWidget * QApplication::mainWidget () const

Returns the main application widget, or 0 if there is not a defined main widget.

See also: setMainWidget().

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

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

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

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

See also: QObject::event() and installEventFilter().

QCursor * QApplication::overrideCursor () `[static]`

Returns the active application override cursor.

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

See also: setOverrideCursor() and restoreOverrideCursor().

QPalette * QApplication::palette () `[static]`

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

See also: setPalette() and QWidget::palette().

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

Stores the event in a queue and returns immediatly.

The event must be allocated on the heap, as it is deleted when the event has been posted.

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

See also: sendEvent().

void QApplication::processEvents ()

Processes pending events, for 3 seconds or until there are no more events to process, then return.

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

See also: processEvents(), exec() and QTimer.

void QApplication::processEvents (int maxtime)

Processes pending events, for *maxtime* milliseconds or until there are no more events to process, then return.

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

See also: processOneEvent(), exec() and QTimer.

void QApplication::processOneEvent ()

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

This function is useful for adapting Qt to situations where the event processing must be grafted into existing program loops. Beware that using this function in new applications may be an indication of design problems.

See also: processEvents(), exec() and QTimer.

void QApplication::quit () `[slot]`

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

This function is a slot, i.e. you may connect any signal to activate quit().

Example:

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

See also: exit().

void QApplication::restoreOverrideCursor () `[static]`

Restores the effect of setOverrideCursor().

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

Application cursors are stored on an internal stack. setOverrideCursor() pushes the cursor onto the stack, and restoreOverrideCursor() pops the active cursor off the stack. Every setOverrideCursor() must have an corresponding restoreOverrideCursor(), otherwise the stack will get out of sync. overrideCursor() returns 0 if the cursor stack is empty.

See also: setOverrideCursor() and overrideCursor().

Examples:

showimg/showimg.cpp

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

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

See also: postEvent() and notify().

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

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

Some event compression may occur. Note that events from the window system are *not* dispatched by this function.

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

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

The color specification controls how your application allocates colors. You must set the color specification before you create the QApplication object.

The choices are:

- `QApplication::NormalColor.` This is the default color allocation strategy. Use this choice if your application uses buttons, menus, texts and pixmaps with few colors. With this choice, the application allocates system global colors. This work fine for most applications under X11, but Windows dithers to the 20 standard colors unless the display has true color support (more than 256 colors).
- `QApplication::CustomColor.` Use this choice if your application needs a small number of custom colors. This choice only makes a difference on Windows - the application gets more colors when it is active, but the background windows look less good. Under X11 this is the same as `NormalColor.` Under Windows, Qt creates a Windows palette if the display supports 256 colors.
- `QApplication::ManyColor.` Use this choice if your application is very color hungry (e.g. it wants thousands of colors). Under Windows, this is equal to `CustomColor.` Under X11 the effect is:
- For 256-color displays which have at best a 256 color true color visual, the default visual is used, and a colors are allocated from a color cube. The color cube is the 6x6x6 (216 color) "Web palette", but the number of colors can be changed by the *-ncols* option. The user can force the application to use the true color visual by the -visual option.
- For 256-color displays which have a true color visual with more than 256 colors, use that visual. Silicon Graphics X servers have this feature. They provide an 8 bit visual as default but can deliver true color when asked.

Example:

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

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

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

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

See also: colorSpec(), QColor::numBitPlanes() and QColor::enterAllocContext().

Examples:
widgets/widgets.cpp

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

Sets the time limit that distinguishes a double click from two consecutive mouse clicks to *ms* milliseconds. This value is ignored under Windows (the control panel value is used.)

The default value is 400 milliseconds.

See also: doubleClickInterval().

void QApplication::setFont (const QFont & font, bool updateAllWidgets=FALSE) `[static]`

Changes the default application font to *font*.

The default font depends on the X server in use.

If *updateAllWidgets* is TRUE, then the font of all existing widgets is set to *font*.

Widgets created after this call get *font* as their font.

See also: font(), fontMetrics() and QWidget::setFont().

Examples:
desktop/desktop.cpp
movies/main.cpp

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

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

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

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

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

See also: hasGlobalMouseTracking() and QWidget::hasMouseTracking().

void QApplication::setMainWidget (QWidget * mainWidget)

Sets the main widget of the application.

The special thing about the main widget is that destroying the main widget (i.e. the program calls QWidget::close() or the user double-clicks the window close box) will leave the main event loop and exit the application.

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

See also: mainWidget(), exec() and quit().

Examples:
forever/forever.cpp
connect/connect.cpp
mainlyQt/editor.cpp
drawdemo/drawdemo.cpp

picture/picture.cpp
xform/xform.cpp
menu/menu.cpp
progress/progress.cpp
cursor/cursor.cpp
layout/layout.cpp
qmag/qmag.cpp
widgets/widgets.cpp

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

Sets the application override cursor to *cursor*.

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

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

Application cursors are stored on an internal stack. setOverrideCursor() pushes the cursor onto the stack, and restoreOverrideCursor() pops the active cursor off the stack. Every setOverrideCursor() must have an corresponding restoreOverrideCursor(), otherwise the stack will get out of sync. overrideCursor() returns 0 if the cursor stack is empty.

If *replace* is TRUE, the new cursor will replace the last override cursor.

Example:

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

See also: overrideCursor(), restoreOverrideCursor() and QWidget::setCursor().

Examples:
showimg/showimg.cpp

void QApplication::setPalette (const QPalette & palette, bool updateAllWidgets=FALSE) `[static]`

Changes the default application palette to *palette.*

If *updateAllWidgets* is TRUE, then the palette of all existing widgets is set to *palette.*

Widgets created after this call get *palette* as their palette.

See also: QWidget::setPalette() and palette().

Examples:
xform/xform.cpp

void QApplication::setStyle (GUIStyle style) `[static]`

Sets the application GUI style to *style.*

The style parameter can be `WindowsStyle` or `MotifStyle.`

See also: style() and QWidget::setStyle().

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

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

The default color is `darkBlue.`

See also: winStyleHighlightColor().

Examples:
widgets/widgets.cpp

bool QApplication::startingUp () `[static]`

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

See also: closingDown().

GUIStyle QApplication::style () `[static]`

Returns the GUI style of the application.

See also: setStyle().

void QApplication::syncX () `[static]`

Synchronizes with the X server in the X11 implementation. Does nothing on other platforms.

See also: flushX().

QWidgetList * QApplication::topLevelWidgets () `[static]`

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

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

The list is empty if there are no top level widgets.

Note that some of the top level widgets may be hidden.

Example:

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

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

Warning: Delete the list away as soon you have finished using it. You can get in serious trouble if you for instance try to access a widget that has been deleted.

See also: allWidgets(), QWidget::isTopLevel(), QWidget::isVisible() and QList::isEmpty().

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

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

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

See also: QCursor::pos(), QWidget::grabMouse() and QWidget::grabKeyboard().

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

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

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

Returns the color used to mark selections in windows style.

See also: setWinStyleHighlightColor().

16.3. Related Functions

void qAddPostRoutine (CleanUpFunction p)

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

CleanUpFunctions is defined as `typedef void (*CleanUpFunction)();` , i.e. a pointer to a function that takes no arguments and returns nothing.

Example of use:

```
static int *global_ptr = 0;
void cleanup_ptr()
{
    delete [] global_ptr;
}
void init_ptr()
{
    global_ptr = new int[100];              // allocate data
    qAddPostRoutine( cleanup_ptr );         // delete later
}
```

Q_EXPORT const char * qVersion ()

Returns the Qt version number for the library, typically "1.30" or "1.31".

Q_EXPORT bool qSysInfo (int * wordSize, bool * bigEndian)

Obtains information about the system.

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

This function calls fatal() with a message if the computer is truely weird (i.e. different endianness for 16 bit and 32 bit integers).

Q_EXPORT void debug (const char * msg, ...)

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

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

Example:

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

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

Warning: The internal buffer is limited to 512 bytes (including the 0-terminator.

See also: warning(), fatal(), qInstallMsgHandler() and Debugging

Q_EXPORT void warning (const char * msg, ...)

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

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

Example:

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

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

Warning: The internal buffer is limited to 512 bytes (including the 0-terminator.

See also: debug(), fatal(), qInstallMsgHandler() and Debugging

Examples:
progress/progress.cpp

Q_EXPORT void fatal (const char * msg, ...)

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

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

Example:

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

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

Warning: The internal buffer is limited to 512 bytes (including the 0-terminator.

See also: debug(), warning(), qInstallMsgHandler() and Debugging

void ASSERT (bool test)

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

This is really a macro defined in qglobal.h.

ASSERT is useful for testing required conditions in your program.

Example:

```
//
// File: div.cpp
//
#include <qglobal.h>
int divide( int a, int b )
{
    ASSERT( b != 0 );                        // this is line 9
    return a/b;
}
```

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

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

See also: warning() and Debugging

void CHECK_PTR (void * p)

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

This is really a macro defined in qglobal.h.

Warning: CHECK_PTR only works for the development release of the Qt library. In the release library, CHECK_PTR will be substituted with nothing.

Example:

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

See also: fatal() and Debugging

Q_EXPORT msg_handler qInstallMsgHandler (msg_handler h)

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

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

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

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

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

Example:

```
#include <qapplication.h>
#include <stdio.h>
#include <stdlib.h>
void myMessageOutput( QtMsgType type, const char *msg )
{
    switch ( type ) {
        case QtDebugMsg:
            fprintf( stderr, "Debug: %s\n", msg );
```

```
                break;
            case QtWarningMsg:
                fprintf( stderr, "Warning: %s\n", msg );
                break;
            case QtFatalMsg:
                fprintf( stderr, "Fatal: %s\n", msg );
                abort();                        // dump core on purpose
        }
    }
    int main( int argc, char **argv )
    {
        qInstallMsgHandler( myMessageOutput );
        QApplication a( argc, argv );
        ...
        return a.exec();
    }
```

See also: debug(), warning(), fatal() and Debugging

16.3.1. URL

Generated from http://www.troll.no/qt/qapplication.html.

17. QArray Class Reference

`#include <qarray.h>`

Inherits QGArray.

Inherited by QByteArray and QPointArray.

17.1. Detailed Description

The QArray class is a template class that provides arrays of simple types.

QArray is implemented as both a template and a macro class. Define a template instance QArray<X> to create an array that contains X items.

The QVector collection class is also a kind of array. Like all collection classes, it has pointers to the contained items.

QArray stores the array elements directly in the array. It can only deal with simple types, i.e. C++ types, structs and classes that have no constructors, destructors or virtual functions. QArray uses bitwise operations to copy and compare array elements.

QArray uses explicit sharing with a reference count. If more than one array share common data, and one array is modified, all arrays will be modified.

The benefit of sharing is that a program does not need to duplicate data when it is not required, which results in less memory usage and less copying of data.

Example:

```
#include <qarray.h>
#include <stdio.h>
QArray<int> fib( int num )                  // returns fibonacci array
{
    ASSERT( num > 2 );
    QArray<int> f( num );                   // array of ints
    f[0] = f[1] = 1;                        // initialize first two numbers
    for ( int i=2; i<num; i++ )
        f[i] = f[i-1] + f[i-2];
    return f;
}
void main()
{
    QArray<int> a = fib( 6 );               // get 6 first fibonaccis
    int i;
    for ( i=0; i<a.size(); i++ )            // print them
        prinf( "%d: %d\n", i, a[i] );
    printf( "1 is found %d time(s)\n", a.contains(1) );
    printf( "5 is found at index %d\n", a.find(5) );
}
```

Program output:

```
0: 1
1: 1
2: 2
3: 3
4: 5
5: 8
1 is found 2 times
5 is found at index 4
```

An array can also be instantiated through a macro expansion, but this is necessary only for compilers that do not support templates. We recommend using templates if your compiler supports it. Macros are much harder to debug.

Macro expansion:

```
  ...
Q_DECLARE(QArrayM,X);                       // declares the array
void main()
{
    QArrayM(X) a;                   // array of X elements
  ...
```

If you want to make your code work for compilers that do not have template support, but use templates if they are available, you can use `typedef`:

```
  ...
#if defined(USE_TEMPLATECLASS)      // defined in qglobal.h
typedef QArrayT<X>         XArray;
#else
typedef Q_DECLARE(QArrayM,X) XArray;
#endif
void main()
{
    XArray a;                       // array of X elements
  ...
```

QArrayT refers to the QArray template and QArrayM refers to the QArray macro. QArray defaults to QArrayT if templates are supported, QArrayM otherwise.

See also: Shared Classes

17.2. Member Function Documentation

QArray::QArray ()

Constructs a null array.

See also: isNull().

QArray::QArray (const QArray<type> & a)

Constructs a shallow copy of *a*.

See also: assign().

QArray::QArray (int size)

Constructs an array with room for *size* elements. Makes a null array if *size* == 0.

Note that the elements are not initialized.

See also: resize() and isNull().

QArray::QArray (int, int) `[protected]`

Constructs an array *without allocating* array space. The arguments should be (0, 0). Use at own risk.

QArray::~QArray ()

Dereferences the array data and deletes it if this was the last reference.

QArray::operator const type * () const

Cast operator. Returns a pointer to the array.

See also: data().

QArray<type> & QArray::assign (const QArray<type> & a)

Shallow copy. Dereferences the current array and references the data contained in *a* instead. Returns a reference to this array.

See also: operator=().

QArray<type> & QArray::assign (const type * data, uint size)

Shallow copy. Dereferences the current array and references the array data *data,* which contains *size* elements. Returns a reference to this array.

Do not delete *data* later, QArray takes care of that.

type & QArray::at (uint index) const

Returns a reference to the element at position *index* in the array.

This can be used to both read and set an element.

See also: operator[]().

int QArray::contains (const type & v) const

Returns the number of times *v* occurs in the array.

See also: find().

QArray<type> QArray::copy () const

Returns a deep copy of this array.

See also: detach() and duplicate().

type * QArray::data () const

Returns a pointer to the actual array data.

The array is a null array if data() == 0 (null pointer).

See also: isNull().

void QArray::detach () `[virtual]`

Detaches this array from shared array data, i.e. makes a private, deep copy of the data.

Copying will only be performed if the reference count is greater than one.

See also: copy().

Reimplemented from QGArray.

QArray<type> & QArray::duplicate (const QArray<type> & a)

Deep copy. Dereferences the current array and obtains a copy of the data contained in *a* instead. Returns a reference to this array.

See also: copy().

QArray<type> & QArray::duplicate (const type * data, uint size)

Deep copy. Dereferences the current array and obtains a copy of the array data *data* instead. Returns a reference to this array.

See also: copy().

bool QArray::fill (const type & v, int size=-1=-1)

Fills the array with the value *v.* If *size* is specified as different from -1, then the array will be resized before filled.

Returns TRUE if successful, or FALSE if the memory cannot be allocated (only when *size* != -1).

See also: resize().

int QArray::find (const type & v, uint index=0) const

Finds the first occurrence of *v,* starting at position *index.*

Returns the position of *v,* or -1 if *v* could not be found.

See also: contains().

bool QArray::isEmpty () const

Returns TRUE if the array is empty, i.e. size() == 0, otherwise FALSE.

isEmpty() is equivalent with isNull() for QArray. Note that this is not the case for QString::isEmpty().

bool QArray::isNull () const

Returns TRUE if the array is null, otherwise FALSE.

A null array has size() == 0 and data() == 0.

uint QArray::nrefs () const

Returns the reference count for the shared array data. This reference count is always greater than zero.

bool QArray::operator!= (const QArray<type> & a) const

Returns TRUE if this array is different from *a,* otherwise FALSE.

The two arrays are bitwise compared.

See also: operator==().

QArray<type> & QArray::operator= (const QArray<type> & a)

Assigns a shallow copy of *a* to this array and returns a reference to this array.

Equivalent to assign(a).

bool QArray::operator== (const QArray<type> & a) const

Returns TRUE if this array is equal to *a,* otherwise FALSE.

The two arrays are bitwise compared.

See also: operator!=().

type & QArray::operator[] (int index) const

Returns a reference to the element at position *index* in the array.

This can be used to both read and set an element. Equivalent to at().

See also: at().

void QArray::resetRawData (const type * data, uint size)

Resets raw data that was set using setRawData().

The arguments must be the data and length that were passed to setRawData(). This is for consistency checking.

See also: setRawData().

bool QArray::resize (uint size)

Resizes (expands or shrinks) the array to *size* elements. The array becomes a null array if *size* == 0.

Returns TRUE if successful, or FALSE if the memory cannot be allocated.

New elements will not be initialized.

See also: size().

QArray<type> & QArray::setRawData (const type * data, uint size)

Sets raw data and returns a reference to the array.

Dereferences the current array and sets the new array data to *data* and the new array size to *size.* Do not attempt to resize or re-assign the array data when raw data has been set. Call resetRawData(d,len) to reset the array.

Setting raw data is useful because it sets QArray data without allocating memory or copying data.

Example I (intended use):

```
static char bindata[] = { 231, 1, 44, ... };
QByteArray  a;
a.setRawData( bindata, sizeof(bindata) );   // a points to bindata
QDataStream s( a, IO_ReadOnly );            // open on a's data
s >> <something>;                           // read raw bindata
a.resetRawData( bindata, sizeof(bindata) ); // finished
```

Example II (you don't want to do this):

```
static char bindata[] = { 231, 1, 44, ... };
QByteArray  a, b;
a.setRawData( bindata, sizeof(bindata) );   // a points to bindata
a.resize( 8 );                              // will crash
b = a;                                      // will crash
a[2] = 123;                                 // might crash
  // forget to resetRawData - will crash
```

Warning: If you do not call resetRawData(), QArray will attempt to deallocate or reallocate the raw data, which might not be too good. Be careful.

See also: resetRawData().

uint QArray::size () const

Returns the size of the array (max number of elements).

The array is a null array if size() == 0.

See also: isNull() and resize().

Examples:
drawdemo/drawdemo.cpp

bool QArray::truncate (uint pos)

Truncates the array at position *pos.*

Returns TRUE if successful, or FALSE if the memory cannot be allocated.

Equivalent to resize(*pos).*

See also: resize().

17.2.1. URL

Generated from http://www.troll.no/qt/qarray.html.

18. QAsyncIO Class Reference

`#include <qasyncio.h>`

Inherited by QDataSink and QDataSource.

18.1. Detailed Description

Encapsulates I/O asynchronicity.

The Qt classes for asynchronous input/output provide a simple mechanism to allow large files or slow datasources to be processed without using large amounts of memory, or blocking the user interface.

This facility is used in Qt to drive animated images. See QImageConsumer and QImageSource.

18.2. Member Function Documentation

QAsyncIO::˜QAsyncIO () `[virtual]`

Destroys the async IO object.

void QAsyncIO::connect (QObject * obj, const char * member)

Ensures only one object can respond to changes in readiness.

void QAsyncIO::ready () `[protected]`

Derived classes should call this when they change from being unready to ready.

18.2.1. URL

Generated from http://www.troll.no/qt/qasyncio.html.

19. QBitArray Class Reference

`#include <qbitarray.h>`

Inherits QByteArray.

19.1. Detailed Description

The QBitArray class provides an array of bits.

QString inherits QByteArray, which is defined as QArray<char>.

Since QBitArray is a QArray, it uses explicit sharing with a reference count.

A QBitArray is a special byte array that can access individual bits and perform bit-operations (AND, OR, XOR and NOT) on entire arrays or bits.

Bits can be manipulated by the setBit() and clearBit() functions, but it is also possible to use the indexing [] operator to test and set individual bits. The [] operator is a little slower than the others, because some tricks are required to implement single-bit assignments.

Example:

```
QBitArray a(3);
a.setBit( 0 );
a.clearBit( 1 );
a.setBit( 2 );                    // a = [1 0 1]
QBitArray b(3);
b[0] = 1;
b[1] = 1;
b[2] = 0;                         // b = [1 1 0]
QBitArray c;
c = ~a & b;                       // c = [0 1 0]
```

19.2. Member Function Documentation

QBitArray::QBitArray ()

Constructs an empty bit array.

QBitArray::QBitArray (const QBitArray & a)

Constructs a shallow copy of *a*.

QBitArray::QBitArray (uint size)

Constructs a bit array of *size* bits. The bits are uninitialized.

bool QBitArray::at (uint index) const

Returns the value (0 or 1) of the bit at position *index*.

See also: operator[].

void QBitArray::clearBit (uint index)

Clears the bit at position *index* (sets it to 0).

See also: setBit() and toggleBit().

QBitArray QBitArray::copy () const

Returns a deep copy of the bit array.

See also: detach().

void QBitArray::detach () `[virtual]`

Detaches from shared bit array data and makes sure that this bit array is the only one referring the data.

If multiple bit arrays share common data, this bit array dereferences the data and gets a copy of the data. Nothing will be done if there is just a single reference.

See also: copy().

Reimplemented from QGArray.

bool QBitArray::fill (bool v, int size = -1)

Fills the bit array with *v* (1's if *v* is TRUE, or 0's if *v* is FALSE).

Will resize the bit array to *size* bits if *size* is nonnegative.

Returns FALSE if a nonnegative *size* was specified and if the bit array could not be resized, otherwise returns TRUE.

See also: resize().

QBitArray & QBitArray::operator&= (const QBitArray & a)

Performs the AND operation between all bits in this bit array and *a.* Returns a reference to this bit array.

The two bit arrays must have the same size. The debug library will warn you if they aren't, the production library blithely ignores the problem.

Example:

```
QBitArray a(3), b(3);
a[0] = 1;  a[1] = 0;        a[2] = 1;        // a = [1 0 1]
b[0] = 0;  b[1] = 0;        b[2] = 1;        // b = [0 0 1]
a &= b;                                      // a = [0 0 1]
```

See also: operator|=(), operator^=() and operator~().

QBitArray & QBitArray::operator= (const QBitArray & a)

Assigns a shallow copy of *a* to this bit array and returns a reference to this array.

QBitVal QBitArray::operator[] (int index)

Implements the [] operator for bit arrays.

The returned QBitVal is a context object. It makes it possible to get and set a single bit value.

Example:

```
QBitArray a( 3 );
a[0] = 0;
a[1] = 1;
a[2] = a[0] ^ a[1];
```

The functions testBit(), setBit() and clearBit() are faster.

See also: at().

QBitArray & QBitArray::operator^= (const QBitArray & a)

Performs the XOR operation between all bits in this bit array and *a*. Returns a reference to this bit array.

The two bit arrays must have the same size. The debug library will warn you if they aren't, the production library blithely ignores the problem.

Example:

```
QBitArray a(3), b(3);
a[0] = 1;  a[1] = 0;        a[2] = 1;       // a = [1 0 1]
b[0] = 0;  b[1] = 0;        b[2] = 1;       // b = [0 0 1]
a ^= b;                                     // a = [1 0 0]
```

See also: operator&=(), operator|=() and operator~().

QBitArray & QBitArray::operator|= (const QBitArray & a)

Performs the OR operation between all bits in this bit array and *a*. Returns a reference to this bit array.

The two bit arrays must have the same size. The debug library will warn you if they aren't, the production library blithely ignores the problem.

Example:

```
QBitArray a(3), b(3);
a[0] = 1;  a[1] = 0;        a[2] = 1;       // a = [1 0 1]
b[0] = 0;  b[1] = 0;        b[2] = 1;       // b = [0 0 1]
a |= b;                                     // a = [1 0 1]
```

See also: operator&=(), operator^=() and operator~().

QBitArray QBitArray::operator~ () const

Returns a bit array which contains the inverted bits of this bit array.

Example:

```
QBitArray a(3);
a[0] = 1;  a[1] = 0; a[2] = 1;      // a = [1 0 1]
QBitArray b = ~a;                   // b = [0 1 0]
```

bool QBitArray::resize (uint size)

Resizes the bit array to *size* bits. Returns TRUE if the bit array could be resized.

When expanding the bit array, the new bits will be uninitialized.

See also: size().

void QBitArray::setBit (uint index)

Sets the bit at position *index* (sets it to 1).

See also: clearBit() and toggleBit().

void QBitArray::setBit (uint index, bool value)

Sets the bit at position *index* to *value.*

Equivalent to:

```
if ( value )
    setBit( index );
else
    clearBit( index );
```

See also: clearBit() and toggleBit().

uint QBitArray::size () const

Returns the size (number of bits) of the bit array.

See also: resize().

bool QBitArray::testBit (uint index) const

Returns TRUE if the bit at position *index* is set.

See also: setBit() and clearBit().

bool QBitArray::toggleBit (uint index)

Toggles the bit at position *index.*

If the previous value was 0, the new value will be 1. If the previous value was 1, the new value will be 0.

See also: setBit() and clearBit().

19.3. Related Functions

QBitArray operator& (const QBitArray & a1, const QBitArray & a2)

Returns the AND result between the bit arrays *a1* and *a2.*

See also: QBitArray::operator&=().

QBitArray operator| (const QBitArray & a1, const QBitArray & a2)

Returns the OR result between the bit arrays *a1* and *a2.*

See also: QBitArray::operator|=().

QBitArray operator^ (const QBitArray & a1, const QBitArray & a2)

Returns the XOR result between the bit arrays *a1* and *a2.*

See also: QBitArray::operator^().

QDataStream & operator<< (QDataStream & s, const QBitArray & a)

Writes a bit array to a stream.

Serialization format:

1 The array size (UINT32)

2 The array bits, (size+7)/8 bytes

QDataStream & operator>> (QDataStream & s, QBitArray & a)

Reads a bit array from a stream.

19.3.1. URL

Generated from http://www.troll.no/qt/qbitarray.html.

20. QBitmap Class Reference

`#include <qbitmap.h>`

Inherits QPixmap.

20.1. Detailed Description

The QBitmap class provides monochrome (1 bit depth) pixmaps.

The QBitmap class is a monochrome off-screen paint device, used mainly for creating custom mouse cursors or brushes.

A QBitmap is a QPixmap with the depth 1. If a pixmap with a depth greater than 1 is assigned to a bitmap, the bitmap will be automatically dithered. A QBitmap is guaranteed to always have the depth 1, unless it is a null bitmap (has depth 0).

When drawing in a QBitmap (or QPixmap with depth 1), we recommend using the global `color0` and `color1` QColor objects. Painting with `color0` sets the bitmap bits to 0, and painting with `color1` sets the bits to 1. For a bitmap, 0-bits indicate background (or white) and 1-bits indicate foreground (or black). Using the `black` and `white` QColor objects make no sense, because the pixel value is not necessarily 0 for black and 1 for white.

Just like the QPixmap class, QBitmap is optimized by the use of implicit sharing, so it is very efficient to pass QBitmap objects as arguments.

See also: QPixmap, QPainter::drawPixmap(), bitBlt() and Shared Classes

Examples:

cursor/cursor.cpp

20.2. Member Function Documentation

QBitmap::QBitmap ()

Constructs a null bitmap.

See also: QPixmap::isNull().

QBitmap::QBitmap (const QSize & size, bool clear = FALSE)

Overloaded constructor; takes a QSize parameter instead of *(w,h).*

QBitmap::QBitmap (const QSize & size, const uchar * bits, bool isXbitmap=FALSE)

Overloaded constructor; takes a QSize parameter instead of *(w,h).*

QBitmap::QBitmap (const QBitmap & bitmap)

Constructs a bitmap which is a copy of *bitmap.*

QBitmap::QBitmap (const char * fileName, const char * format=0)

Constructs a pixmap from the file *fileName.* If the file does not exist, or is of an unknown format, the pixmap becomes a null pixmap.

The parameters are passed on to QPixmap::load().

See also: QPixmap::isNull(), QPixmap::load(), QPixmap::loadFromData(), QPixmap::save() and QPixmap::imageFormat().

QBitmap::QBitmap (int w, int h, bool clear = FALSE)

Constructs a bitmap with *w* width and *h* height.

The contents of the bitmap is uninitialized if *clear* is FALSE, otherwise it is filled with pixel value 0 (the QColor `color0)`.

QBitmap::QBitmap (int w, int h, const uchar * bits, bool isXbitmap=FALSE)

Constructs a bitmap with *w* width and *h* height and sets the contents to *bits.*

The *isXbitmap* should be TRUE if *bits* was generated by the X11 bitmap program. The X bitmap bit order is little endian. The QImage documentation discusses bit order of monochrome images.

Example (creates an arrow bitmap):

```
uchar arrow_bits[] = { 0x3f, 0x1f, 0x0f, 0x1f, 0x3b, 0x71, 0xe0, 0xc0 };
QBitmap bm( 8, 8, arrow_bits, TRUE );
```

QBitmap & QBitmap::operator= (const QImage & image)

Converts the image *image* to a bitmap that is assigned to this bitmap. Returns a reference to the bitmap.

Dithering will be performed if the image has a depth greater than 1.

QBitmap & QBitmap::operator= (const QPixmap & pixmap)

Assigns the pixmap *pixmap* to this bitmap and returns a reference to this bitmap.

Dithering will be performed if the pixmap has a depth greater than 1.

QBitmap & QBitmap::operator= (const QBitmap & bitmap)

Assigns the bitmap *bitmap* to this bitmap and returns a reference to this bitmap.

QBitmap QBitmap::xForm (const QWMatrix & matrix) const

Returns a copy of the bitmap that is transformed using *matrix.*

This function does exactly the same as QPixmap::xForm(), except that it returns a QBitmap instead of a QPixmap.

See also: QPixmap::xForm().

20.2.1. URL

Generated from http://www.troll.no/qt/qbitmap.html.

21. QBitVal Class Reference

```
#include <qbitarray.h>
```

21.1. Detailed Description

The QBitVal class is an internal class, used with QBitArray.

The QBitVal is required by the indexing [] operator on bit arrays. It is probably a bad idea to use it in any other context.

21.2. Member Function Documentation

QBitVal::QBitVal (QBitArray * a, uint i)

Constructs a reference to an element in a QBitArray. This is what QBitArray::operator[] contructs its return value with.

QBitVal::operator int ()

Returns the value referenced by the QBitVal.

QBitVal& QBitVal::operator= (const QBitVal & v)

Sets the value referenced by the QBitVal to that referenced by another QBitVal.

QBitVal& QBitVal::operator= (int v)

Sets the value referenced by the QBitVal.

21.2.1. URL

Generated from http://www.troll.no/qt/qbitval.html.

22. QBoxLayout Class Reference

`#include <qlayout.h>`

Inherits QLayout.

Inherited by QHBoxLayout and QVBoxLayout.

22.1. Detailed Description

The QBoxLayout class lines up child widgets horizontally or vertically.

QBoxLayout takes the space it gets (from its parent layout or from the mainWindget()), divides it up into a row of boxes and makes each managed widget fill one box.

If the QBoxLayout is `Horizontal,` the boxes are beside each other, with suitable sizes. Each widget (or other box) will get at least its minimum sizes and at most its maximum size, and any excess space is shared according to the stretch factors (more about that below).

If the QBoxLayout is `Vertical,` the boxes are above and below each other, again with suitable sizes.

The easiest way to create a QBoxLayout is to use one of the convenience classes QHBoxLayout (for `Horizontal` boxes) or QVBoxLayout (for `Vertical` boxes). You can also use the QBoxLayout constuctor directly, specifying its direction as `LeftToRight, Down, RightToLeft` or `Up.`

If the QBoxLayout is not the top-level layout (ie. is not managing all of the widget's area and children), you must add it to its parent layout before you can do anything with it. The normal way to add a layout is by calling parentLayout->addLayout().

Once you have done that, you can add boxes to the QBoxLayout using one of four functions:

- addWidget() to add a widget to the QBoxLayout and set the widget's stretch factor. (The stretch factor is along the row if boxes.)
- addSpacing() to create an empty box; this is one of the functions you use to create nice and spacious dialogs. See below for ways to set margins.
- addStretch() to create an empty, stretchable box.
- addLayout() to add a box containing another QLayout to the row and set that layout's stretch factor.

Finally, if the layout is a top-level one, you activate() it.

QBoxLayout also includes two margin widths: The border width and the inter-box width. The border width is the width of the reserved space along each of the QBoxLayout's four sides. The intra-widget width is the width of the automatically allocated spacing between neighbouring boxes. (You can use addSpacing() to get more space.)

The border width defaults to 0, and the intra-widget width defaults to the same as the border width. Both are set using arguments to the constructor.

You will almost always want to use the convenience classes for QBoxLayout: QVBoxLayout and QHBoxLayout, because of their simpler constructors.

Examples:
    pref/pref.cpp
    layout/layout.cpp

22.2. Member Function Documentation

QBoxLayout::QBoxLayout (QWidget * parent, Direction d, int border=0, int autoBorder = -1, const char * name=0)

Creates a new QBoxLayout with direction *d* and main widget *parent. parent* may not be 0.

border is the number of pixels between the edge of the widget and the managed children. *autoBorder* is the default number of pixels between neighbouring children. If *autoBorder* is -1 the value of *border* is used.

name is the internal object name

See also: direction().

Examples:
layout/layout.cpp

QBoxLayout::QBoxLayout (Direction d, int autoBorder = -1, const char * name=0)

If *autoBorder* is -1, this QBoxLayout will inherit its parent's defaultBorder(), otherwise *autoBorder* is used.

You have to insert this box into another layout before using it.

QBoxLayout::~QBoxLayout ()

Destroys this box.

void QBoxLayout::addLayout (QLayout * layout, int stretch = 0)

Adds *layout* to the box, with serial stretch factor *stretch.*

See also: addWidget() and addSpacing().

Examples:
pref/pref.cpp
layout/layout.cpp
widgets/widgets.cpp

void QBoxLayout::addSpacing (int size)

Adds a non-stretchable space with size *size.* QBoxLayout gives default border and spacing. This function adds additional space.

See also: addStretch().

Examples:
widgets/widgets.cpp

void QBoxLayout::addStretch (int stretch = 0)

Adds a stretchable space with zero minimum size and stretch factor *stretch.*

See also: addSpacing().

Examples:
pref/pref.cpp
widgets/widgets.cpp

void QBoxLayout::addStrut (int size)

Limits the perpendicular dimension of the box (e.g. height if the box is LeftToRight) to a minimum of *size*. Other constraints may increase the limit.

void QBoxLayout::addWidget (QWidget * widget, int stretch = 0, int align = AlignCenter)

Adds *widget* to the box, with stretch factor *stretch* and alignment *align*.

The stretch factor applies only in the direction of the QBoxLayout, and is relative to the other boxes and widgets in this QBoxLayout. Widgets and boxes with higher stretch factor grow more.

If the stretch factor is 0 and nothing else in the QBoxLayout can grow at all, the widget may still grow up to its maximum size.

Alignment is perpendicular to direction(), alignment in the serial direction is done with addStretch().

For horizontal boxes, the possible alignments are

- `AlignCenter` centers vertically in the box.
- `AlignTop` aligns to the top border of the box.
- `AlignBottom` aligns to the bottom border of the box.

For vertical boxes, the possible alignments are

- `AlignCenter` centers horizontally in the box.
- `AlignLeft` aligns to the left border of the box.
- `AlignRight` aligns to the right border of the box.

Alignment only has effect if the size of the box is greater than the widget's maximum size.

See also: addLayout() and addSpacing().

Examples:
pref/pref.cpp
layout/layout.cpp
widgets/widgets.cpp

QBoxLayout::Direction QBoxLayout::direction() const

Returns the (serial) direction of the box. addWidget(), addBox() and addSpacing() works in this direction; the stretch stretches in this direction. Alignment works perpendicular to this direction.

The directions are `LeftToRight, RightToLeft, TopToBottom` and `BottomToTop.` For the last two, the shorter aliases `Down` and `Up` are also available.

See also: addWidget(), addBox() and addSpacing().

void QBoxLayout::initGM () `[virtual protected]`

Initializes this box.

Reimplemented from QLayout.

QChain * QBoxLayout::mainHorizontalChain () `[virtual protected]`

Returns the main horizontal chain, so that a box can be put into other boxes (or other types of QLayout).

Reimplemented from QLayout.

QChain * QBoxLayout::mainVerticalChain () `[virtual protected]`

Returns the main vertical chain, so that a box can be put into other boxes (or other types of QLayout).

Reimplemented from QLayout.

22.2.1. URL

Generated from http://www.troll.no/qt/qboxlayout.html.

23. QBrush Class Reference

`#include <qbrush.h>`

23.1. Detailed Description

The QBrush class defines the fill pattern of shapes drawn by a QPainter.

A brush has a style and a color. One of the brush styles is a custom pattern, which is defined by a QPixmap.

The brush style defines the fill pattern. The default brush style is `NoBrush` (depends on how you construct a brush). This style tells the painter to not fill shapes. The standard style for filling is called `SolidPattern`.

The brush color defines the color of the fill pattern. The QColor documentation lists the predefined colors.

Use the QPen class for specifying line/outline styles.

Example:

```
QPainter painter;
QBrush   brush( yellow );           // yellow solid pattern
painter.begin( &anyPaintDevice );   // paint something
painter.setBrush( brush );          // set the yellow brush
painter.setPen( NoPen );            // do not draw outline
painter.drawRect( 40,30, 200,100 ); // draw filled rectangle
painter.setBrush( NoBrush );        // do not fill
painter.setPen( black );            // set black pen, 0 pixel width
painter.drawRect( 10,10, 30,20 );   // draw rectangle outline
painter.end();                      // painting done
```

See the setStyle() function for a complete list of brush styles.

See also: QPainter, QPainter::setBrush() and QPainter::setBrushOrigin().

Examples:
- tictac/tictac.cpp
- drawdemo/drawdemo.cpp
- picture/picture.cpp

23.2. Member Function Documentation

QBrush::QBrush ()

Constructs a default black brush with the style `NoBrush` (will not fill shapes).

QBrush::QBrush (BrushStyle style)

Constructs a black brush with the specified style.

See also: setStyle().

QBrush::QBrush (const QColor & color, BrushStyle style=SolidPattern)

Constructs a brush with a specified color and style.

See also: setColor() and setStyle().

QBrush::QBrush (const QColor & color, const QPixmap & pixmap)

Constructs a brush with a specified color and a custom pattern.

The color will only have an effect for monochrome pixmaps, i.e. QPixmap::depth() == 1.

See also: setColor() and setPixmap().

QBrush::QBrush (const QBrush & b)

Constructs a brush which is a shallow copy of *b.*

QBrush::~QBrush ()

Destroys the brush.

const QColor & QBrush::color () const

Returns the brush color.

See also: setColor().

bool QBrush::operator!= (const QBrush & b) const

Returns TRUE if the brush is different from *b,* or FALSE if the brushes are equal.

Two brushes are different if they have different styles, colors or pixmaps.

See also: operator==().

QBrush & QBrush::operator= (const QBrush & b)

Assigns *b* to this brush and returns a reference to this brush.

bool QBrush::operator== (const QBrush & b) const

Returns TRUE if the brush is equal to *b,* or FALSE if the brushes are different.

Two brushes are equal if they have equal styles, colors and pixmaps.

See also: operator!=().

QPixmap * QBrush::pixmap () const

Returns a pointer to the custom brush pattern.

A null pointer is returned if no custom brush pattern has been set.

See also: setPixmap().

void QBrush::setColor (const QColor & c)

Sets the brush color to *c.*

See also: color() and setStyle().

Examples:

picture/picture.cpp

void QBrush::setPixmap (const QPixmap & pixmap)

Sets the brush pixmap. The style is set to `CustomPattern`.

The curren brush color will only have an effect for monochrome pixmaps, i.e. QPixmap::depth() == 1.

See also: pixmap() and color().

void QBrush::setStyle (BrushStyle s)

Sets the brush style to *s*.

The brush styles are:

- `NoBrush` will not fill shapes (default).
- `SolidPattern` solid (100%) fill pattern.
- `Dense1Pattern` 94% fill pattern.
- `Dense2Pattern` 88% fill pattern.
- `Dense3Pattern` 63% fill pattern.
- `Dense4Pattern` 50% fill pattern.
- `Dense5Pattern` 37% fill pattern.
- `Dense6Pattern` 12% fill pattern.
- `Dense7Pattern` 6% fill pattern.
- `HorPattern` horizontal lines pattern.
- `VerPattern` vertical lines pattern.
- `CrossPattern` crossing lines pattern.
- `BDiagPattern` diagonal lines (directed /) pattern.
- `FDiagPattern` diagonal lines (directed \ ) pattern.
- `DiagCrossPattern` diagonal crossing lines pattern.
- `CustomPattern` set when a pixmap pattern is being used.

See also: style().

BrushStyle QBrush::style () const

Returns the brush style.

See also: setStyle().

23.3. Related Functions

QDataStream & operator<< (QDataStream & s, const QBrush & b)

Writes a brush to the stream and returns a reference to the stream.

The serialization format is:

1 The brush style (UINT8)

2 The brush color (QColor)

3 If style == `CustomPattern`: the brush pixmap (QPixmap)

QDataStream & operator>> (QDataStream & s, QBrush & b)

Reads a brush from the stream and returns a reference to the stream.

23.3.1. URL

Generated from http://www.troll.no/qt/qbrush.html.

24. QBuffer Class Reference

`#include <qbuffer.h>`

Inherits QIODevice.

24.1. Detailed Description

The QBuffer class is an I/O device that operates on a QByteArray

QBuffer is an I/O device for reading and writing a memory buffer. A QBuffer may be used directly (readBlock() and writeBlock()) or more conveniently via QDataStream or QTextStream. Most of its behavior is inherited from QIODevice.

A QBuffer has an associated QByteArray which holds the buffer data. Writing data at the end (i.e. size()) of the buffer expands the byte array.

For convenience, the byte stream classes QDataStream and QTextStream can operate on a QByteArray (or a QString) via an internal QBuffer:

```
QString str;
QTextStream ts( str, IO_WriteOnly );
ts << "pi = " << 3.14;                    // str == "pi = 3.14"
```

See also: QFile, QDataStream and QTextStream.

Examples:
grapher/grapher.cpp

24.2. Member Function Documentation

QBuffer::QBuffer ()

Constructs an empty buffer.

QBuffer::QBuffer (QByteArray buf)

Constructs a buffer and sets the buffer contents to *buf.*

See also: setBuffer().

QBuffer::~QBuffer ()

Destroys the buffer.

bool QBuffer::at (int pos) `[virtual]`

Sets the buffer index to *pos.* Returns TRUE if successful, otherwise FALSE.

See also: size().

Reimplemented from QIODevice.

int QBuffer::at () const `[virtual]`

Returns the buffer index.

See also: size().

Reimplemented from QIODevice.

QByteArray QBuffer::buffer () const

Returns the buffer most recently set by setBuffer(), or at construction.

void QBuffer::close () `[virtual]`

Closes an open buffer.

See also: open().

Examples:
grapher/grapher.cpp

Reimplemented from QIODevice.

void QBuffer::flush () `[virtual]`

The flush function does nothing.

Reimplemented from QIODevice.

int QBuffer::getch () `[virtual]`

Reads a single byte/character from the buffer.

Returns the byte/character read, or -1 if the end of the buffer has been reached.

See also: putch() and ungetch().

Reimplemented from QIODevice.

bool QBuffer::open (int m) `[virtual]`

Opens the file specified by the file name currently set, using the mode *m*. Returns TRUE if successful, otherwise FALSE.

The mode parameter *m* must be a combination of the following flags.

- `IO_ReadOnly` opens a buffer in read-only mode.
- `IO_WriteOnly` opens a buffer in write-only mode.
- `IO_ReadWrite` opens a buffer in read/write mode.
- `IO_Append` sets the buffer index to the end of the buffer.
- `IO_Truncate` truncates the buffer.

See also: close() and isOpen().

Examples:
grapher/grapher.cpp

Reimplemented from QIODevice.

int QBuffer::putch (int ch) `[virtual]`

Writes the character *ch* into the buffer, overwriting the character at the current index, extending the buffer if necessary.

Returns *ch,* or -1 if some error occurred.

See also: getch() and ungetch().

Examples:
grapher/grapher.cpp

Reimplemented from QIODevice.

int QBuffer::readBlock (char * p, uint len) `[virtual]`

Reads at most *len* bytes from the buffer into *p* and returns the number of bytes actually read.

Returns -1 if a serious error occurred.

See also: writeBlock().

Reimplemented from QIODevice.

int QBuffer::readLine (char * p, uint maxlen) `[virtual]`

Reads a line of text.

Reads bytes from the buffer until end-of-line is reached, or up to *maxlen* bytes.

See also: readBlock().

Reimplemented from QIODevice.

bool QBuffer::setBuffer (QByteArray buf)

Replaces the buffer's contents with *buf.*

This may not be done while the buffer is open.

Note that if you open the buffer in write mode (`IO_WriteOnly` or IO_ReadWrite) and write something into the buffer, *buf* is also modified because QByteArray is an explicitly shared class.

Example:

```
QString str = "abc";
QBuffer b( str );
b.open( IO_WriteOnly );
b.at( 3 );                                  // position at \0
b.writeBlock( "def", 4 );                   // write including \0
b.close();
  // Now, str == "abcdef"
```

See also: open and Shared Classes

uint QBuffer::size () const `[virtual]`

Returns the number of bytes in the buffer.

See also: at().

Reimplemented from QIODevice.

int QBuffer::ungetch (int ch) `[virtual]`

Puts the character *ch* back into the buffer and decrements the index if it is not zero.

This function is normally called to "undo" a getch() operation.

Returns *ch,* or -1 if some error occurred.

See also: getch() and putch().

Reimplemented from QIODevice.

int QBuffer::writeBlock (const char * p, uint len) `[virtual]`

Writes *len* bytes from *p* into the buffer at the current index, overwriting any characters there and extending the buffer if necessary. Returns the number of bytes actually written.

Returns -1 if a serious error occurred.

See also: readBlock().

Reimplemented from QIODevice.

24.2.1. URL

Generated from http://www.troll.no/qt/qbuffer.html.

25. QButton Class Reference

`#include <qbutton.h>`

Inherits QWidget.

Inherited by QCheckBox, QPushButton, QRadioButton and QToolButton.

25.1. Detailed Description

The QButton class is the abstract base class of button widgets, providing functionality common to buttons.

The QButton class implements an abstract button, and lets subclasses specify how to reply to user actions and how to draw the button.

QButton provides both push and toggle buttons. The QRadioButton and QCheckBox classes provide only toggle buttons, QPushButton and QToolButton provide both toggle and push buttons.

Any button can have either a text or pixmap label. setText() sets the button to be a text button and setPixmap() sets it to be a pixmap button. The text/pixmap is manipulated as necessary to create "disabled" appearance when the button is disabled.

QButton provides most of the states used for buttons:

- isDown() determines whether the button is *pressed* down.
- isOn() determines whether the button is *on.* Only toggle buttons can be switched on and off (see below).
- isEnabled() determines whether the button can be pressed by the user.
- setAutoRepeat() determines whether the button will auto-repeat if the user holds it down.
- setToggleButton() determines whether the button is a toggle button or not.

The difference between isDown() and isOn() is as follows: When the user clicks a toggle button to toggle it on, the button is first *pressed,* then released into *on* state. When the user clicks it again (to toggle it off) the button moves first to the *pressed* state, then to the *off* state (isOn() and isDown() are both FALSE).

Default buttons (as used in many dialogs) are provided by QPushButton::setDefault() and QPushButton::setAutoDefault().

QButton provides four signals:

- pressed() is emitted when the left mouse button is pressed while the mouse cursor is inside the button.
- released() is emitted when the left mouse button is released.
- clicked() is emitted when the button is first pressed then released, or when the accelerator key is typed, or when animateClick() is called.
- toggled() is emitted when the state of a toggle button changes.

If the button is a text button with "&" in its text, QButton creates an automatic accelerator key. This code creates a push button labelled "Rock & Roll" (where the c is underscored). The button gets an automatic accelerator key, Alt-C:

```
QPushButton *p = new QPushButton( "Ro&ck && Roll", this );
```

In this example, when the user presses Alt-C the button will animate a click.

You can also set a custom accelerator using the setAccel() function. This is useful mostly for pixmap buttons since they have no automatic accelerator.

```
QPushButton *p;
p->setPixmap( QPixmap("print.gif") );
p->setAccel( ALT+Key_F7 );
```

All of the buttons provided by Qt (QPushButton, QToolButton, QCheckBox and QRadioButton) can display both text and pixmaps.

To subclass QButton, you have to reimplement at least drawButton() (to draw the button's outskirts) and drawButtonLabel() (to draw its text or pixmap). It is generally advisable to reimplement sizeHint() as well, and sometimes hitButton() (to determine whether a button press is within the button).

To reduce flickering the QButton::paintEvent() sets up a pixmap that the drawButton() function draws in. You should not reimplement paintEvent() for a subclass of QButton unless you want to take over all drawing.

See also: QButtonGroup and Microsoft Style Guide

Examples:
    tictac/tictac.cpp

25.2. Member Function Documentation

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

Constructs a standard button with a parent widget and a name.

If *parent* is a QButtonGroup, this constructor calls QButtonGroup::insert().

QButton::~QButton ()

Destroys the button and all its child widgets.

int QButton::accel () const

Returns the accelerator key currently set for the button, or 0 if no accelerator key has been set.

See also: setAccel().

void QButton::animateClick () `[slot]`

Performs an animated click: The button is pressed and a short while later released.

pressed(), released(), clicked() and toggled() signals are emitted as appropriate.

This function does nothing if the button is disabled.

See also: setAccel().

bool QButton::autoRepeat () const

Returns TRUE if the button is auto-repeating, else FALSE.

The default is FALSE.

See also: setAutoRepeat().

bool QButton::autoResize () const

Returns TRUE if auto-resizing is enabled, or FALSE if auto-resizing is disabled.

Auto-resizing is disabled by default.

See also: setAutoResize().

void QButton::clicked () `[signal]`

This signal is emitted when the button is activated (i.e. first pressed down and then released when the mouse cursor is inside the button).

See also: pressed(), released() and toggled().

void QButton::drawButton (QPainter *) `[virtual protected]`

Draws the button. The default implementation does nothing.

This virtual function is reimplemented by subclasses to draw real buttons.

Reimplemented in QToolButton, QPushButton, QCheckBox and QRadioButton.

void QButton::drawButtonLabel (QPainter *) `[virtual protected]`

Draws the button text or pixmap. The default implementation does nothing.

This virtual function is reimplemented by subclasses to draw real buttons.

Reimplemented in QToolButton, QPushButton, QCheckBox and QRadioButton.

void QButton::enabledChange (bool e) `[virtual protected]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QWidget.

void QButton::focusInEvent (QFocusEvent *) `[virtual protected]`

Handles focus in events for the button.

See also: focusOutEvent().

Reimplemented from QWidget.

void QButton::focusOutEvent (QFocusEvent *) `[virtual protected]`

Handles focus out events for the button.

See also: focusInEvent().

Reimplemented from QWidget.

bool QButton::hitButton (const QPoint & pos) const `[virtual protected]`

Returns TRUE if *pos* is inside the widget rectangle, or FALSE if it is outside.

This virtual function is reimplemented by subclasses.

Reimplemented in QRadioButton.

bool QButton::isDown () const

Returns TRUE if the button pressed down, or FALSE if it is standing up.

See also: setDown().

Examples:

tictac/tictac.cpp

bool QButton::isOn () const

Returns TRUE if this toggle button is switched on, or FALSE if it is switched off.

See also: setOn() and toggleButton().

Examples:
qmag/qmag.cpp

bool QButton::isToggleButton () const

Returns TRUE if the button is a toggle button.

See also: setToggleButton().

void QButton::keyPressEvent (QKeyEvent * e) `[virtual protected]`

Handles keyboard events for the button.

Space is the only key that has any effect; it calls animateClick().

Reimplemented from QWidget.

void QButton::mouseMoveEvent (QMouseEvent * e) `[virtual protected]`

Handles mouse move events for the button.

See also: mousePressEvent() and mouseReleaseEvent().

Reimplemented from QWidget.

void QButton::mousePressEvent (QMouseEvent * e) `[virtual protected]`

Handles mouse press events for the button.

See also: mouseReleaseEvent().

Reimplemented from QWidget.

void QButton::mouseReleaseEvent (QMouseEvent * e) `[virtual protected]`

Handles mouse release events for the button.

See also: mousePressEvent().

Reimplemented from QWidget.

void QButton::paintEvent (QPaintEvent * event) `[virtual protected]`

Handles paint events for the button.

Opens the painter on the button and calls drawButton().

Reimplemented from QWidget.

const QPixmap * QButton::pixmap () const

Returns the button pixmap, or 0 if the button has no pixmap.

void QButton::pressed () `[signal]`

This signal is emitted when the button is pressed down.

See also: released() and clicked().

void QButton::released () `[signal]`

This signal is emitted when the button is released.

See also: pressed(), clicked() and toggled().

void QButton::setAccel (int key)

Specifies an accelerator *key* for the button, or removes the accelerator if *key* is 0.

Setting a button text containing a shortcut character (for example the 'x' in E&xit) automatically defines an ALT+letter accelerator for the button. You only need to call this function in order to specify a custom accelerator.

Example:

```
QPushButton *b1 = new QPushButton;
b1->setText( "&OK" );               // sets accel ALT+'O'
QPushButton *b2 = new QPushButton;
b2->setPixmap( printIcon );         // pixmap instead of text
b2->setAccel( CTRL+'P' );           // custom accel
```

See also: accel(), setText() and QAccel.

void QButton::setAutoRepeat (bool enable)

Turns on auto-repeat for the button if *enable* is TRUE, or turns it off if *enable* is FALSE.

When auto-repeat is enabled, the clicked() signal is emitted at regular intervals while the buttons is down.

setAutoRepeat() has no effect for toggle buttons.

See also: isDown(), autoRepeat() and clicked().

void QButton::setAutoResize (bool enable)

Enables auto-resizing if *enable* is TRUE, or disables it if *enable* is FALSE.

When auto-resizing is enabled, the button will resize itself whenever the contents change.

See also: autoResize() and adjustSize().

void QButton::setDown (bool enable)

Sets the state of the button to pressed down if *enable* is TRUE or to standing up if *enable* is FALSE.

If the button is a toggle button, it is *not* toggled. Call toggle() as well if you need to do that. The pressed() and released() signals are not emitted by this function.

This method is provided in case you need to override the mouse event handlers.

See also: isDown(), setOn(), toggle() and toggled().

void QButton::setOn (bool enable) `[protected]`

Switches a toggle button on if *enable* is TRUE or off if *enable* is FALSE. This function should be called only for toggle buttons.

See also: isOn() and toggleButton().

void QButton::setPixmap (const QPixmap & pixmap)

Sets the button to display *pixmap* and repaints at once.

If *pixmap* is monochrome (i.e. it is a QBitmap or its depth is 1) and it does not have a mask, this function sets the pixmap to be its own mask. The purpose of this is to draw transparent bitmaps, which is important for e.g. toggle buttons.

The button resizes itself if auto-resizing is enabled, changes its minimum size if autoMinimumSize() is enabled, and always disables any accelerator.

See also: pixmap(), setText(), setAutoMinimumSize(), setAutoResize(), setAccel() and QPixmap::mask().

void QButton::setText (const char * text)

Sets the button to display *text* and repaints

The button resizes itself if auto-resizing is enabled, changes its minimum size if autoMinimumSize() is enabled, and sets the appropriate accelerator.

See also: text(), setPixmap(), setAutoMinimumSize(), setAutoResize(), setAccel() and QPixmap::mask().

Examples:
xform/xform.cpp
layout/layout.cpp
qmag/qmag.cpp
widgets/widgets.cpp

void QButton::setToggleButton (bool enable) `[protected]`

Makes the button a toggle button if *enable* is TRUE, or a normal button if *enable* is FALSE.

Note that this function is protected. It is called from subclasses to enable the toggle functionality. QCheckBox and QRadioButton are toggle buttons. QPushButton is initially not a toggle button, but QPushButton::setToggleButton() can be called to create toggle buttons.

See also: isToggleButton().

const char * QButton::text () const

Returns the button text, or 0 if the button has no text.

See also: setText().

void QButton::toggle () `[slot]`

if this is a toggle button, toggles it.

void QButton::toggled (bool on) `[signal]`

This signal is emitted whenever a toggle button changes status. *on* is TRUE if the button is on, or FALSE if the button is off.

This may be the result of a user action, toggle() slot activation, or because setOn() was called.

See also: clicked().

25.2.1. URL

Generated from http://www.troll.no/qt/qbutton.html.

26. QButtonGroup Class Reference

```
#include <qbuttongroup.h>
```

Inherits QGroupBox.

26.1. Detailed Description

The QButtonGroup widget organizes QButton widgets in a group.

A button group widget makes it easier to deal with groups of buttons. A button in a button group is associated with a unique identifer. The button group emits a clicked() signal with this identifier when the button is clicked. Thus, a button group is an ideal solution when you have several similar buttons and want to connect all their clicked() signals, for example, to one slot.

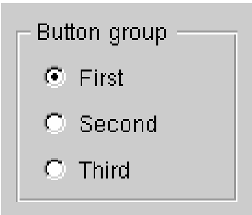
An exclusive button group switches off all toggle buttons except the one that was clicked. A button group is by default non-exclusive, however, it automatically becomes an exclusive group when a QRadioButton is inserted.

There are two ways of using a button group:

1 The button group is a parent widget of a number of buttons, i.e. the button group is the parent argument in the button contructor. The buttons are assigned identifiers 0, 1, 2 etc. in the order they are created. A QButtonGroup can display a frame and a title because it inherits QGroupBox.

2 The button group is an invisible widget and the contained buttons have some other parent widget. A button must then be manually inserted using the insert() function with an identifer.

By default, the button group's setFont() and setPalette() functions do not change the appearance of the buttons, but you can use setFontPropagation() and setPalettePropagation() to change that.



See also: QButton, QPushButton, QCheckBox and QRadioButton.

Examples:

drawdemo/drawdemo.cpp
xform/xform.cpp
pref/pref.cpp
widgets/widgets.cpp

26.2. Member Function Documentation

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

Constructs a button group with no title.

The *parent* and *name* arguments are passed to the QWidget constructor.

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

Constructs a button group with a title.

The *parent* and *name* arguments are passed to the QWidget constructor.

QButtonGroup::~QButtonGroup ()

Destroys the button group and its child widgets.

void QButtonGroup::buttonClicked () `[protected slot]`

For internal use only.

void QButtonGroup::buttonPressed () `[protected slot]`

For internal use only.

void QButtonGroup::buttonReleased () `[protected slot]`

For internal use only.

void QButtonGroup::buttonToggled (bool on) `[protected slot]`

For internal use only.

void QButtonGroup::clicked (int id) `[signal]`

This signal is emitted when a button in the group is clicked. The *id* argument is the button's identifier.

QButton * QButtonGroup::find (int id) const

Finds and returns a pointer to the button with the specified identifier *id.*

Returns null if the button was not found.

int QButtonGroup::insert (QButton * button, int id=-1)

Inserts a button with the identifier *id* into the button group. Returns the button identifier.

It is not necessary to manually insert buttons that have this button group as their parent widget. An exception is when you want custom identifiers instead of the default 0, 1, 2 etc.

The button is assigned the identifier *id* or an automatically generated identifier. It works as follows: If *id* >= 0, this identifier is assigned. If *id* == -1 (default), the identifier is equal to the number of buttons in the group. If *id* is any other negative integer, for instance -2, a unique identifier (negative integer <= -2) is generated.

Inserting several buttons with *id* = -1 assigns the identifers 0, 1, 2, etc.

This function calls setExclusive(TRUE) if *button* is a QRadioButton.

See also: find(), remove() and setExclusive().

bool QButtonGroup::isExclusive () const

Returns TRUE if the button group is exclusive, otherwise FALSE.

See also: setExclusive().

void QButtonGroup::pressed (int id) `[signal]`

This signal is emitted when a button in the group is pressed. The *id* argument is the button's identifier.

void QButtonGroup::released (int id) `[signal]`

This signal is emitted when a button in the group is released. The *id* argument is the button's identifier.

void QButtonGroup::remove (QButton * button)

Removes a button from the button group.

See also: insert().

void QButtonGroup::setButton (int id)

Sets the button with id *id* to be on, and if this is an exclusive group, all other button in the group to be off.

void QButtonGroup::setExclusive (bool enable)

Sets the button group to be exclusive if *enable* is TRUE, or to be non-exclusive if *enable* is FALSE.

An exclusive button group switches off all other toggle buttons when one is switched on. This is ideal for groups of radio buttons A non-exclusive group allow many buttons to be swithed on at the same time.

The default setting is FALSE. A button group automatically becomes an exclusive group when a QRadioButton is inserted.

See also: isExclusive().

26.2.1. URL

Generated from http://www.troll.no/qt/qbuttongroup.html.

27. QByteArray Class Reference

`#include <qstring.h>`

Inherits QArray.

Inherited by QBitArray and QString.

27.1. Detailed Description

The QByteArray class provides an array of bytes.

QByteArray is defined as QArray<char>.

27.2. Member Function Documentation

QByteArray::QByteArray ()

Constructs a null array.

See also: isNull().

QByteArray::QByteArray (const QByteArray & a)

Constructs a shallow copy of *a*.

See also: assign().

QByteArray::QByteArray (int size)

Constructs an array with room for *size* elements. Makes a null array if *size* == 0.

Note that the elements are not initialized.

See also: resize() and isNull().

QByteArray::QByteArray (int, int) `[protected]`

Constructs an array *without allocating* array space. The arguments should be (0, 0). Use at own risk.

QByteArray::~QByteArray ()

Dereferences the array data and deletes it if this was the last reference.

QByteArray::operator const char * () const

Cast operator. Returns a pointer to the array.

See also: data().

QByteArray & QByteArray::assign (const QByteArray & a)

Shallow copy. Dereferences the current array and references the data contained in *a* instead. Returns a reference to this array.

See also: operator=().

QByteArray & QByteArray::assign (const char * data, uint size)

Shallow copy. Dereferences the current array and references the array data *data,* which contains *size* elements. Returns a reference to this array.

Do not delete *data* later, QArray takes care of that.

char & QByteArray::at (uint index) const

Returns a reference to the element at position *index* in the array.

This can be used to both read and set an element.

See also: operator[]().

int QByteArray::contains (const char & v) const

Returns the number of times *v* occurs in the array.

See also: find().

QByteArray QByteArray::copy () const

Returns a deep copy of this array.

See also: detach() and duplicate().

char * QByteArray::data () const

Returns a pointer to the actual array data.

The array is a null array if data() == 0 (null pointer).

See also: isNull().

void QByteArray::detach () `[virtual]`

Detaches this array from shared array data, i.e. makes a private, deep copy of the data.

Copying will only be performed if the reference count is greater than one.

See also: copy().

Reimplemented from QGArray.

QByteArray & QByteArray::duplicate (const QByteArray & a)

Deep copy. Dereferences the current array and obtains a copy of the data contained in *a* instead. Returns a reference to this array.

See also: copy().

QByteArray & QByteArray::duplicate (const char * data, uint size)

Deep copy. Dereferences the current array and obtains a copy of the array data *data* instead. Returns a reference to this array.

See also: copy().

bool QByteArray::fill (const char & v, int size=-1=-1)

Fills the array with the value *v.* If *size* is specified as different from -1, then the array will be resized before filled.

Returns TRUE if successful, or FALSE if the memory cannot be allocated (only when *size* != -1).

See also: resize().

int QByteArray::find (const char & v, uint index=0) const

Finds the first occurrence of *v,* starting at position *index.*

Returns the position of *v,* or -1 if *v* could not be found.

See also: contains().

bool QByteArray::isEmpty () const

Returns TRUE if the array is empty, i.e. size() == 0, otherwise FALSE.

isEmpty() is equivalent with isNull() for QArray. Note that this is not the case for QString::isEmpty().

bool QByteArray::isNull () const

Returns TRUE if the array is null, otherwise FALSE.

A null array has size() == 0 and data() == 0.

uint QByteArray::nrefs () const

Returns the reference count for the shared array data. This reference count is always greater than zero.

bool QByteArray::operator!= (const QByteArray & a) const

Returns TRUE if this array is different from *a,* otherwise FALSE.

The two arrays are bitwise compared.

See also: operator==().

QByteArray & QByteArray::operator= (const QByteArray & a)

Assigns a shallow copy of *a* to this array and returns a reference to this array.

Equivalent to assign(a).

bool QByteArray::operator== (const QByteArray & a) const

Returns TRUE if this array is equal to *a,* otherwise FALSE.

The two arrays are bitwise compared.

See also: operator!=().

char & QByteArray::operator[] (int index) const

Returns a reference to the element at position *index* in the array.

This can be used to both read and set an element. Equivalent to at().

See also: at().

void QByteArray::resetRawData (const char * data, uint size)

Resets raw data that was set using setRawData().

The arguments must be the data and length that were passed to setRawData(). This is for consistency checking.

See also: setRawData().

bool QByteArray::resize (uint size)

Resizes (expands or shrinks) the array to *size* elements. The array becomes a null array if *size* == 0.

Returns TRUE if successful, or FALSE if the memory cannot be allocated.

New elements will not be initialized.

See also: size().

QByteArray & QByteArray::setRawData (const char * data, uint size)

Sets raw data and returns a reference to the array.

Dereferences the current array and sets the new array data to *data* and the new array size to *size*. Do not attempt to resize or re-assign the array data when raw data has been set. Call resetRawData(d,len) to reset the array.

Setting raw data is useful because it sets QArray data without allocating memory or copying data.

Example I (intended use):

```
static char bindata[] = { 231, 1, 44, ... };
QByteArray  a;
a.setRawData( bindata, sizeof(bindata) );   // a points to bindata
QDataStream s( a, IO_ReadOnly );            // open on a's data
s >> <something>;                           // read raw bindata
a.resetRawData( bindata, sizeof(bindata) ); // finished
```

Example II (you don't want to do this):

```
static char bindata[] = { 231, 1, 44, ... };
QByteArray  a, b;
a.setRawData( bindata, sizeof(bindata) );   // a points to bindata
a.resize( 8 );                              // will crash
b = a;                                      // will crash
a[2] = 123;                                 // might crash
  // forget to resetRawData - will crash
```

Warning: If you do not call resetRawData(), QArray will attempt to deallocate or reallocate the raw data, which might not be too good. Be careful.

See also: resetRawData().

uint QByteArray::size () const

Returns the size of the array (max number of elements).

The array is a null array if size() == 0.

See also: isNull() and resize().

bool QByteArray::truncate (uint pos)

Truncates the array at position *pos.*

Returns TRUE if successful, or FALSE if the memory cannot be allocated.

Equivalent to resize(*pos).*

See also: resize().

27.3. Related Functions

Q_UINT16 qChecksum (const char * data, uint len)

Returns the CRC-16 checksum of *len* bytes starting at *data.*

The checksum is independent of the byte order (endianness).

UINT16 qchecksum (const char * data, uint len)

Please use qChecksum; this function is provided only for upward compatibility.

QDataStream & operator<< (QDataStream & s, const QByteArray & a)

Writes a byte array to a stream and returns a reference to the stream.

The serialization format is the byte array size (`Q_UINT32)` followed by this number of bytes.

QDataStream & operator>> (QDataStream & s, QByteArray & a)

Reads a byte array from a stream and returns a reference to the stream.

27.3.1. URL

Generated from http://www.troll.no/qt/qbytearray.html.

28. QCache Class Reference

```
#include <qcache.h>
```

Inherits QGCache.

28.1. Detailed Description

The QCache class is a template class that provides a cache based on `char*` keys.

QCache is implemented as both a template and a macro class. Define a template instance QCache<X> to create a cache that operates on pointers to X, or X*.

A cache is a least recently used (LRU) list of cache items. Each cache item has a cost. The sum of item costs, totalCost(), will not exceed the maximum cache cost, maxCost(). If inserting a new item would cause the total cost to exceed the maximum cost, the least recently used items in the cache are deleted.

Apart from insert, by far the most important function is find() (which also exists as operatior[]). This function looks up an item, returns it, and by default marks it as being the least recently used item.

There are also methods to remove() an object from the cache and delete it, and to take() an object out of the cache without deleting it.

The keys for the cache can be case-sensitive or not, or deeply copied or not. The constructor lets you decide.

QCache, as it must, takes over ownership of the items you insert. That implies that it will delete the items at its discretion, and that you must *not* delete them. There is one exception, however: If the cost of an object you try to insert() is higher than the allowed total cost of the cache, insert() returns FALSE and in this case you must delete the object yourself.

In addition to totalCost() and maxCost()/setMaxCost(), there are access functions to obtain the number of items in the cache (count()), the size of the hash array (size(), set using the constructor) and whether the cache is empty (you guessed it: isEmpty()). Finally, there is a statistics(), which prints various cache statistics on `stdout.`

There is also a QCacheIterator which may be used to traverse the items in the cache in arbritrary order.

See also: QCacheIterator, QIntCache and Collection Classes

28.2. Member Function Documentation

QCache::QCache (const QCache<type> & c)

For internal use only.

QCache::QCache (int maxCost=100, int size=17, bool cs=TRUE, bool ck=TRUE)

Constructs an empty cache with maximum allowed total cost *maxCost,* optimized to contain a bit less than *size* items, with case-sensitive keys if *cs* is TRUE (the default) and case-insensitive keys if *cs* is FALSE, and whose keys are deeply copied on insertion (the default) and shallowly copied of *ck* is FALSE.

QCache::˜QCache ()

Destroys the object and all objects in it.

void QCache::clear () `[virtual]`

Removes all items from the cache, and deletes them.

All cache iterators that operate this on cache are reset.

See also: remove() and take().

Reimplemented from QCollection.

uint QCache::count () const `[virtual]`

Returns the number of items in the cache.

See also: totalCost().

Reimplemented from QCollection.

type * QCache::find (const char * k, bool ref=TRUE) const

Returns the item associated with \ k, or null if the key does not exist in the cache. If *ref* is TRUE (the default), the item is moved to the front of the LRU list.

If there are two or more items with equal keys, then the one that was inserted last is returned.

bool QCache::insert (const char * k, const type * d, int c=1, int p=0)

Inserts the item *d* into the cache with key *k,* cost `and` priority *p,* and returns TRUE if it is successful and FALSE if it fails.

The cache's size is limited, and if the total cost is too high, QCache will delete some of the least-used objects.

Warning: If this function returns FALSE, you must delete *d* yourself. Additionally, be very careful about using *d* after calling this function, as any other insertions into the cache, from anywhere in the application, or within Qt itself, could cause the object to be discarded from the cache, and the pointer to become invalid.

bool QCache::isEmpty () const

Returns TRUE if the cache is empty, or TRUE if there is at least one object in it.

int QCache::maxCost () const

Returns the maximum allowed total cost of the cache.

See also: setMaxCost() and totalCost().

QCache<type>& QCache::operator= (const QCache<type> & c)

For internal use only.

type * QCache::operator[] (const char * k) const

Returns the item associated with *k,* or null if *k* does not exist in the cache, and moves the item to the front of the LRU list.

If there are two or more items with equal keys, then the one that was inserted last is returned.

This is the same as find(k, TRUE).

See also: find().

bool QCache::remove (const char * k)

Deletes the item associated with *k,* and returns TRUE if the item was present in the cache or FALSE if it was not.

If there are two or more items with equal keys, then the one that was inserted last is is deleted.

All iterators that refer to the removed item are set to point to the next item in the cache's traversal order.

See also: take() and clear().

void QCache::setMaxCost (int m)

Sets the maximum allowed total cost of the cache to *m.* If the current total cost is above *m,* some items are deleted immediately.

See also: maxCost() and totalCost().

uint QCache::size () const

Returns the size of the hash array used to implement the cache. This should be a bit bigger than count() is likely to be.

type * QCache::take (const char * k)

Takes the item associated with *k* out of the cache without deleting it, and returns a pointer to the item taken out, or null if the key does not exist in the cache.

If there are two or more items with equal keys, then the one that was inserted last is taken.

All iterators that refer to the taken item are set to point to the next item in the cache's traversal order.

See also: remove() and clear().

int QCache::totalCost () const

Returns the total cost of the items in the cache. This is an integer in the range 0 to maxCost().

See also: setMaxCost().

28.2.1. URL

Generated from http://www.troll.no/qt/qcache.html.

29. QCacheIterator Class Reference

`#include <qcache.h>`

Inherits QGCacheIterator.

29.1. Detailed Description

The QCacheIterator class provides an iterator for QCache collections.

Note that the traversal order is arbitrary, you are not guaranteed any particular order. If new objects are inserted into the cache while the iterator is active, the iterator may or may not see them.

Multiple iterators are completely independent, even when they operate on the same QCache. QCache updates all iterators that refer an item when that itam is removed.

QCacheIterator provides an operator++(), and an operator+=() to traverse the cache, current() and currentKey() to access the current cache item and its key, atFirst() atLast() which return TRUE if the iterator points to the first/last item in the cache, isEmpty() which returns TRUE if the cache is empty and finally count() which returns the number of items in the cache.

Note that atFirst() and atLast() refer to the iterator's arbitrary ordering, not to the cache's internal LRU list.

See also: QCache.

29.2. Member Function Documentation

QCacheIterator::QCacheIterator (const QCache<type> & cache)

Constructs an iterator for *cache*. The current iterator item is set to point on the first item in the *cache* (or rather, the first item is defined to be the item at which this constructor sets the iterator to point).

QCacheIterator::QCacheIterator (const QCacheIterator<type> & ci)

Constructs an iterator for the same cache as *ci*. The new iterator starts at the same item as ci.current(), but moves independently thereafter.

QCacheIterator::operator type * () const

Cast operator. Returns a pointer to the current iterator item. Same as current().

uint QCacheIterator::count () const

Returns the number of items in the cache on whichthis iterator operates.

See also: isEmpty().

type * QCacheIterator::current () const

Returns a pointer to the current iterator item.

const char * QCacheIterator::currentKey () const

Returns a pointer to the key for the current iterator item.

bool QCacheIterator::isEmpty () const

Returns TRUE if the cache is empty, i.e. count() == 0, otherwise FALSE.

See also: count().

type * QCacheIterator::operator() ()

Makes the succeeding item current and returns the original current item.

If the current iterator item was the last item in the cache or if it was null, null is returned.

type * QCacheIterator::operator++ ()

Prefix ++ makes the iterator point to the item just after *current()*, and makes that the new current item for the iterator. If current() was the last item, operator--() returns 0.

type * QCacheIterator::operator+= (uint jump)

Returns the item *jump* positions after the current item, or null if it is beyond the last item. Makes this the current item.

type * QCacheIterator::operator-- ()

Prefix -- makes the iterator point to the item just before *current()*, and makes that the new current item for the iterator. If current() was the first item, operator--() returns 0.

type * QCacheIterator::operator-= (uint jump)

Returns the item *jump* positions before the current item, or null if it is beyond the first item. Makes this the current item.

QCacheIterator<type>& QCacheIterator::operator= (const QCacheIterator<type> & ci)

Makes this an iterator for the same cache as *ci*. The new iterator starts at the same item as ci.current(), but moves independently thereafter.

type * QCacheIterator::toFirst ()

Sets the iterator to point to the first item in the cache and returns a pointer to the item.

Sets the iterator to null and returns null if if the cache is empty.

See also: toLast() and isEmpty().

type * QCacheIterator::toLast ()

Sets the iterator to point to the last item in the cache and returns a pointer to the item.

Sets the iterator to null and returns null if if the cache is empty.

See also: toLast() and isEmpty().

29.2.1. URL

Generated from http://www.troll.no/qt/qcacheiterator.html.

30. QCheckBox Class Reference

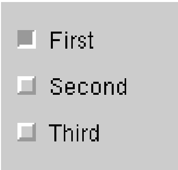
```
#include <qcheckbox.h>
```

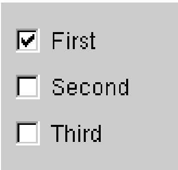
Inherits QButton.

30.1. Detailed Description

The QCheckBox widget provides a check box with a text label.

QCheckBox and QRadioButton are both toggle buttons, but a check box represents an independent switch that can be on (checked) or off (unchecked).





See also: QButton, QRadioButton and Fowler: Check Box.

Examples:
 xform/xform.cpp
 widgets/widgets.cpp

30.2. Member Function Documentation

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

Constructs a check box with no text.

The *parent* and *name* arguments are sent to the QWidget constructor.

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

Constructs a check box with a text.

The *parent* and *name* arguments are sent to the QWidget constructor.

void QCheckBox::drawButton (QPainter * paint) `[virtual protected]`

Draws the check box. Calls drawButtonLabel() to draw the content.

See also: drawButtonLabel().

Reimplemented from QButton.

void QCheckBox::drawButtonLabel (QPainter * p) `[virtual protected]`

Draws the check box label.

See also: drawButton().

Reimplemented from QButton.

bool QCheckBox::isChecked () const

Returns TRUE if the check box is checked, or FALSE if it is not checked.

See also: setChecked().

Examples:
xform/xform.cpp

void QCheckBox::setChecked (bool check)

Checks the check box if *check* is TRUE, or unchecks it if *check* is FALSE.

See also: isChecked().

Examples:
xform/xform.cpp

QSize QCheckBox::sizeHint () const `[virtual]`

Returns a size which fits the contents of the check box.

Reimplemented from QWidget.

30.2.1. URL

Generated from http://www.troll.no/qt/qcheckbox.html.

31. QCheckListItem Class Reference

`#include <qlistview.h>`

Inherits QListViewItem.

31.1. Detailed Description

The QCheckListItem class implements checkable list view items.

There are three types of check list items: CheckBox, RadioButton and Controller.

Checkboxes may be inserted at top level in the list view. A radio button must be child of a controller.

31.2. Member Function Documentation

QCheckListItem::QCheckListItem (QListView * parent, const char * text, Type tt = Controller)

Constructs a checkable item with parent *parent,* text *text* and type *tt.* Note that *tt* must not be RadioButton, if so it will not toggle.

QCheckListItem::QCheckListItem (QListView * parent, const char * text, const QPixmap & p)

Constructs a Controller item with parent *parent,* text *text* and pixmap *p.*

QCheckListItem::QCheckListItem (QListViewItem * parent, const char * text, const QPixmap & p)

Constructs a Controller item with parent *parent,* text *text* and pixmap *p.*

QCheckListItem::QCheckListItem (QCheckListItem * parent, const char * text, Type tt = Controller)

Constructs a checkable item with parent *parent,* text *text* and type *tt.* Note that a RadioButton must be child of a Controller, otherwise it will not toggle.

void QCheckListItem::activate () `[virtual protected]`

Toggle checkbox, or set radiobutton on.

Reimplemented from QListViewItem.

bool QCheckListItem::isOn () const

Returns TRUE if this item is toggled on, FALSE otherwise.

void QCheckListItem::paintBranches (QPainter * p, const QColorGroup & cg, int w, int, int h, GUIStyle) `[virtual protected]`

Fills the rectangle. No decoration is drawn.

Reimplemented from QListViewItem.

void QCheckListItem::paintCell (QPainter * p, const QColorGroup & cg, int column, int width, int align) `[virtual]`

Paints this item.

Reimplemented from QListViewItem.

void QCheckListItem::setOn (bool b)

Sets this button on of *b* is TRUE, off otherwise. Maintains radiobutton exclusivity.

void QCheckListItem::setup () `[virtual]`

Performs setup.

Reimplemented from QListViewItem.

void QCheckListItem::stateChange (bool) `[virtual protected]`

This virtual function is called when the item changes its on/off state.

const char * QCheckListItem::text () const

Returns the text of this item.

const char * QCheckListItem::text (int n) const `[virtual]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QListViewItem.

void QCheckListItem::turnOffChild () `[protected]`

If this is a Controller that has RadioButton children, turn off the child that is on.

QCheckListItem::Type QCheckListItem::type() const

Returns the type of this item.

31.2.1. URL

Generated from http://www.troll.no/qt/qchecklistitem.html.

32. QChildEvent Class Reference

```
#include <qevent.h>
```

Inherits QEvent.

32.1. Detailed Description

The QChildEvent class contains event parameters for child widget events.

Child events are sent to widgets when children are inserted or removed.

In this release of Qt, no event handlers are defined to receive child events. Reimplement QObject::event() or install an QObject::eventFilter() to handle these events.

32.2. Member Function Documentation

QChildEvent::QChildEvent (int type, QWidget * child)

Constructs a child event object.

The *type* parameter must be either *Event_ChildInserted* or *Event_ChildRemoved.*

QWidget * QChildEvent::child () const

Returns the child widget inserted or removed.

bool QChildEvent::inserted () const

Returns TRUE if the widget received a new child.

bool QChildEvent::removed () const

Returns TRUE if the widget lost a child.

32.2.1. URL

Generated from http://www.troll.no/qt/qchildevent.html.

33. QClipboard Class Reference

`#include <qclipboard.h>`

Inherits QObject.

33.1. Detailed Description

The QClipboard class provides access to the window system clipboard.

The clipboard offers a simple mechanism to copy and paste data between applications.

QClipboard supports these formats (a format is identified by a string):

- "TEXT", zero-terminated char *.
- "PIXMAP" as provided by QPixmap.

The "PIXMAP" format is not implemented in this version of Qt.

Only a single QClipboard object may exist in an application. This is because QClipboard is a shared window system resource. It is not possible to create a QClipboard object the standard C++ way (the constructor and destructor are private member functions, but accessible to QApplication since it is a friend class). Call QApplication::clipboard() to access the clipboard.

Example:

```
QClipboard *cb = QApplication::clipboard();
const char *text;
// Copy text from the clipboard (paste)
text = cb->text();
if ( text )
    debug( "The clipboard contains: %s", text );
// Copy text into the clipboard
cb->setText( "This text can be pasted by other programs" );
```

33.2. Member Function Documentation

void QClipboard::clear ()

Clears the clipboard contents.

void QClipboard::connectNotify (const char *) `[virtual protected]`

For internal use only.

Reimplemented from QObject.

void * QClipboard::data (const char * format) const

Returns a pointer to the clipboard data, where *format* is the clipboard format.

We recommend that you use text() or pixmap() instead.

void QClipboard::dataChanged () `[signal]`

This signal is emitted when the clipboard data is changed.

bool QClipboard::event (QEvent * e) `[virtual protected]`

Handles clipboard events (very platform-specific).

Reimplemented from QObject.

QPixmap * QClipboard::pixmap () const

Returns the clipboard pixmap, or null if the clipboard does not contains any pixmap.

See also: setText().

void QClipboard::setData (const char * format, void * data)

Copies text into the clipboard, where *format* is the clipboard format string and *data* is the data to be copied.

We recommend that you use setText() or setPixmap() instead.

void QClipboard::setPixmap (const QPixmap & pixmap)

Copies *pixmap* into the clipboard.

See also: pixmap().

void QClipboard::setText (const char * text)

Copies *text* into the clipboard.

See also: text().

const char * QClipboard::text () const

Returns the clipboard text, or null if the clipboard does not contains any text.

See also: setText().

33.2.1. URL

Generated from http://www.troll.no/qt/qclipboard.html.

34. QCloseEvent Class Reference

```
#include <qevent.h>
```

Inherits QEvent.

34.1. Detailed Description

The QCloseEvent class contains parameters that describe a close event.

Close events are sent to widgets that the user wants to close, usually by choosing "Close" from the window menu. They are also sent when you call QWidget::close() to close a widget from inside the program.

Close events contain a special accept flag which tells whether the receiver wants the widget to be closed. When a widget accepts the close event, it is hidden. If it refuses to accept the close event, nothing happens.

The main widget of the application is a special case. When it accepts the close event, the application is immediately terminated.

The event handler QWidget::closeEvent() receives close events.

The default implementation of this event handler accepts the close event. This makes Qt hide the widget.

```
void QWidget::closeEvent( QCloseEvent *e )
{
    e->accept();                            // hides the widget
}
```

If you do not want your widget to be hidden, you should reimplement the event handler.

```
void MyWidget::closeEvent( QCloseEvent *e )
{
    e->ignore();                            // does not hide the widget
}
```

If you want your widget to be deleted when it is closed, simply delete it in the close event. In this case, calling QCloseEvent::accept() or QCloseEvent::ignore() makes no difference.

Warning: Be careful. The code below assumes that the widget was created on the heap using the `new` operator. Even when the widget has been created by new doing this is a tricky opreation. Be sure that you cannot have any other pointers to the widget hanging around.

```
void MyWidget::closeEvent( QCloseEvent * )
{
    delete this;
}
```

QObject emits the destroyed() signal when it is deleted. This is a useful signal if a widget needs to know when another widget is deleted.

See also: QWidget::close(), QWidget::hide(), QObject::destroyed(), QApplication::setMainWidget() and QApplication::quit().

34.2. Member Function Documentation

QCloseEvent::QCloseEvent ()

Constructs a close event object with the accept parameter flag set to FALSE.

void QCloseEvent::accept ()

Sets the accep flag of the close event object.

Setting the accept flag indicates that the receiver of this event agrees to close the widget.

The accept flag is not set by default.

If you choose to accept in QWidget::closeEvent(), the widget will be hidden.

See also: ignore() and QWidget::hide().

void QCloseEvent::ignore ()

Clears the accept flag of the close event object.

Clearing the accept flag indicates that the receiver of this event does not want the widget to be hidden.

The accept flag is not set by default.

See also: accept().

bool QCloseEvent::isAccepted () const

Returns TRUE if the receiver of the event has agreed to close the widget.

See also: accept() and ignore().

34.2.1. URL

Generated from http://www.troll.no/qt/qcloseevent.html.

35. QCollection Class Reference

`#include <qcollection.h>`

Inherited by QGCache, QGDict and QGList.

35.1. Detailed Description

The QCollection class is the base class of all Qt collections.

The QCollection class is an abstract base class for the Qt collection classes QDict, QList etc. via QGDict, QGList etc.

A QCollection knows only about the number of objects in the collection and the deletion strategy.

A collection is implemented using the `GCI` (generic collection item) type, which is a `void*`. The template (or macro) classes that create the real collections cast the `GCI` to the required type.

See also: Collection Classes

35.2. Member Function Documentation

QCollection::QCollection () `[protected]`

Constructs a collection. The constructor is protected because QCollection is an abstract class.

QCollection::QCollection (const QCollection & source) `[protected]`

Constructs a copy of *source* with autoDelete() set to FALSE. The constructor is protected because QCollection is an abstract class.

Note that if *source* has autoDelete turned on, copying it is a good way to get memory leaks, reading freed memory, or both.

QCollection::~QCollection () `[virtual protected]`

Destroys the collection. The destructor is protected because QCollection is an abstract class.

bool QCollection::autoDelete () const

Returns the setting of the auto-delete option (default is FALSE).

See also: setAutoDelete().

virtual void QCollection::clear () `[virtual]`

Removes all objects from the collection. The objects will be deleted if auto-delete has been enabled.

See also: setAutoDelete().

Reimplemented in QPtrDict, QIntCache, QGCache, QDict, QIntDict, QGDict, QList, QCache and QGList.

virtual uint QCollection::count () const `[virtual]`

Returns the number of objects in the collection.

Reimplemented in QPtrDict, QIntCache, QGCache, QDict, QIntDict, QGDict, QList, QCache and QGList.

void QCollection::deleteItem (GCI d) `[virtual protected]`

Virtual function that deletes an item that is about to be removed from the collection.

The default implementation deletes *d* pointer if and only if auto-delete has been enabled.

This function is always reimplemented in the collection template classes.

See also: newItem() and setAutoDelete().

GCI QCollection::newItem (GCI d) `[virtual protected]`

Virtual function that creates a copy of an object that is about to be inserted into the collection.

The default implementation returns the *d* pointer, i.e. no copy is made.

This function is seldom reimplemented in the collection template classes. It is not common practice to make a copy of something that is being inserted.

See also: deleteItem().

void QCollection::setAutoDelete (bool enable)

Sets the auto-delete option of the collection.

Enabling auto-delete (*enable* is TRUE) will delete objects that are removed from the collection. This can be useful if the collection has the only reference to the objects. (Note that the object can still be copied using the copy constructor - copying such objects is a good way to get memory leaks, reading freed memory or both.)

Disabling auto-delete (*enable* is FALSE) will *not* delete objects that are removed from the collection. This is useful if the objects are part of many collections.

The default setting is FALSE.

See also: autoDelete().

Examples:
grapher/grapher.cpp

35.2.1. URL

Generated from http://www.troll.no/qt/qcollection.html.

36. QColor Class Reference

`#include <qcolor.h>`

36.1. Detailed Description

The QColor class provides colors based on RGB.

A color is normally specified in terms of RGB (red,green and blue) components, but it is also possible to specify HSV (hue,saturation and value) or set a color name (the names are copied from from the X11 color database).

In addition to the RGB value, a QColor also has a pixel value. This value is used by the underlying window system to refer to a color. It can be thought of as an index into the display hardware's color table.

There are 19 predefined global QColor objects: `black, white, darkGray, gray, lightGray, red, green, blue, cyan, magenta, yellow, darkRed, darkGreen, darkBlue, darkCyan, darkMagenta, darkYellow, color0` and `color1.`

The colors `color0` (zero pixel value) and `color1` (non-zero pixel value) are special colors for drawing in bitmaps.

The QColor class has an efficient, dynamic color allocation strategy. A color is normally allocated the first time it is used (lazy allocation), that is, whenever the pixel() function is called:

1 Is the pixel value valid? If it is, just return it, otherwise, allocate a pixel value.

2 Check an internal hash table to see if we allocated an equal RGB value earlier. If we did, set the pixel value and return.

3 Try to allocate the RGB value. If we succeed, we get a pixel value which we save in the internal table with the RGB value. Return the pixel value.

4 The color could not be allocated. Find the closest matching color and save it in the internal table.

See also: QPalette, QColorGroup and QApplication::setColorSpec().

Examples:
grapher/grapher.cpp
forever/forever.cpp
desktop/desktop.cpp
connect/connect.cpp
drawdemo/drawdemo.cpp
hello/hello.cpp
picture/picture.cpp
xform/xform.cpp
progress/progress.cpp
widgets/widgets.cpp

36.2. Member Function Documentation

QColor::QColor ()

Constructs an invalid color with the RGB value (0,0,0). An invalid color is a color that is not properly set up for the underlying window system.

See also: isValid().

QColor::QColor (QRgb rgb, uint pixel=0xffffffff)

Constructs a color with a RGB value and a custom pixel value.

If the *pixel* = 0xffffffff, then the color uses the RGB value in a standard way. If *pixel* is something else, then the pixel value will be set directly to *pixel* (skips the normal allocation procedure).

QColor::QColor (const QColor & c)

Constructs a color that is a copy of *c*.

QColor::QColor (const char * name)

Constructs a named color, i.e. loads the color from the color database.

See also: setNamedColor().

QColor::QColor (int r, int g, int b)

Constructs a color with the RGB value *(r,g,b). r, g* and *b* must be in the range 0..255.

See also: setRgb().

QColor::QColor (int x, int y, int z, Spec colorSpec)

Constructs a color with the RGB *or* HSV value *(x,y,z).*

The *(x,y,z)* triplet defines an RGB value if *colorSpec* == `QColor::Rgb.` *x* (red), *y* (green) and *z* (blue) must be in the range 0..255.

The *(x,y,z)* triplet defines a HSV value if *colorSpec* == `QColor::Hsv.` *x* (hue) must be in the range -1..360 (-1 means achromatic), and *y* (saturation) and *z* (value) must be in the range 0..255.

See also: setRgb() and setHsv().

uint QColor::alloc ()

Allocates the RGB color and returns the pixel value.

Allocating a color means to obtain a pixel value from the RGB specification. The pixel value is an index into the global color table.

Calling the pixel() function will allocate automatically if the color was not already allocated.

See also: setLazyAlloc() and enterAllocContext().

int QColor::blue () const

Returns the blue component of the RGB value.

void QColor::cleanup () `[static]`

Internal clean up required for QColor. This function is called from the QApplication destructor.

See also: initialize().

int QColor::currentAllocContext () `[static]`

Returns the current color allocation context.

The default context is 0.

See also: enterAllocContext() and leaveAllocContext().

QColor QColor::dark (int factor = 200) const

Returns a darker (or lighter) color.

Returns a darker color if *factor* is greater than 100. Setting *factor* to 300 returns a color that has one third the brightness.

Returns a lighter color if *factor* is less than 100, equal to light(10000 / *factor).*

This function converts the current RGB color to HSV, divides V by *factor* and converts back to RGB.

See also: light().

Examples:
desktop/desktop.cpp
xform/xform.cpp

void QColor::destroyAllocContext (int context) `[static]`

Destroys a color allocation context, *context.*

This function deallocates all colors that were allocated in the specified *context.* If *context* == -1, it frees up all colors that the application has allocated. If *context* == -2, it frees up all colors that the application has allocated, except those in the default context.

The function does nothing for true color displays.

See also: enterAllocContext() and alloc().

Examples:
showimg/showimg.cpp

int QColor::enterAllocContext () `[static]`

Enters a color allocation context and returns a nonzero unique identifier.

Color allocation contexts are useful for programs that need to allocate many colors and throw them away later, like image viewers. The allocation context functions work for true color displays as well as colormap display, except that QColor::destroyAllocContext() does nothing for true color.

Example:

```
QPixmap loadPixmap( const char *fileName )
{
    static int alloc_context = 0;
    if ( alloc_context )
        QColor::destroyAllocContext( alloc_context );
    alloc_context = QColor::enterAllocContext();
    QPixmap pm( fileName );
    QColor::leaveAllocContext();
    return pm;
}
```

The example code loads a pixmap from file. It frees up all colors that were allocated the last time loadPixmap() was called.

The initial/default context is 0. Qt keeps a list of colors associated with their allocation contexts. You can call destroyAllocContext() to get rid of all colors that were allocated in a specific context.

Calling enterAllocContext() enters an allocation context. The allocation context lasts until you call leaveAllocContext(). QColor has an internal stack of allocation contexts. Each call to enterAllocContex() must have a corresponding leaveAllocContext().

```
  // context 0 active
int c1 = QColor::enterAllocContext();       // enter context c1
  // context c1 active
int c2 = QColor::enterAllocContext();       // enter context c2
  // context c2 active
QColor::leaveAllocContext();                // leave context c2
  // context c1 active
QColor::leaveAllocContext();                // leave context c1
  // context 0 active
  // Now, free all colors that were allocated in context c2
QColor::destroyAllocContext( c2 );
```

You may also want to set the application's color specification. See QApplication::setColorSpec() for more information.

See also: leaveAllocContext(), currentAllocContext(), destroyAllocContext() and QApplication::setColorSpec().

int QColor::green () const

Returns the green component of the RGB value.

void QColor::hsv (int * h, int * s, int * v) const

Returns the current RGB value as HSV.

Arguments:

- **h,* hue.
- **s,* saturation.
- **v,* value. The hue defines the color. Its range is 0..359 if the color is chromatic and -1 if the color is achromatic. The saturation and value both vary between 0 and 255 inclusive.

See also: setHsv() and rgb().

void QColor::initialize () `[static]`

Internal initialization required for QColor. This function is called from the QApplication constructor.

See also: cleanup().

bool QColor::isDirty () const

Returns TRUE if the color is dirty, i.e. lazy allocation is enabled and an RGB/HSV value has been set but not allocated.

See also: setLazyAlloc(), alloc() and pixel().

bool QColor::isValid () const

Returns TRUE if the color is invalid, i.e. it was constructed using the default constructor.

bool QColor::lazyAlloc () `[static]`

Returns TRUE if lazy color allocation is enabled (on-demand allocation), or FALSE if it is disabled (immediate allocation).

See also: setLazyAlloc().

void QColor::leaveAllocContext () `[static]`

Leaves a color allocation context.

See enterAllocContext() for a detailed explanation.

See also: enterAllocContext() and currentAllocContext().

Examples:
showimg/showimg.cpp

QColor QColor::light (int factor = 150) const

Returns a lighter (or darker) color.

Returns a lighter color if *factor* is greater than 100. Setting *factor* to 150 returns a color that is 50% brighter.

Returns a darker color if *factor* is less than 100, equal to dark(10000 / *factor)*.

This function converts the current RGB color to HSV, multiplies V with *factor* and converts back to RGB.

See also: dark().

Examples:
desktop/desktop.cpp
xform/xform.cpp

int QColor::maxColors () `[static]`

Returns the maximum number of colors supported by the underlying window system.

int QColor::numBitPlanes () `[static]`

Returns the number of color bit planes for the underlying window system.

The returned values is equal to the default pixmap depth;

See also: QPixmap::defaultDepth().

bool QColor::operator!= (const QColor & c) const

Returns TRUE if this color has different RGB value from *c,* or FALSE if they have equal RGB values.

QColor & QColor::operator= (const QColor & c)

Assigns a copy of the color `and` returns a reference to this color.

bool QColor::operator== (const QColor & c) const

Returns TRUE if this color has the same RGB value as *c,* or FALSE if they have different RGB values.

uint QColor::pixel () const

Returns the pixel value.

This value is used by the underlying window system to refer to a color. It can be thought of as an index into the display hardware's color table.

See also: setLazyAlloc() and alloc().

int QColor::red () const

Returns the red component of the RGB value.

QRgb QColor::rgb () const

Returns the RGB value.

Bits 0-7 = red, bits 8-15 = green, bits 16-23 = blue.

The return type *QRgb* is equivalent to `unsigned int.`

See also: setRgb() and hsv().

void QColor::rgb (int * r, int * g, int * b) const

Returns the red, green and blue components of the RGB value in **r,* **g* and **b.* The value range for a component is 0..255.

See also: setRgb() and hsv().

void QColor::setHsv (int h, int s, int v)

Sets a HSV color value.

Arguments:

- *h,* hue (-1,0..360). -1 means achromatic.
- *s,* saturation (0..255).
- *v,* value (0..255).

See also: hsv() and setRgb().

Examples:

grapher/grapher.cpp
drawdemo/drawdemo.cpp
progress/progress.cpp

void QColor::setLazyAlloc (bool enable) `[static]`

Enables or disables lazy color allocation.

If lazy allocation is enabled, colors are allocated the first time they are used (upon calling the pixel() function). If lazy allocation is disabled, colors are allocated when they are constructed or when either setRgb() or setHsv() is called.

Lazy color allocation is enabled by default.

See also: lazyAlloc(), pixel() and alloc().

void QColor::setNamedColor (const char * name)

Sets the RGB value to that of the named color.

The "#RRGGBB" color format is supported on all platforms, where RR, GG, and BB and hex digits.

A color name may also be a color in the X color database (rgb.txt) , e.g. "steelblue" or "gainsboro". The X color names also work under Qt for Windows.

void QColor::setRgb (QRgb rgb)

Sets the RGB value to *rgb.*

Bits 0-7 = red, bits 8-15 = green, bits 16-23 = blue.

Warning: The bit encoding may change in a future version of Qt. Please use the qRgb() function to compose RGB triplets.

The type *QRgb* is equivalent to `unsigned int.`

See also: rgb() and setHsv().

void QColor::setRgb (int r, int g, int b)

Sets the RGB value to *(r,g,b). r, g* and *b* must be in the range 0..255.

See also: rgb() and setHsv().

Examples:
widgets/widgets.cpp

36.3. Related Functions

QDataStream & operator<< (QDataStream & s, const QColor & c)

Writes a color object to the stream.

Serialization format: RGB value serialized as an UINT32.

QDataStream & operator>> (QDataStream & s, QColor & c)

Reads a color object from the stream.

int qRed (QRgb rgb)

Returns the red component of the RGB triplet *rgb.*

See also: qRgb() and QColor::red().

Examples:
qmag/qmag.cpp

int qGreen (QRgb rgb)

Returns the green component of the RGB triplet *rgb.*

See also: qRgb() and QColor::green().

Examples:
qmag/qmag.cpp

int qBlue (QRgb rgb)

Returns the blue component of the RGB triplet *rgb*.

See also: qRgb() and QColor::blue().

Examples:
qmag/qmag.cpp

QRgb qRgb (int r, int g, int b)

Returns the RGB triplet *(r,g,b)*.

Bits 0-7 = *r* (red), bits 8-15 = *g* (green), bits 16-23 = *b* (blue).

Warning: The bit encoding may change in a future version of Qt.

The return type *QRgb* is equivalent to `unsigned int.`

See also: qRed(), qGreen() and qBlue().

Examples:
desktop/desktop.cpp

int qGray (int r, int g, int b)

Returns a gray value 0..255 from the *(r,g,b)* triplet.

The gray value is calculated using the formula: `(r*11 + g*16 + b*5)/32`

int qGray (qRgb rgb)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

36.3.1. URL

Generated from http://www.troll.no/qt/qcolor.html.

37. QColorGroup Class Reference

```
#include <qpalette.h>
```

37.1. Detailed Description

The QColorGroup class contains a group of widget colors.

A color group contains a group of colors used by widgets for drawing themselves. Widgets should not use colors like "red" and "turqoise" but rather "foreground" and "base", where possible.

We have identified eight distinct color roles:

1. Foreground (graphics foreground color)
2. Background (general background color)
3. Light (lighter than background color, for shadow effects)
4. Midlight (between Background and Light, for shadow effects)
5. Dark (darker than the background color, for shadow effects)
6. Medium (between background and dark, used for shadow and contrast effects)
7. Text (usually the same as the foreground color, but sometimes text and other foreground are not the same)
8. Base (used as background color for some widgets). Usually white or another light color.

We have not seen any good, well-made and usable widgets that use more than these eight color roles.

A QPalette contains three color groups.

The current widget color group is returned by QWidget::colorGroup().

See also: QColor and QPalette.

Examples:

xform/xform.cpp

37.2. Member Function Documentation

QColorGroup::QColorGroup ()

Constructs a color group with all colors set to black.

QColorGroup::QColorGroup (const QColor & foreground, const QColor & background, const QColor & light, const QColor & dark, const QColor & mid, const QColor & text, const QColor & base)

Constructs a color group with the specified colors.

QColorGroup::~QColorGroup ()

Destroys the color group.

const QColor & QColorGroup::background () const

Returns the background color of the color group.

const QColor & QColorGroup::base () const

Returns the base color of the color group.

const QColor & QColorGroup::dark () const

Returns the dark color of the color group.

const QColor & QColorGroup::foreground () const

Returns the foreground color of the color group.

const QColor & QColorGroup::light () const

Returns the light color of the color group.

const QColor & QColorGroup::mid () const

Returns the medium color of the color group.

QColor QColorGroup::midlight () const

Returns the midlight color of the color group. Currently, this is a lightened version of the background, but this may change in the future, to return a `const QColor &` from the palette.

bool QColorGroup::operator!= (const QColorGroup & g) const

Returns TRUE if this color group is different from *g,* or FALSE if it is equal to *g.*

See also: operator!=().

bool QColorGroup::operator== (const QColorGroup & g) const

Returns TRUE if this color group is equal to *g,* or FALSE if it is different from *g.*

See also: operator==().

const QColor & QColorGroup::text () const

Returns the text foreground color of the color group.

37.3. Related Functions

QDataStream & operator<< (QDataStream & s, const QColorGroup & g)

Writes a color group to the stream.

Serialization format:

1 QColor foreground

2 QColor background

3 QColor light
4 QColor dark
5 QColor mid
6 QColor text
7 QColor base The colors are serialized in the listed order.

QDataStream & operator>> (QDataStream & s, QColorGroup & g)

Reads a color group from the stream.

37.3.1. URL

Generated from http://www.troll.no/qt/qcolorgroup.html.

38. QComboBox Class Reference

`#include <qcombobox.h>`

Inherits QWidget.

38.1. Detailed Description

The QComboBox widget is a combined button and popup list.

A combo box may be defined as a selection widget which displays the current selection, and which can pop up a list of possible selections. Some combo boxes also allow the user to select arbitrary strings, using a line editor.

Since combo boxes occupy little screen space and always display the current selection, they are very well suited to displaying and selecting modes (such as font family and size): The user can always see what mode he/she is in, and the majority of the screen space is available for real work.

QComboBox supports three different appearances: Motif 1.x, Motif 2.0 and Windows 95. In Motif 1.x, a combo box was called XmOptionMenu. In Motif 2.0, OSF introduced an improved combo box and named that XmComboBox. QComboBox provides both.

QComboBox provides two different constructors. The simplest one creates an old-style combo box in Motif style:

```
QComboBox * c = new QCombBox( this, "read-only combo" );
```

The other one creates a new-style combo box in Motif style, and can create both read-only and read-write combo boxes:

```
QComboBox * c1 = new QCombBox( FALSE, this, "read-only combo" );
QComboBox * c2 = new QCombBox( TRUE, this, "read-write combo" );
```

New-style combo boxes use a list box in both Motif and Windows styles, and both the content size and the on-screen size of the list box can be limited. Old-style combo boxes use a popup in Motif style, and that popup will happily grow larger than the desktop if you put enough data in it.

The two constructors create identical-looking combos in Windows style.

Read-only combo boxes can contain pixmaps as well as texts; the insert() and changeItem() functions are suitably overloaded. If you try to insert a pixmap in a read-write combo box, QComboBox simply ignores you. For read-write combo boxes, the function clearEdit() is provided, to clear the displayed string without changing the combo box' contents.

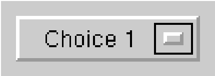
A combo box emits two signals, activated() and highlighted(), when a new item has been activated (selected) or highlighted (set to current). Both signals exist in two versions, one with a `char*` argument and one with an `int` argument. If the user highlights or activates a pixmap, only the `int` signals are emitted.

Read-write combo boxes offer four policies for dealing with typed input:

- `NoInsertion` means to simply emit the activated() signal,
- `AtBottom` means to insert the string at the bottom of the combo box and emit activated(),
- `AtTop` means to insert the string at the top of the combo box and emit activated(), and finally
- `AtCurrent` means to replace the previously selected item with the typed string, and emit activated(). If inserting the typed string would cause the combo box to breach its content size limit, the item at the other end of the list is deleted. The default insertion policy is `AtBottom`, you can change it using setInsertionPolicy().

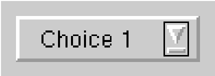
It is possible to constrain the input to an editable combo box using QValidator; see setValidator(). By default, all input is accepted.

A combo box has a default focusPolicy() of `TabFocus,` i.e. it will not grab focus if clicked. This differs from both Windows and Motif.



(Motif 1, read-only)

(Motif 2, read-write)



(Motif 2, read-only)

(Windows style)

See also: QLineEdit, QListBox, QSpinBox, QRadioButton, QButtonGroup, GUI Design Handbook: Combo Box, and GUI Design Handbook: Drop-Down List Box.

Examples:
tictac/tictac.cpp
qmag/qmag.cpp
widgets/widgets.cpp

38.2. Member Function Documentation

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

Constructs a combo box widget with a parent and a name.

This constructor creates a popup menu if the program uses Motif look and feel; this is compatible with Motif 1.x.

QComboBox::QComboBox (bool rw, QWidget * parent=0, const char * name=0)

Constructs a combo box with a maximum size and either Motif 2.0 or Windows look and feel.

The input field can be edited if *rw* is TRUE, otherwise the user may only choose one of the items in the combo box.

QComboBox::~QComboBox ()

Destroys the combo box.

void QComboBox::activated (const char * string) `[signal]`

This signal is emitted when a new item has been activated (selected). *string* is the activated string.

You can also use activated(int) signal, but be aware that its argument meaningful only for selected strings, not for typed strings.

void QComboBox::activated (int index) `[signal]`

This signal is emitted when a new item has been activated (selected). The *index* is the position of the item in the popup list.

bool QComboBox::autoCompletion () const

Returns TRUE if this combo box is in auto-completion mode.

See also: setAutoCompletion().

bool QComboBox::autoResize () const

Returns TRUE if auto-resizing is enabled, or FALSE if auto-resizing is disabled.

Auto-resizing is disabled by default.

See also: setAutoResize().

void QComboBox::changeItem (const QPixmap & im, int index)

Replaces the item at position *index* with a pixmap, unless the combo box is writable.

See also: insertItem().

void QComboBox::changeItem (const char * t, int index)

Replaces the item at position *index* with a text.

void QComboBox::clear ()

Removes all combo box items.

void QComboBox::clearEdit () `[slot]`

Clears the line edit without changing the combo's contents. Does nothing if the combo isn't editable.

This is particularly handy when using a combo box as a line edit with history. For example you can connect the combo's activated() signal to clearEdit() in order to present the user with a new, empty line as soon as return is pressed.

See also: setEditText().

void QComboBox::clearValidator () `[slot]`

This slot is equivalent to setValidator(0).

int QComboBox::count () const

Returns the number of items in the combo box.

int QComboBox::currentItem () const

Returns the index of the current combo box item.

See also: setCurrentItem().

const char * QComboBox::currentText () const

Returns the text item being edited, or the current text item if the combo box is not editable.

See also: text().

bool QComboBox::eventFilter (QObject * object, QEvent * event) `[virtual]`

This event filter is used to manipulate the line editor in magic ways. In Qt 2.0 it will all change, until then binary compatibility lays down the law.

The event filter steals events from the popup or listbox when they are popped up. It makes the popup stay up after a short click in motif style. In windows style it toggles the arrow button of the combo box field, and activates an item and takes down the listbox when the mouse button is released.

Reimplemented from QObject.

void QComboBox::focusInEvent (QFocusEvent *) `[virtual protected]`

Handles focus in events.

Reimplemented from QWidget.

void QComboBox::highlighted (const char * string) `[signal]`

This signal is emitted when a new item has been highlighted. *string* is the highlighted string.

You can also use highlighted(int) signal.

void QComboBox::highlighted (int index) `[signal]`

This signal is emitted when a new item has been set to current. The *index* is the position of the item in the popup list.

void QComboBox::insertItem (const QPixmap & pixmap, int index=-1)

Inserts a pixmap item at position *index.* The item will be appended if *index* is negative.

If the combo box is writable, the pixmap is not inserted.

void QComboBox::insertItem (const char * t, int index=-1)

Inserts a text item at position *index.* The item will be appended if *index* is negative.

Examples:
widgets/widgets.cpp

void QComboBox::insertStrList (const QStrList * list, int index=-1)

Inserts the list of strings at the index *index* in the combo box.

Examples:
qmag/qmag.cpp

void QComboBox::insertStrList (const char ** strings, int numStrings=-1, int index=-1)

Inserts the array of strings at the index *index* in the combo box.

The *numStrings* argument is the number of strings. If *numStrings* is -1 (default), the *strs* array must be terminated with 0.

Example:

```
static const char *items[] = { "red", "green", "blue", 0 };
combo->insertStrList( items );
```

QComboBox::Policy QComboBox::insertionPolicy() const

Returns the current insertion policy of the combo box.

See also: setPolicy().

void QComboBox::keyPressEvent (QKeyEvent * e) `[virtual protected]`

Handles key press events for the combo box.

In Motif style, up and down change the selected item and both enter and return pops up the list. In Windows style, all four arrow keys change the selected item, and Space pops up the list.

Reimplemented from QWidget.

QListBox * QComboBox::listBox () const

Returns the current list box, or 0 if there is no list box currently. (QComboBox can use QPopupMenu instead of QListBox.) Provided to match setListBox().

See also: setListBox().

int QComboBox::maxCount () const

Returns the current maximum size of the combo box. By default, there is no limit, so this function returns INT_MAX.

See also: setMaxCount() and count().

void QComboBox::mouseDoubleClickEvent (QMouseEvent * e) `[virtual protected]`

Handles mouse double click events for the combo box.

Reimplemented from QWidget.

void QComboBox::mouseMoveEvent (QMouseEvent *) `[virtual protected]`

Handles mouse move events for the combo box.

Reimplemented from QWidget.

void QComboBox::mousePressEvent (QMouseEvent * e) `[virtual protected]`

Handles mouse press events for the combo box.

Reimplemented from QWidget.

void QComboBox::mouseReleaseEvent (QMouseEvent *) `[virtual protected]`

Handles mouse release events for the combo box.

Reimplemented from QWidget.

void QComboBox::paintEvent (QPaintEvent * event) `[virtual protected]`

Handles paint events for the combo box.

Reimplemented from QWidget.

const QPixmap * QComboBox::pixmap (int index) const

Returns the pixmap item at a given index, or 0 if the item is not a pixmap.

void QComboBox::popup () `[protected]`

Popups the combo box popup list. If the list is empty, inserts a single, empty, string.

void QComboBox::removeItem (int index)

Removes the item at position *index.*

void QComboBox::resizeEvent (QResizeEvent *) `[virtual protected]`

Handles resize events for the combo box.

Reimplemented from QWidget.

void QComboBox::setAutoCompletion (bool enable)

Sets this combo box to offer auto-completion while the user is editing if *enable* is TRUE, or not to offer auto-completion of *enable* is FALSE.

The combo box uses the list of items as candidates for completion.

See also: autoCompletion() and setEditText().

Examples:
widgets/widgets.cpp

void QComboBox::setAutoResize (bool enable)

Enables auto-resizing if *enable* is TRUE, or disables it if *enable* is FALSE.

When auto-resizing is enabled, the combo box button will resize itself whenever the current combo box item change.

See also: autoResize() and adjustSize().

void QComboBox::setBackgroundColor (const QColor & color) `[virtual]`

Reimplements QWidget::setBackgroundColor().

Sets the background color for both the combo box button and the combo box popup list.

Reimplemented from QWidget.

void QComboBox::setCurrentItem (int index)

Sets the current combo box item. This is the item to be displayed on the combo box button.

See also: currentItem().

Examples:
qmag/qmag.cpp

void QComboBox::setEditText (const char * newText) `[slot]`

Sets the text in the embedded line edit to *newText* without changing the combo's contents. Does nothing if the combo isn't editable.

This is useful e.g. for providing a good starting point for the user's editing and entering the change in the combo only when the user presses enter.

See also: clearEditText() and insertItem().

void QComboBox::setEnabled (bool enable) `[virtual]`

Reimplemented for internal purposes.

Reimplemented from QWidget.

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

Reimplements QWidget::setFont().

Sets the font for both the combo box button and the combo box popup list.

Reimplemented from QWidget.

void QComboBox::setInsertionPolicy (Policy policy)

Sets the insertion policy of the combo box to *policy.*

The insertion policy governs where items typed in by the user are inserted in the list. The possible values are

- `NoInsertion:` Strings typed by the user aren't inserted anywhere
- `AtTop:` Strings typed by the user are inserted above the top item in the list
- AtCurrent: Strings typed by the user replace the last selected item
- AtBottom: Strings typed by the user are inserted at the bottom of the list.

The default insertion policy is `AtBottom.`

See also: policy().

void QComboBox::setListBox (QListBox * newListBox)

Sets the combo box to use *newListBox* instead of the current list box or popup. As a site effect, clears the combo box of its current contents.

Warning: QComboBox assumes that newListBox->text(n) returns non-null for 0 <= n < newListbox->count(). This assumption is necessary becuase of the line edit in QComboBox.

void QComboBox::setMaxCount (int count)

Sets the maximum number of items the combo box can hold to *count.*

If *count* is smaller than the current number of items, the list is truncated at the end. There is no limit by default.

See also: maxCount() and count().

void QComboBox::setPalette (const QPalette & palette) `[virtual]`

Reimplements QWidget::setPalette().

Sets the palette for both the combo box button and the combo box popup list.

Reimplemented from QWidget.

void QComboBox::setSizeLimit (int lines)

Sets the maximum on-screen size of the combo box to *lines.* This is disregarded in Motif 1.x style. The default limit is ten lines.

If the number of items in the combo box is/grows larger than `lines,` a list box is added.

See also: sizeLimit(), count() and setMaxCount().

void QComboBox::setStyle (GUIStyle s) `[virtual]`

Reimplemented for implementational reasons.

Note that QComboBox always turns into a new-style Motif combo box when it is changed from Windows to Motif style (even if it was an old-style combo box before).

Reimplemented from QWidget.

void QComboBox::setValidator (QValidator * v)

Sets this combo box to be editable only as allowed by *v.*

This function does nothing if the combo is not editable.

See also: validator(), clearValidator() and QValidator.

QSize QComboBox::sizeHint () const `[virtual]`

Returns a size which fits the contents of the combo box button.

Examples:
widgets/widgets.cpp

Reimplemented from QWidget.

int QComboBox::sizeLimit () const

Returns the current maximum on-screen size of the combo box. The default is ten lines.

See also: setSizeLimit(), count() and maxCount().

const char * QComboBox::text (int index) const

Returns the text item at a given index, or 0 if the item is not a string.

See also: currentText().

QValidator * QComboBox::validator () const

Returns the validator which constrains editing for this combo box if there is any, or else 0.

See also: setValidator(), clearValidator() and QValidator.

38.2.1. URL

Generated from http://www.troll.no/qt/qcombobox.html.

39. QConnection Class Reference

```
#include <qconnection.h>
```

39.1. Detailed Description

The QConnection class is an internal class, used in the signal/slot mechanism.

Do not use this class directly in application programs.

39.2. Member Function Documentation

QConnection::QConnection (const QObject * object, QMember member, const char * memberName)

For internal use only.

QConnection::~QConnection ()

For internal use only.

bool QConnection::isConnected () const

For internal use only.

QMember * QConnection::member () const

For internal use only.

const char * QConnection::memberName () const

For internal use only.

int QConnection::numArgs () const

For internal use only.

QObject * QConnection::object () const

For internal use only.

39.2.1. URL

Generated from http://www.troll.no/qt/qconnection.html.

40. QCursor Class Reference

```
#include <qcursor.h>
```

40.1. Detailed Description

The QCursor class provides a mouse cursor with an arbitrary shape.

This class is used for mainly two things; to create mouse cursors to be associated with widgets and to get and set the position of the mouse cursor.

Qt has a number of standard cursor shapes, but you can also make custom cursor shapes based on a bitmap, a mask and a hotspot.

To associate a cursor with a widget, use QWidget::setCursor(). To associate a cursor with all widgets (maybe for a short period of time), use QApplication::setOverrideCursor().

The predefined cursor objects are:

- arrowCursor : standard arrow cursor
- upArrowCursor : upwards arrow
- crossCursor : crosshair
- waitCursor : hourglass/watch
- ibeamCursor : ibeam/text entry
- sizeVerCursor : vertical resize
- sizeHorCursor : horizontal resize
- sizeBDiagCursor : diagonal resize (/)
- sizeFDiagCursor : diagonal resize (\)
- sizeAllCursor : all directions resize
- blankCursor : blank/invisible cursor

See also: QWidget and GUI Design Handbook: Cursors.

Examples:
cursor/cursor.cpp

40.2. Member Function Documentation

QCursor::QCursor ()

Constructs a cursor with the default arrow shape.

QCursor::QCursor (const QBitmap & bitmap, const QBitmap & mask, int hotX=-1, int hotY=-1)

Constructs a custom bitmap cursor.

Arguments:

- *bitmap* and
- *mask* make up the bitmap.
- *hotX* and
- *hotY* define the hot spot of this cursor. If *hotX* is negative, it is set to the bitmap().width()/2. If *hotY* is negative, it is set to the bitmap().height()/2.

The cursor *bitmap* (B) and *mask* (M) bits are combined this way:

1 B=1 and M=1 gives black.

2 B=0 and M=1 gives white.

3 B=0 and M=0 gives transparency.

4 B=1 and M=0 gives an undefined result.

Use the global color `color0` to draw 0-pixels and `color1` to draw 1-pixels in the bitmaps.

Allowed cursor sizes depend on the display hardware (or the underlying window system). We recommend using 32x32 cursors, because this size is supported on all platforms. Some platforms also support 16x16, 48x48 and 64x64 cursors.

QCursor::QCursor (const QCursor & c)

Constructs a copy of the cursor *c*.

QCursor::QCursor (int shape)

Constructs a cursor with the specified *shape*.

QCursor::~QCursor ()

Destroys the cursor.

const QBitmap * QCursor::bitmap () const

Returns the cursor bitmap, or 0 if it is one of the standard cursors.

void QCursor::cleanup () `[static]`

Internal function that cleans up the predefined cursors. This function is called from the QApplication destructor.

See also: initialize().

HANDLE QCursor::handle () const

Returns the window system cursor handle.

Warning: Portable in principle, but if you use it you are probably about to do something non-portable. Be careful.

QPoint QCursor::hotSpot () const

Returns the cursor hot spot, or (0,0) if it is one of the standard cursors.

void QCursor::initialize () `[static]`

Internal function that initializes the predefined cursors. This function is called from the QApplication constructor.

See also: cleanup().

const QBitmap * QCursor::mask () const

Returns the cursor bitmap mask, or 0 if it is one of the standard cursors.

QCursor & QCursor::operator= (const QCursor & c)

Assigns *c* to this cursor and returns a reference to this cursor.

QPoint QCursor::pos () `[static]`

Returns the position of the cursor (hot spot) in global screen coordinates.

You can call QWidget::mapFromGlobal() to translate it to widget coordinates.

See also: setPos(), QWidget::mapFromGlobal() and QWidget::mapToGlobal().

void QCursor::setPos (int x, int y) `[static]`

Moves the cursor (hot spot) to the global screen position *(x,y)*.

You can call QWidget::mapToGlobal() to translate widget coordinates to global screen coordinates.

See also: pos(), QWidget::mapFromGlobal() and QWidget::mapToGlobal().

void QCursor::setPos (const QPoint &) `[static]`

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

void QCursor::setShape (int shape)

Sets the cursor to the shape identified by *shape*.

The allowed shapes are: `ArrowCursor, UpArrowCursor, CrossCursor, WaitCursor, IbeamCursor, SizeVerCursor, SizeHorCursor, SizeBDiagCursor, SizeFDiagCursor, SizeAllCursor, BlankCursor.` These correspond to the predefined global QCursor objects.

See also: shape().

int QCursor::shape () const

Returns the cursor shape identifer. If a custom bitmap has been set, `BitmapCursor` is returned.

See also: setShape().

40.3. Related Functions

QDataStream & operator<< (QDataStream & s, const QCursor & c)

Writes a cursor to the stream.

QDataStream & operator>> (QDataStream & s, QCursor & c)

Reads a cursor from the stream.

40.3.1. URL

Generated from http://www.troll.no/qt/qcursor.html.

41. QCustomEvent Class Reference

`#include <qevent.h>`

Inherits QEvent.

41.1. Detailed Description

The QCustomEvent class provides support for custom events.

QCustomEvent is a user-defined event type which contains a `void*`.

Warning: This event class is internally used to implement Qt enhancements. It is not advisable to use QCustomEvent in normal applications, where other event types and the signal/slot mechanism can do the job.

41.2. Member Function Documentation

QCustomEvent::QCustomEvent (int type, void * data)

Constructs a custom event object with the event type *type* and a pointer to *data*.

void * QCustomEvent::data () const

Returns a pointer to the event data (specified in the constructor).

The event data can be anything and must be casted to something useful based on the event type. Again, it is not recommended to use custom events unless you are implementing Qt kernel enhancements.

41.2.1. URL

Generated from http://www.troll.no/qt/qcustomevent.html.

42. QDataPump Class Reference

`#include <qasyncio.h>`

Inherits QObject.

42.1. Detailed Description

Moves data from a QDataSource to a QDataSink during event processing.

For a QDataSource to provide data to a QDataSink, a controller must exist to examine the QDataSource::readyToSend() and QDataSink::readyToReceive() methods and respond to the QASyncIO::activate() signal of the source and sink. One very useful way to do this is interleaved with other event processing. QDataPump provides this - create a pipe between a source and a sink, and data will be moved during subsequent event processing.

Note that each source can only provide data to one sink and each sink can only receive data from one source (although it is quite possible to write a multiplexing sink that is multiple sources).

42.2. Member Function Documentation

QDataPump::QDataPump (QDataSource * data_source, QDataSink * data_sink)

Creates a QDataPump to move data from a given *data_source* to a given *data_sink*.

42.2.1. URL

Generated from http://www.troll.no/qt/qdatapump.html.

43. QDataSink Class Reference

`#include <qasyncio.h>`

Inherits QAsyncIO.

43.1. Detailed Description

A QDataSink is an asynchronous consumer of data.

A data sink is an object which receives data from some source in an asynchronous manner. This means that at some time not determined by the data sink, blocks of data are given to it from processing. The data sink is able to limit the maximum size of such blocks which it is currently able to process.

See also: QAsyncIO, QDataSource and QDataPump.

43.2. Member Function Documentation

void QDataSink::eof () `[virtual]`

This function will be called when no more data is available for processing.

void QDataSink::maybeReady ()

This should be called whenever readyToReceive() might have become non-zero. It is merely calls QAsyncIO::ready() if readyToReceive() is non-zero.

int QDataSink::readyToReceive () `[virtual]`

The data sink should return a value indicating how much data it is ready to consume. This may be 0.

void QDataSink::receive (const uchar *, int count) `[virtual]`

This function is called to provide data for the data sink. The count will be no more than the amount indicated by the most recent call to readyToReceive(). The sink must use all the provided data.

43.2.1. URL

Generated from http://www.troll.no/qt/qdatasink.html.

44. QDataSource Class Reference

`#include <qasyncio.h>`

Inherits QAsyncIO.

Inherited by QIODeviceSource.

44.1. Detailed Description

A QDataSource is an asynchronous producer of data.

A data source is an object which provides data from some source in an asynchronous manner. This means that at some time not determined by the data source, blocks of data will be taken from it for processing. The data source is able to limit the maximum size of such blocks which it is currently able to provide.

See also: QAsyncIO, QDataSink and QDataPump.

44.2. Member Function Documentation

void QDataSource::enableRewind (bool) `[virtual]`

If this function is called with *on* set to TRUE, and rewindable() is TRUE, then the data source must take measures to allow the rewind() function to subsequently operate as described. If rewindable() is FALSE, the function should call QDataSource::enableRewind(), which aborts with a fatal() error.

For example, a network connection may choose to utilize a disk cache of input only if rewinding is enabled before the first buffer-full of data is discarded, returning FALSE in rewindable() if that first buffer is discarded.

Reimplemented in QIODeviceSource.

void QDataSource::maybeReady ()

This should be called whenever readyToSend() might have become non-zero. It is merely calls QAsyncIO::ready() if readyToSend() is non-zero.

int QDataSource::readyToSend () `[virtual]`

The data source should return a value indicating how much data it is ready to provide. This may be 0. If the data source knows it will never be able to provide any more data (until after a rewind()), it may return -1.

Reimplemented in QIODeviceSource.

void QDataSource::rewind () `[virtual]`

This function rewinds the data source. This may only be called if enableRewind(TRUE) has been previously called.

bool QDataSource::rewindable () const `[virtual]`

This function should return TRUE if the data source can be rewound.

The default returns FALSE.

Reimplemented in QIODeviceSource.

void QDataSource::sendTo (QDataSink *, int count) `[virtual]`

This function is called to extract data from the source, by sending it to the given data sink. The count will be no more than the amount indicated by the most recent call to readyToSend(). The source must use all the provided data, and the sink will be prepared to accept at least this much data.

Reimplemented in QIODeviceSource.

44.2.1. URL

Generated from http://www.troll.no/qt/qdatasource.html.

45. QDataStream Class Reference

```
#include <qdatastream.h>
```

45.1. Detailed Description

The QDataStream class provides basic functions for serialization of binary data to a QIODevice.

A data stream is a binary stream of encoded information which is 100% independent of the host computer operation system, CPU or byte order. A stream that is written by a PC under DOS/Windows can easily be read by a Sun SPARC running Solaris.

The QDataStream class implements serialization of primitive types, like `char`, `short`, `int`, `char*` etc. Serialization of more complex data is accomplished by breaking up the data into primitive units.

The programmer can select which byte order to use when serializing data. The default setting is big endian (MSB first). Changing it to little endian breaks the portability. We therefore recommend keeping this setting unless you have special needs or requirements.

A data stream cooperates closely with a QIODevice. A QIODevice represents an input/output medium one can read data from and write data to. The QFile class is an example of an IO device.

Example (write data to a stream):

```
QFile f( "file.dta" );
f.open( IO_WriteOnly );                    // open file for writing
QDataStream s( &f );                       // serialize using f
s << "the answer is";                      // serialize string
s << (Q_INT32)42;                          // serialize integer
f.close();                                 // done
```

Example (read data from a stream):

```
QFile f( "file.dta" );
f.open( IO_ReadOnly );                     // open file for reading
QDataStream s( &f );                       // serialize using f
char   *str;
Q_INT32 a;
s >> str >> a;                             // "the answer is" and 42
f.close();                                 // done
delete str;                                // delete string
```

In the last example, if you read into a QString instead of a `char*` you do not have to delete it.

See also: QTextStream.

45.2. Member Function Documentation

QDataStream::QDataStream ()

Constructs a data stream that has no IO device.

QDataStream::QDataStream (QByteArray a, int mode)

Constructs a data stream that operates on a byte array throught an internal QBuffer device.

Example:

```
static char bindata[] = { 231, 1, 44, ... };
QByteArray  a;
a.setRawData( bindata, sizeof(bindata) );   // a points to bindata
QDataStream s( a, IO_ReadOnly );            // open on a's data
```

```
s >> <something>;                           // read raw bindata
a.resetRawData( bindata, sizeof(bindata) ); // finished
```

The QArray::setRawData() function is not for the inexperienced.

QDataStream::QDataStream (QIODevice * d)

Constructs a data stream that uses the IO device *d*.

QDataStream::~QDataStream () [virtual]

Destroys the data stream.

The destructor will not affect the current IO device, unless it is an internal IO device processing a QByteArray passed in the constructor.

int QDataStream::byteOrder () const

Returns the current byte order setting.

See also: setByteOrder().

QIODevice * QDataStream::device () const

Returns the IO device currently set.

See also: setDevice() and unsetDevice().

bool QDataStream::eof () const

Returns TRUE if the IO device has reached the end position (end of stream or file) or if there is no IO device set.

Returns FALSE if the current position of the read/write head of the IO device is somewhere before the end position.

See also: QIODevice::atEnd().

bool QDataStream::isPrintableData () const

Returns TRUE if the printable data flag has been set.

See also: setPrintableData().

QDataStream & QDataStream::operator<< (Q_INT16 i)

Writes a signed 16-bit integer to the stream and returns a reference to the stream.

QDataStream & QDataStream::operator<< (Q_INT32 i)

Writes a signed 32-bit integer to the stream and returns a reference to the stream.

QDataStream & QDataStream::operator<< (Q_INT8 i)

Writes a signed byte to the stream.

QDataStream & QDataStream::operator<< (Q_UINT16 i)

Writes an unsigned 16-bit integer to the stream and returns a reference to the stream.

QDataStream & QDataStream::operator<< (Q_UINT32 i)

Writes an unsigned 32-bit integer to the stream and returns a reference to the stream.

QDataStream & QDataStream::operator<< (Q_UINT8 i)

Writes an unsigned byte to the stream and returns a reference to the stream.

QDataStream & QDataStream::operator<< (const char * s)

Writes the '\0'-terminated string *s* to the stream and returns a reference to the stream.

The string is serialized using writeBytes().

QDataStream & QDataStream::operator<< (double f)

Writes a 64-bit floating point number to the stream using the standard IEEE754 format. Returns a reference to the stream.

QDataStream & QDataStream::operator<< (float f)

Writes a 32-bit floating point number to the stream using the standard IEEE754 format. Returns a reference to the stream.

QDataStream & QDataStream::operator>> (Q_INT16 & i)

Reads a signed 16-bit integer from the stream and returns a reference to the stream.

QDataStream & QDataStream::operator>> (Q_INT32 & i)

Reads a signed 32-bit integer from the stream and returns a reference to the stream.

QDataStream & QDataStream::operator>> (Q_INT8 & i)

Reads a signed byte from the stream.

QDataStream & QDataStream::operator>> (Q_UINT16 & i)

Reads an unsigned 16-bit integer from the stream and returns a reference to the stream.

QDataStream & QDataStream::operator>> (Q_UINT32 & i)

Reads an unsigned 32-bit integer from the stream and returns a reference to the stream.

QDataStream & QDataStream::operator>> (Q_UINT8 & i)

Reads an unsigned byte from the stream and returns a reference to the stream.

QDataStream & QDataStream::operator>> (char *& s)

Reads the '\0'-terminated string *s* from the stream and returns a reference to the stream.

The string is read using readBytes(), which allocates space using `new`.

QDataStream & QDataStream::operator>> (double & f)

Reads a 64-bit floating point number from the stream using the standard IEEE754 format. Returns a reference to the stream.

QDataStream & QDataStream::operator>> (float & f)

Reads a 32-bit floating point number from the stream using the standard IEEE754 format. Returns a reference to the stream.

QDataStream & QDataStream::readBytes (char *& s, uint & l)

Reads the buffer *s* from the stream and returns a reference to the stream.

The buffer *s* is allocated using `new`. Destroy it with the `delete` operator. If the length is zero or *s* cannot be allocated, *s* is set to 0.

The *l* parameter will be set to the length of the buffer.

The serialization format is an Q_UINT32 length specifier first, then the data (*length* bytes).

See also: readRawBytes() and writeBytes().

QDataStream & QDataStream::readRawBytes (char * s, uint len)

Reads *len* bytes from the stream into *e* s and returns a reference to the stream.

The buffer *s* must be preallocated.

See also: readBytes(), QIODevice::readBlock() and writeRawBytes().

void QDataStream::setByteOrder (int bo)

Sets the serialization byte order to *bo*.

The *bo* parameter can be `QDataStream::BigEndian` or `QDataStream::LittleEndian`.

The default setting is big endian. We recommend leaving this setting unless you have special requirements.

See also: byteOrder().

void QDataStream::setDevice (QIODevice * d)

void QDataStream::setDevice(QIODevice *d) Sets the IO device to *d*.

See also: device() and unsetDevice().

void QDataStream::setPrintableData (bool enable)

Sets or clears the printable data flag.

If this flag is set, the write functions will generate output that consists of printable characters (7 bit ASCII).

We recommend enabling printable data only for debugging purposes (it is slower and creates bigger output).

void QDataStream::unsetDevice ()

Unsets the IO device. This is the same as calling setDevice(0).

See also: device() and setDevice().

QDataStream & QDataStream::writeBytes (const char * s, uint len)

Writes the length specifier *len* and the buffer *s* to the stream and returns a reference to the stream.

The *len* is serialized as an Q_UINT32, followed by *len* bytes from *s*.

See also: writeRawBytes() and readBytes().

QDataStream & QDataStream::writeRawBytes (const char * s, uint len)

Writes *len* bytes from *s* to the stream and returns a reference to the stream.

See also: writeBytes(), QIODevice::writeBlock() and readRawBytes().

45.2.1. URL

Generated from http://www.troll.no/qt/qdatastream.html.

46. QDate Class Reference

`#include <qdatetime.h>`

46.1. Detailed Description

The QDate class provides date functions.

The QDate is based on the Gregorian (modern western) calendar. England adopted the Gregorian calendar on September 14th 1752, which is the earliest date that is supported by QDate. Using earlier dates will give undefined results. Some countries adopted the Gregorian calendar later than England, thus the week day of early dates might be incorrect for these countries (but correct for England). The end of time is reached around 8000AD, by which time we expect Qt to be obsolete.

See also: QTime and QDateTime.

Examples:

dclock/dclock.cpp

46.2. Member Function Documentation

QDate::QDate ()

Constructs a null date. Null dates are invalid.

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

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

QDate QDate::addDays (int ndays) const

Returns this date plus *ndays* days.

QDate QDate::currentDate () `[static]`

Returns the current date.

See also: QTime::currentTime() and QDateTime::currentDateTime().

int QDate::day () const

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

See also: year(), month() and dayOfWeek().

Examples:

dclock/dclock.cpp

const char * QDate::dayName (int weekday) const `[virtual]`

Returns the name of the *weekday.*

Weekday 1 == "Mon", day 2 == "Tue" etc.

int QDate::dayOfWeek () const

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

See also: day().

int QDate::dayOfYear () const

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

int QDate::daysInMonth () const

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

int QDate::daysInYear () const

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

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

Returns the number of days from this date to *d,* which is negative if *d* is in the past.

Example:

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

See also: addDays().

uint QDate::greg2jul (int y, int m, int d) `[static protected]`

For internal use only.

bool QDate::isNull () const

Returns TRUE if the date is null. A null date is invalid.

See also: isValid().

bool QDate::isValid () const

Returns TRUE if the date is valid.

See also: isNull().

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

Returns TRUE if the specified date is valid.

Note that years 00-99 are treated as 1900-1999.

See also: isNull().

void QDate::jul2greg (uint jd, int & y, int & m, int & d) `[static protected]`

For internal use only.

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

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

int QDate::month () const

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

See also: year() and day().

Examples:
dclock/dclock.cpp

const char * QDate::monthName (int month) const `[virtual]`

Returns the name of the *month.*

Month 1 == "Jan", month 2 == "Feb" etc.

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

Returns TRUE if this date is different from *d,* or FALSE if they are equal.

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

Returns TRUE if this date is before *d,* otherwise FALSE.

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

Returns TRUE if this date is before or equal to *d,* otherwise FALSE.

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

Returns TRUE if this date is equal to *d,* or FALSE if they are different.

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

Returns TRUE if this date is after *d,* otherwise FALSE.

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

Returns TRUE if this date is equal to or after *d,* otherwise FALSE.

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

Sets the year *y,* month *m* and day *d.* Returns TRUE if the date is valid, otherwise FALSE.

QString QDate::toString () const

Returns the date as a string.

The string format is "Sat May 20 1995".

int QDate::year () const

Returns the year (>= 1752) for this date.

See also: month() and day().

46.3. Related Functions

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

Writes the date to the stream.

Serialization format: [Q_UINT32], Julian day.

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

Reads a date from the stream.

46.3.1. URL

Generated from http://www.troll.no/qt/qdate.html.

47. QDateTime Class Reference

`#include <qdatetime.h>`

47.1. Detailed Description

The QDateTime class combines QDate and QTime into a single class.

QDateTime provides high precision date and time functions since it can work with Gregorian dates up to about year 8000.

Most countries that use the Gregorian calendar switched to it between 1550 and 1920.

See also: QDate and QTime.

47.2. Member Function Documentation

QDateTime::QDateTime ()

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

QDateTime::QDateTime (const QDate & date)

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

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

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

QDateTime QDateTime::addDays (int ndays) const

Returns the datetime plus *ndays* days.

See also: daysTo().

QDateTime QDateTime::addSecs (int nsecs) const

Returns the datetime plus *nsecs* seconds.

See also: secsTo().

QDateTime QDateTime::currentDateTime () `[static]`

Returns the current datetime.

See also: QDate::currentDate() and QTime::currentTime().

QDate QDateTime::date () const

Returns the date part of this datetime.

See also: time().

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

Returns the number of days from this datetime to *dt,* which is negaitive if *dt* is in the past.

See also: addDays() and secsTo().

bool QDateTime::isNull () const

Returns TRUE if both the date and the time are null. A null date is invalid.

See also: QDate::isNull() and QTime::isNull().

bool QDateTime::isValid () const

Returns TRUE if both the date and the time are value.

See also: QDate::isValid() and QTime::isValid().

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

Returns TRUE if this datetime is different from *dt,* or FALSE if they are equal.

See also: operator==().

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

Returns TRUE if this datetime is before *dt,* otherwise FALSE.

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

Returns TRUE if this datetime is before or equal to *dt,* otherwise FALSE.

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

Returns TRUE if this datetime is equal to *dt,* or FALSE if they are different.

See also: operator!=().

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

Returns TRUE if this datetime is after *dt,* otherwise FALSE.

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

Returns TRUE if this datetime is equal to or after *dt,* otherwise FALSE.

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

Returns the number of seconds from this datetime to *dt,* which is negative if *t* is in the past.

Example:

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

See also: addSecs(), daysTo() and QTime::secsTo().

void QDateTime::setDate (const QDate & date)

Sets the date part of this datetime.

See also: setTime().

void QDateTime::setTime (const QTime & time)

Sets the time part of this datetime.

See also: setDate().

void QDateTime::setTime_t (uint secsSince1Jan1970UTC)

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

QTime QDateTime::time () const

Returns the time part of this datetime.

See also: date().

QString QDateTime::toString () const

Returns the datetime as a string.

The string format is "Sat May 20 1995 03:40:13".

47.3. Related Functions

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

Writes a datetime to the stream.

Serialization format: [QDate QTime].

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

Reads a datetime from the stream.

47.3.1. URL

Generated from http://www.troll.no/qt/qdatetime.html.

48. QDialog Class Reference

`#include <qdialog.h>`

Inherits QWidget.

Inherited by QFileDialog, QMessageBox, QPrintDialog and QTabDialog.

48.1. Detailed Description

The QDialog class is the base class of dialog windows.

A dialog window is a widget used to communicate with the user. It offers mechanisms such as default buttons.

The dialog window can either be modeless or modal. A modeless dialog is a normal window, while a modal window must be finished before the user can continue with other parts of the program. The third constructor argument must be set to TRUE to create a modal dialog, otherwise it will create a modeless dialog.

Example (your own modal dialog):

```
class Modal : public QDialog {
    Q_OBJECT
public:
    Modal( QWidget *parent, const char *name );
};
Modal::Modal( QWidget *parent, const char *name )
    : QDialog( parent, name, TRUE )
{
    QPushButton *ok, *cancel;
    ok = new QPushButton( "Ok", this );
    ok->setGeometry( 10,10, 100,30 );
    connect( ok, SIGNAL(clicked()), SLOT(accept()) );
    cancel = new QPushButton( "Cancel", this );
    cancel->setGeometry( 10,60, 100,30 );
    connect( cancel, SIGNAL(clicked()), SLOT(reject()) );
}
```

Note that the parent widget has a different meaning for modal dialogs than for other types of widgets. A dialog is placed on top of the parent widget. The dialog is centered on the screen if the parent widget is zero.

You would normally call exec() to start a modal dialog. This enters a local event loop, which is terminated when the modal dialog calls done() (or accept() or reject()).

Example (using a modal dialog):

```
Modal m;
if ( m.exec() ) {
   // ok was pressed, then fetch the interesting dialog data
}
```

Modeless dialogs behave just like ordinary widgets. The only difference is that they have the default button mechanism.

See also: QTabDialog, QWidget, QSemiModal and GUI Design Handbook: Dialogs, Standard.

Examples:

xform/xform.cpp

48.2. Member Function Documentation

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

Constructs a dialog named *name,* which has a parent widget *parent.*

The dialog will by default be modeless, unless you set *modal* to TRUE to contruct a modal dialog.

The *f* argument is the widget flags, which can be used to customize the window frame style.

Warning: Creating a modeless dialog with a parent makes it an ordinary child widget, which is probably not what you want. Expect strange behavior (QDialog has a default button mechanism).

QDialog::~QDialog ()

Destroys the QDialog and all its children.

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

Closes the dialog and sets the result code to `Accepted.`

Equivalent to done(Accepted);.

void QDialog::closeEvent (QCloseEvent * e) `[virtual protected]`

Calls reject() if it is a modal dialog, or accepts the close event if it is a modeless dialog.

Reimplemented from QWidget.

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

Closes the dialog and sets the result code to *r.*

Equivalent to calling hide(), then setResult(*r*).

This function is very useful for modal dialogs. It leaves the local event loop and returns from the exec() or show() function.

Warning: Although done() will return to the caller if this dialog is modal, the local event loop is then marked for termination. Hence, a program should not try to do anything that depends on event handling before the corresponding exec() or show() has returned.

See also: accept() and reject().

int QDialog::exec ()

Starts the dialog and returns the result code.

Equivalent to calling show(), then result().

This function is very useful for modal dialogs. It enters a new local event loop. The event loop is terminated when the dialog is hidden, usually by calling done().

void QDialog::keyPressEvent (QKeyEvent * e) `[virtual protected]`

Handles key press events for the dialog.

Calls reject() if Escape is pressed. Simulates a button click for the default button if Enter is pressed. All other keys are ignored.

Reimplemented from QWidget.

void QDialog::move (const QPoint & p)

Reimplements QWidget::move() for internal purposes.

void QDialog::move (int x, int y) `[virtual]`

Reimplements QWidget::move() for internal purposes.

Reimplemented from QWidget.

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

Closes the dialog and sets the result code to `Rejected`.

Equivalent to done(Rejected);.

void QDialog::resize (const QSize & s)

Reimplements QWidget::resize() for internal purposes.

Examples:
  pref/pref.cpp

void QDialog::resize (int w, int h) `[virtual]`

Reimplements QWidget::resize() for internal purposes.

Reimplemented from QWidget.

int QDialog::result () const

Returns the result code of the dialog.

void QDialog::setGeometry (const QRect & r)

Reimplements QWidget::setGeometry() for internal purposes.

void QDialog::setGeometry (int x, int y, int w, int h) `[virtual]`

Reimplements QWidget::setGeometry() for internal purposes.

Examples:
  xform/xform.cpp

Reimplemented from QWidget.

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

Sets the result code of the dialog.

void QDialog::show () `[virtual]`

Shows the dialog box on the screen, as QWidget::show() and enters a local event loop if this dialog is modal (see constructor).

This implementation also does automatic resizing and automatic positioning. If you have not already resized or moved the dialog, it will find a size that fits the contents and a position near the middle of the screen (or centered relative to the parent widget if any).

Warning: Calling show() for a modal dialog enters a local event loop. The event loop is terminated when the dialog is hidden, usually by calling done().

See also: exec().

Examples:
movies/main.cpp
xform/xform.cpp

Reimplemented from QWidget.

48.2.1. URL

Generated from http://www.troll.no/qt/qdialog.html.

49. QDict Class Reference

`#include <qdict.h>`

Inherits QGDict.

49.1. Detailed Description

The QDict class is a template class that provides a dictionary based on `char*` keys.

QDict is implemented as both a template and a macro class. Define a template instance QDict<X> to create a dictionary that operates on pointers to X, or X*.

A dictionary is a collection that associates an item with a key. The key is used for inserting and looking up an item. QDict has `char*` keys.

The dictionary has very fast insertion and lookup.

Example:

```
#include <qdict.h>
#include <stdio.h>
void main()
{
  // Creates a dictionary that maps char* ==> char* (case insensitive)
    QDict<char> dict( 17, FALSE );
    dict.insert( "France", "Paris" );
    dict.insert( "Russia", "Moscow" );
    dict.insert( "Norway", "Oslo" );
    printf( "%s\n", dict["Norway"] );
    printf( "%s\n", dict["FRANCE"] );
    printf( "%s\n", dict["russia"] );
    if ( !dict["Italy"] )
        printf( "Italy not defined\n" );
}
```

Program output:

```
Oslo
Paris
Moscow
Italy not defined
```

The dictionary in our example maps `char*` keys to `char*` items. Note that the mapping is case insensitive (set in the constructor). QDict implements the [] operator to lookup an item.

A QDict can also be instantiated through a macro expansion, but this is necessary only for compilers that do not support templates. See the collection classes documentation for a general discussion on template-based versus macro-based collections.

QDict is implemented by QGDict as a hash array with a fixed number of entries. Each array entry points to a singly linked list of buckets, in which the dictionary items are stored.

When an item is inserted with a key, the key is converted (hashed) to an integer index into the hash array. The item is inserted before the first bucket in the list of buckets.

Looking up an item is normally very fast. The key is again hashed to an array index. Then QDict scans the list of buckets and returns the item found or null if the item was not found. You cannot insert null pointers into a dictionary.

The size of the hash array is very important. In order to get good performance, you should use a suitably large prime number. Suitable means equal to or larger than the maximum expected number of dictionary items.

Items with equal keys are allowed. When inserting two items with the same key, only the last inserted item will be visible (last in, first out) until it is removed.

Example:

```
#include <qdict.h>
#include <stdio.h>
void main()
{
  // Creates a dictionary that maps char* ==> char* (case sensitive)
    QDict<char> dict;
    dict.insert( "Germany", "Berlin" );
    dict.insert( "Germany", "Bonn" );
    printf( "%s\n", dict["Germany"] );
    dict.remove( "Germany" );       // Oct 3rd 1990
    printf( "%s\n", dict["Germany"] );
}
```

Program output:

```
Bonn
Berlin
```

The QDictIterator class can traverse the dictionary contents, but only in an arbitrary order. Multiple iterators may independently traverse the same dictionary.

Calling setAutoDelete(TRUE) for a dictionary tells it to delete items that are removed . The default is to not delete items when they are removed.

When inserting an item into a dictionary, only the pointer is copied, not the item itself. This is called a shallow copy. It is possible to make the dictionary copy all of the item's data (known as a deep copy) when an item is inserted. insert() calls the virtual function QCollection::newItem() for the item to be inserted. Inherit a dictionary and reimplement it if you want deep copies.

When removing a dictionary item, the virtual function QCollection::deleteItem() is called. QDict's default implementation is to delete the item if auto-deletion is enabled.

See also: QDictIterator, QIntDict and Collection Classes

49.2. Member Function Documentation

QDict::QDict (const QDict<type> & dict)

Constructs a copy of *dict.*

Each item in *dict* are inserted into this dictionary. Only the pointers are copied (shallow copy).

QDict::QDict (int size=17, bool caseSensitive=TRUE, bool copyKeys=TRUE)

Constructs a dictionary with the following properties:

Arguments:

- *size* is the size of the internal hash array.
- *caseSensitive* specifies whether to use case sensitive lookup or not.
- *copyKeys* specifies whether to copy the key strings. Setting *size* to a suitably large prime number (equal to or greater than the expected number of entries) makes the hash distribution better and hence the loopup faster.

Setting *caseSensitive* to TRUE will treat "abc" and "Abc" as different keys. Setting it to FALSE will make the dictionary ignore case. Case insensitive comparision includes only the 26 standard letters A..Z, not French or German accents or Scandinavian letters.

Setting *copyKeys* to TRUE will make the dictionary copy the key when an item is inserted. Setting it to FALSE will make the dictionary only use the pointer to the key.

QDict::~QDict ()

Removes all items from the dictionary and destroys it.

All iterators that access this dictionary will be reset.

See also: setAutoDelete().

void QDict::clear () `[virtual]`

Removes all items from the dictionary.

The removed items are deleted if auto-deletion is enabled.

All dictionary iterators that operate on dictionary are reset.

See also: remove(), take() and setAutoDelete().

Reimplemented from QCollection.

uint QDict::count () const `[virtual]`

Returns the number of items in the dictionary.

See also: isEmpty().

Reimplemented from QCollection.

type * QDict::find (const char * key) const

Returns the item associated with *key,* or null if the key does not exist in the dictionary.

This function uses an internal hashing algorithm to optimize lookup.

If there are two or more items with equal keys, then the last inserted of these will be found.

Equivalent to the [] operator.

See also: operator[].

void QDict::insert (const char * key, const type * item)

Inserts the *key* with the *item* into the dictionary.

The key does not have to be a unique dictionary key. If multiple items are inserted with the same key, only the last item will be visible.

Null items are not allowed.

See also: replace().

bool QDict::isEmpty () const

Returns TRUE if the dictionary is empty, i.e. count() == 0. Returns FALSE otherwise.

See also: count().

QDict<type> & QDict::operator= (const QDict<type> & dict)

Assigns *dict* to this dictionary and returns a reference to this dictionary.

This dictionary is first cleared, then each item in *dict* is inserted into this dictionary. Only the pointers are copied (shallow copy), unless newItem() has been reimplemented().

type * QDict::operator[] (const char * key) const

Returns the item associated with *key,* or null if the key does not exist in the dictionary.

This function uses an internal hashing algorithm to optimize lookup.

If there are two or more items with equal keys, then the last inserted of these will be found.

Equivalent to the find() function.

See also: find().

bool QDict::remove (const char * key)

Removes the item associated with *key* from the dictionary. Returns TRUE if successful, or FALSE if the key does not exist in the dictionary.

If there are two or more items with equal keys, then the last inserted of these will be removed.

The removed item is deleted if auto-deletion is enabled.

All dictionary iterators that refer to the removed item will be set to point to the next item in the dictionary traversing order.

See also: take(), clear() and setAutoDelete().

void QDict::replace (const char * key, const type * item)

Replaces an item which has a key equal to *key* with *item.*

If the item does not already exist, it will be inserted.

Null items are not allowed.

Equivalent to:

```
QDict<char> dict;
    ...
if ( dict.find(key) )
    dict.remove( key );
dict.insert( key, item );
```

If there are two or more items with equal keys, then the last inserted of these will be replaced.

See also: insert().

void QDict::resize (uint newsize)

Changes the size of the hashtable the *newsize.* The contents of the dictionary are preserved, but all iterators on the dictionary become invalid.

uint QDict::size () const

Returns the size of the internal hash array (as specified in the constructor).

See also: count().

void QDict::statistics () const

Debugging-only function that prints out the dictionary distribution using debug().

type * QDict::take (const char * key)

Takes the item associated with *key* out of the dictionary without deleting it (even if auto-deletion is enabled).

If there are two or more items with equal keys, then the last inserted of these will be taken.

Returns a pointer to the item taken out, or null if the key does not exist in the dictionary.

All dictionary iterators that refer to the taken item will be set to point to the next item in the dictionary traversal order.

See also: remove(), clear() and setAutoDelete().

49.2.1. URL

Generated from http://www.troll.no/qt/qdict.html.

50. QDictIterator Class Reference

`#include <qdict.h>`

Inherits QGDictIterator.

50.1. Detailed Description

The QDictIterator class provides an iterator for QDict collections.

QDictIterator is implemented as both a template and a macro class. Define a template instance QDictIterator<X> to create a dictionary iterator that operates on QDict<X> (dictionary of X*).

Example:

```
#include <qdict.h>
#include <stdio.h>
void main()
{
  // Creates a dictionary that maps char* ==> char* (case insensitive)
    QDict<char> dict( 17, FALSE );
    dict.insert( "France", "Paris" );
    dict.insert( "Russia", "Moscow" );
    dict.insert( "Norway", "Oslo" );
    QDictIterator<char> it( dict ); // iterator for dict
    while ( it.current() ) {
        printf( "%s -> %s\n", it.currentKey(), it.current() );
        ++it;
    }
}
```

Program output:

```
Russia -> Moscow
Norway -> Oslo
France -> Paris
```

Note that the traversal order is arbitrary, you are not guaranteed the order above.

A QDictIterator can also be instantiated through a macro expansion, but this is necessary only for compilers that do not support templates. See the collection classes documentation for a general discussion on template-based versus macro-based collections.

Multiple iterators may independently traverse the same dictionary. A QDict knows about all iterators that are operating on the dictionary. When an item is removed from the dictionary, QDict update all iterators that are referring the removed item to point to the next item in the traversing order.

See also: QDict and Collection Classes

50.2. Member Function Documentation

QDictIterator::QDictIterator (const QDict<type> & dict)

Constructs an iterator for *dict*. The current iterator item is set to point on the first item in the *dict*.

QDictIterator::~QDictIterator ()

Destroys the iterator.

QDictIterator::operator type * () const

Cast operator. Returns a pointer to the current iterator item. Same as current().

uint QDictIterator::count () const

Returns the number of items in the dictionary this iterator operates on.

See also: isEmpty().

type * QDictIterator::current () const

Returns a pointer to the current iterator item.

const char * QDictIterator::currentKey () const

Returns a pointer to the key for the current iterator item.

bool QDictIterator::isEmpty () const

Returns TRUE if the dictionary is empty, i.e. count() == 0, otherwise FALSE.

See also: count().

type * QDictIterator::operator() ()

Makes the succeeding item current and returns the original current item.

If the current iterator item was the last item in the dictionary or if it was null, null is returned.

type * QDictIterator::operator++ ()

Prefix ++ makes the succeeding item current and returns the new current item.

If the current iterator item was the last item in the dictionary or if it was null, null is returned.

type * QDictIterator::operator+= (uint jump)

Sets the current item to the item *jump* positions after the current item, and returns a pointer to that item.

If that item is beyond the last item or if the dictionary is empty, it sets the current item to null and returns null.

type * QDictIterator::toFirst ()

Sets the current iterator item to point to the first item in the dictionary and returns a pointer to the item. If the dictionary is empty it sets the current item to null and returns null.

50.2.1. URL

Generated from http://www.troll.no/qt/qdictiterator.html.

51. QDir Class Reference

```
#include <qdir.h>
```

51.1. Detailed Description

Traverses directory structures and contents in a platform-independent way.

A QDir can point to a file using either a relative or an absolute file path. Absolute file paths begin with the directory separator ('/') or a drive specification (not applicable to UNIX). Relative file names begin with a directory name or a file name and specify a path relative to the current directory.

An example of an absolute path is the string "/tmp/quartz", a relative path might look like "src/fatlib". You can use the function isRelative() to check if a QDir is using a relative or an absolute file path. You can call the function convertToAbs() to convert a relative QDir to an absolute one.

The directory "example" under the current directory is checked for existence in the example below:

```
QDir d( "example" );                        // "./example"
if ( !d.exists() )
    warning( "Cannot find the example directory" );
```

If you always use '/' as a directory separator, Qt will translate your paths to conform to the underlying operating system.

cd() and cdUp() can be used to navigate the directory tree. Note that the logical cd and cdUp operations are not performed if the new directory does not exist.

Example:

```
QDir d = QDir::root();                      // "/"
if ( !d.cd("tmp") ) {                       // "/tmp"
    warning( "Cannot find the \"/tmp\" directory" );
} else {
    QFile f( d.filePath("ex1.txt") );       // "/tmp/ex1.txt"
    if ( !f.open(IO_ReadWrite) )
        warning( "Cannot create the file %s", f.name() );
}
```

To read the contents of a directory you can use the entryList() and entryInfoList() functions.

Example:

```
#include <stdio.h>
#include <qdir.h>
//
// This program scans the current directory and lists all files
// that are not symbolic links, sorted by size with the smallest files
// first.
//
int main( int argc, char **argv )
{
    QDir d;
    d.setFilter( QDir::Files | QDir::Hidden | QDir::NoSymLinks );
    d.setSorting( QDir::Size | QDir::Reversed );
    const QFileInfoList *list = d.entryInfoList();
    QFileInfoListIterator it( *list );      // create list iterator
    QFileInfo *fi;                          // pointer for traversing
    printf( "     BYTES FILENAME\n" );      // print header
    while ( (fi=it.current()) ) {           // for each file...
        printf( "%10li %s\n", fi->size(), fi->fileName().data() );
        ++it;                               // goto next list element
    }
}
```

Examples:
dirview/dirview.cpp

51.2. Member Function Documentation

QDir::QDir ()

Constructs a QDir pointing to the current directory.

See also: currentDirPath().

QDir::QDir (const QDir & d)

Constructs a QDir that is a copy of the given directory.

See also: operator=().

QDir::QDir (const char * path, const char * nameFilter = 0, int sortSpec = Name | IgnoreCase, int filterSpec = All)

Constructs a QDir.

Arguments:

- *path* is the directory.
- *nameFilter* is the file name filter.
- *sortSpec* is the sort specification, which describes how to sort the files in the directory.
- *filterSpec* is the filter specification, which describes how to filter the files in the directory. Most of these arguments (except *path)* have optional values.

Example:

```
// lists all files in /tmp
QDir d( "/tmp" );
for ( int i=0; i<d.count(); i++ )
    printf( "%s\n", d[i] );
```

If *path* is "" or null, the directory is set to "." (the current directory). If *nameFilter* is "" or null, it is set to "*" (all files).

No check is made to ensure that the directory exists.

See also: exists(), setPath(), setNameFilter(), setFilter() and setSorting().

QDir::˜QDir ()

Destroys the QDir and cleans up.

QString QDir::absFilePath (const char * fileName, bool acceptAbsPath = TRUE) const

Returns the absolute path name of a file in the directory. Does NOT check if the file actually exists in the directory. Redundant multiple separators or "." and ".." directories in *fileName* will NOT be removed (see cleanDirPath()).

If *acceptAbsPath* is TRUE a *fileName* starting with a separator ('/') will be returned without change. if *acceptAbsPath* is FALSE an absolute path will be appended to the directory path.

See also: filePath().

QString QDir::absPath () const

Returns the absolute (a path that starts with '/') path, which may contain symbolic links, but never contains redundant ".", ".." or multiple separators.

See also: setPath(), canonicalPath(), exists(), cleanDirPath(), dirName() and absFilePath().

QString QDir::canonicalPath () const

Returns the canonical path, i.e. a path without symbolic links.

On systems that do not have symbolic links this function will always return the same string that absPath returns. If the canonical path does not exist a null string is returned.

See also: path(), absPath(), exists(), cleanDirPath(), dirName(), absFilePath() and QString::isNull().

bool QDir::cd (const char * dirName, bool acceptAbsPath = TRUE)

Changes directory by descending into the given directory. Returns TRUE if the new directory exists and is readable. Note that the logical cd operation is NOT performed if the new directory does not exist.

If *acceptAbsPath* is TRUE a path starting with a separator ('/') will cd to the absolute directory, if *acceptAbsPath* is FALSE any number of separators at the beginning of *dirName* will be removed.

Example:

```
QDir d = QDir::home();  // now points to home directory
if ( !d.cd("c++") ) {   // now points to "c++" under home directory if OK
    QFileInfo fi( d, "c++" );
    if ( fi.exists() ) {
        if ( fi.isDir() )
            warning( "Cannot cd into \"%s\".", (char*)d.absFilePath("c++") );
        else
            warning( "Cannot create directory \"%s\"\n"
                     "A file named \"c++\" already exists in \"%s\"",
                     (const char *)d.absFilePath("c++"),
                     (const char *)d.path() );
        return;
    } else {
        warning( "Creating directory \"%s\"",
                 (const char *) d.absFilePath("c++") );
        if ( !d.mkdir( "c++" ) ) {
            warning("Could not create directory \"%s\"",
                    (const char *)d.absFilePath("c++") );
            return;
        }
    }
}
```

Calling cd("..") is equivalent to calling cdUp().

See also: cdUp(), isReadable(), exists() and path().

bool QDir::cdUp ()

Changes directory by moving one directory up the path followed to arrive at the current directory.

Returns TRUE if the new directory exists and is readable. Note that the logical cdUp() operation is not performed if the new directory does not exist.

See also: cd(), isReadable(), exists() and path().

QString QDir::cleanDirPath (const char * filePath) `[static]`

Removes all multiple directory separators ('/') and resolves any "." or ".." found in the path.

Symbolic links are kept. This function does not return the canonical path.

QString QDir::convertSeparators (const char * pathName) `[static]`

Converts the '/' separators in *pathName* to system native separators. Returns the translated string.

On Windows, convertSeparators("c:/winnt/system32") returns "c:\winnt\system32".

No conversion is done on UNIX.

void QDir::convertToAbs ()

Converts the directory path to an absolute path. If it is already absolute nothing is done.

See also: isRelative().

uint QDir::count () const

Returns the number of files that was found. Equivalent to entryList()->count().

See also: operator[] and entryList().

QDir QDir::current () `[static]`

Returns the current directory.

See also: currentDirPath() and QDir::QDir().

QString QDir::currentDirPath () `[static]`

Returns the absolute path of the current directory.

See also: current().

QString QDir::dirName () const

Returns the name of the directory, this is NOT the same as the path, e.g. a directory with the name "mail", might have the path "/var/spool/mail". If the directory has no name (e.g. the root directory) a null string is returned.

No check is made to ensure that a directory with this name actually exists.

See also: path(), absPath(), absFilePath(), exists() and QString::isNull().

const QFileInfoList * QDir::drives () `[static]`

Returns a list if the root directories on this system. On win32, this returns a number of QFileInfo objects containing "C:/", "D:/" etc. On other operating systems, it returns a list containing just one root directory (e.g. "/").

const QFileInfoList * QDir::entryInfoList (const char * nameFilter, int filterSpec = DefaultFilter, int sortSpec = DefaultSort) const

Returns a list of QFileInfo objects for all files and directories in the directory pointed to using the setSorting(), setFilter() and setNameFilter() specifications.

The the filter and sorting specifications can be overridden using the *nameFilter, filterSpec* and *sortSpec* arguments.

Returns 0 if the directory is unreadable or does not exist.

See also: entryList(), setNameFilter(), setSorting() and setFilter().

const QFileInfoList * QDir::entryInfoList (int filterSpec = DefaultFilter, int sortSpec = DefaultSort) const

Returns a list of QFileInfo objects for all files and directories in the directory pointed to using the setSorting(), setFilter() and setNameFilter() specifications.

The the filter and sorting specifications can be overridden using the *filterSpec* and *sortSpec* arguments.

Returns 0 if the directory is unreadable or does not exist.

See also: entryList(), setNameFilter(), setSorting() and setFilter().

Examples:
dirview/dirview.cpp

const QStrList * QDir::entryList (const char * nameFilter, int filterSpec = DefaultFilter, int sortSpec = DefaultSort) const

Returns a list of the names of all files and directories in the directory pointed to using the setSorting(), setFilter() and setNameFilter() specifications.

The the filter and sorting specifications can be overridden using the *nameFilter, filterSpec* and *sortSpec* arguments.

Returns 0 if the directory is unreadable or does not exist.

See also: entryInfoList(), setNameFilter(), setSorting() and setFilter().

const QStrList * QDir::entryList (int filterSpec = DefaultFilter, int sortSpec = DefaultSort) const

Returns a list of the names of all files and directories in the directory pointed to using the setSorting(), setFilter() and setNameFilter() specifications.

The the filter and sorting specifications can be overridden using the *filterSpec* and *sortSpec* arguments.

Returns 0 if the directory is unreadable or does not exist.

See also: entryInfoList(), setNameFilter(), setSorting() and setFilter().

bool QDir::exists () const

Returns TRUE if the directory exists. (If a file with the same name is found this function will of course return FALSE).

See also: QFileInfo::exists() and QFile::exists().

bool QDir::exists (const char * name, bool acceptAbsPath = TRUE)

Checks for existence of a file.

If *acceptAbsPaths* is TRUE a path starting with a separator ('/') will check the file with the absolute path, if *acceptAbsPath* is FALSE any number of separators at the beginning of *name* will be removed.

Returns TRUE if the file exists, otherwise FALSE.

See also: QFileInfo::exists() and QFile::exists().

QString QDir::filePath (const char * fileName, bool acceptAbsPath = TRUE) const

Returns the path name of a file in the directory. Does NOT check if the file actually exists in the directory. If the QDir is relative the returned path name will also be relative. Redundant multiple separators or "." and ".." directories in *fileName* will not be removed (see cleanDirPath()).

If *acceptAbsPath* is TRUE a *fileName* starting with a separator ('/') will be returned without change. If *acceptAbsPath* is FALSE an absolute path will be appended to the directory path.

See also: absFilePath(), isRelative() and canonicalPath().

QDir::FilterSpec QDir::filter() const

Returns the value set by setFilter()

See also: setFilter().

QDir QDir::home () `[static]`

Returns the home directory.

See also: homeDirPath().

QString QDir::homeDirPath () `[static]`

Returns the absolute path for the user's home directory,

See also: home().

bool QDir::isReadable () const

Returns TRUE if the directory is readable AND we can open files by name. This function will return FALSE if only one of these is present.

Warning: A FALSE value from this function is not a guarantee that files in the directory are not accessible.

See also: QFileInfo::isReadable().

Examples:
dirview/dirview.cpp

bool QDir::isRelative () const

Returns TRUE if the directory path is relative to the current directory, FALSE if the path is absolute (e.g. under UNIX a path is relative if it does not start with a '/').

According to Einstein this function should always return TRUE.

See also: convertToAbs().

bool QDir::isRelativePath (const char * path) `[static]`

Returns TRUE if the path is relative, FALSE if it is absolute.

See also: isRelative().

bool QDir::isRoot () const

Returns TRUE if the directory is the root directory, otherwise FALSE.

Note: If the directory is a symbolic link to the root directory this function returns FALSE. If you want to test for this you can use canonicalPath():

Example:

```
QDir d( "/tmp/root_link" );
d = d.canonicalPath();
if ( d.isRoot() )
    warning( "It IS a root link!" );
```

See also: root() and rootDirPath().

bool QDir::match (const char * filter, const char * fileName) `[static]`

Returns TRUE if the *fileName* matches the wildcard *filter*.

See also: QRegExp.

bool QDir::matchAllDirs () const

Returns the value set by setMatchAllDirs()

See also: setMatchAllDirs().

bool QDir::mkdir (const char * dirName, bool acceptAbsPath = TRUE) const

Creates a directory.

If *acceptAbsPath* is TRUE a path starting with a separator ('/') will create the absolute directory, if *acceptAbsPath* is FALSE any number of separators at the beginning of *dirName* will be removed.

Returns TRUE if successful, otherwise FALSE.

See also: rmdir().

const char * QDir::nameFilter () const

Returns the string set by setNameFilter()

See also: setNameFilter().

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

Returns TRUE if the *d* and this dir havee different path or different sort/filter settings, otherwise FALSE.

QDir & QDir::operator= (const QDir & d)

Makes a copy of d and assigns it to this QDir.

QDir & QDir::operator= (const char * path)

Sets the directory path to be the given path.

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

Returns TRUE if the *d* and this dir have the same path and all sort and filter settings are equal, otherwise FALSE.

const char * QDir::operator[] (int index) const

Returns the file name at position *index* in the list of found file names. Equivalent to entryList()->at(index).

Returns null if the *index* is out of range or if the entryList() function failed.

See also: count() and entryList().

const char * QDir::path () const

Returns the path, this may contain symbolic links, but never contains redundant ".", ".." or multiple separators.

The returned path can be either absolute or relative (see setPath()).

See also: setPath(), absPath(), exists(), cleanDirPath(), dirName(), absFilePath() and convertSeparators().

bool QDir::remove (const char * fileName, bool acceptAbsPath = TRUE)

Removes a file.

If *acceptAbsPath* is TRUE a path starting with a separator ('/') will remove the file with the absolute path, if *acceptAbsPath* is FALSE any number of separators at the beginning of *fileName* will be removed.

Returns TRUE if successful, otherwise FALSE.

bool QDir::rename (const char * name, const char * newName, bool acceptAbsPaths = TRUE)

Renames a file.

If *acceptAbsPaths* is TRUE a path starting with a separator ('/') will rename the file with the absolute path, if *acceptAbsPath* is FALSE any number of separators at the beginning of *name* will be removed.

Returns TRUE if successful, otherwise FALSE.

On most file systems, rename() fails only if oldName does not exist or if *newName* and *oldName* are not on the same partition, but there are also other reasons why rename() can fail. For example, on at least one file system rename() fails if newName points to an open file.

bool QDir::rmdir (const char * dirName, bool acceptAbsPath = TRUE) const

Removes a directory.

If *acceptAbsPath* is TRUE a path starting with a separator ('/') will remove the absolute directory, if *acceptAbsPath* is FALSE any number of separators at the beginning of *dirName* will be removed.

The directory must be empty for rmdir() to succeed.

Returns TRUE if successful, otherwise FALSE.

See also: mkdir().

QDir QDir::root () `[static]`

Returns the root directory.

See also: rootDirPath() and drives().

QString QDir::rootDirPath () `[static]`

Returns the absolute path for the root directory ("/" under UNIX).

See also: root() and drives().

char QDir::separator () `[static]`

Returns the native directory separator; '/' under UNIX and '\' under MS-DOS, Windows NT and OS/2.

You do not need to use this function to build file paths. If you always use '/', Qt will translate your paths to conform to the underlying operating system.

bool QDir::setCurrent (const char * path) `[static]`

Sets the the current directory. Returns TRUE if successful.

void QDir::setFilter (int filterSpec)

Sets the filter used by entryList() and entryInfoList(). The filter is used to specify the kind of files that should be returned by entryList() and entryInfoList(). The filter is specified by or-ing values from the enum FilterSpec. The different values are:

Dirs List directories only.

Files List files only.

Drives List drives.

NoSymLinks
Do not list symbolic links.

Readable
List files with read permission.

Writable
List files with write permission.

Executable
List files with execute permission.

Setting none of the three flags above is equivalent to setting all of them.

Modified
Only list files that have been modified (does nothing under UNIX).

Hidden
List hidden files also (.* under UNIX).

System
List system files (does nothing under UNIX).

See also: nameFilter().

void QDir::setMatchAllDirs (bool enable)

If *enable* is TRUE, all directories will be listed (even if they do not match the filter or the name filter), otherwise only matched directories will be listed.

See also: matchAllDirs().

Bugs and limitations:

- Currently, directories that do not match the filter will not be included (the name filter will be ignored as expected).

void QDir::setNameFilter (const char * nameFilter)

Sets the name filter used by entryList() and entryInfoList().

The name filter is a wildcarding filter that understands "*" and "?" wildcards, if you want entryList() and entryInfoList() to list all files ending with ".cpp", you simply call dir.setNameFilter("*.cpp");

See also: nameFilter().

void QDir::setPath (const char * path)

Sets the path of the directory. The path is cleaned of redundant ".", ".." and multiple separators. No check is made to ensure that a directory with this path exists.

The path can be either absolute or relative. Absolute paths begin with the directory separator ('/') or a drive specification (not applicable to UNIX). Relative file names begin with a directory name or a file name and specify a path relative to the current directory. An example of an absolute path is the string "/tmp/quartz", a relative path might look like "src/fatlib". You can use the function isRelative() to check if a QDir is using a relative or an absolute file path. You can call the function convertToAbs() to convert a relative QDir to an absolute one.

See also: path(), absPath(), exists, cleanDirPath(), dirName(), absFilePath(), isRelative() and convertToAbs().

void QDir::setSorting (int sortSpec)

Sets the sorting order used by entryList() and entryInfoList().

The *sortSpec* is specified by or-ing values from the enum SortSpec. The different values are:

One of these:

Name Sort by name (alphabetical order).

Time Sort by time (most recent first).

Size Sort by size (largest first).

Unsorted
Use the operating system order (UNIX does NOT sort alphabetically).

ORed with zero or more of these:

DirsFirst
: Always put directory names first.

Reversed
: Reverse sort order.

IgnoreCase
: Ignore case when sorting by name.

QDir::SortSpec QDir::sorting() const

Returns the value set by setSorting()

See also: setSorting().

51.2.1. URL

Generated from http://www.troll.no/qt/qdir.html.

52. QDoubleValidator Class Reference

`#include <qvalidator.h>`

Inherits QValidator.

52.1. Detailed Description

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

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

See also: QIntValidator.

52.2. Member Function Documentation

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

Creates a validator object which accepts all double from 2.7182818 to 3.1415926 (please, no bug reports) with at most seven digits after the decimal point.

This constructor is not meant to be useful; it is provided for completeness.

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

Creates a validator object which accepts all doubles from *bottom* up to and including *top* with at most *decimals* digits after the decimal point.

QDoubleValidator::~QDoubleValidator ()

Deletes the validator and frees any storage and other resources used.

double QDoubleValidator::bottom () const

Returns the lowest valid number according to this validator.

See also: top(), decimals() and setRange().

int QDoubleValidator::decimals () const

Returns the largest number of digits a valid number can have after its decimal point.

See also: bottom(), top() and setRange().

void QDoubleValidator::setRange (double bottom, double top, int decimals = 0) `[virtual]`

Sets the validator to accept numbers from *bottom* up to and including *top* with at most *decimals* digits after the decimal point.

double QDoubleValidator::top () const

Returns the highest valid number according to this validator.

See also: bottom(), decimals and setRange().

52.2.1. URL

Generated from http://www.troll.no/qt/qdoublevalidator.html.

53. QDragMoveEvent Class Reference

`#include <qevent.h>`

Inherits QEvent.

Inherited by QDragEnterEvent.

53.1. Detailed Description

Event sent as a drag-and-drop is in progress.

When a widgets accepts drop events, it will receive this event repeatedly while the the drag is inside that widget. The widget should examine the event, especially seeing what data it provides, and accept() the drop if appropriate.

53.2. Member Function Documentation

QDragMoveEvent::QDragMoveEvent (const QPoint & pos)

Creates a QDragMoveEvent for which the mouse is at point *pos.*

Note that internal state is also involved with QDragMoveEvent, so it is not useful to create these yourself.

QDragMoveEvent::QDragMoveEvent (const QPoint & pos, int type) `[protected]`

Creates a QDragMoveEvent for which the mouse is at point *pos,* and the given event *type.*

Note that internal state is also involved with QDragMoveEvent, so it is not useful to create these yourself.

void QDragMoveEvent::accept ()

Call this to indicate that the event provides data which your widget can process. Use provides(), or preferrably, the canDecode() methods of existing QDragObject subclasses, such as QTextDrag::canDecode(), or your own subclasses.

void QDragMoveEvent::accept (const QRect & r)

The same as accept() above, but also notifies that future moves will also be acceptable if they remain withing the rectangle *r* on the widget - this can improve performance, but may also be ignored by the underlying system.

QRect QDragMoveEvent::answerRect () const

{ return rect; }

Returns the rectangle for which the acceptance of the move event applies.

QByteArray QDragMoveEvent::data (const char * format)

Returns a byte array containing the payload data of this drag, in *format.*

data() normally needs to get the data from the drag source, which is potentially very slow, so it's advisable to call this function only if you're sure that you will need the data in *format.*

See also: format().

const char * QDragMoveEvent::format (int n = 0)

Returns a string describing one of the available data types for this drag. Common examples are "text/plain" and "image/gif". If *n* is less than zero or greater than the number of available data types, format() returns 0.

This function is provided mainly for debugging. Most drop targets will use provides().

See also: data() and provides().

void QDragMoveEvent::ignore ()

The opposite of accept().

void QDragMoveEvent::ignore (const QRect & r)

The opposite of accept(const QRect&).

bool QDragMoveEvent::isAccepted () const

Returns TRUE if accept() has been called.

const QPoint& QDragMoveEvent::pos () const

Returns the position of the mouse when the event occurred.

bool QDragMoveEvent::provides (const char * mimeType)

Returns TRUE if this drag object provides format *mimeType* or FALSE if it does not.

See also: data().

53.2.1. URL

Generated from http://www.troll.no/qt/qdragmoveevent.html.

54. QDragObject Class Reference

`#include <qdragobject.h>`

Inherits QObject.

Inherited by QImageDrag and QStoredDrag.

54.1. Detailed Description

The QDragObject encapsulates MIME-based drag-and-drop interaction.

Drag-and-drop in Qt uses the MIME open standard, to allow independently developers applications to exchange information.

To start a drag, for example in a mouse motion event, create an object of the QDragObject subclass appropriate for your media, such as QTextDrag for text and QImageDrag for images. Then call the drag() method. This is all you need for simple dragging of existing types.

To be able to receive media dropped on a widget, multiply-inherit the QDropSite class and override the dragEnterEvent(), dragMoveEvent(), dragLeaveEvent(), and dropEvent() event handler methods.

Support for specific media types is provided by subclasses of QDragObject. For example, QTextDrag provides support for the "`text/plain`" MIME type (ordinary unformated text), QImageDrag provides for "`image/*`", where `*` is all the image formats that Qt supports, and the QUrlDrag subclass provides "`url/url`", a standard format for transfering a list of filenames.

To implement drag-and-drop of some type of media for which there is no available QDragObject subclass, the first and most important step is to look for existing formats that are appropriate - the Internet Assigned Numbers Authority (IANA) provides a hierarchical list of MIME media types at the Information Sciences Institute (ISI). This maximizes inter-operability of your software.

To support an additional media type, subclass either QDragObject or QStoredDrag. Subclass QDragObject when you need to provide support for multiple media types. Subclass the simpler QStoredDrag when one type is sufficient.

Subclasses of QDragObject will override the format() and encodedData() members, and provide a set-method to encode the media data and static members canDecode() and decode() to decode incoming data. QImageDrag is an example of such a class in Qt. Of course, you can provide drag-only or drop-only support for a media type by omitting some of these methods.

Subclasses of QStoredDrag provide a set-method to encode the media data and static members canDecode() and decode() to decode incoming data. QTextDrag in Qt 1.x is an example of such a class in Qt.

Inter-operating with existing applications

On X11, the public XDND protocol is used, while on Windows Qt uses the OLE standard. On X11, XDND uses MIME, so no translation is necessary. On Windows, MIME-aware applications can communicate by using clipboard format names that are MIME types. Internally, Qt has facilities for translating all proprietary clipboard formats to and from MIME. This interface will be made public at some time, but if you need to do such translations now, contact your Qt Technical Support service.

54.2. Member Function Documentation

QDragObject::QDragObject (QWidget * dragSource = 0, const char * name = 0)

Creates a drag object which is a child of *dragSource* and named *name.*

Note that the drag object will be deleted when *dragSource* is.

QDragObject::~QDragObject ()

Deletes the drag object and frees up the storage used.

bool QDragObject::drag ()

Starts a drag operation using the contents of this object, using DragDefault mode.

See drag(DragMove) for important further information.

bool QDragObject::drag (DragMode mode) `[virtual protected]`

Starts a drag operation using the contents of this object.

At this point, the object becomes owned by Qt, not the application. You should not delete the drag object nor anything it references. The actual transfer of data to the target application will be done during future event processing - after that time the drag object will be deleted.

Returns TRUE if the dragged data was dragged as a *move,* indicating that the caller should remove the original source of the data (the drag object must continue to have a copy).

The *mode* is one of:

- `DragDefault` - the mode is determined heuristically.
- `DragCopy` - the data is copied, never moved.
- `DragMove` - the data is moved, if dragged at all.
- `DragCopyOrMove` - the user chooses the mode by using control key to switch from the default.

Normally one of simpler drag(), dragMove(), or dragCopy() functions would be used instead.

void QDragObject::dragCopy ()

Starts a drag operation using the contents of this object, using DragCopy mode.

See drag(DragMove) for important further information.

bool QDragObject::dragMove ()

Starts a drag operation using the contents of this object, using DragMove mode.

See drag(DragMove) for important further information.

QByteArray QDragObject::encodedData (const char *) const `[virtual]`

Returns the encoded payload of this object. The drag manager calls this when the recipient needs to see the content of the drag; this generally doesn't happen until the actual drop.

Subclasses must override this function.

const char * QDragObject::format (int i) const `[virtual]`

Returns the *ith* format, or NULL.

QPixmap QDragObject::pixmap () const

Returns the currently set pixmap (which isNull() if none is set).

QPoint QDragObject::pixmapHotSpot () const

Returns the currently set pixmap hitspot.

bool QDragObject::provides (const char * mimeType) const `[virtual]`

Returns TRUE if the drag object can provide the data in format *mimeType*. The default implementation iterates over format().

void QDragObject::setPixmap (QPixmap pm, QPoint hotspot)

Set the pixmap *pm* to display while dragging the object. The platform-specific implementation will use this in a loose fashion - so provide a small masked pixmap, but do not require that the user ever sees it in all its splendour. In particular, cursors on Windows 95 are of limited size.

The *hotspot* is the point on (or off) the pixmap that should be under the cursor as it is dragged. It is relative to the top-left pixel of the pixmap.

void QDragObject::setPixmap (QPixmap pm)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Uses a hotspot that positions the pixmap below and to the right of the mouse pointer. This allows the user to clearly see the point on the window which they are dragging the data onto.

QWidget * QDragObject::source ()

Returns a pointer to the drag source where this object originated.

54.2.1. URL

Generated from http://www.troll.no/qt/qdragobject.html.

55. QDropEvent Class Reference

`#include <qevent.h>`

Inherits QEvent.

55.1. Detailed Description

Event sent when a drag-and-drop is completed.

When a widgets accepts drop events, it will receive this event if it has accepted the most recent QDragEnterEvent or QDragMoveEvent sent to it.

The widget should use data() to extract data in an appropriate format.

55.2. Member Function Documentation

QDropEvent::QDropEvent (const QPoint & pos)

Creates a QDropEvent for which the mouse is at point *pos.*

Note that internal state is also involved with QDropEvent, so it is not useful to create these yourself.

void QDropEvent::accept ()

Call this to indicate that the event provided data which your widget processed. Use data(), or preferrably, the decode() methods of existing QDragObject subclasses, such as QTextDrag::decode(), or your own subclasses.

QByteArray QDropEvent::data (const char * format)

Returns a byte array containing the payload data of this drag, in *format.*

data() normally needs to get the data from the drag source, which is potentially very slow, so it's advisable to call this function only if you're sure that you will need the data in *format.*

The resulting data will have a size of 0 if the format was not available.

See also: QDragMoveEvent::data() and QDragMoveEvent::format().

void QDropEvent::ignore ()

The opposite of accept().

55.2.1. URL

Generated from http://www.troll.no/qt/qdropevent.html.

56. QEvent Class Reference

`#include <qevent.h>`

Inherited by QChildEvent, QCloseEvent, QCustomEvent, QDragLeaveEvent, QDragMoveEvent, QDragResponseEvent, QDropEvent, QFocusEvent, QHideEvent, QKeyEvent, QMouseEvent, QMoveEvent, QPaintEvent, QResizeEvent, QShowEvent and QTimerEvent.

56.1. Detailed Description

The QEvent class is base class of all event classes. Event objects contain event parameters.

The main event loop of Qt fetches native window system events from the event queue, translates them into QEvent and sends the translated events to QObjects.

Generally, events come from the underlying window system, but it is also possible to manually send events through the QApplication class using QApplication::sendEvent() and QApplication::postEvent().

QObject recevied events by having its QObject::event() function called. The default implementation simply calls QObject::timerEvent() for timer events and ignores all other events. QWidget reimplements event() and dispatches incoming events to various event handles such as QWidget::keyPressEvent() and QWidget::mouseMoveEvent().

The basic QEvent only contains an event type parameter. Subclasses of QEvent contain additional parameters that descripe the particular event.

See also: QObject::event(), QObject::installEventFilter(), QWidget::event(), QApplication::sendEvent(), QApplcation::postEvent() and QApplication::processEvents().

56.2. Member Function Documentation

QEvent::QEvent (int type)

Contructs an event object with a *type.* The file qevent.h lists all event types.

QEvent::~QEvent ()

Destroys the event. Reports an error if the event has been posted.

int QEvent::type () const

Returns the event type.

56.2.1. URL

Generated from http://www.troll.no/qt/qevent.html.

57. QFile Class Reference

```
#include <qfile.h>
```

Inherits QIODevice.

57.1. Detailed Description

The QFile class is an I/O device that operates on files.

QFile is an I/O device for reading and writing binary and text files. A QFile may be used by itself (readBlock and writeBlock) or by more conveniently using QDataStream or QTextStream.

Here is a code fragment that uses QTextStream to read a text file line by line. It prints each line with a line number.

```
QFile f("file.txt");
if ( f.open(IO_ReadOnly) ) {    // file opened successfully
    QTextStream t( &f );        // use a text stream
    QString s;
    int n = 1;
    while ( !t.eof() ) {        // until end of file...
        s = t.readLine();       // line of text excluding '\n'
        printf( "%3d: %s\n", n++, (const char *)s );
    }
    f.close();
}
```

The QFileInfo class holds detailed information about a file, such as access permissions, file dates and file types.

The QDir class manages directories and lists of file names.

See also: QDataStream and QTextStream.

Examples:

application/application.cpp

57.2. Member Function Documentation

QFile::QFile ()

Constructs a QFile with no name.

QFile::QFile (const char * name)

Constructs a QFile with a file name *name.*

See also: setName().

QFile::~QFile ()

Destroys a QFile. Calls close().

bool QFile::at (int pos) `[virtual]`

Sets the file index to *pos.* Returns TRUE if successful, otherwise FALSE.

Example:

```
QFile f( "data.bin" );
f.open( IO_ReadOnly );                       // index set to 0
```

```
f.at( 100 );                          // set index to 100
f.at( f.at()+50 );                    // set index to 150
f.at( f.size()-80 );                  // set index to 80 before EOF
f.close();
```

Warning: The result is undefined if the file was opened using the `IO_Append` specifier.

See also: size() and open().

Reimplemented from QIODevice.

int QFile::at () const `[virtual]`

Returns the file index.

See also: size().

Reimplemented from QIODevice.

bool QFile::atEnd () const `[virtual]`

Returns TRUE if the end of file has been reached, otherwise FALSE.

See also: size().

Reimplemented from QIODevice.

void QFile::close () `[virtual]`

Closes an open file.

The file is closed even if it was opened with an existing file handle or a file descriptor, *except* that stdin, stdout and stderr are never closed.

See also: open() and flush().

Examples:
application/application.cpp

Reimplemented from QIODevice.

bool QFile::exists () const

Returns TRUE if the file exists, otherwise FALSE.

See also: name().

bool QFile::exists (const char * fileName) `[static]`

Returns TRUE if the file given by *fileName* exists, otherwise FALSE.

void QFile::flush () `[virtual]`

Flushes the file buffer to the disk.

close() also flushes the file buffer.

Reimplemented from QIODevice.

int QFile::getch () `[virtual]`

Reads a single byte/character from the file.

Returns the byte/character read, or -1 if the end of the file has been reached.

See also: putch() and ungetch().

Reimplemented from QIODevice.

int QFile::handle () const

Returns the file handle of the file.

This is a small positive integer, suitable for use with C library functions such as fdopen() and fcntl(), as well as with QSocketNotifier.

If the file is not open or there is an error, handle() returns -1.

See also: QSocketNotifier.

const char * QFile::name () const

Returns the name set by setName().

See also: setName() and QFileInfo::fileName().

bool QFile::open (int m) `[virtual]`

Opens the file specified by the file name currently set, using the mode *m*. Returns TRUE if successful, otherwise FALSE.

The mode parameter *m* must be a combination of the following flags:

- `IO_Raw` specified raw (non-buffered) file access.
- `IO_ReadOnly` opens the file in read-only mode.
- `IO_WriteOnly` opens the file in write-only mode (and truncates).
- `IO_ReadWrite` opens the file in read/write mode, equivalent to `(IO_ReadOnly|IO_WriteOnly).`
- `IO_Append` opens the file in append mode. This mode is very useful when you want to write something to a log file. The file index is set to the end of the file. Note that the result is undefined if you position the file index manually using at() in append mode.
- `IO_Truncate` truncates the file.
- `IO_Translate` enables carriage returns and linefeed translation for text files under MS-DOS, Windows and OS/2.

The raw access mode is best when I/O is block-operated using 4kB block size or greater. Buffered access works better when reading small portions of data at a time.

Important: When working with buffered files, data may not be written to the file at once. Call flush to make sure the data is really written.

Warning: We have experienced problems with some C libraries when a buffered file is opened for both reading and writing. If a read operation takes place immediately after a write operation, the read buffer contains garbage data. Worse, the same garbage is written to the file. Calling flush() before readBlock() solved this problem.

If the file does not exist and `IO_WriteOnly` or `IO_ReadWrite` is specified, it is created.

Example:

```
QFile f1( "/tmp/data.bin" );
QFile f2( "readme.txt" );
f1.open( IO_Raw | IO_ReadWrite | IO_Append );
f2.open( IO_ReadOnly | IO_Translate );
```

See also: name(), close(), isOpen() and flush().

Examples:
application/application.cpp

Reimplemented from QIODevice.

bool QFile::open (int m, FILE * f)

Opens a file in the mode *m* using an existing file handle *f*. Returns TRUE if successful, otherwise FALSE.

Example:

```
#include <stdio.h>
void printError( const char *msg )
{
    QFile f;
    f.open( IO_WriteOnly, stderr );
    f.writeBlock( msg, strlen(msg) );       // write to stderr
    f.close();
}
```

When a QFile is opened using this function, close() does not actually close the file, only flushes it.

Warning: If *f* is `stdin, stdout, stderr,` you may not be able to seek. See QIODevice::isSequentialAccess() for more information.

See also: close().

bool QFile::open (int m, int f)

Opens a file in the mode *m* using an existing file descriptor *f*. Returns TRUE if successful, otherwise FALSE.

When a QFile is opened using this function, close() does not actually close the file.

Warning: If *f* is one of 0 (stdin), 1 (stdout) or 2 (stderr), you may not be able to seek. size() is set to `INT_MAX` (in limits.h).

See also: close().

int QFile::putch (int ch) `[virtual]`

Writes the character *ch* to the file.

Returns *ch,* or -1 if some error occurred.

See also: getch() and ungetch().

Reimplemented from QIODevice.

int QFile::readBlock (char * p, uint len) `[virtual]`

Reads at most *len* bytes from the file into *p* and returns the number of bytes actually read.

Returns -1 if a serious error occurred.

Warning: We have experienced problems with some C libraries when a buffered file is opened for both reading and writing. If a read operation takes place immediately after a write operation, the read buffer contains garbage data. Worse, the same garbage is written to the file. Calling flush() before readBlock() solved this problem.

See also: writeBlock().

Reimplemented from QIODevice.

int QFile::readLine (char * p, uint maxlen) `[virtual]`

Reads a line of text.

Reads bytes from the file until end-of-line is reached, or up to *maxlen* bytes, and returns the number of bytes read, or -1 in case of error. The terminating newline is not stripped.

This function is efficient only for buffered files. Avoid readLine() for files that have been opened with the `IO_Raw` flag.

See also: readBlock() and QTextStream::readLine().

Reimplemented from QIODevice.

bool QFile::remove ()

Removes the file specified by the file name currently set. Returns TRUE if successful, otherwise FALSE.

The file is closed before it is removed.

bool QFile::remove (const char * fileName) `[static]`

Removes the file *fileName.* Returns TRUE if successful, otherwise FALSE.

void QFile::setName (const char * name)

Sets the name of the file. The name can include an absolute directory path or it can be a name or a path relative to the current directory.

Do not call this function if the file has already been opened.

Note that if the name is relative QFile does not associate it with the current directory. If you change directory before calling open(), open uses the new current directory.

Example:

```
QFile f;
QDir::setCurrent( "/tmp" );
f.setName( "readme.txt" );
QDir::setCurrent( "/home" );
f.open( IO_ReadOnly );        // opens "/home/readme.txt" under UNIX
```

Also note that the directory separator '/' works for all operating systems supported by Qt.

See also: name(), QFileInfo and QDir.

uint QFile::size () const `[virtual]`

Returns the file size.

See also: at().

Reimplemented from QIODevice.

int QFile::ungetch (int ch) `[virtual]`

Puts the character *ch* back into the file and decrements the index if it is not zero.

This function is normally called to "undo" a getch() operation.

Returns *ch,* or -1 if some error occurred.

See also: getch() and putch().

Reimplemented from QIODevice.

int QFile::writeBlock (const char * p, uint len) `[virtual]`

Writes *len* bytes from *p* to the file and returns the number of bytes actually written.

Returns -1 if a serious error occurred.

Important: When working with buffered files, data may not be written to the file at once. Call flush to make sure the data is really written.

See also: readBlock().

Reimplemented from QIODevice.

57.2.1. URL

Generated from http://www.troll.no/qt/qfile.html.

58. QFileDialog Class Reference

`#include <qfiledialog.h>`

Inherits QDialog.

58.1. Detailed Description

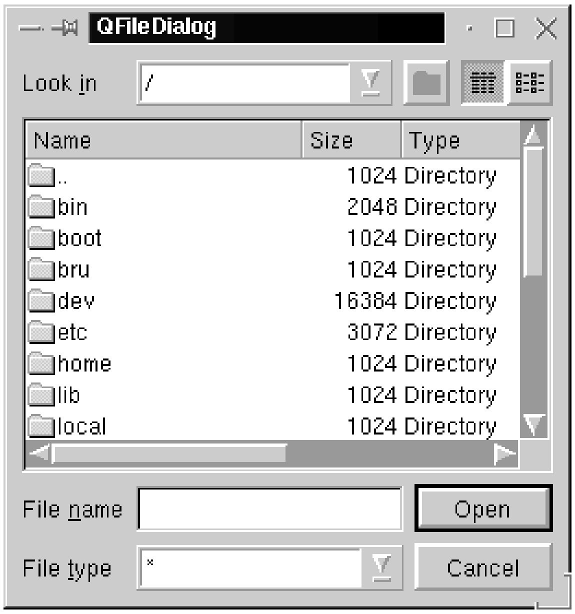
The QFileDialog provides a dialog widget for inputting file names.

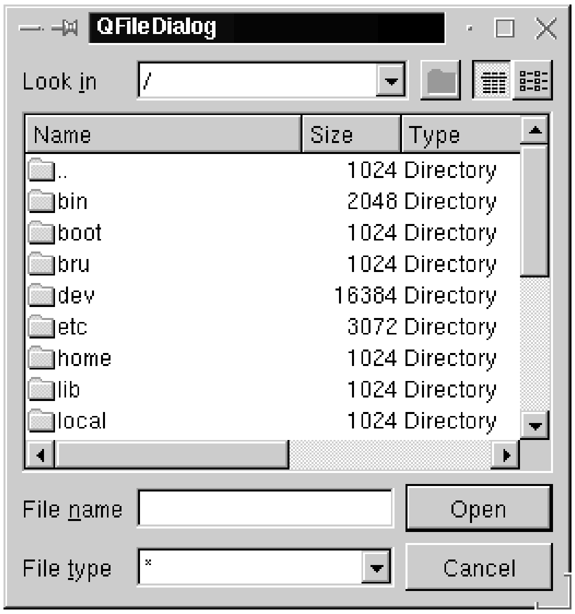
Example:

```
QString fileName = QFileDialog::getOpenFileName();
if ( !fileName.isNull() ) {                    // got a file name
    ...
}
```

There are two ready-made convenience functions, getOpenFileName() and getSaveFileName(), which may be used like this:

```
QString s( QFileDialog::getOpenFileName() );
if ( s.isNull() )
    return;
open( s ); // open() being your function to read the file
```





See also: QPrintDialog.

Examples:
movies/main.cpp

58.2. Member Function Documentation

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

Constructs a file dialog with a *parent, name* and *modal* flag.

The dialog becomes modal if *modal* is TRUE, otherwise modeless.

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

Constructs a file dialog with a *parent, name* and *modal* flag.

The dialog becomes modal if *modal* is TRUE, otherwise modeless.

QFileDialog::~QFileDialog ()

Destroys the file dialog.

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

Adds 1-3 widgets to the bottom of the file dialog. *l* is the (optional) label, which is put beneath the "file name" and "file type" labels, *w* is a (optional) widget, which is put beneath the file type combo box, and *b* is the (you guessed it - optional) button, which is put beneath the cancel button.

If you don’t want to add something in one of the columns, pass 0.

It is not currently possible to add more than one row.

const QDir * QFileDialog::dir () const

Returns the active directory in the file dialog.

See also: setDir().

void QFileDialog::dirEntered (const char *) `[signal]`

This signal is emitted when the user has selected a new directory.

const char * QFileDialog::dirPath () const

Returns the active directory path string in the file dialog.

See also: dir() and setDir().

bool QFileDialog::eventFilter (QObject * o, QEvent * e) `[virtual]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QObject.

void QFileDialog::fileHighlighted (const char *) `[signal]`

This signal is emitted when the user highlights a file.

void QFileDialog::fileSelected (const char *) `[signal]`

This signal is emitted when the user selects a file.

QString QFileDialog::getExistingDirectory (const char * dir = 0, QWidget * parent = 0, const char * name = 0) `[static]`

Ask the user for the name of an existing directory, starting at *dir*. Returns the name of the directory the user selected.

If *dir* is null, getExistingDirectory() starts wherever the previous file dialog left off.

QString QFileDialog::getOpenFileName (const char * startWith = 0, const char * filter= 0, QWidget * parent = 0, const char * name = 0) `[static]`

Opens a modal file dialog and returns the name of the file to be opened.

If *startWith* is the name of a directory, the dialog starts off in that directory. If *startWith* is the name of an existing file, the dialogs starts in that directory, and with *startWith* selected.

Only files matching *filter* are selectable. If *filter* is 0, all files are selectable.

If *widget* and/or *name* is provided, the dialog will be centered over *widget* and named *name.*

getOpenFileName() returns a null string if the user cancelled the dialog.

This static function is less capable than the full QFileDialog object, but is convenient and easy to use.

Example:

```
// start at the current working directory and with *.cpp as filter
QString f = QFileDialog::getOpenFileName( 0, "*.cpp", this );
if ( !f.isEmpty() ) {
```

```
        // the user selected a valid existing file
    } else {
        // the user cancelled the dialog
    }
```

getSaveFileName() is another convenience function, equal to this one except that it allows the user to specify the name of a nonexistent file name.

See also: getSaveFileName().

QStrList QFileDialog::getOpenFileNames (const char * filter= 0, const char * dir = 0, QWidget * parent = 0, const char * name = 0) `[static]`

Lets the user select N files from a single directory, and returns a list of the selected files. The list may be empty, and the file names are fully qualified (i.e. "/usr/games/quake" or "c:\\quake\\quake").

filter is the default glob pattern (which the user can change). The default is all files. *dir* is the starting directory. If *dir* is not supplied, QFileDialog picks something presumably useful (such as the directory where the user selected something last, or the current working directory).

parent is a widget over which the dialog should be positioned and *name* is the object name of the temporary QFileDialog object.

Note that the returned list has auto-delete turned off. It is the application developer's responsibility to delete the strings in the list, for example using code such as:

```
    QStrList s( QFileDialog::getOpenFileNames() );
    // do something with the files in s.
    s.setAutoDelete();
    s.clear(); // or just go out of scope.
```

QString QFileDialog::getSaveFileName (const char * startWith = 0, const char * filter= 0, QWidget * parent = 0, const char * name = 0) `[static]`

Opens a modal file dialog and returns the name of the file to be saved.

If *startWith* is the name of a directory, the dialog starts off in that directory. If *startWith* is the name of an existing file, the dialogs starts in that directory, and with *startWith* selected.

Only files matching *filter* are selectable. If *filter* is 0, all files are selectable.

If *widget* and/or *name* is provided, the dialog will be centered over *widget* and named *name.*

Returns a null string if the user cancelled the dialog.

This static function is less capable than the full QFileDialog object, but is convenient and easy to use.

Example:

```
    // start at the current working directory and with *.cpp as filter
    QString f = QFileDialog::getSaveFileName( 0, "*.cpp", this );
    if ( !f.isEmpty() ) {
        // the user gave a file name
    } else {
        // the user cancelled the dialog
    }
```

getOpenFileName() is another convenience function, equal to this one except that it does not allow the user to specify the name of a nonexistent file name.

See also: getOpenFileName().

QFileIconProvider * QFileDialog::iconProvider () `[static]`

Returns the icon provider currently in use. By default there is no icon provider and this function returns 0.

See also: setIconProvider() and QFileIconProvider.

void QFileDialog::keyPressEvent (QKeyEvent * ke) `[virtual protected]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QWidget.

QFileDialog::Mode QFileDialog::mode() const

Returns the file mode of this dialog.

See also: setMode().

void QFileDialog::rereadDir ()

Re-reads the active directory in the file dialog.

It is seldom necessary to call this function. It is provided in case the directory contents change and you want to refresh the directory list box.

void QFileDialog::resizeEvent (QResizeEvent *) `[virtual protected]`

Handles resize events for the file dialog.

Reimplemented from QWidget.

QString QFileDialog::selectedFile () const

Returns the selected file name.

If a file name was selected, the returned string contains the absolute path name. The returned string is a null string if no file name was selected.

See also: QString::isNull().

void QFileDialog::setDir (const QDir & dir)

Sets a directory path for the file dialog.

See also: dir().

void QFileDialog::setDir (const char * pathstr) `[slot]`

Sets a directory path string for the file dialog.

See also: dir().

void QFileDialog::setFilter (const char * newFilter) `[slot]`

Sets the filter spec in use to *newFilter*.

If *newFilter* matches the regular expression `([a-zA-Z0-9\.\*\?]*)$` (ie. it ends with a normal wildcard expression enclosed in parentheses), only the parenthesized is used. This means that these two calls are equivalent:

```
fd->setFilter( "All perl files (*.pl)" );
fd->setFilter( "*.pl" )
```

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

Sets this file dialog to offer *types* in the File Type combo box. *types* must be a null-terminated list of strings; each string must be in the format described in the documentation for setFilter().

See also: setFilter().

void QFileDialog::setFilters (const QStrList &) `[slot]`

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

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

Sets all file dialogs to use *provider* to select icons to draw for each file. By default there is no icon provider, and QFileDialog simply draws a "folder" icon next to each directory and nothing next to the files.

See also: QFileIconProvider and iconProvider().

void QFileDialog::setMode (Mode newMode)

Sets this file dialog to *newMode,* which can be one of `Directory` (directories are accepted), `ExistingFile` (existing files are accepted) or `AnyFile` (any valid file name is accepted).

See also: mode().

void QFileDialog::setSelection (const char * filename)

Sets the default selection to *filename.* If *filename* is absolute, setDir() is also called.

58.2.1. URL

Generated from http://www.troll.no/qt/qfiledialog.html.

59. QFileIconProvider Class Reference

`#include <qfiledialog.h>`

Inherits QObject.

59.1. 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's advisable to make all the icons QFileIconProvider returns be of the same size, or at least the same width. This makes the list view look much better.

See also: QFileDialog.

59.2. Member Function Documentation

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

Constructs an empty file icon provider.

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

Returns a pointer to a pixmap suitable for display when the file dialog next to the name of *file.*

If pixmap() returns 0, QFileDialog draws nothing.

The default implementation returns 0 in Qt 1.40. In future versions of Qt it may be extended.

59.2.1. URL

Generated from http://www.troll.no/qt/qfileiconprovider.html.

60. QFileInfo Class Reference

```
#include <qfileinfo.h>
```

60.1. Detailed Description

The QFileInfo class provides system-independent file information.

QFileInfo provides information about a file's name and position (path) in the file system, its access rights and whether it is a directory or a symbolic link. Its size and last modified/read times are also available.

To speed up performance QFileInfo caches information about the file. Since files can be changed by other users or programs, or even by other parts of the same program there is a function that refreshes the file information; refresh(). If you would rather like a QFileInfo to access the file system every time you request information from it, you can call the function setCaching(FALSE).

A QFileInfo can point to a file using either a relative or an absolute file path. Absolute file paths begin with the directory separator ('/') or a drive specification (not applicable to UNIX). Relative file names begin with a directory name or a file name and specify a path relative to the current directory. An example of an absolute path is the string "/tmp/quartz". A relative path might look like "src/fatlib". You can use the function isRelative() to check if a QFileInfo is using a relative or an absolute file path. You can call the function convertToAbs() to convert a relative QFileInfo to an absolute one.

If you need to read and traverse directories, see the QDir class.

Examples:
dirview/dirview.cpp
biff/biff.cpp

60.2. Member Function Documentation

QFileInfo::QFileInfo ()

Constructs a new empty QFileInfo.

QFileInfo::QFileInfo (const QDir & d, const char * fileName)

Constructs a new QFileInfo that gives information about the file named *fileName* in the directory *d.*

If the directory has a relative path, the QFileInfo will also have one.

See also: isRelative().

QFileInfo::QFileInfo (const QFile & file)

Constructs a new QFileInfo that gives information about *file.*

If the file has a relative path, the QFileInfo will also have one.

See also: isRelative().

QFileInfo::QFileInfo (const QFileInfo & fi)

Constructs a new QFileInfo that is a copy of *fi.*

QFileInfo::QFileInfo (const char * file)

Constructs a new QFileInfo that gives information about the given file. The string given can be an absolute or a relative file path.

See also: bool, setFile(const, char*), isRelative(), QDir::setCurrent() and QDir::isRelativePath().

QFileInfo::˜QFileInfo ()

Destroys the QFileInfo.

QString QFileInfo::absFilePath () const

Returns the absolute path name.

The absolute path name is the file name including the absolute path. If the QFileInfo is absolute (i.e. not relative) this function will return the same string as filePath().

Note that this function can be time-consuming under UNIX. (in the order of milliseconds on a 486 DX2/66 running Linux).

See also: isRelative() and filePath().

Examples:
biff/biff.cpp

QString QFileInfo::baseName () const

Returns the base name of the file.

The base name consists of all characters in the file name up to (but not including) the first '.' character. The path is not included.

Example:

```
QFileInfo fi( "/tmp/abdomen.lower" );
QString base = fi.baseName();              // base = "abdomen"
```

See also: fileName() and extension().

bool QFileInfo::caching () const

Returns TRUE if caching is enabled.

See also: setCaching() and refresh().

bool QFileInfo::convertToAbs ()

Converts the file path name to an absolute path.

If it is already absolute nothing is done.

See also: filePath() and isRelative().

QDir QFileInfo::dir (bool absPath = FALSE) const

Returns the directory path of the file.

If the QFileInfo is relative and *absPath* is FALSE, the QDir will be relative, otherwise it will be absolute.

See also: dirPath(), filePath(), fileName() and isRelative().

QString QFileInfo::dirPath (bool absPath = FALSE) const

Returns the directory path of the file.

If *absPath* is TRUE an absolute path is always returned.

See also: dir(), filePath(), fileName() and isRelative().

bool QFileInfo::exists () const

Returns TRUE if the file pointed to exists, otherwise FALSE.

Examples:
biff/biff.cpp

QString QFileInfo::extension () const

Returns the extension name of the file.

The file name extension consists of all characters in the file name after (but not including) the first '.' character.

For a file named "archive.tar.gz" this function will return "tar.gz".

Example:

```
QFileInfo fi( "/tmp/abdomen.lower" );
QString ext = fi.extension();              // ext = "lower"
```

See also: fileName() and baseName().

QString QFileInfo::fileName () const

Returns the name of the file, the file path is not included.

Example:

```
QFileInfo fi( "/tmp/abdomen.lower" );
QString name = fi.fileName();              // name = "abdomen.lower"
```

See also: isRelative(), filePath(), baseName() and extension().

Examples:
dirview/dirview.cpp

const char * QFileInfo::filePath () const

Returns the name, i.e. the file name including the path (which can be absolute or relative).

See also: isRelative() and absFilePath().

const char * QFileInfo::group () const

Returns the group the file belongs to.

On systems where files do not have groups this function always returns 0.

Note that this function can be time-consuming under UNIX (in the order of milliseconds on a 486 DX2/66 running Linux).

See also: groupId(), owner() and ownerId().

uint QFileInfo::groupId () const

Returns the id of the group the file belongs to.

On systems where files do not have groups this function always returns ((uind) -2).

See also: group(), owner() and ownerId().

bool QFileInfo::isDir () const

Returns TRUE if we are pointing to a directory or a symbolic link to a directory.

See also: isFile() and isSymLink().

Examples:
dirview/dirview.cpp

bool QFileInfo::isExecutable () const

Returns TRUE if the file is executable.

See also: isReadable(), isWritable() and permission().

bool QFileInfo::isFile () const

Returns TRUE if we are pointing to a file or a symbolic link to a file.

See also: isDir() and isSymLink().

Examples:
dirview/dirview.cpp

bool QFileInfo::isReadable () const

Returns TRUE if the file is readable.

See also: isWritable(), isExecutable() and permission().

bool QFileInfo::isRelative () const

Returns TRUE if the file path name is relative to the current directory, FALSE if the path is absolute (e.g. under UNIX a path is relative if it does not start with a '/').

According to Einstein this function should always return TRUE.

bool QFileInfo::isSymLink () const

Returns TRUE if we are pointing to a symbolic link.

See also: isFile(), isDir() and readLink().

Examples:
dirview/dirview.cpp

bool QFileInfo::isWritable () const

Returns TRUE if the file is writable.

See also: isReadable(), isExecutable() and permission().

QDateTime QFileInfo::lastModified () const

Returns the date and time when the file was last modified.

See also: lastRead().

Examples:
biff/biff.cpp

QDateTime QFileInfo::lastRead () const

Returns the date and time when the file was last read (accessed).

On systems that do not support last read times, the modification time is returned.

See also: lastModified().

Examples:
biff/biff.cpp

QFileInfo & QFileInfo::operator= (const QFileInfo & fi)

Makes a copy of *fi* and assigns it to this QFileInfo.

const char * QFileInfo::owner () const

Returns the owner of the file.

On systems where files do not have owners this function returns 0.

Note that this function can be time-consuming under UNIX. (in the order of milliseconds on a 486 DX2/66 running Linux).

See also: ownerId(), group() and groupId().

uint QFileInfo::ownerId () const

Returns the id of the owner of the file.

On systems where files do not have owners this function returns ((uint) -2).

See also: owner(), group() and groupId().

bool QFileInfo::permission (int permissionSpec) const

Tests for file permissions. The *permissionSpec* argument can be several flags of type PermissionSpec or'ed together to check for permission combinations.

On systems where files do not have permissions this function always returns TRUE.

Example:

```
QFileInfo fi( "/tmp/tonsils" );
if ( fi.permission( QFileInfo::WriteUser | QFileInfo::ReadGroup ) )
    warning( "Tonsils can be changed by me, and the group can read them.");
if ( fi.permission( QFileInfo::WriteGroup | QFileInfo::WriteOther ) )
    warning( "Danger! Tonsils can be changed by the group or others!" );
```

See also: isReadable(), isWritable() and isExecutable().

QString QFileInfo::readLink () const

Returns the name a symlink points to, or a null QString if the object does not refer to a symbolic link.

This name may not represent an existing file; it is only a string. QFileInfo::exists() returns TRUE if the symlink points to an existing file.

See also: exists(), isSymLink(), isDir() and isFile().

void QFileInfo::refresh () const

Refresh the information about the file, i.e. read in information from the file system the next time a cached property is fetched.

See also: setCaching().

void QFileInfo::setCaching (bool enable)

Enables caching of file information if *enable* is TRUE, or disables it if *enable* is FALSE.

When caching is enabled, QFileInfo reads the file information the first time

Caching is enabled by default.

See also: refresh() and caching().

void QFileInfo::setFile (const QDir & d, const char * fileName)

Sets the file to obains information about to *fileName* in the directory *d.*

If the directory has a relative path, the QFileInfo will also have one.

See also: isRelative().

void QFileInfo::setFile (const QFile & file)

Sets the file to obtain information about.

If the file has a relative path, the QFileInfo will also have one.

See also: isRelative().

void QFileInfo::setFile (const char * file)

Sets the file to obtain information about.

The string given can be an absolute or a relative file path. Absolute file paths begin with the directory separator (e.g. '/' under UNIX) or a drive specification (not applicable to UNIX). Relative file names begin with a directory name or a file name and specify a path relative to the current directory.

Example:

```
#include <qfileinfo.h>
#include <qdir.h>
void test()
{
    const char *absolute = "/liver/aorta";
    const char *relative = "liver/aorta";
    QFileInfo fi1( absolute );
    QFileInfo fi2( relative );
    QDir::setCurrent( QDir::rootDirPath() );
```

```
                                // fi1 and fi2 now point to the same file
        QDir::setCurrent( "/tmp" );
                                // fi1 now points to "/liver/aorta",
                                // while fi2 points to "/tmp/liver/aorta"
    }
```

See also: isRelative(), QDir::setCurrent() and QDir::isRelativePath().

Examples:
biff/biff.cpp

uint QFileInfo::size () const

Returns the file size in bytes, or 0 if the file does not exist if the size cannot be fetched.

60.2.1. URL

Generated from http://www.troll.no/qt/qfileinfo.html.

61. QFocusData Class Reference

```
#include <qfocusdata.h>
```

61.1. Detailed Description

Maintains the list of widgets which can take focus.

This read-only list always contains at least one widget (the top-level widget, actually). It provides a simple cursor, which can be reset to the current focus widget using home(), or moved to its neighboring widgets using next() and prev(), and a count() of widgets in the list.

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

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

The cursor may change at any time; this class is not thread-safe.

See also: QWidget::focusNextPrevChild(), QWidget::setTabOrder() and QWidget::setFocusPolicy().

61.2. Member Function Documentation

int QFocusData::count () const

Returns a count of the number of widgets in the hierarchy which accept focus.

QWidget* QFocusData::focusWidget () const

Returns the widgets in the hierarchy which currently has focus.

QWidget* QFocusData::home ()

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

QWidget* QFocusData::next ()

Moves the cursor to the right. Note that the focus widgets are a *loop* of widgets. If you keep calling next(), it will loop, without ever returning 0.

QWidget* QFocusData::prev ()

Moves the cursor to the left. Note that the focus widgets are a *loop* of widgets. If you keep calling prev(), it will loop, without ever returning 0.

61.2.1. URL

Generated from http://www.troll.no/qt/qfocusdata.html.

62. QFocusEvent Class Reference

`#include <qevent.h>`

Inherits QEvent.

62.1. Detailed Description

The QFocusEvent class contains event parameters for widget focus events.

Focus events are sent to widgets when the keyboard input focus changes.

The event handlers QWidget::focusInEvent() and QWidget::focusOutEvent() receive focus events.

See also: QWidget::setFocus() and QWidget::setFocusEnabled().

Examples:

table/table.cpp
qmag/qmag.cpp

62.2. Member Function Documentation

QFocusEvent::QFocusEvent (int type)

Constructs a focus event object.

The *type* parameter must be either *Event_FocusIn* or *Event_FocusOut*.

bool QFocusEvent::gotFocus () const

Returns TRUE if the widget received the text input focus.

bool QFocusEvent::lostFocus () const

Returns TRUE if the widget lost the text input focus.

62.2.1. URL

Generated from http://www.troll.no/qt/qfocusevent.html.

63. QFont Class Reference

```
#include <qfont.h>
```

63.1. Detailed Description

The QFont class specifies a font used for drawing text.

QFont, more precisely, is a collection of attributes of a font. When Qt needs to draw text, it will look up and load the closest matching installed font and draw using that.

The most important attributes of a QFont are family, point size, weight and italic. There are QFont constructors that take these attributes as arguments, as shown in this example:

```
void MyWidget::paintEvent( QPaintEvent * )
{
    QPainter p( this );
    // times, 12pt, normal
    p.setFont( QFont( "times" ) );
    p.drawText( 10, 20, "Text1" );
    // helvetica, 18pt, normal
    p.setFont( QFont( "helvetica", 18 ) );
    p.drawText( 10, 120, "Text2" );
    // courier, 24pt, bold
    p.setFont( QFont( "courier", 24, QFont::Bold ) );
    p.drawText( 10, 220, "Text3" );
    // lucida, 36pt, bold, italic
    p.setFont( QFont( "lucida", 36, QFont::Bold, TRUE ) );
    p.drawText( 10, 320, "Text4" );
}
```

The default QFont constructor makes a copy of the default font.

You can also change these attributes of an existing QFont object using functions such as setFamily(), setPointSize(), setWeight() and setItalic().

There are also some less-used attributes. setUnderline() decides whether the font is underlined or not; setStrikeOut() can be used to get overstrike (a horizontal line through the middle of the characters); setFixedPitch() determines whether Qt should give preference to fixed-pitch or variable-pitch fonts when it needs to choose an installed font; setStyleHint() can be used to offer more general help to the font matching algorithm, and on X11 setRawMode() can be used to bypass the entire font matching and use an X11 XLFD.

Of course there is also a reader function for each of these set*() functions. Note that the reader functions return the values previously set, *not* the attributes of the actual window system font that will be used for drawing. You can get information about the font that will be used for drawing by using QFontInfo, but be aware that QFontInfo may be slow and that its results depend on what fonts are installed.

In general font handling and loading are costly operations, especially on X11. The QFont class contains extensive optimizations to make copying of QFont objects fast, and to cache the results of the slow window system functions it uses.

QFont also offers a few static functions, mostly to tune the font matching algorithm: You can control what happens if a font's family isn't installed using insertSubstitution() and removeSubstitution(), ask what happens for a single family uing substitute() and you can get a complete list of the fallback families using listSubstitutions().

cacheStatistics() offers cache effectiveness information; this is useful mostly for debugging.

Finally, setDefaultFont() allows you to set the default font. The default default font is chosen at application startup from a set of common installed fonts that support the correct character set. Of course, the initialization algorithm has a default, too: The default default default font!

For more general information on fonts, see the comp.fonts FAQ.

See also: QFontMetrics, QFontInfo, QApplication::setFont(), QWidget::setFont(), QPainter::setFont(), QFont::StyleHint, QFont::CharSet and QFont::Weight.

Examples:
- grapher/grapher.cpp
- desktop/desktop.cpp
- drawdemo/drawdemo.cpp
- movies/main.cpp
- picture/picture.cpp
- xform/xform.cpp
- widgets/widgets.cpp

63.2. Member Function Documentation

QFont::QFont ()

Constructs a font object that refers to the default font.

QFont::QFont (const QFont & font)

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

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

Constructs a font object with the specified *family, pointSize, weight* and *italic* settings. The character set is copied from the default font and the rest of the settings ser set reasonably.

If *pointSize* is less than or equal to 0 it is set to 1.

See also: setFamily(), setPointSize(), setWeight() and setItalic().

QFont::QFont (const char * family, int pointSize, int weight, bool italic, CharSet charSet)

Constructs a font object with the specified *family, pointSize, weight, italic* and *charSet* settings. If *pointSize* is less than or equal to 0 it is set to 1.

See also: setFamily(), setPointSize(), setWeight() and setItalic().

QFont::~QFont () `[virtual]`

Destroys the font object.

bool QFont::bold () const

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

See also: weight(), setBold() and QFontInfo::bold().

void QFont::cacheStatistics () `[static]`

Internal function that dumps font cache statistics.

QFont::CharSet QFont::charSet() const

Returns the character set by setCharSet().

Use QFontInfo to find the CharSet of the window system font actually used.

See also: setCharSet().

void QFont::cleanup () `[static]`

Internal function that cleans up the font system.

int QFont::deciPointSize () const `[protected]`

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

See also: pointSize().

QString QFont::defaultFamily () const `[protected]`

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

const QFont & QFont::defaultFont () `[static]`

Returns the system default font.

bool QFont::dirty () const `[protected]`

Returns TRUE if the font attributes have been changed and the font has to be (re)loaded, or FALSE if no changes have been made.

bool QFont::exactMatch () const

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

See also: QFontInfo and font matching

const char * QFont::family () const

Returns the family name set by setFamily().

Use QFontInfo to find the family name of the window system font that is actually used for drawing.

Example:

```
QFont     font( "Nairobi" );
QFontInfo info( font );
debug( "Font family requested is    : \"%s\"", font.family() );
debug( "Font family actually used is: \"%s\"", info.family() );
```

See also: setFamily() and substitute().

bool QFont::fixedPitch () const

Returns the value set by setFixedPitch().

Use QFontInfo to find the fixed pitch value of the window system font actually used.

See also: setFixedPitch() and QFontInfo::fixedPitch().

HANDLE QFont::handle (HANDLE=0) const

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

void QFont::initialize () `[static]`

Internal function that initializes the font system.

void QFont::insertSubstitution (const char * familyName, const char * replacementName) `[static]`

Inserts a new font family name substitution in the family substitution table.

If *familyName* already exists in the substitution table, it will be replaced with this new substitution.

See also: removeSubstitution(), listSubstitutions() and substitute().

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

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

See also: operator= and operator==.

bool QFont::italic () const

Returns the value set by setItalic().

Use QFontInfo to find the italic value of the window system font actually used.

See also: setItalic().

QString QFont::key () const

Returns the font's key, which is a textual representation of the font settings. It is typically used to insert and find fonts in a dictionary or a cache.

See also: QDict and QCache.

QString QFont::lastResortFamily () const `[protected]`

Returns a last resort family name for the font matching algorithm.

See also: lastResortFont().

QString QFont::lastResortFont () const `[protected]`

Returns a last resort raw font name for the font matching algorithm.

This is used if not even the last resort family is available.

See also: lastResortFamily().

void QFont::listSubstitutions (QStrList * list) [static]

Returns a sorted list of substituted family names in *list.*

See also: insertSubstitution(), removeSubstitution() and substitute().

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

Returns TRUE if the this font is different from *f,* or FALSE if they are equal.

Two QFonts are different if their font attributes are different. If raw mode is enabled for both fonts, then only the family fields are compared.

See also: operator==().

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

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

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

Returns TRUE if the this font is equal to *f,* or FALSE if they are different.

Two QFonts are equal if their font attributes are equal. If raw mode is enabled for both fonts, then only the family fields are compared.

See also: operator!=().

int QFont::pointSize () const

Returns the point size set by setPointSize().

Use QFontInfo to find the point size of the window system font actually used.

Example of use:

```
QFont     font( "helvetica" );
QFontInfo info( font );
font.setPointSize( 53 );
debug( "Font size requested is    : %d", font.pointSize() );
debug( "Font size actually used is: %d", info.pointSize() );
```

See also: setPointSize() and deciPointSize().

Examples:
xform/xform.cpp

bool QFont::rawMode () const

Returns the value set by setRawMode().

See also: setRawMode().

const char* QFont::rawName () const

Returns the name of the font within the underlying system. *Using the return value of this function is usually not portable.*

See also: setRawMode() and rawMode().

void QFont::removeSubstitution (const char * familyName) `[static]`

Removes a font family name substitution from the family substitution table.

See also: insertSubstitution(), listSubstitutions() and substitute().

void QFont::setBold (bool enable)

Sets the weight to `QFont::Bold` if *enable* is TRUE, or to `QFont::Normal` if *enable* is FALSE.

Use setWeight() to set the weight to other values.

See also: bold() and setWeight().

void QFont::setCharSet (CharSet charset)

Sets the character set encoding (e.g. `Latin1`).

If the character set encoding is not available another will be used for drawing. for most non-trivial applications you will probably not want this to happen since it can totally obscure the text shown to the user. This is why the font matching algorithm gives high priority to finding the correct character set.

You can test that the character set is correct using the QFontInfo class.

Example:

```
QFont     font( "times", 14 );           // default character set is Latin1
QFontInfo info( font );
if ( info.charSet() != Latin1 )          // check actual font
    fatal( "Cannot find a Latin 1 Times font" );
```

In Qt 1.40, the following character set encodings are available:

- `QFont::Latin1` - ISO 8859-1, common in much of Europe
- `QFont::Latin2` - ISO 8859-2, less common European character set
- `QFont::Latin3` - ISO 8859-3, less common European character set
- `QFont::Latin4` - ISO 8859-4, less common European character set
- `QFont::Latin5` - ISO 8859-5, Cyrillic
- `QFont::Latin6` - ISO 8859-6, Arabic
- `QFont::Latin7` - ISO 8859-7, Greek
- `QFont::Latin8` - ISO 8859-8, Hebrew
- `QFont::Latin9` - ISO 8859-9, less common European character set
- `QFont::KOI8R` - KOI8-R, Cyrillic, defined in RFC 1489.
- `QFont::AnyCharSet` - whatever is handiest.

See also: charSet(), QFontInfo and font matching

void QFont::setDefaultFont (const QFont & f) `[static]`

Sets the system default font.

void QFont::setFamily (const char * family)

Sets the family name of the font (e.g. "Helvetica" or "times").

The family name is case insensitive.

If the family is not available a default family is used.

See also: family(), setStyleHint(), QFontInfo and font matching

Examples:
xform/xform.cpp

void QFont::setFixedPitch (bool enable)

Sets fixed pitch on or off.

A fixed pitch font is a font where all characters have the same width.

See also: fixedPitch(), QFontInfo and font matching

void QFont::setItalic (bool enable)

Sets italic on or off.

See also: italic(), QFontInfo and font matching

Examples:
xform/xform.cpp

void QFont::setPointSize (int pointSize)

Sets the point size to *pointSize.* The point size must be greater than zero.

Example:

```
QFont font( "courier" );
font.setPointSize( 18 );
```

See also: pointSize(), QFontInfo and font matching

Examples:
xform/xform.cpp

void QFont::setRawMode (bool enable)

Turns raw mode on if *enable* is TRUE, or turns it off if *enable* is FALSE.

Calling this function only has effect under X windows. If raw mode is enabled, Qt will search for an X font with a complete font name matching the family name, ignoring all other values set for the QFont. If the font name matches several fonts, Qt will use the first font returned by X. QFontInfo *cannot* be used to fetch information about a QFont using raw mode (it will return the values set in the QFont for all parameters, including the family name).

Example:

```
#if defined(_WS_X11_)
    QFont font( "-*-fixed-*-*-*-*-*-140-75-75-c-*-iso8859-1" );
    font.setRawMode( TRUE );
    if ( !font.exactMatch() )
        fatal( "Sorry, could not find the X specific font" );
#endif
```

Warning: Do not use raw mode unless you really need it!

See also: rawMode().

void QFont::setStrikeOut (bool enable)

Sets strike out on or off.

See also: strikeOut(), QFontInfo and font matching

Examples:
xform/xform.cpp

void QFont::setStyleHint (StyleHint hint)

Sets the style hint.

The style hint is used by the font matching algorithm when a selected font family cannot be found and is used to find an appropriate default family.

The style hint has a default value of `AnyStyle` which leaves the task of finding a good default family to the font matching algorithm.

In this example (which is a complete program) the push button will display its text label with the Bavaria font family if this family is available, if not it will display its text label with another serif font:

```
#include <qapplication.h>
#include <qpushbutton.h>
#include <qfont.h>
int main( int argc, char **argv )
{
    QApplication app( argc, argv );
    QPushButton  push("Push me");
    QFont font( "Bavaria", 18 );        // preferrred family is Bavaria
    font.setStyleHint( QFont::Serif );  // can also use any serif font
    push.setFont( font );
    return app.exec( &push );
}
```

The other available style hints are `QFont::SansSerif, QFont::TypeWriter, QFont::OldEnglish, QFont::System`

See also: styleHint(), QFontInfo and font matching

Examples:
desktop/desktop.cpp

void QFont::setUnderline (bool enable)

Sets underline on or off.

See also: underline(), QFontInfo and font matching.

Examples:
xform/xform.cpp

void QFont::setWeight (int weight)

Sets the weight (or boldness).

The enum `QFont::Weight` contains the predefined font weights:

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

Strictly speaking you can use all values in the range [0,99] (where 0 is ultralight and 99 is extremely black), but there is such a thing as asking too much of the underlying window system.

Example:

```
QFont font( "courier" );
font.setWeight( QFont::Bold );
```

If the specified weight is not available the closest available will be used. Use QFontInfo to check the actual weight.

See also: weight(), QFontInfo and font matching

Examples:
xform/xform.cpp

bool QFont::strikeOut () const

Returns the value set by setStrikeOut().

Use QFontInfo to find the strike out value of the window system font actually used.

See also: setStrikeOut() and QFontInfo::strikeOut().

QFont::StyleHint QFont::styleHint() const

Returns the StyleHint set by setStyleHint().

See also: setStyleHint() and QFontInfo::styleHint().

const char * QFont::substitute (const char * familyName) `[static]`

Returns the font family name to be used whenever *familyName* is specified, and not found by the font matching algorithm. The lookup is case insensitive.

If there is no substitution for *familyName,* then *familyName* is returned.

Example:

```
QFont::insertSubstitution( "NewYork", "London" );
QFont::insertSubstitution( "Paris",   "Texas" );
QFont::substitute( "NewYork" );     // returns "London"
QFont::substitute( "PARIS" );       // returns "Texas"
QFont::substitute( "Rome" );        // returns "Rome"
QFont::removeSubstitution( "newyork" );
QFont::substitute( "NewYork" );     // returns "NewYork"
```

See also: setFamily(), insertSubstitution() and removeSubstitution().

bool QFont::underline () const

Returns the value set by setUnderline().

Use QFontInfo to find the underline value of the window system font actually used for drawing.

See also: setUnderline() and QFontInfo::underline().

int QFont::weight () const

Returns the weight set by setWeight().

Use QFontInfo to find the weight of the window system font actually used.

See also: setWeight() and QFontInfo.

63.3. Related Functions

QDataStream & operator<< (QDataStream & s, const QFont & f)

Writes a font to the stream.

QDataStream & operator>> (QDataStream & s, QFont & f)

Reads a font from the stream.

63.3.1. URL

Generated from http://www.troll.no/qt/qfont.html.

64. QFontInfo Class Reference

```
#include <qfontinfo.h>
```

64.1. Detailed Description

The QFontInfo class provides general information about fonts.

The QFont class might not always map exactly to the specified font for a paint device. The QFontInfo class provides information of the actual font that matched a QFont specification.

There are three ways you can create a QFontInfo object:

1 The QFontInfo constructor with a QFont creates a font info object for a screen-compatible font, i.e. the font must not be a printer font.

2 QWidget::fontInfo() returns the font info for a widget's current font. The font info object is automatically updated if somebody sets a new widget font. *Please read the note below.*

3 QPainter::fontInfo() returns the font info for a painter's current font. The font info object is automatically updated if somebody sets a new painter font. *Please read the note below.*

Example:

```
QFont font("reykjavik",24);
QFontInfo fi(font);
if ( strcmp(font.family(),fi.family()) == 0 ) {
    ; // got this font
} else {
    ; // got fi.family() instead
}
```

In Qt 2.0 the font info object will no longer be automatically updated when the widget or painter gets a new font. Make sure you program does not depend on this feature.

About efficiency: We recommend that you use the QFontInfo constructor if you can. This is more efficient than getting the font info from a widget or a painter. The QWidget::fontInfo() and QPainter::fontInfo() will become faster in Qt 2.0, when we have removed the automatic update policy.

See also: QFont and QFontMetrics.

Examples:
xform/xform.cpp

64.2. Member Function Documentation

QFontInfo::QFontInfo (const QFont & font)

Constructs a font info object for *font.*

The font must be screen-compatible, i.e. a font you use when drawing text in widgets or pixmaps. If *font* is a printer font, you'll probably get wrong results.

Use the QPainter::fontInfo() to get the font info when painting. This is a little slower than using this constructor, but it always gives correct results.

QFontInfo::QFontInfo (const QFontInfo & fi)

Constructs a copy of *fi.*

QFontInfo::˜QFontInfo ()

Destroys the font info object.

bool QFontInfo::bold () const

Returns TRUE if weight() would return a greater than `QFont::Normal`, and FALSE otherwise.

See also: weight() and QFont::bold().

QFont::CharSet QFontInfo::charSet() const

Returns the character set of the matched window system font.

See also: QFont::charSet().

bool QFontInfo::exactMatch () const

Returns TRUE if the matched window system font is exactly the one specified by the font.

See also: QFont::exactMatch().

const char * QFontInfo::family () const

Returns the family name of the matched window system font.

See also: QFont::family().

Examples:
xform/xform.cpp

bool QFontInfo::fixedPitch () const

Returns the fixed pitch value of the matched window system font. A fixed pitch font is a font that has constant character pixel width.

See also: QFont::fixedPitch().

bool QFontInfo::italic () const

Returns the italic value of the matched window system font.

See also: QFont::italic().

Examples:
xform/xform.cpp

QFontInfo & QFontInfo::operator= (const QFontInfo & fi)

Font info assignment.

int QFontInfo::pointSize () const

Returns the point size of the matched window system font.

See also: QFont::pointSize().

Examples:
xform/xform.cpp

bool QFontInfo::rawMode () const

Returns TRUE if the font is a raw mode font.

If it is a raw mode font, all other functions in QFontInfo will return the same values set in the QFont, regardless of the font actually used.

Warning: The default font is a raw-mode font.

See also: QFont::rawMode().

bool QFontInfo::strikeOut () const

Returns the strike out value of the matched window system font.

See also: QFont::strikeOut().

QFont::StyleHint QFontInfo::styleHint() const

Returns the style of the matched window system font.

Currently only returns the hint set in QFont.

See also: QFont::styleHint().

bool QFontInfo::underline () const

Returns the underline value of the matched window system font.

See also: QFont::underline().

int QFontInfo::weight () const

Returns the weight of the matched window system font.

See also: QFont::weight() and bold().

Examples:
xform/xform.cpp

64.2.1. URL

Generated from http://www.troll.no/qt/qfontinfo.html.

65. QFontMetrics Class Reference

`#include <qfontmetrics.h>`

65.1. Detailed Description

The QFontMetrics class provides font metrics information about fonts.

QFontMetrics functions calculate size of characters and strings for a given font.

There are three ways you can create a QFontMetrics object:

1 The QFontMetrics constructor with a QFont creates a font metrics object for a screen-compatible font, i.e. the font must not be a printer font.

2 QWidget::fontMetrics() returns the font metrics for a widget's current font. The font metrics object is automatically updated if somebody sets a new widget font. *Please read the note below.*

3 QPainter::fontMetrics() returns the font metrics for a painter's current font. The font metrics object is automatically updated if somebody sets a new painter font. *Please read the note below.*

Example:

```
QFont font("times",24);
QFontMetrics fm(font);
int w = fm.width("What's the width of this text");
int h = fm.height();
```

NOTE: In Qt 2.0 the font metrics object will no longer be automatically updated when the widget or painter gets a new font. Make sure you program does not depend on this feature.

About efficiency: We recommend that you use the QFontMetrics constructor if you can. This is more efficient than getting the font metrics from a widget or a painter. The QWidget::fontMetrics() and QPainter::fontMetrics() will become faster in Qt 2.0, when we have removed the automatic update policy.

See also: QFont and QFontInfo.

Examples:

grapher/grapher.cpp
drawdemo/drawdemo.cpp
hello/hello.cpp
movies/main.cpp
xform/xform.cpp
application/application.cpp

65.2. Member Function Documentation

QFontMetrics::QFontMetrics (const QFont & font)

Constructs a font metrics object for *font.*

The font must be screen-compatible, i.e. a font you use when drawing text in widgets or pixmaps. If *font* is a printer font, you'll probably get wrong results.

Use the QPainter::fontMetrics() to get the font metrics when painting. This is a little slower than using this constructor, but it always gives correct results.

QFontMetrics::QFontMetrics (const QFontMetrics & fm)

Constructs a copy of *fm.*

QFontMetrics::~QFontMetrics ()

Destroys the font metrics object.

int QFontMetrics::ascent () const

Returns the maximum ascent of the font.

The ascent is the distance from the base line to the uppermost line where pixels may be drawn.

See also: descent().

Examples:
drawdemo/drawdemo.cpp

QRect QFontMetrics::boundingRect (char ch) const

Returns the bounding rectangle of *ch* relative to the leftmost point on the base line.

Note that the bounding rectangle may extend to the left of (0,0), e.g. for italicized fonts, and that the text output may cover *all* pixels in the bounding rectangle.

Note that the rectangle usually extends both above and below the base line.

See also: width().

QRect QFontMetrics::boundingRect (const char * str, int len = -1) const

Returns the bounding rectangle of the first *len* characters of *str,* which is the set of pixels the text would cover if drawn at (0,0).

If *len* is negative (default value), the whole string is used.

Note that the bounding rectangle may extend to the left of (0,0), e.g. for italicized fonts, and that the text output may cover *all* pixels in the bounding rectangle.

Newline characters are processed as regular characters, *not* as linebreaks.

Due to the different actual character heights, the height of the bounding rectangle of "Yes" and "yes" may be different.

See also: width() and QPainter::boundingRect().

Examples:
xform/xform.cpp

QRect QFontMetrics::boundingRect (int x, int y, int w, int h, int flags, const char * str, int len=-1, int tabstops=0, int * tabarray=0, char ** intern=0) const

Returns the bounding rectangle of the first *len* characters of *str,* which is the set of pixels the text would cover if drawn at (0,0).

If *len* is negative (default value), the whole string is used.

The *flags* argument is the bitwise OR of the following flags:

- `AlignLeft` aligns to the left border.
- `AlignRight` aligns to the right border.
- `AlignHCenter` aligns horizontally centered.
- `AlignTop` aligns to the top border.

- `AlignBottom` aligns to the bottom border.
- `AlignVCenter` aligns vertically centered
- `AlignCenter` (= `AlignHCenter` | AlignVCenter)
- `SingleLine` ignores newline characters in the text.
- `ExpandTabs` expands tabulators.
- `ShowPrefix` interprets "&x" as "x" underlined.
- `WordBreak` breaks the text to fit the rectangle.

Horizontal alignment defaults to AlignLeft and vertical alignment defaults to AlignTop.

If several of the horizontal or several of the vertical alignment flags are set, the resulting alignment is undefined.

These flags are defined in qwindowdefs.h.

If `ExpandTabs` is set in *flags,* then: if *tabarray* is non.zero, it specifies a 0-terminated sequence of pixel-positions for tabs; otherwise if *tabstops* is non-zero, it is used as the tab spacing (in pixels).

Note that the bounding rectangle may extend to the left of (0,0), e.g. for italicized fonts, and that the text output may cover *all* pixels in the bounding rectangle.

Newline characters are processed as linebreaks.

Despite the different actual character heights, the heights of the bounding rectangles of "Yes" and "yes" are the same.

The bounding rectangle given by this function is somewhat larger than that calculated by the simpler boundingRect() function. This function uses the maximum left and right font bearings as is necessary for multi-line text to align correctly. Also, fontHeight() and lineSpacing() are used to calculate the height, rather than individual character heights.

The *internal* argument is for internal purposes.

See also: width() and QPainter::boundingRect().

int QFontMetrics::descent () const

Returns the maximum descent of the font.

The descent is the distance from the base line to the lowermost line where pixels may be drawn. (Note that this is different from X, which adds 1 pixel.)

See also: ascent().

Examples:
drawdemo/drawdemo.cpp
hello/hello.cpp

int QFontMetrics::height () const

Returns the height of the font.

This is always equal to ascent()+descent()+1 (the 1 is for the base line).

See also: leading() and lineSpacing().

Examples:
grapher/grapher.cpp
hello/hello.cpp

bool QFontMetrics::inFont (char ch) const

Returns TRUE if *ch* is a valid character in the font.

int QFontMetrics::leading () const

Returns the leading of the font.

This is the natural inter-line spacing.

See also: height() and lineSpacing().

int QFontMetrics::leftBearing (char ch) const

Returns the left bearing of character *ch* in the font.

The left bearing is the rightward distance of the left-most pixel of the character from the logical origin of the character. This value is negative if the pixels of the character extend to the left of the logical origin.

See width(char) for a graphical description of this metric.

See also: rightBearing(char), minLeftBearing() and width().

int QFontMetrics::lineSpacing () const

Returns the distance from one base line to the next.

This value is always equal to leading()+height().

See also: height() and leading().

Examples:
application/application.cpp

int QFontMetrics::lineWidth () const

Returns the width of the underline and strike-out lines, adjusted for the point size of the font.

See also: underlinePos() and strikeOutPos().

int QFontMetrics::maxWidth () const

Returns the width of the widest character in the font.

int QFontMetrics::minLeftBearing () const

Returns the minimum left bearing of the font.

This is the smallest leftBearing(char) of all characters in the font.

See also: minRightBearing() and leftBearing(char).

int QFontMetrics::minRightBearing () const

Returns the minimum right bearing of the font.

This is the smallest rightBearing(char) of all characters in the font.

See also: minLeftBearing() and rightBearing(char).

QFontMetrics & QFontMetrics::operator= (const QFontMetrics & fm)

Font metrics assignment.

int QFontMetrics::rightBearing (char ch) const

Returns the right bearing of character *ch* in the font.

The right bearing is the leftward distance of the right-most pixel of the character from the logical origin of a subsequent character. This value is negative if the pixels of the character extend to the right of the width() of the character.

See width() for a graphical description of this metric.

See also: leftBearing(char), minRightBearing() and width().

QSize QFontMetrics::size (int flags, const char * str, int len=-1, int tabstops=0, int * tabarray=0, char ** intern=0) const

Returns the size in pixels of the first *len* characters of *str*.

If *len* is negative (default value), the whole string is used.

The *flags* argument is the bitwise OR of the following flags:

- `SingleLine` ignores newline characters in the text.
- `ExpandTabs` expands tabulators.
- `ShowPrefix` interprets "&x" as "x" underlined.
- `WordBreak` breaks the text to fit the rectangle.

These flags are defined in qwindowdefs.h.

If `ExpandTabs` is set in *flags,* then: if *tabarray* is non.zero, it specifies a 0-terminated sequence of pixel-positions for tabs; otherwise if *tabstops* is non-zero, it is used as the tab spacing (in pixels).

Newline characters are processed as linebreaks.

Despite the different actual character heights, the heights of the bounding rectangles of "Yes" and "yes" are the same.

The *internal* argument is for internal purposes.

See also: boundingRect().

int QFontMetrics::strikeOutPos () const

Returns the distance from the base line to where the strike-out line should be drawn.

See also: underlinePos() and lineWidth().

int QFontMetrics::underlinePos () const

Returns the distance from the base line to where an underscore should be drawn.

See also: strikeOutPos() and lineWidth().

int QFontMetrics::width (char ch) const

Returns the logical width of a *ch* in pixels. This is a distance appropriate for drawing a subsequent character after *ch.*

Some of the metrics are described in the image to the right. The tall dark rectangle covers the logical width() of a character. The shorter pale rectangles cover the left and right bearings of the characters. Notice that the bearings of "f" in this particular font are both negative, while the bearings of "o" are both positive.

See also: boundingRect().

int QFontMetrics::width (const char * str, int len = -1) const

Returns the width in pixels of the first *len* characters of *str*.

If *len* is negative (default value), the whole string is used.

Note that this value is *not* equal to boundingRect().width(); boundingRect() returns a rectangle describing the pixels this string will cover whereas width() returns the distance to where the next string should be drawn. Thus, width(stra)+width(strb) is always equal to width(strcat(stra, strb)). This is almost never the case with boundingRect().

See also: boundingRect().

Examples:
drawdemo/drawdemo.cpp
hello/hello.cpp
movies/main.cpp

65.2.1. URL

Generated from http://www.troll.no/qt/qfontmetrics.html.

66. QFrame Class Reference

`#include <qframe.h>`

Inherits QWidget.

Inherited by QGroupBox, QLCDNumber, QLabel, QMenuBar, QProgressBar, QScrollView, QSpinBox, QSplitter, QTableView and QWidgetStack.

66.1. Detailed Description

The QFrame class is the base class of widgets that have an (optional) frame.

It draws a label and calls a virtual function, drawContents(), to fill in the frame. This function is reimplemented by essentially all subclasses. There are also two other less useful functions, drawFrame() and frameChanged().

QMenuBar uses this to "raise" the menu bar above the surrounding screen:

```
if ( style() == MotifStyle ) {
    setFrameStyle( QFrame::Panel | QFrame::Raised );
    setLineWidth( 2 );
} else {
    setFrameStyle( QFrame::NoFrame );
}
```

The QFrame class can also be used directly for creating simple frames without any contents, for example like this:

```
QFrame *emptyFrame = new QFrame( parentWidget );
emptyFrame->setFrameStyle( Panel | Sunken );
emptyFrame->setLineWidth( 2 );
```

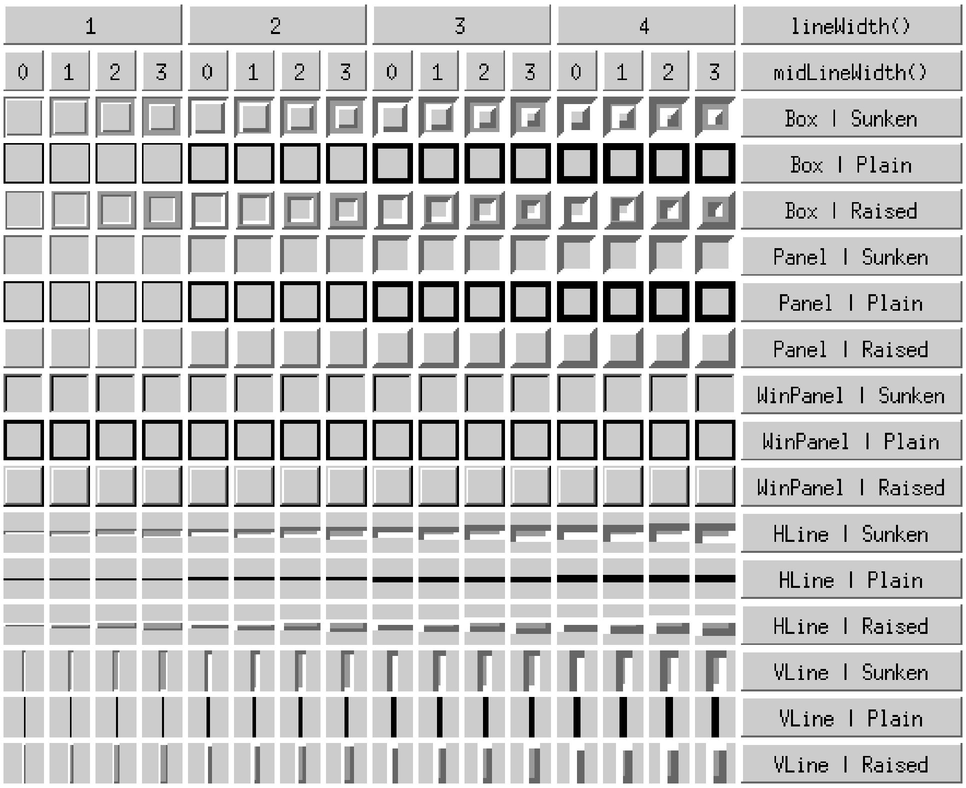
A frame widget has three attributes: frame style, a line width and a mid-line width.

The frame style is specified by a frame shape and a shadow style. The frame shapes are `NoFrame, Box, Panel, WinPanel, HLine` and `VLine,` and the shadow styles are `Plain, Raised` and `Sunken.`

The line width is the width of the frame border.

The mid-line width specifies the width of an extra line in the middle of the frame, that uses a third color to obtain a special 3D effect. Notice that a mid-line is only drawn for `Box, HLine` and `VLine` frames that are raised or sunken.

This table shows the most useful combinations of styles and widths (and some rather useless ones):



For obvious reasons, `NoFrame` isn't shown. The gray areas next to the `VLine` and `HLine` examples are there because the widgets are taller/wider than the natural width of the lines. frameWidth() returns the natural width of the line.

The labels on the top and right are QLabel objects with frameStyle() `Raised|Panel` and lineWidth() 1.

Examples:

- tictac/tictac.cpp
- life/life.cpp
- movies/main.cpp
- xform/xform.cpp
- widgets/widgets.cpp

66.2. Member Function Documentation

QFrame::QFrame (QWidget * parent=0, const char * name=0, WFlags f=0, bool allowLines=TRUE)

Constructs a frame widget with frame style `NoFrame` and a 1 pixel frame width.

The *allowLines* argument can be set to FALSE to disallow `HLine` and `VLine` shapes.

The *parent, name* and *f* arguments are passed to the QWidget constructor.

QRect QFrame::contentsRect () const

Returns the rectangle inside the frame.

See also: frameRect() and drawContents().

void QFrame::drawContents (QPainter *) `[virtual protected]`

Virtual function that draws the contents of the frame.

The QPainter is already open when you get it, and you must leave it open. Painter transformations are switched off on entry. If you transform the painter, remember to take the frame into account and reset transformation before returning.

This function is reimplemented by subclasses that draw something inside the frame. It should draw only inside contentsRect(). The default function does nothing.

See also: contentsRect() and QPainter::setClipRect().

Reimplemented in QLCDNumber, QMenuBar, QLabel and QProgressBar.

void QFrame::drawFrame (QPainter * p) `[virtual protected]`

Draws the frame using the current frame attributes and color group. The rectangle inside the frame is not affected.

This function is virtual, but in general you do not need to reimplement it. If you do, note that the QPainter is already open and must remain open.

See also: frameRect(), contentsRect(), drawContents(), frameStyle() and setPalette().

Examples:
    life/life.cpp

void QFrame::frameChanged () `[virtual protected]`

Virtual function that is called when the frame style, line width or mid-line width changes.

This function can be reimplemented by subclasses that need to know when the frame attributes change.

Reimplemented in QScrollView and QWidgetStack.

QRect QFrame::frameRect () const

Returns the frame rectangle.

The default frame rectangle is equivalent to the widget rectangle.

See also: setFrameRect().

int QFrame::frameShadow () const

Returns the frame shadow value from the frame style.

See also: frameStyle() and frameShape().

int QFrame::frameShape () const

Returns the frame shape value from the frame style.

See also: frameStyle() and frameShadow().

int QFrame::frameStyle () const

Returns the frame style.

The default value is QFrame::NoFrame.

See also: setFrameStyle(), frameShape() and frameShadow().

int QFrame::frameWidth () const

Returns the width of the frame that is drawn.

Note that the frame width depends on the frame style, not only the line width and the mid line width. For example, the style `NoFrame` always has a frame width 0, while the style `Panel` has a frame width equivalent to the line width.

See also: lineWidth(), midLineWidth() and frameStyle().

Examples:
widgets/widgets.cpp

bool QFrame::lineShapesOk () const

Returns TRUE if line shapes (`HLine` or `VLine`) are allowed, or FALSE if they are not allowed.

It is only possible to disallow line shapes in the constructor. The default value is TRUE.

int QFrame::lineWidth () const

Returns the line width. (Note that the *total* line width for `HLine` and `VLine` is given by frameWidth(), not lineWidth().)

The default value is 1.

See also: setLineWidth(), midLineWidth() and frameWidth().

int QFrame::margin () const

Returns the width of the margin. The margin is the distance between the innermost pixel of the frame and the outermost pixel of contentsRect(). It is included in frameWidth().

The margin is filled according to backgroundMode().

The default value is 0.

See also: setMargin(), lineWidth() and frameWidth().

int QFrame::midLineWidth () const

Returns the width of the mid-line.

The default value is 0.

See also: setMidLineWidth(), lineWidth() and frameWidth().

void QFrame::paintEvent (QPaintEvent * event) `[virtual protected]`

Handles paint events for the frame.

Paints the frame and the contents.

Opens the painter on the frame and calls first drawFrame(), then drawContents().

Reimplemented from QWidget.

void QFrame::resizeEvent (QResizeEvent * e) `[virtual protected]`

Handles resize events for the frame.

Adjusts the frame rectangle for the resized widget. The frame rectangle is elastic, the surrounding area is static.

The resulting frame rectangle may be null or invalid. You can use setMinimumSize() to avoid that possibility.

Nothing is done if the frame rectangle is a null rectangle already.

Reimplemented from QWidget.

void QFrame::setFrameRect (const QRect & r) `[protected]`

Sets the frame rectangle to *r*.

The frame rectangle is the rectangle the frame is drawn in. By default, this is the entire widget. Calling setFrameRect() does *not* cause a widget update.

If *r* is a null rectangle (for example `QRect(0,0,0,0)`), then the frame rectangle is equivalent to the widget rectangle.

See also: frameRect() and contentsRect().

void QFrame::setFrameStyle (int style)

Sets the frame style to *style*.

The *style* is the bitwise OR between a frame shape and a frame shadow style. See the illustration in the class documentation.

The frame shapes are:

- `NoFrame` draws nothing. Naturally, you should not specify a shadow style if you use this.
- `Box` draws a rectangular box. The contents appear to be level with the surrounding screen, but the border itself may be raised or sunken.
- `Panel` draws a rectangular panel that can be raised or sunken.
- `WinPanel` draws a rectangular panel that can be raised or sunken. Specifying this shape sets the line width to 2 pixels. WinPanel provides fancy Windows 95-like shadows.
- `HLine` draws a horizontal line (vertically centered).
- `VLine` draws a vertical line (horizontally centered).

The shadow styles are:

- `Plain` draws using the palette foreground color (without any 3D effect).
- `Raised` draws a 3D raised line using the light and dark colors of the current color group.
- `Sunken` draws a 3D sunken line using the light and dark colors of the current color group.

If a mid-line width greater than 0 is specified, an additional line is drawn for `Raised` or `Sunken Box`, `HLine` and `VLine` frames. The mid color of the current color group is used for drawing middle lines.

Warning: Attempts to set the frame style to `HLine` or `VLine` (with any shadow style) are disregarded unless line shapes are allowed. Line shapes are allowed by default.

See also: Illustration, frameStyle(), lineShapesOk(), colorGroup() and QColorGroup.

Examples:

tictac/tictac.cpp
dclock/dclock.cpp

xform/xform.cpp
cursor/cursor.cpp
layout/layout.cpp
widgets/widgets.cpp

void QFrame::setLineWidth (int w)

Sets the line width to *w*.

See also: frameWidth(), lineWidth() and setMidLineWidth().

Examples:
dclock/dclock.cpp
xform/xform.cpp

void QFrame::setMargin (int w)

Sets the width of the margin to *w*.

See also: margin() and setLineWidth().

void QFrame::setMidLineWidth (int w)

Sets the width of the mid-line to *w*.

See also: midLineWidth() and setLineWidth().

66.2.1. URL

Generated from http://www.troll.no/qt/qframe.html.

67. QGArray Class Reference

`#include <qgarray.h>`

Inherited by QArray.

67.1. Detailed Description

The QGArray class is an internal class for implementing the QArray class.

QGArray is a strictly internal class that acts as base class for the QArray template array.

It contains an array of bytes and has no notion of an array element.

67.2. Member Function Documentation

QGArray::QGArray ()

For internal use only.

QGArray::QGArray (const QGArray & a) `[protected]`

For internal use only.

QGArray::QGArray (int size) `[protected]`

For internal use only.

QGArray::QGArray (int, int) `[protected]`

For internal use only.

QGArray::~QGArray () `[virtual protected]`

For internal use only.

QGArray & QGArray::assign (const QGArray & a) `[protected]`

For internal use only.

QGArray & QGArray::assign (const char * d, uint len) `[protected]`

For internal use only.

char * QGArray::at (uint index) const `[protected]`

For internal use only.

int QGArray::contains (const char * d, uint sz) const `[protected]`

For internal use only.

char * QGArray::data () const `[protected]`

For internal use only.

void QGArray::deleteData (array_data * p) `[virtual protected]`

For internal use only.

void QGArray::detach () `[virtual protected]`

For internal use only.

Reimplemented in QBitArray, QArray and QByteArray.

QGArray & QGArray::duplicate (const QGArray & a) `[protected]`

For internal use only.

QGArray & QGArray::duplicate (const char * d, uint len) `[protected]`

For internal use only.

bool QGArray::fill (const char * d, int len, uint sz) `[protected]`

For internal use only.

int QGArray::find (const char * d, uint index, uint sz) const `[protected]`

For internal use only.

bool QGArray::isEqual (const QGArray & a) const `[protected]`

For internal use only.

array_data * QGArray::newData () `[virtual protected]`

For internal use only.

uint QGArray::nrefs () const `[protected]`

For internal use only.

QGArray & QGArray::operator= (const QGArray & a) `[protected]`

For internal use only.

void QGArray::resetRawData (const char * d, uint len) `[protected]`

For internal use only.

bool QGArray::resize (uint newsize) `[protected]`

For internal use only.

bool QGArray::setExpand (uint index, const char * d, uint sz) `[protected]`

For internal use only.

QGArray & QGArray::setRawData (const char * d, uint len) `[protected]`

For internal use only.

void QGArray::setSharedBlock (array_data * p) `[protected]`

For internal use only.

array_data * QGArray::sharedBlock () const `[protected]`

For internal use only.

uint QGArray::size () const `[protected]`

For internal use only.

void QGArray::store (const char * d, uint len) `[protected]`

For internal use only.

67.2.1. URL

Generated from http://www.troll.no/qt/qgarray.html.

68. QGCache Class Reference

`#include <qgcache.h>`

Inherits QCollection.

Inherited by QCache and QIntCache.

68.1. Detailed Description

The QGCache class is an internal class for implementing QCache and QIntCache.

QGCache is a strictly internal class that acts as a base class for the collection classes QCache and QIntCache.

68.2. Member Function Documentation

QGCache::QGCache (const QGCache &) `[protected]`

For internal use only.

QGCache::QGCache (int maxCost, uint size, bool caseS, bool copyKeys, bool trivial) `[protected]`

For internal use only.

QGCache::~QGCache () `[protected]`

For internal use only.

void QGCache::clear () `[virtual protected]`

For internal use only.

Reimplemented from QCollection.

uint QGCache::count () const `[virtual protected]`

For internal use only.

Reimplemented from QCollection.

GCI QGCache::find (const char * key, bool ref=TRUE) const `[protected]`

For internal use only.

bool QGCache::insert (const char * key, GCI data, int cost, int priority) `[protected]`

For internal use only.

int QGCache::maxCost () const `[protected]`

For internal use only.

QGCache & QGCache::operator= (const QGCache &) `[protected]`

For internal use only.

bool QGCache::remove (const char * key) `[protected]`

For internal use only.

void QGCache::setMaxCost (int maxCost) `[protected]`

For internal use only.

uint QGCache::size () const `[protected]`

For internal use only.

void QGCache::statistics () const `[protected]`

For internal use only.

GCI QGCache::take (const char * key) `[protected]`

For internal use only.

int QGCache::totalCost () const `[protected]`

For internal use only.

68.2.1. URL

Generated from http://www.troll.no/qt/qgcache.html.

69. QGCacheIterator Class Reference

`#include <qgcache.h>`

Inherited by QCacheIterator and QIntCacheIterator.

69.1. Detailed Description

An internal class for implementing QCacheIterator and QIntCacheIterator.

QGCacheIterator is a strictly internal class that does the heavy work for QCacheIterator and QIntCacheIterator.

69.2. Member Function Documentation

QGCacheIterator::QGCacheIterator (const QGCache & c) `[protected]`

For internal use only.

QGCacheIterator::QGCacheIterator (const QGCacheIterator & ci) `[protected]`

For internal use only.

QGCacheIterator::~QGCacheIterator () `[protected]`

For internal use only.

bool QGCacheIterator::atFirst () const `[protected]`

For internal use only.

bool QGCacheIterator::atLast () const `[protected]`

For internal use only.

uint QGCacheIterator::count () const `[protected]`

For internal use only.

GCI QGCacheIterator::get () const `[protected]`

For internal use only.

const char * QGCacheIterator::getKey () const `[protected]`

For internal use only.

GCI QGCacheIterator::operator() () `[protected]`

For internal use only.

GCI QGCacheIterator::operator++ () `[protected]`

For internal use only.

GCI QGCacheIterator::operator+= (uint jump) `[protected]`

For internal use only.

GCI QGCacheIterator::operator-- () `[protected]`

For internal use only.

GCI QGCacheIterator::operator-= (uint jump) `[protected]`

For internal use only.

QGCacheIterator & QGCacheIterator::operator= (const QGCacheIterator & ci) `[protected]`

For internal use only.

GCI QGCacheIterator::toFirst () `[protected]`

For internal use only.

GCI QGCacheIterator::toLast () `[protected]`

For internal use only.

69.2.1. URL

Generated from http://www.troll.no/qt/qgcacheiterator.html.

70. QGDict Class Reference

`#include <qgdict.h>`

Inherits QCollection.

Inherited by QDict, QIntDict and QPtrDict.

70.1. Detailed Description

The QGDict class is an internal class for implementing QDict and QIntDict.

QGDict is a strictly internal class that acts as a base class for the collection classes QDict and QIntDict.

QGDict has some virtual functions that can be reimplemented to customize the subclasses.

- hashKey() implements the hashing function for the dictionary.
- read() reads a collection/dictionary item from a QDataStream.
- write() writes a collection/dictionary item to a QDataStream. Normally, you do not have to reimplement any of these functions.

70.2. Member Function Documentation

QGDict::QGDict (const QGDict & dict) `[protected]`

For internal use only.

QGDict::QGDict (uint len, bool cs, bool ck, bool th) `[protected]`

For internal use only.

QGDict::~QGDict () `[protected]`

For internal use only.

void QGDict::clear () `[virtual protected]`

For internal use only.

Reimplemented from QCollection.

uint QGDict::count () const `[virtual]`

For internal use only.

Reimplemented from QCollection.

int QGDict::hashKey (const char * key) `[virtual protected]`

Returns the hash key for *key,* when key is a string.

GCI QGDict::look (const char * key, GCI d, int op)

For internal use only.

QGDict & QGDict::operator= (const QGDict & dict) `[protected]`

For internal use only.

QDataStream & QGDict::read (QDataStream & s)

For internal use only.

QDataStream& QGDict::read (QDataStream & s, GCI & item) `[virtual protected]`

Reads a collection/dictionary item from the stream *s* and returns a reference to the stream.

The default implementation sets *item* to 0.

See also: write().

bool QGDict::remove (const char * key) `[protected]`

For internal use only.

void QGDict::resize (uint newsize) `[protected]`

For internal use only.

uint QGDict::size () const

For internal use only.

void QGDict::statistics () const `[protected]`

For internal use only.

GCI QGDict::take (const char * key) `[protected]`

For internal use only.

QDataStream& QGDict::write (QDataStream & s) const

For internal use only.

QDataStream& QGDict::write (QDataStream & s, GCI) const `[virtual protected]`

Writes a collection/dictionary item to the stream *s* and returns a reference to the stream.

The default implementation does nothing.

See also: read().

70.2.1. URL

Generated from http://www.troll.no/qt/qgdict.html.

71. QGDictIterator Class Reference

`#include <qgdict.h>`

Inherited by QDictIterator, QIntDictIterator and QPtrDictIterator.

71.1. Detailed Description

An internal class for implementing QDictIterator and QIntDictIterator.

QGDictIterator is a strictly internal class that does the heavy work for QDictIterator and QIntDictIterator.

71.2. Member Function Documentation

QGDictIterator::QGDictIterator (const QGDict & d)

For internal use only.

QGDictIterator::QGDictIterator (const QGDictIterator & it)

For internal use only.

QGDictIterator::~QGDictIterator ()

For internal use only.

GCI QGDictIterator::get () const

For internal use only.

const char * QGDictIterator::getKey () const

For internal use only.

GCI QGDictIterator::operator() ()

For internal use only.

GCI QGDictIterator::operator++ ()

For internal use only.

GCI QGDictIterator::operator+= (uint jumps)

For internal use only.

QGDictIterator & QGDictIterator::operator= (const QGDictIterator & it)

For internal use only.

GCI QGDictIterator::toFirst ()

For internal use only.

71.2.1. URL

Generated from http://www.troll.no/qt/qgdictiterator.html.

72. QGIFFormat Class Reference

`#include <qasyncimageio.h>`

Inherits QImageFormat.

72.1. Detailed Description

Incremental image decoder for GIF image format.

This subclass of QImageFormat decodes GIF format images, including animated GIFs. Internally in.

72.2. Member Function Documentation

QGIFFormat::QGIFFormat ()

Constructs a QGIFFormat.

QGIFFormat::~QGIFFormat () `[virtual]`

Destructs a QGIFFormat.

int QGIFFormat::decode (QImage & img, QImageConsumer * consumer, const uchar * buffer, int length) `[virtual]`

This function decodes some data into image changes.

Returns the number of bytes consumed.

Reimplemented from QImageFormat.

72.2.1. URL

Generated from http://www.troll.no/qt/qgifformat.html.

73. QGIFFormatType Class Reference

```
#include <qasyncimageio.h>
```

Inherits QImageFormatType.

73.1. Detailed Description

Incremental image decoder for GIF image format.

This subclass of QImageFormatType recognizes GIF format images, creating a QGIFFormat when required. An instance of this class is created automatically before any other factories, so you should have no need for such objects.

73.1.1. URL

Generated from http://www.troll.no/qt/qgifformattype.html.

74. QGLContext Class Reference

`#include <qgl.h>`

74.1. Detailed Description

This class is defined in the **Qt OpenGL Extension**, which can be found in the `qt/extensions` directory. It is not included in the main Qt API.

The QGLContext class encapsulates an OpenGL rendering context.

An OpenGL rendering context is a complete set of OpenGL state variables.

74.2. Member Function Documentation

QGLContext::QGLContext (const QGLFormat & format, QPaintDevice * device)

Constructs an OpenGL context for the paint device *device,* which must be a widget or a pixmap. The *format* specifies several display options for this context.

The context will be invalid if the *format* settings cannot be satified by the underlying OpenGL/Window system.

See also: isValid().

QGLContext::˜QGLContext () `[virtual]`

Destroys the OpenGL context.

bool QGLContext::chooseContext (const QGLContext * shareContext = 0 = 0) `[virtual protected]`

This semi-internal function is called by create(). It creates a system-dependent OpenGL handle that matches the specified format.

Windows: Calls choosePixelFormat() which finds a matching pixel format identifier.

X11: Calls chooseVisual() which finds an appropriate X visual.

choosePixelFormat() and chooseVisual() can be reimplemented in a subclass if you need to choose a very custom context.

bool QGLContext::create (const QGLContext * shareContext = 0) `[virtual]`

Creates the GL context. Returns TRUE if it was successful in creating a context that satisfies the requested format, otherwise FALSE is returned (the context is invalid).

If successful and the *shareContext* points to a valid QGLContext, this context will share OpenGL display lists with *shareContext.*

Implementation note: Initialization of C++ class members usually takes place in the class constructor. QGLContext is an exception because it must be simple to customize. The virtual functions chooseContext() (and chooseVisual() for X11) can be reimplemented in a subclass to select a particular context. The trouble is that virtual functions are not properly called during construction (which is indeed correct C++), hence we need a create() function.

See also: chooseContext() and isValid().

QPaintDevice* QGLContext::device () const

Returns the paint device set for this context.

See also: QGLContext::QGLContext().

void QGLContext::doneCurrent () `[virtual protected]`

Resets the current OpenGL rendering context. Normally, you do not need to call this function, because makeCurrent() takes care of that.

const QGLFormat& QGLContext::format () const

Returns the format.

See also: setFormat().

bool QGLContext::isValid () const

Returns TRUE if the characteristics specified in the display format could be fulfilled by the OpenGL driver/window system.

void QGLContext::makeCurrent () `[virtual]`

Makes this context the current OpenGL rendering context. All gl functions you call operate on this context until another context is made current.

void QGLContext::reset () `[virtual]`

Resets the context and makes it invalid.

See also: create() and isValid().

void QGLContext::setFormat (const QGLFormat & format) `[virtual]`

Sets a new OpenGL context specification *format.* The context is reset. Call create() to create a new context that matches this format.

```
QGLContext *cx;
  ...
QGLFormat f;
f.setAlpha( TRUE );
f.setStereo( TRUE );
cx->setFormat( f );
if ( !cx->create() )
    ; // could not create context
```

See also: format(), reset() and create().

void QGLContext::swapBuffers () `[virtual]`

Swaps the screen contents with an off-screen buffer. Works only if the context is in double buffer mode.

See also: QGLFormat::setDoubleBuffer().

74.2.1. URL

Generated from http://www.troll.no/qt/qglcontext.html.

75. QGLFormat Class Reference

`#include <qgl.h>`

75.1. Detailed Description

This class is defined in the **Qt OpenGL Extension**, which can be found in the `qt/extensions` directory. It is not included in the main Qt API.

The QGLFormat class specifies the display format of an OpenGL rendering context.

A display format has several characteristics:

- Double or single buffering.
- Depth buffer.
- RGBA or color index mode.
- Alpha channel.
- Accumulation buffer.
- Stencil buffer.
- Stereo buffer.

You create and tell a QGLFormat object what rendering options you want from an OpenGL rendering context.

OpenGL drivers or accelerated hardware may or may not support advanced features like alpha channel or stereographic viewing. If you request such features and the driver/hardware does not provide it, you will get an invalid context, which can be detected by calling the isValid() method. You can then try again with a less demanding format.

There are different ways of defining the display characteristics of a rendering context. One is to create a QGLFormat and make it default for the entire application:

```
QGLFormat f;
f.setAlpha( TRUE );
f.setStereo( TRUE );
QGLFormat::setDefaultFormat( f );
```

You can also set a format for a specific widget (subclassed from QGLWidget):

```
MyGLWidget *w;
  ...
QGLFormat f;
f.setAlpha( TRUE );
f.setStereo( TRUE );
w->setFormat( f );
if ( !w->isValid() ) {
    f.setStereo( FALSE );   // ok, goggles off
    w->setFormat( f );
    if ( !w->isValid() ) {
        fatal( "Cool hardware wanted" );
    }
}
```

Implementation note: QGLFormat objects use implicit sharing. This is an internal optimization that minimizes copying and memory overhead. When you pass a QGLFormat as an argument to a function, or return a QGLFormat from a function, a reference count is incremented or decremented. Objects stay shared until one of them is changed.

See also: QGLContext and QGLWidget.

75.2. Member Function Documentation

QGLFormat::QGLFormat (bool doubleBuffer=TRUE)

Constructs a QGLFormat object with the default settings:

- Double buffer: Enabled if *doubleBuffer* is TRUE, otherwise single buffer.
- Depth buffer: Enabled.
- RGBA: Enabled (i.e. color index disabled).
- Alpha channel: Disabled.
- Accumulator buffer: Disabled.
- Stencil buffer: Disabled.
- Stereo: Disabled.

QGLFormat::QGLFormat (const QGLFormat & f)

Constructs a QGLFormat object which is a copy of *f*.

QGLFormat::~QGLFormat () `[virtual]`

Destroys the object.

bool QGLFormat::accum () const

Returns TRUE if the accumulation buffer is enabled, otherwise FALSE. The accumulation buffer is disabled by default.

See also: setAccum().

bool QGLFormat::alpha () const

Returns TRUE if the alpha channel of the framebuffer is enabled, otherwise FALSE. The alpha channel is disabled by default.

See also: setAlpha().

const QGLFormat & QGLFormat::defaultFormat () `[static]`

Returns the default QGLFormat for the application. All QGLWidgets that are created are assigned this format unless anything else is specified.

bool QGLFormat::depth () const

Returns TRUE if the depth buffer is enabled, otherwise FALSE. The depth buffer is enabled by default.

See also: setDepth().

bool QGLFormat::doubleBuffer () const

Returns TRUE if double buffering is enabled, otherwise FALSE. Double buffering is enabled by default.

See also: setDoubleBuffer().

bool QGLFormat::hasOpenGL () `[static]`

Returns TRUE if the window system has any OpenGL support, otherwise FALSE.

QGLFormat & QGLFormat::operator= (const QGLFormat & f)

Assigns *f* to this QGLFormat object and returns a reference to this object.

bool QGLFormat::rgba () const

Returns TRUE if RGBA color mode is set, or FALSE if color index mode is set. The default color mode is RGBA.

See also: setRgba().

void QGLFormat::setAccum (bool enable)

Enables the accumulation buffer if *enable* is TRUE, or disables it if *enable* is FALSE.

The accumulation buffer is disabled by default.

The accumulation buffer is used for create blur effects and multiple exposures.

See also: accum().

void QGLFormat::setAlpha (bool enable)

Enables the alpha channel of the framebuffer if *enable* is TRUE, or disables it if *enable* is FALSE.

The alpha buffer is disabled by default.

The alpha channel is typically used for implementing transparency or translucency. The A in RGBA specifies the transparency of a pixel.

See also: alpha().

void QGLFormat::setDefaultFormat (const QGLFormat & f) `[static]`

Sets a new default QGLFormat for the application. For example, to set single buffering as default instead of double buffering, your main() can contain:

```
QApplication a(argc, argv);
QGLFormat f;
f.setDoubleBuffer( FALSE );
QGLFormat::setDefaultFormat( f );
```

void QGLFormat::setDepth (bool enable)

Enables the depth buffer if *enable* is TRUE, or disables it if *enable* is FALSE.

The depth buffer is enabled by default.

The purpose of a depth buffer (or z-buffering) is to remove hidden surfaces. Pixels are assigned z values based on the distance to the viewer. A pixel with a high z value is closer to the viewer than a pixel with a low z value. This information is used to decide whether to draw a pixel or not.

See also: depth().

void QGLFormat::setDoubleBuffer (bool enable)

Sets double buffering if *enable* is TRUE og single buffering if *enable* is FALSE.

Double buffering is enabled by default.

Double buffering is a technique where graphics is rendered to an off-screen buffer and not directly to the screen. When the drawing has been completed, the program calls a swapBuffers function to exchange the screen contents with the buffer. The result is flicker-free drawing and often better performance.

See also: doubleBuffer(), QGLContext::swapBuffers() and QGLWidget::swapBuffers().

void QGLFormat::setRgba (bool enable)

Sets RGBA mode if *enable* is TRUE, or color index mode if *enable* is FALSE.

The default color mode is RGBA.

RGBA is the preferred mode for most OpenGL applications. In RGBA color mode you specify colors as a red + green + blue + alpha quadruplet

In color index mode you specify an index into a color lookup table.

See also: rgba().

void QGLFormat::setStencil (bool enable)

Enables the stencil buffer if *enable* is TRUE, or disables it if *enable* is FALSE.

The stencil buffer is disabled by default.

The stencil buffer masks away drawing from certain parts of the screen.

See also: stencil().

void QGLFormat::setStereo (bool enable)

Enables stereo buffering if *enable* is TRUE, or disables it if *enable* is FALSE.

Stereo buffering is disabled by default.

Stereo buffering provides extra color buffers to generate left-eye and right-eye images.

See also: stereo().

bool QGLFormat::stencil () const

Returns TRUE if the stencil buffer is enabled, otherwise FALSE. The stencil buffer is disabled by default.

See also: setStencil().

bool QGLFormat::stereo () const

Returns TRUE if stereo buffering is enabled, otherwise FALSE. Stereo buffering is disabled by default.

See also: setStereo().

75.3. Related Functions

const char * qGLVersion ()

Returns the version number string for the Qt OpenGL extension, e.g. "1.0".

75.3.1. URL

Generated from http://www.troll.no/qt/qglformat.html.

76. QGList Class Reference

`#include <qglist.h>`

Inherits QCollection.

Inherited by QList, QQueue and QStack.

76.1. Detailed Description

The QGList class is an internal class for implementing Qt collection classes.

QGList is a strictly internal class that acts as a base class for several collection classes; QList, QQueue and QStack.

QGList has some virtual functions that can be reimplemented to customize the subclasses.

- compareItems() compares two collection/list items.
- read() reads a collection/list item from a QDataStream.
- write() writes a collection/list item to a QDataStream. Normally, you do not have to reimplement any of these functions. If you still want to reimplement them, see the QStrList class (qstrlist.h), which is a good example.

76.2. Member Function Documentation

QGList::QGList () `[protected]`

For internal use only.

QGList::QGList (const QGList & list) `[protected]`

For internal use only.

QGList::~QGList () `[protected]`

For internal use only.

void QGList::append (GCI d) `[protected]`

For internal use only.

GCI QGList::at (uint index) `[protected]`

For internal use only.

int QGList::at () const `[protected]`

For internal use only.

GCI QGList::cfirst () const `[protected]`

For internal use only.

GCI QGList::clast () const `[protected]`

For internal use only.

void QGList::clear () `[virtual protected]`

For internal use only.

Reimplemented from QCollection.

int QGList::compareItems (GCI item1, GCI item2) `[virtual protected]`

This virtual function compares two list items.

Returns:

- 0 if *item1* == *item2*
- non-zero if *item1* != *item2*

This function returns *int* rather than *bool* so that reimplementations can return three values and use it to sort by:

- 0 if *item1* == *item2*
- > 0 (positive integer) if *item1* > *item2*
- < 0 (negative integer) if *item1* < *item2*

The QList::inSort() function requires that compareItems() is implemented as described here.

This function should not modify the list because some const functions call compareItems().

The default implementation compares the pointers:

uint QGList::contains (GCI d) const `[protected]`

For internal use only.

uint QGList::containsRef (GCI d) const `[protected]`

For internal use only.

uint QGList::count () const `[virtual]`

For internal use only.

Reimplemented from QCollection.

QLNode * QGList::currentNode () const `[protected]`

For internal use only.

int QGList::find (GCI d, bool fromStart = TRUE) `[protected]`

For internal use only.

int QGList::findRef (GCI d, bool fromStart = TRUE) `[protected]`

For internal use only.

GCI QGList::first () `[protected]`

For internal use only.

GCI QGList::get () const `[protected]`

For internal use only.

void QGList::inSort (GCI d) `[protected]`

For internal use only.

bool QGList::insertAt (uint index, GCI d) `[protected]`

For internal use only.

GCI QGList::last () `[protected]`

For internal use only.

GCI QGList::next () `[protected]`

For internal use only.

QGList& QGList::operator= (const QGList & list) `[protected]`

For internal use only.

GCI QGList::prev () `[protected]`

For internal use only.

QDataStream & QGList::read (QDataStream & s)

For internal use only.

QDataStream & QGList::read (QDataStream & s, GCI & item) `[virtual protected]`

Reads a collection/list item from the stream *s* and returns a reference to the stream.

The default implementation sets *item* to 0.

See also: write().

void QGList::relinkNode (QLNode * n) `[protected]`

For internal use only.

bool QGList::remove (GCI d = 0) `[protected]`

For internal use only.

bool QGList::removeAt (uint index) `[protected]`

For internal use only.

bool QGList::removeFirst () `[protected]`

For internal use only.

bool QGList::removeLast () `[protected]`

For internal use only.

bool QGList::removeNode (QLNode * n) `[protected]`

For internal use only.

bool QGList::removeRef (GCI d = 0) `[protected]`

For internal use only.

GCI QGList::take () `[protected]`

For internal use only.

GCI QGList::takeAt (uint index) `[protected]`

For internal use only.

GCI QGList::takeFirst () `[protected]`

For internal use only.

GCI QGList::takeLast () `[protected]`

For internal use only.

GCI QGList::takeNode (QLNode * n) `[protected]`

For internal use only.

void QGList::toVector (QGVector * vector) const `[protected]`

For internal use only.

QDataStream & QGList::write (QDataStream & s) const

For internal use only.

QDataStream & QGList::write (QDataStream & s, GCI) const `[virtual protected]`

Writes a collection/list item to the stream *s* and returns a reference to the stream.

The default implementation does nothing.

See also: read().

76.2.1. URL

Generated from http://www.troll.no/qt/qglist.html.

77. QGListIterator Class Reference

`#include <qglist.h>`

Inherited by QListIterator.

77.1. Detailed Description

The QGListIterator is an internal class for implementing QListIterator.

QGListIterator is a strictly internal class that does the heavy work for QListIterator.

77.2. Member Function Documentation

QGListIterator::QGListIterator (const QGList & l) `[protected]`

For internal use only.

QGListIterator::QGListIterator (const QGListIterator & it) `[protected]`

For internal use only.

QGListIterator::~QGListIterator () `[protected]`

For internal use only.

bool QGListIterator::atFirst () const `[protected]`

For internal use only.

bool QGListIterator::atLast () const `[protected]`

For internal use only.

GCI QGListIterator::get () const `[protected]`

For internal use only.

GCI QGListIterator::operator() () `[protected]`

For internal use only.

GCI QGListIterator::operator++ () `[protected]`

For internal use only.

GCI QGListIterator::operator+= (uint jumps) `[protected]`

For internal use only.

GCI QGListIterator::operator-- () `[protected]`

For internal use only.

GCI QGListIterator::operator-= (uint jumps) `[protected]`

For internal use only.

QGListIterator & QGListIterator::operator= (const QGListIterator & it) `[protected]`

For internal use only.

GCI QGListIterator::toFirst () `[protected]`

For internal use only.

GCI QGListIterator::toLast () `[protected]`

For internal use only.

77.2.1. URL

Generated from http://www.troll.no/qt/qglistiterator.html.

78. QGLWidget Class Reference

`#include <qgl.h>`

Inherits QWidget.

78.1. Detailed Description

This class is defined in the **Qt OpenGL Extension**, which can be found in the `qt/extensions` directory. It is not included in the main Qt API.

The QGLWidget class is a widget for rendering OpenGL graphics.

QGLWidget provides functionality for displaying OpenGL graphics integrated in a Qt application. It is very simple to use: you inherit from it and use the subclass like any other QWidget, only that instead of drawing the widget's contents using QPainter & al., you use the standard OpenGL rendering commands.

QGLWidget provides three convenient virtual functions that you can reimplement in your subclass to perform the typical OpenGL tasks:

- paintGL() - Render the OpenGL scene. Gets called whenever the widget needs to be updated.
- resizeGL() - Set up OpenGL viewport, projection etc. Gets called whenever the the widget has been resized (and also when it shown for the first time, since all newly created widgets get a resize event automatically).
- initializeGL() - Set up the OpenGL rendering context, define display lists etc. Gets called once before the first time resizeGL() or paintGL() is called.

Here is a rough outline of how your QGLWidget subclass may look:

```
class MyGLDrawer : public QGLWidget
{
    Q_OBJECT        // must include this if you use Qt signals/slots
public:
    MyGLDrawer( QWidget *parent, const char *name )
        : QGLWidget(parent,name) {}
protected:
    void initializeGL()
    {
      // Set up the rendering context, define display lists etc.:
      ...
      glClearColor( 0.0, 0.0, 0.0, 0.0 );
      glEnable(GL_DEPTH_TEST);
      ...
    }
    void resizeGL( int w, int h )
    {
      // setup viewport, projection etc.:
      glViewport( 0, 0, (GLint)w, (GLint)h );
      ...
      glFrustum( ... );
      ...
    }
    void paintGL()
    {
      // draw the scene:
      ...
      glRotatef( ... );
      glMaterialfv( ... );
      glBegin( GL_QUADS );
      glVertex3f( ... );
      glVertex3f( ... );
      ...
      glEnd();
```

```
        ...
    }
};
```

If you need to trigger a repaint from other places than paintGL() (a typical example is when using timers to animate scenes), you should call the widget's updateGL() function.

When paintGL(), resizeGL() or initializeGL() is called, your widget's OpenGL rendering context has been made current. If you need to call the standard OpenGL API functions from other places (e.g. in your widget's constructor), you must call makeCurrent() first.

QGLWidget provides advanced functions for requesting a new display format, and you can even set a new rendering context.

You can achieve sharing of OpenGL display lists between QGLWidgets, see the documentation of the QGLWidget constructors for details. *Portability note:* Some window systems require that the sharing widgets/contexts have the same format

78.2. Member Function Documentation

QGLWidget::QGLWidget (QWidget * parent=0, const char * name=0, const QGLWidget * shareWidget = 0, WFlags f=0)

Constructs an OpenGL widget with a *parent* widget and a *name.*

The default format is used.

The *parent, name* and *f* arguments are passed to the QWidget constructor.

If the *shareWidget* parameter points to a valid QGLWidget, this widget will share OpenGL display lists with *shareWidget.*

Note: Initialization of OpenGL rendering state etc. should be done by overriding the initializeGL() function, not in the constructor of your QGLWidget subclass.

See also: QGLFormat::defaultFormat().

QGLWidget::QGLWidget (const QGLFormat & format, QWidget * parent=0, const char * name=0, const QGLWidget * shareWidget = 0, WFlags f=0)

Constructs an OpenGL widget with a *parent* widget and a *name.*

The *format* argument specifies the rendering capabilities. The widget becomes invalid if the driver/hardware cannot satisfy the requested format.

The *parent, name* and *f* arguments are passed to the QWidget constructor.

If the *shareWidget* parameter points to a valid QGLWidget, this widget will share OpenGL display lists with *shareWidget.*

Note: Initialization of OpenGL rendering state etc. should be done by overriding the initializeGL() function, not in the constructor of your QGLWidget subclass.

See also: QGLFormat::defaultFormat() and isValid().

QGLWidget::~QGLWidget ()

Destroys the widget.

const QGLContext * QGLWidget::context () const

Returns the current context.

See also: setContext().

bool QGLWidget::doubleBuffer () const

Returns TRUE if double buffering is set in the format for this widget.

See also: QGLFormat::doubleBuffer().

const QGLFormat & QGLWidget::format () const

Returns the widget's format.

See also: setFormat().

void QGLWidget::gl_init () `[protected]`

Initializes OpenGL for this widget's context. Calls the virtual function initializeGL().

void QGLWidget::initializeGL () `[virtual protected]`

This virtual function is called one time before the first call to paintGL() or resizeGL(), and then one time whenever the widget has been assigned a new QGLContext. Reimplement it in a subclass.

This function should take care of setting any required OpenGL context rendering flags, defining display lists, etc.

There is no need to call makeCurrent() because this has already been done when this function is called.

bool QGLWidget::isValid () const

Returns TRUE if the widget was able to satisfy the specified constraints.

void QGLWidget::makeCurrent () `[virtual]`

Makes this widget the current widget for OpenGL operations. I.e. makes this widget's rendering context the current OpenGL rendering context.

void QGLWidget::paintEvent (QPaintEvent *) `[virtual protected]`

Handles paint events. Calls the virtual function paintGL().

Reimplemented from QWidget.

void QGLWidget::paintGL () `[virtual protected]`

This virtual function is called whenever the widget needs to be painted. Reimplement it in a subclass.

There is no need to call makeCurrent() because this has already been done when this function is called.

void QGLWidget::resizeEvent (QResizeEvent *) `[virtual protected]`

Handles resize events. Calls the virtual function resizeGL().

Reimplemented from QWidget.

void QGLWidget::resizeGL (int width, int height) `[virtual protected]`

This virtual function is called whenever the widget has been resized. Reimplement it in a subclass.

There is no need to call makeCurrent() because this has already been done when this function is called.

void QGLWidget::setContext (QGLContext * context, const QGLContext * shareContext = 0) `[virtual]`

Sets a new context for this widget. The context must be created using *new*. QGLWidget will delete the context when another context is set or when the widget is destroyed.

The initializeGL() function will be executed for the new context before the first resizeGL() or paintGL().

Calling setContext() will not alter any display list sharing this widget currently has, unless you explicitly request sharing with a different context by giving a *shareContext* parameter that points to a valid context.

See also: context() and setFormat().

void QGLWidget::setFormat (const QGLFormat & format) `[virtual]`

Sets a new format for this widget. The widget becomes invalid if the requested format cannot be satisfied.

The widget will be assigned a new QGLContext, and the initializeGL() function will be executed for this new context before the first resizeGL() or paintGL().

Calling setFormat() will not alter any display list sharing this widget currently has.

See also: format(), setContext() and isValid().

void QGLWidget::swapBuffers () `[virtual]`

Swaps the screen contents with an off-screen buffer. Works only if the widget's format specifies double buffer mode.

See also: doubleBuffer() and QGLFormat::setDoubleBuffer().

void QGLWidget::updateGL () `[virtual slot]`

Updates the widget by calling paintGL().

78.2.1. URL

Generated from http://www.troll.no/qt/qglwidget.html.

79. QGManager Class Reference

`#include <qgmanager.h>`

Inherits QObject.

79.1. Detailed Description

The QGManager class provides one-dimensional geometry management.

This class is intended for those who write geometry managers and graphical designers. **It is not for the faint of heart.** The QHBoxLayout, QVBoxLayout and QGridLayout classes are available for normal application programming.

Each dimension (horizontal and vertical) is handled independently. Widgets are organized in chains, which can be parallel or serial.

In a serial chain, elements are added one after another. Available space is divided among the elements according to stretch and max-/minsize.

In parallel chains, elements are added on top of each other. All elements are given the full length of the chain, and are placed at the same position.

See also: QHBoxLayout, QVBoxLayout and QGridLayout.

79.2. Member Function Documentation

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

Creates a new QGManager which manages *parent's* children.

QGManager::~QGManager ()

Destroys the QGManager, deleting all add()ed chains.

bool QGManager::activate ()

Starts geometry management.

bool QGManager::add (QChain * destination, QChain * source, int stretch = 0)

Adds the chain *source* to the chain *destination.*

bool QGManager::addBranch (QChain * destination, QChain * branch, int fromIndex, int toIndex)

Adds *branch* to *destination* as a branch going from *fromIndex* to *toIndex.* A branch is a chain that is anchored at two locations in a serial chain. The branch does not influence the main chain; if the branch's minimum size is greater than the minimum distance between the anchors, things will look ugly.

The branch goes from the beginning of the item at *fromIndex* to the end of the item at *toIndex.* Note: remember to count spacing when calculating indices.

bool QGManager::addSpacing (QChain * d, int minSize, int stretch = 0, int maxSize = unlimited)

Adds the spacing *w* to the chain *d.* If *d* is a serial chain, this means screen space between widgets. If *d* is parallel, this influences the maximum and minimum size.

bool QGManager::addWidget (QChain * d, QWidget * w, int stretch = 0)

Adds the widget *w* to the chain *d.*

bool QGManager::eventFilter (QObject * o, QEvent * e) `[virtual protected]`

Grabs all resize events for my parent, and does child widget resizing.

Reimplemented from QObject.

void QGManager::freeze (int w = 0, int h = 0)

Fixes the size of the main widget and distributes the available space to the child widgets. The size is adjusted to a valid value. Thus freeze(0,0) (the default) will fix the widget to its minimum size.

QWidget * QGManager::mainWidget ()

Returns the main widget of the manager.

QChain * QGManager::newParChain (Direction d)

Creates a new QChain which is *parallel.*

QChain * QGManager::newSerChain (Direction d)

Creates a new QChain which is *serial.*

void QGManager::remove (QWidget * w)

Removes the widget *w* from geometry management. Will not remove spacing around the widget. This function is called automatically if the widget is deleted.

void QGManager::setBorder (int b)

Sets the border around the edge of the widget. *b* is the number of pixels between the edge of the widget and the area controlled by the manager.

void QGManager::setMenuBar (QWidget * w)

Makes the geometry manager take account of the menu bar *w.* All child widgets are placed below the bottom edge of the menu bar.

void QGManager::setName (QChain * chain, const char * name)

Sets the name of *chain* to *name.* The name is used for debugging purposes.

void QGManager::setStretch (QChain * c, int s)

Sets the stretch factor on *c* to *s.* This stretch factor is overridden by add(), so there's no point in calling this function before you add() the chain.

See also: add().

void QGManager::unFreeze ()

Undoes the effect of a previous freeze(). The main widget will now again be resizable.

QChain * QGManager::xChain ()

Returns the main horizontal chain of the manager. All horizontal chains should be inserted into this chain or one of its descendants, otherwise they will be ignored.

QChain * QGManager::yChain ()

Returns the main vertical chain of the manager. All vertical chains should be inserted into this chain or one of its descendants, otherwise they will be ignored.

79.2.1. URL

Generated from http://www.troll.no/qt/qgmanager.html.

80. QGridLayout Class Reference

#include <qlayout.h>

Inherits QLayout.

80.1. Detailed Description

The QGridLayout class lays out child widgets in a grid.

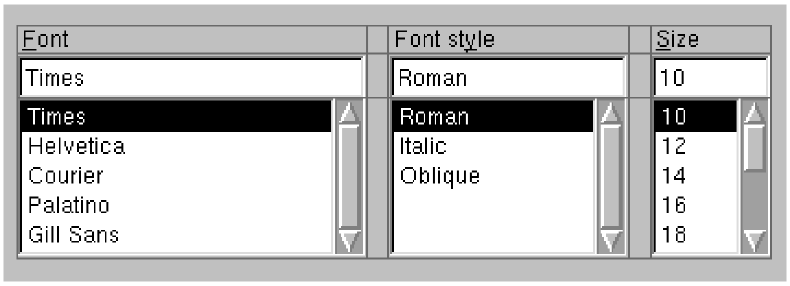
QGridLayout takes the space it gets (from its parent layout or from the mainWidget()), divides it up into rows and columns, and puts each of the child widgets it manages into the correct cell(s).

Columns and rows behave identically; we will discuss columns but there are eqivalent functions for rows.

Each column has a minimum width and a stretch factor. The minimum width is the greatest of that set using addColSpacing() and the minimum width of each widget in that column. The stretch factor is set using setColStretch() and determines how much of the available space the column will get, over and above its necessary minimum.

Normally, each child widget or layout is put into a cell of its own using addWidget() or addLayout(), but you can also put widget into multiple cells using addMultiCellWidget(). However, if you do that, QGridLayout does not take the child widget's minimum size into consideration (because it cannot know what column the minimum width should belong to). Thus you must set the minimum width of each column using addColSpacing().

This illustration shows a fragment of a dialog with a five-column, three-row grid (the grid is shown overlaid in magenta):



Columns 0, 2 and 4 in this dialog fragment are made up of a QLabel, a QLineEdit and a QListBox. Columns 1 and 2 are placeholders, made with setColSpacing(). Row 0 consists of three QLabel objects, row 1 of three QLineEdit objects and row 2 of three QListBox objects.

Since we did not want any space between the rows, we had to use placeholder columns to get the right amount of space between the columns.

Note that the columns and rows are not equally wide/tall: If you want two columns to be equally wide, you must set the columns' minimum widths and stretch factors to be the same yourself. You do this using addColSpacing() and setStretch().

If the QGridLayout is not the top-level layout (ie. is not managing all of the widget's area and children), you must add it to its parent layout when you have created it, but before you can do anything with it. The normal way to add a layout is by calling parentLayout->addLayout().

Once you have done that, you can start putting widgets and other layouts in the cells of your grid layout using addWidget(), addLayout() and addMultiCellWidget().

Finally, if the grid is the top-level layout, you activate() it.

QGridLayout also includes two margin widths: The border width and the inter-box width. The border width is the width of the reserved space along each of the QGridLayout's four sides. The intra-widget width is the width of the automatically allocated spacing between neighbouring boxes.

The border width defaults to 0, and the intra-widget width defaults to the same as the border width. Both are set using arguments to the constructor.

Examples:
pref/pref.cpp
layout/layout.cpp
widgets/widgets.cpp

80.2. Member Function Documentation

QGridLayout::QGridLayout (QWidget * parent, int nRows, int nCols, int border=0, int autoBorder = -1, const char * name=0)

Constructs a new QGridLayout with *nRows* rows, *nCols* columns and main widget *parent. parent* may not be 0.

border is the number of pixels between the edge of the widget and the managed children. *autoBorder* is the default number of pixels between cells. If *autoBorder* is -1 the value of *border* is used.

name is the internal object name.

QGridLayout::QGridLayout (int nRows, int nCols, int autoBorder = -1, const char * name=0)

Constructs a new grid with *nRows* rows and *nCols* columns, If *autoBorder* is -1, this QGridLayout will inherits its parent's defaultBorder(), otherwise *autoBorder* is used.

You have to insert this grid into another layout before using it.

QGridLayout::~QGridLayout ()

Deletes this grid. Geometry management is terminated if this is a top-level grid.

void QGridLayout::addColSpacing (int col, int minsize)

Sets the minimum width of *col* to *minsize* pixels.

void QGridLayout::addLayout (QLayout * layout, int row, int col)

Places another layout at position (*row, col)* in the grid. The top left position is (0,0).

Examples:
widgets/widgets.cpp

void QGridLayout::addMultiCellWidget (QWidget * w, int fromRow, int toRow, int fromCol, int toCol, int align = 0)

Adds the widget *w* to the cell grid, spanning multiple rows/columns.

Note that multicell widgets do not influence the minimum or maximum size of columns/rows they span. Use addColSpacing() or addRowSpacing() to set minimum sizes explicitly.

Alignment is specified by *align* which takes the same arguments as QLabel::setAlignment(), alignment has no effect unless you have set QWidget::maximumSize().

Examples:
layout/layout.cpp

widgets/widgets.cpp

void QGridLayout::addRowSpacing (int row, int minsize)

Sets the minimum height of *row* to *minsize* pixels.

void QGridLayout::addWidget (QWidget * w, int row, int col, int align = 0)

Adds the widget *w* to the cell grid at *row, col.* The top left position is (0,0)

Alignment is specified by *align* which takes the same arguments as QLabel::setAlignment(). Note that widgets take all the space they can get; alignment has no effect unless you have set QWidget::maximumSize().

Examples:
pref/pref.cpp
layout/layout.cpp
widgets/widgets.cpp

void QGridLayout::expand (int nRows, int nCols)

Expands this grid so that it will have *nRows* rows and *nCols* columns. Will not shrink the grid.

void QGridLayout::initGM () `[virtual protected]`

Initializes this grid.

Reimplemented from QLayout.

QChain * QGridLayout::mainHorizontalChain () `[virtual protected]`

This function returns the main horizontal chain.

Reimplemented from QLayout.

QChain * QGridLayout::mainVerticalChain () `[virtual protected]`

This function returns the main vertical chain.

Reimplemented from QLayout.

int QGridLayout::numCols () const

Returns the number of columns in this grid.

int QGridLayout::numRows () const

Returns the number of rows in this grid.

void QGridLayout::setColStretch (int col, int stretch)

Sets the stretch factor of column *col* to *stretch.* The first column is number 0.

The stretch factor is relative to the other columns in this grid. Columns with higher stretch factor take more of the available space.

The default stretch factor is 0. If the stretch factor is 0 and no other column in this table can grow at all, the column may still grow.

Examples:
pref/pref.cpp
layout/layout.cpp
widgets/widgets.cpp

void QGridLayout::setRowStretch (int row, int stretch)

Sets the stretch factor of row *row* to *stretch.* The first row is number 0.

The stretch factor is relative to the other rows in this grid. Rows with higher stretch factor take more of the available space.

The default stretch factor is 0. If the stretch factor is 0 and no other row in this table can grow at all, the row may still grow.

Examples:
widgets/widgets.cpp

80.2.1. URL

Generated from http://www.troll.no/qt/qgridlayout.html.

81. QGroupBox Class Reference

`#include <qgroupbox.h>`

Inherits QFrame.

Inherited by QButtonGroup.

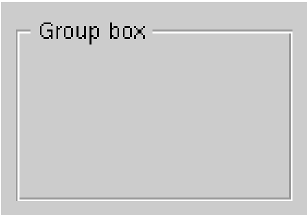
81.1. Detailed Description

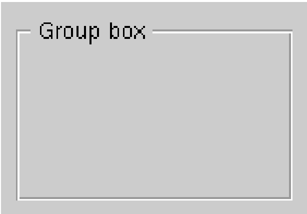
The QGroupBox widget provides a group box frame with a title.

The intended use of a group box is to show that a number of widgets (i.e. child widgets) are logically related. QPrintDialog is a good example; each of its three panes is one group box.

By default, the group's setFont() and setPalette() functions do not change the appearance of the widgets it containsn, but you can use setFontPropagation() and setPalettePropagation() to change that.

Qt also provides a more specialized group box, QButtonGroup, that is very useful for organizing buttons in a group.





Examples:
xform/xform.cpp

81.2. Member Function Documentation

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

Constructs a group box widget with no title.

The *parent* and *name* arguments are passed to the QWidget constructor.

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

Constructs a group box with a title.

The *parent* and *name* arguments are passed to the QWidget constructor.

int QGroupBox::alignment () const

Returns the alignment of the group box title.

The default alignment is `AlignHCenter.`

See also: setAlignment().

void QGroupBox::paintEvent (QPaintEvent * event) `[virtual protected]`

Handles paint events for the group box.

Reimplemented from QWidget.

void QGroupBox::setAlignment (int alignment)

Sets the alignment of the group box title.

The title is always placed on the upper frame line, however, the horizontal alignment can be specified by the *alignment* parameter.

The *alignment* is the bitwise OR of the following flags:

- `AlignLeft` aligns the title text to the left.
- `AlignRight` aligns the title text to the right.
- `AlignHCenter` aligns the title text centered.

See also: alignment().

void QGroupBox::setTitle (const char * title)

Sets the group box title text to *title.*

Examples:
xform/xform.cpp
widgets/widgets.cpp

const char* QGroupBox::title () const

Returns the group box title text.

81.2.1. URL

Generated from http://www.troll.no/qt/qgroupbox.html.

82. QHBoxLayout Class Reference

`#include <qlayout.h>`

Inherits QBoxLayout.

82.1. Detailed Description

The QHBoxLayout class lines up child widgets horizontally.

This class provides an easier way to construct horizontal box layout objects. See QBoxLayout for more details.

The simplest way to use this class is:

```
QBoxLayout * l = new QHBoxLayout( widget );
l->addWidget( aWidget );
l->addWidget( anotherWidget );
l->activate()
```

See also: QVBoxLayout and QGridLayout.

Examples:
layout/layout.cpp
widgets/widgets.cpp

82.2. Member Function Documentation

QHBoxLayout::QHBoxLayout (QWidget * parent, int border=0, int autoBorder = -1, const char * name=0)

Creates a new top-level horizontal box.

QHBoxLayout::QHBoxLayout (int autoBorder = -1, const char * name=0)

Creates a new horizontal box. You have to add it to another layout before using it.

QHBoxLayout::~QHBoxLayout ()

Destroys this box.

82.2.1. URL

Generated from http://www.troll.no/qt/qhboxlayout.html.

83. QHeader Class Reference

```
#include <qheader.h>
```

Inherits QTableView.

83.1. Detailed Description

The QHeader class provides a table header.

This is a table heading of the type used in a list view. It gives the user the opportunity to resize and move the columns (or rows for vertical headings).

This class can be used without a table view, if you need to control table-like structures.

See also: QListView, QTableView and Microsoft Style Guide

83.2. Member Function Documentation

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

Constructs a horizontal header.

The *parent* and *name* arguments are sent to the QWidget constructor.

QHeader::QHeader (int n, QWidget * parent=0, const char * name=0)

Constructs a horizontal header with *n* sections.

The *parent* and *name* arguments are sent to the QWidget constructor.

QHeader::~QHeader ()

Destroys the header.

int QHeader::addLabel (const char * s, int size = -1)

Adds a new section, with label text *s.* Returns the index. If *size* is non-negative, the section width is set to *size,* otherwise a size currently sufficient for the label text is used.

int QHeader::cellAt (int c) const

Returns the actual index of the section at position *c,* or -1 if outside.

int QHeader::cellHeight (int row) `[virtual protected]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QTableView.

int QHeader::cellPos (int i) const

Returns the position in pixels of section *i* of the header. *i* is the actual index.

int QHeader::cellSize (int i) const

Returns the size in pixels of section *i* of the header. *i* is the actual index.

int QHeader::cellWidth (int col) `[virtual protected]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QTableView.

int QHeader::count () const

Returns the number of sections in the header.

const char* QHeader::label (int i)

Returns the text set on logical section *i*.

int QHeader::mapToActual (int l) const

Translates from logical index *l* to actual index. Returns -1 if *l* is outside the legal range.

int QHeader::mapToLogical (int a) const

Translates from actual index *a* to logical index. Returns -1 if *a* is outside the legal range.

void QHeader::moved (int from, int to) `[signal]`

This signal is emitted when the user has moved column *from* to position *to*.

int QHeader::offset () const

int QHeader::offset() const Returns the leftmost (or uppermost for vertical headers) visible pixel.

QHeader::Orientation QHeader::orientation() const

Returns `Horizontal` if the header is horizontal, `Vertical` if the header is vertical.

void QHeader::paintCell (QPainter * p, int row, int col) `[virtual protected]`

paints a section of the header.

Reimplemented from QTableView.

void QHeader::resizeEvent (QResizeEvent *) `[virtual protected]`

Handles resize events.

Reimplemented from QWidget.

QRect QHeader::sRect (int i) `[protected]`

Returns the rectangle covered by actual section *i*.

void QHeader::sectionClicked (int logical) `[signal]`

This signal is emitted when a part of the header is clicked. In a list view, this signal would typically be connected to a slot which sorts the specified column.

void QHeader::setCellSize (int i, int s)

Sets the size of logical cell *i* to *s* pixels.

Warning: does not repaint or send out signals at present.

void QHeader::setClickEnabled (bool enable, int i = -1)

Enable clicking in column *i* if *enable* is TRUE, disable otherwise. If *i* is negative, clicking is enabled/disabled for all columns.

If enabled, the sectionClicked() signal is emitted when the user clicks.

See also: setMovingEnabled() and setResizeEnabled().

void QHeader::setLabel (int i, const char * s, int size = -1)

Sets the text on logical section *i* to *s*. If the section does not exist, nothing happens. If *size* is non-negative, the section width is set to *size*.

void QHeader::setMovingEnabled (bool enable)

Enable the user to exchange columns if *enable* is TRUE, disable otherwise.

See also: setClickEnabled() and setResizeEnabled().

void QHeader::setOffset (int x) `[slot]`

Scrolls the header such that *x* becomes the leftmost (or uppermost for vertical headers) visible pixel.

void QHeader::setOrientation (Orientation orientation)

Sets the header orientation. The *orientation* must be QQHeader::Vertical or QHeader::Horizontal.

See also: orientation().

void QHeader::setResizeEnabled (bool enable, int i = -1)

Enable user resizing of column *i* if *enable* is TRUE, disable otherwise. If *i* is negative, resizing is enabled/disabled for all columns.

See also: setMovingEnabled() and setClickEnabled().

void QHeader::setTracking (bool enable)

Sets tracking if *enable* is TRUE, otherwise turns off tracking. If tracking is on, the sizeChange() signal is emitted continuously while the mouse is moved, otherwise it is only emitted when the mouse button is released.

See also: tracking().

void QHeader::setupPainter (QPainter * p) `[virtual protected]`

sets up the painter.

Reimplemented from QTableView.

void QHeader::sizeChange (int section, int oldSize, int newSize) `[signal]`

This signal is emitted when the user has changed the size of some of the parts of the header. This signal is typically connected to a slot that repaints the table. *section* is the logical section resized.

QSize QHeader::sizeHint () const `[virtual]`

Returns the recommended size of the QHeader.

Reimplemented from QWidget.

bool QHeader::tracking () const

Returns TRUE if tracking is on, FALSE otherwise.

See also: setTracking().

83.2.1. URL

Generated from http://www.troll.no/qt/qheader.html.

84. QIconSet Class Reference

```
#include <qiconset.h>
```

84.1. Detailed Description

The QIconSet class provides a set of icons (normal, disabled, various sizes) for e.g. buttons.

QIconSet must be fed at least one icon, and can generate the other icons from the ones it is fed, or use programmer-specified icons.

Using the icon or icons specified, QIconSet generates a set of six icons:

- Small, normal
- Small, disabled
- Small, active
- Large, normal
- Large, disabled
- Large, active

You can set any of the icons using setPixmap() and when you retrieve one using pixmap(), QIconSet will compute and cache that from the closest other icon.

The `Disabled` appearance is computed using a "shadow" algorithm which produces results very similar to that used in of Microsoft Windows 95.

The `Active` appearance is identical to the `Normal` appearance unless you use setPixmap() to set it to something special.

QIconSet provides a function, isGenerated(), that indicates whether an icon was set by the application programmer or computed by QIconSet itself.

In Qt 1.41 only QToolButton uses QIconSet. In Qt 2.0 we will use it in more classes, including the menu system.

See also: QPixmap, QLabel, QToolButton, GUI Design Handbook: Iconic Label,, <q, href="http://www.microsoft.com/clipgallerylive/icons.asp">Microsoft, Icon and Gallery.

84.2. Member Function Documentation

QIconSet::QIconSet (const QPixmap & defaultPixmap, Size defaultSize = Automatic)

Constructs an icon set that will generate its members from *defaultPixmap,* which is assumed to be of *defaultSize.*

The default for *defaultSize* is `Automatic,` which means that QIconSet will determine the icon's size from its actual size.

See also: reset().

QIconSet::QIconSet (const QIconSet & other)

Constructs an a copy of *other.* This is very fast.

QIconSet::~QIconSet () `[virtual]`

Destroys the icon set and frees any allocated resources.

bool QIconSet::isGenerated (Size s, Mode m) const

Returns TRUE if the variant with size *s* and mode *m* was automatically generated, and FALSE if it was not.

QIconSet & QIconSet::operator= (const QIconSet & p)

Assigns *other* to this icon set and returns a reference to this icon set.

This is very fast.

QPixmap QIconSet::pixmap () const

Returns the pixmap originally provided to the constructor or reset().

See also: reset().

QPixmap QIconSet::pixmap (Size s, Mode m) const

Returns a pixmap with size *s* and mode *m,* generating one if needed.

void QIconSet::reset (const QPixmap & pm, Size s)

Set this icon set to display *pm,* which is assumed to be in size *s.* If *s* is `Automatic`, QIconSet guesses the size from the size of *pm* using and unspecified algorithm.

void QIconSet::setPixmap (const QPixmap & pm, Size s, Mode m = Normal)

Sets this icon set to display *pn* in size *s/mode m,* and perhaps to use *pm* for deriving some other varieties.

s must be Large or Small; it cannot be Automatic.

void QIconSet::setPixmap (const char * fileName, Size s, Mode m = Normal)

Sets this icon set to load *fileName* as a pixmap and display it in size *s/mode m,* and perhaps to use *pm* for deriving some other varieties.

84.2.1. URL

Generated from http://www.troll.no/qt/qiconset.html.

85. QImage Class Reference

```
#include <qimage.h>
```

85.1. Detailed Description

The QImage class provides a hardware-independent pixmap representation with direct access to the pixel data.

It is one of the two classes Qt provides for dealing with images, the other being QPixmap. QImage is designed and optimized for I/O and for direct pixel access/manipulation, QPixmap is designed and optimized for drawing. There are (slow) functions to convert between QImage and QPixmp; QPixmap::convertToImage() and QPixmap::convertFromImage().

An image has the parameters width, height and depth (bits per pixel, bpp), a color table and the actual pixels. QImage supports 1-bpp, 8-bpp and 32-bpp image data. 1-bpp and 8-bpp images use a color lookup table; the pixel value is a color table index.

32-bpp images encode an RGB value in 24 bits and ignore the color table. The most significant byte is used for the alpha buffer.

An entry in the color table is an RGB triplet encoded as `uint.` Use the qRed, qGreen and qBlue functions (qcolor.h) to access the components, and qRgb to make an RGB triplet (see the QColor class documentation).

1-bpp (monochrome) images have a color table with maximum 2 colors. There are two different formats; big endian (MSB first) or little endian (LSB first) bit order. To access a single bit, you will have to do some bitshifts:

```
QImage image;
  // sets bit at (x,y) to 1
if ( image.bitOrder() == QImage::LittleEndian )
    *(image.scanLine(y) + (x >> 3)) |= 1 << (x & 7);
else
    *(image.scanLine(y) + (x >> 3)) |= 1 << (7 -(x & 7));
```

If this looks complicated, it might be a good idea to convert the 1-bpp image to an 8-bpp image using convertDepth().

8-bpp images are much easier to work with than 1-bpp images because they have a single byte per pixel:

```
QImage image;
  // set entry 19 in the color table to yellow
image.setColor( 19, qRgb(255,255,0) );
  // set 8 bit pixel at (x,y) to value yellow (in color table)
*(image.scanLine(y) + x) = 19;
```

32-bpp images ignore the color table, instead each pixel contains the RGB triplet. 24 bits contain the RGB value and the most significant byte is reserved for the alpha buffer.

```
QImage image;
  // sets 32 bit pixel at (x,y) to yellow.
uint *p = (uint *)image.scanLine(y) + x;
*p = qRgb(255,255,0);
```

The scanlines are 32-bit aligned for all depths.

The QImage class uses explicit sharing, similar to that of QArray and QString.

See also: QImageIO, QPixmap and Shared Classes

Examples:

qtimage/qtimage.cpp
desktop/desktop.cpp
qmag/qmag.cpp

85.2. Member Function Documentation

QImage::QImage ()

Constructs a null image.

See also: isNull().

QImage::QImage (const QImage & image)

Constructs a shallow copy of *image.*

QImage::QImage (const char * fileName, const char * format=0)

Constructs an image from loading *fileName* and an optional *format.*

See also: load().

QImage::QImage (const char * xpm[])

Constructs an image from *xpm,* which must be a valid XPM image.

Errors are silently ignored.

QImage::QImage (int w, int h, int depth, int numColors=0, Endian bitOrder=IgnoreEndian)

Constructs an image with *w* width, *h* height, *depth* bits per pixel, *numColors* colors and bit order *bitOrder.*

Using this constructor is the same as first constructing a null image and then calling the create() function.

See also: create().

QImage::QImage (const QSize &, int depth, int numColors=0, Endian bitOrder=IgnoreEndian)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

QImage::~QImage ()

Destroys the image and cleans up.

bool QImage::allGray () const

Returns TRUE if all the colors in the image are shades of gray, that is their R, G, and B components are equal. This function is slow for large 32-bit images.

QImage::Endian QImage::bitOrder() const

Returns the bit order for the image.

If it is a 1-bpp image, this function returns either QImage::BigEndian or QImage::LittleEndian.

If it is not a 1-bpp image, this function returns QImage::IgnoreEndian.

See also: depth().

uchar * QImage::bits () const

Returns a pointer to the first pixel data. Equivalent to scanLine(0).

See also: scanLine().

int QImage::bytesPerLine () const

Returns the number of bytes per image scanline. This is equivalent to numBytes()/height().

QRgb QImage::color (int i) const

Returns the color in the color table at index *i*.

A color value is an RGB triplet. Use the QRED, QGREEN and QBLUE functions (defined in qcolor.h) to get the color value components.

See also: setColor() and QColor.

QRgb * QImage::colorTable () const

Returns a pointer to the color table.

QImage QImage::convertBitOrder (Endian bitOrder) const

Converts the bit order of the image to *bitOrder* and returns the converted image.

Returns `*this` if the *bitOrder* is equal to the image bit order, or a null image if this image cannot be converted.

See also: bitOrder() and setBitOrder().

QImage QImage::convertDepth (int depth, int conversion_flags) const

Converts the depth (bpp) of the image to *depth* and returns the converted image. The original image is left undisturbed.

The *depth* argument must be 1, 8 or 32.

See QPixmap::convertFromImage for a description of the *conversion_flags* argument.

Returns `*this` if *depth* is equal to the image depth, or a null image if this image cannot be converted.

See also: depth() and isNull().

QImage QImage::convertDepth (int depth) const

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

QImage QImage::convertDepthWithPalette (int d, QRgb * palette, int palette_count, int conversion_flags=0) const

Note: currently no closest-color search is made. If colors are found that are not in the palette, the palette may not be used at all. This result should not be considered valid, as it may change in future implementations.

Currently inefficient for non 32-bit images.

QImage QImage::copy () const

Returns a deep copy of the image.

QImage QImage::copy (int x, int y, int w, int h, int conversion_flags=0) const

Returns a deep copy of a sub-area of the image.

See also: bitBlt().

QImage QImage::copy (QRect & r) const

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

bool QImage::create (int width, int height, int depth, int numColors=0, Endian bitOrder=IgnoreEndian)

Sets the image width, height, depth, number of colors and bit order. Returns TRUE if successful, or FALSE if the parameters are incorrect or if memory cannot be allocated.

The *width* and *height* is limited to 32767. *depth* must be 1, 8 or 32. If *depth* is 1, then *bitOrder* must be set to either QImage::LittleEndian or QImage::BigEndian. For other depths, *bitOrder* must be QImage::IgnoreEndian.

This function allocates a color table and a buffer for the image data. The image data is not initialized.

The image buffer is allocated as a single block that consists of a table of scanline pointers (jumpTable()) and the image data (bits()).

See also: width(), height(), depth(), numColors(), bitOrder(), jumpTable(), scanLine(), bits(), bytesPerLine() and numBytes().

bool QImage::create (const QSize &, int depth, int numColors=0, Endian bitOrder=IgnoreEndian)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

QImage QImage::createAlphaMask (int conversion_flags) const

Builds and returns a 1-bpp mask from the alpha buffer in this image. Returns a null image if alpha buffer mode is disabled.

See QPixmap::convertFromImage for a description of the *conversion_flags* argument.

The returned image has little-endian bit order, which you can convert to big-endianness using convertBitOrder().

QImage QImage::createHeuristicMask (bool clipTight=TRUE) const

Creates and returns a 1-bpp heuristic mask for this image. It works by selecting a color from one of the corners, then chipping away pixels of that color, starting at all the edges.

The four corners vote over which color is to be masked away. In case of a draw (this generally means that this function is not applicable to the image) the voting results are undocumented.

The returned image has little-endian bit order, which you can convert to big-endianness using convertBitOrder().

This function disregards the alpha buffer.

int QImage::depth () const

Returns the depth of the image.

The image depth is the number of bits used to encode a single pixel, also called bits per pixel (bpp) or bit planes of an image.

The supported depths are 1, 8 and 32.

void QImage::detach ()

Detaches from shared image data and makes sure that this image is the only one referring the data.

If multiple images share common data, this image makes a copy of the data and detaches itself from the sharing mechanism. Nothing is done if there is just a single reference.

void QImage::fill (uint pixel)

Fills the entire image with the pixel value *pixel.*

If the depth of this image is 1, only the lowest bit is used. If you say fill(0), fill(2) etc., the image is filled with 0s. If you say fill(1), fill(3) etc., the image is filled with 1s. If the depth is 8, the lowest 8 bits are used.

If the depth is 32 and the image has no alpha buffer, the *pixel* value is written to each pixel in the image. If the image has an alpha buffer, only the 24 RGB bits are set and the upper 8 bits (alpha value) are left unchanged.

bool QImage::hasAlphaBuffer () const

Returns TRUE if alpha buffer mode is enabled, otherwise FALSE.

See also: setAlphaBuffer().

int QImage::height () const

Returns the height of the image.

See also: width(), size() and rect().

const char * QImage::imageFormat (const char * fileName) `[static]`

Returns a string that specifies the image format of the file *fileName,* or null if the file cannot be read or if the format cannot be recognized.

The QImageIO documentation lists the guaranteed supported image formats, or use the QImage::inputFormats() QImage::outputFormats() to get lists that include installed formats.

See also: load() and save().

QStrList QImage::inputFormats () `[static]`

Returns a list of image formats which are supported for image input.

bool QImage::isGrayscale () const

Returns TRUE if the image is allGray(), *and* if the image is 32-bpp or a 256-color 8-bpp image for which color(i) is QRgb(i,i,i).

bool QImage::isNull () const

Returns TRUE if it is a null image.

A null image has all parameters set to zero and no allocated data.

Examples:
qtimage/qtimage.cpp

uchar ** QImage::jumpTable () const

Returns a pointer to the scanline pointer table.

This is the beginning of the data block for the image.

bool QImage::load (const char * fileName, const char * format=0)

Loads an image from the file *fileName*. Returns TRUE if successful, or FALSE if the image could not be loaded.

If *format* is specified, the loader attempts to read the image using the specified format. If *format* is not specified (default), the loader reads a few bytes from the header to guess the file format.

The QImageIO documentation lists the supported image formats and explains how to add extra formats.

See also: loadFromData(), save(), imageFormat(), QPixmap::load() and QImageIO.

bool QImage::loadFromData (const uchar * buf, uint len, const char * format=0)

Loads an image from the binary data in *buf* (*len* bytes). Returns TRUE if successful, or FALSE if the image could not be loaded.

If *format* is specified, the loader attempts to read the image using the specified format. If *format* is not specified (default), the loader reads a few bytes from the header to guess the file format.

The QImageIO documentation lists the supported image formats and explains how to add extra formats.

See also: load(), save(), imageFormat(), QPixmap::loadFromData() and QImageIO.

bool QImage::loadFromData (QByteArray buf, const char * format=0)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

int QImage::numBytes () const

Returns the number of bytes occupied by the image data.

See also: bytesPerLine().

int QImage::numColors () const

Returns the size of the color table for the image.

Notice that numColors() returns 0 for 32-bpp images, since these images do not use color tables, but instead encode pixel values as RGB triplets.

QImage & QImage::operator= (const QPixmap & pixmap)

Sets the image bits to the *pixmap* contents and returns a reference to the image.

If the image shares data with other images, it will first dereference the shared data.

Makes a call to QPixmap::convertToImage().

QImage & QImage::operator= (const QImage & image)

Assigns a shallow copy of *image* to this image and returns a reference to this image.

See also: copy().

QStrList QImage::outputFormats () `[static]`

Returns a list of image formats which are supported for image output.

QRgb QImage::pixel (int x, int y) const

Returns the actual color of the pixel at the given coordinates.

If (x,y) is not on the image, the results are undefined.

Examples:
qmag/qmag.cpp

int QImage::pixelIndex (int x, int y) const

Returns the pixel index at the given coordinates.

If (x,y) is not valid, or if the image is not a paletted image (depth() > 8), the results are undefined.

QRect QImage::rect () const

Returns the enclosing rectangle (0,0,width(),height()) of the image.

See also: width(), height() and size().

void QImage::reset ()

Resets all image parameters and deallocates the image data.

Examples:
qtimage/qtimage.cpp

bool QImage::save (const char * fileName, const char * format) const

Saves the image to the file *fileName,* using the image file format *format.* Returns TRUE if successful, or FALSE if the image could not be saved.

See also: load(), loadFromData(), imageFormat(), QPixmap::save() and QImageIO.

uchar * QImage::scanLine (int i) const

Returns a pointer to the pixel data at the *i'th* scanline.

The scanline data is aligned on a 32-bit boundary.

Warning: If you are accessing 32-bpp image data, cast the returned pointer to `QRgb*` (QRgb has a 32 bit size) and use it to read/write the pixel value. You cannot use the `uchar*` pointer directly, because the pixel format depends on the byte order on the underlying platform. Hint: use qRed() and friends (qcolor.h) to access the pixels.

See also: bits().

Examples:
desktop/desktop.cpp

void QImage::setAlphaBuffer (bool enable)

Enables alpha buffer mode if *enable* is TRUE, otherwise disables it. The default setting is disabled.

An 8-bpp image has 8 bit pixels. A pixel is an index into the color table, which contains 32-bit color values. In a 32-bpp image, the 32 bit pixels are the color values.

This 32 bit value is encoded as follows: The lower 24 bits are used for the red, green and blue components. The upper 8 bits contain the alpha component.

The alpha component specifies the transparency of a pixel. 0 means completely transparent and 255 means opaque. The alpha component is ignored if you do not enable alpha buffer mode.

The alpha buffer is used to set a mask when a QImage is translated to a QPixmap.

See also: hasAlphaBuffer() and createAlphaMask().

void QImage::setColor (int i, QRgb c)

Sets a color in the color table at index *i* to *c.*

A color value is an RGB triplet. Use the qRgb function (defined in qcolor.h) to make RGB triplets.

See also: color().

Examples:
desktop/desktop.cpp

void QImage::setNumColors (int numColors)

Resizes the color table to *numColors* colors.

If the color table is expanded, then all new colors will be set to black (RGB 0,0,0).

See also: color() and setColor().

void QImage::setPixel (int x, int y, uint index_or_rgb)

Sets the pixel index or color at the given coordinates.

If (x,y) is not valid, or if the image is a paletted image (depth() <= 8) and *index_or_rgb* >= numColors(), the results are undefined.

QSize QImage::size () const

Returns the size of the image.

See also: width(), height() and rect().

QImage QImage::smoothScale (int width, int height) const

Returns a copy of the image smoothly scaled to *width* by *height* pixels. For 32-bpp images, and 1-bpp/8-bpp color images, the result will be 32-bpp, while all-gray images (including black-and-white 1-bpp) will produce 8-bit grayscale images with the palette spanning 256 grays from black to white.

This function uses code based on:

pnmscale.c - read a portable anymap and scale it

Copyright (C) 1989, 1991 by Jef Poskanzer.

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation. This software is provided "as is" without express or implied warranty.

QImage::Endian QImage::systemBitOrder() `[static]`

Determines the bit order of the display hardware. Returns QImage::LittleEndian (LSB first) or QImage::BigEndian (MSB first).

QImage::Endian QImage::systemByteOrder() `[static]`

Determines the host computer byte order. Returns QImage::LittleEndian (LSB first) or QImage::BigEndian (MSB first).

bool QImage::valid (int x, int y) const

Tests if the (*x, y*) is a valid coordinate in the image.

Examples:
qmag/qmag.cpp

int QImage::width () const

Returns the width of the image.

See also: heigth(), size() and rect().

85.3. Related Functions

QDataStream & operator<< (QDataStream & s, const QImage & image)

Writes an image to the stream as a BMP image.

See also: QImage::save().

QDataStream & operator>> (QDataStream & s, QImage & image)

Reads an image from the stream.

See also: QImage::load().

85.3.1. URL

Generated from http://www.troll.no/qt/qimage.html.

86. QImageConsumer Class Reference

`#include <qasyncimageio.h>`

86.1. Detailed Description

An abstraction used by QImageDecoder.

A QImageConsumer consumes information about changes to the QImage maintained by a QImageDecoder. It represents the a view of the image which the decoder produces.

See also: QImageDecoder.

86.2. Member Function Documentation

void QImageConsumer::changed (const QRect &) `[virtual]`

Called when the given area of the image has changed.

void QImageConsumer::end () `[virtual]`

Called when all data of all frames has been decoded and revealed as changed().

void QImageConsumer::frameDone () `[virtual]`

Called when a frame of an animated image has ended and been revealed as changed(). The decoder will not make any further changes to the image until the next call to QImageFormat::decode().

void QImageConsumer::setFramePeriod (int milliseconds) `[virtual]`

Notes that the frame about to be decoded should not be displayed until the given number of *milliseconds* after the time that this function is called. Of course, the image may not have been decoded by then, in which case the frame should not be displayed until it is complete. A value of -1 (the assumed default) indicates that the image should be diplayed even while it is only partially loaded.

void QImageConsumer::setLooping (int n) `[virtual]`

Called to indicate that the sequence of frames in the image should be repeated *n* times, including the sequence during decoding.

- 0 = Forever
- 1 = Only display frames the first time through
- 2 = Repeat once after first pass through images
- etc.

To make the QImageDecoder do this just delete it and pass the information to it again for decoding (setLooping() will be called again of course, but that can be ignored), or keep copies of the changed areas at the ends of frames.

void QImageConsumer::setSize (int, int) `[virtual]`

This function is called as soon as the size of the image has been determined.

86.2.1. URL

Generated from http://www.troll.no/qt/qimageconsumer.html.

87. QImageDecoder Class Reference

```
#include <qasyncimageio.h>
```

87.1. Detailed Description

Incremental image decoder for all supported image formats.

New formats are installed by creating objects of class QImageFormatType, and the QMovie class can be used for using all installed incremental image formats; QImageDecoder is only useful for creating new ways of feeding data to an QImageConsumer.

87.2. Member Function Documentation

QImageDecoder::QImageDecoder (QImageConsumer * c)

Constructs a QImageDecoder which will send change information to a given QImageConsumer.

QImageDecoder::~QImageDecoder ()

Destroys a QImageDecoder. The image it built is destroyed. The decoder built by the factory for the file format is destroyed. The consumer for which it decoded the image is *not* destroyed.

int QImageDecoder::decode (const uchar * buffer, int length)

Call this function to decode some data into image changes. The data will be decoded, sending change information to the QImageConsumer of this QImageDecoder, until one of the change functions of the consumer returns FALSE.

Returns the number of bytes consumed, 0 if consumption is complete, and -1 if decoding fails dur to invalid data.

const char* QImageDecoder::formatName (const uchar * buffer, int length) `[static]`

Call this function to find the name of the format of the given header. The returned string is statically allocated.

Returns 0 if the format is not recognized.

const QImage& QImageDecoder::image ()

Returns the image currently being decoded.

QStrList QImageDecoder::inputFormats () `[static]`

Returns a sorted list of formats for which asynchronous loading is supported.

void QImageDecoder::registerDecoderFactory (QImageFormatType * f) `[static]`

Registers a new QImageFormatType. This is not needed in application code as factories call this themselves.

void QImageDecoder::unregisterDecoderFactory (QImageFormatType * f) `[static]`

Unregisters a new QImageFormatType. This is not needed in application code as factories call this themselves.

87.2.1. URL

Generated from http://www.troll.no/qt/qimagedecoder.html.

88. QImageDrag Class Reference

`#include <qdragobject.h>`

Inherits QDragObject.

88.1. Detailed Description

The QImageDrag provides a drag-and-drop object for tranferring images.

Images are offered to the receiving application in multiple formats, determined by the output formats in Qt.

For detailed information about drag-and-drop, see the QDragObject class.

88.2. Member Function Documentation

QImageDrag::QImageDrag (QImage image, QWidget * dragSource = 0, const char * name = 0)

Creates an image drag object and sets it to *image. dragSource* must be the drag source, *name* is the object name.

QImageDrag::QImageDrag (QWidget * dragSource = 0, const char * name = 0)

Creates a default text drag object. *dragSource* must be the drag source, *name* is the object name.

QImageDrag::~QImageDrag ()

Destroys the image drag object and frees all allocated resources.

bool QImageDrag::canDecode (QDragMoveEvent * e) `[static]`

Returns TRUE if the information in *e* can be decoded into an image.

See also: decode().

bool QImageDrag::decode (QDropEvent * e, QImage & img) `[static]`

Attempts to decode the dropped information in *e* into *img,* returning TRUE if successful.

See also: canDecode().

bool QImageDrag::decode (QDropEvent * e, QPixmap & pm) `[static]`

Attempts to decode the dropped information in *e* into *pm,* returning TRUE if successful.

This is a convenience function that converts to *pm* via a QImage.

See also: canDecode().

void QImageDrag::setImage (QImage image)

Sets the image to be dragged. You will need to call this if you did not pass the image during construction.

88.2.1. URL

Generated from http://www.troll.no/qt/qimagedrag.html.

89. QImageFormat Class Reference

`#include <qasyncimageio.h>`

Inherited by QGIFFormat.

89.1. Detailed Description

Incremental image decoder for a specific image format.

By making a derived classes of QImageFormatType which in turn creates objects that are a subclass of QImageFormat, you can add support for more incremental image formats, allowing such formats to be sources for a QMovie, or for the first frame of the image stream to be loaded as a QImage or QPixmap.

Your new subclass must override the decode() function in order to process your new format.

New QImageFormat objects are generated by new QImageFormatType factories.

89.2. Member Function Documentation

QImageFormat::˜QImageFormat () `[virtual]`

Destructs the object.

int QImageFormat::decode (QImage & img, QImageConsumer * consumer, const uchar * buffer, int length) `[virtual]`

New subclasses must override this method.

It should decode some or all of the bytes from *buffer* into *img,* calling the methods of *consumer* as the decoding proceeds to inform that consumer of changes to the image. The consumer may be 0, in which case the function should just process the data into *img* without telling any consumer about the changes. Note that the decoder must store enough state to be able to continue in subsequent calls to this method - this is the essence of the incremental image loading.

The function should return without processing all the data if it reaches the end of a frame in the input.

The function must return the number of bytes it has processed.

Reimplemented in QGIFFormat.

89.2.1. URL

Generated from http://www.troll.no/qt/qimageformat.html.

90. QImageFormatType Class Reference

`#include <qasyncimageio.h>`

Inherited by QGIFFormatType.

90.1. Detailed Description

Factory that makes QImageFormat objects.

While the QImageIO class allows for *complete* loading of images, QImageFormatType allows for *incremental* loading of images.

New image file formats are installed by creating objects of derived classes of QImageFormatType. They must implement decoderFor() and formatName().

QImageFormatType is a very simple class. Its only task is to recognize image data in some format and make a new object, subclassed from QImageFormat, which can decode that format.

The factories for formats built into Qt (currently only GIF) are automatically defined before any other factory is initialized. If two factories would recognize an image format, the factory created last will override the earlier one, thus you can override current and future built-in formats.

90.2. Member Function Documentation

QImageFormatType::QImageFormatType () `[protected]`

Creates a factory. It automatically registers itself with QImageDecoder.

QImageFormatType::~QImageFormatType () `[virtual]`

Destroys a factory. It automatically unregisters itself from QImageDecoder.

virtual QImageFormat* QImageFormatType::decoderFor (const uchar * buffer, int length) `[virtual]`

Returns a decoder for decoding an image which starts with the give bytes. This function should only return a decoder if it is definate that the decoder applies to data with the given header. Returns 0 if there is insufficient data in the header to make a positive identification, or if the data is not recognized.

virtual const char* QImageFormatType::formatName () const `[virtual]`

Returns the name of the format supported by decoders from this factory. The string is statically allocated.

90.2.1. URL

Generated from http://www.troll.no/qt/qimageformattype.html.

91. QImageIO Class Reference

`#include <qimage.h>`

91.1. Detailed Description

The QImageIO class contains parameters for loading and saving images.

QImageIO contains a QIODevice object that is used for image data I/O. The programmer can install new image file formats in addition to those that Qt implements.

Qt currently supports the following image file formats: GIF (reading only), BMP, XBM, XPM and PNM. The different PNM formats are: PBM (P1), PGM (P2), PPM (P3), PBMRAW (P4), PGMRAW (P5) and PPMRAW (P6).

Additional formats are available with the Qt Image IO Extension package.

You will normally not need to use this class, QPixmap::load(), QPixmap::save() and QImage contain most of the needed functionality.

For image files which contain sequences of images, only the first is read. See the QMovie for loading multiple images.

PBM, PGM, and PPM format output is only supported in PPMRAW format.

Warning: Unisys has changed its position regarding GIF. If you are in a country where Unisys holds a patent on LZW compression and/or decompression, Unisys may require you to license that technology. These countries include Canada, Japan, the USA, France, Germany, Italy and the UK. There is more information on Unisys web site: Overview of Unisys' position. GIF support may be removed in a future version of Qt. We recommend using the PNG format, which is available in the Qt Image IO Extension package.

See also: QImage, QPixmap, QFile and QMovie.

91.2. Member Function Documentation

QImageIO::QImageIO ()

Constructs a QImageIO object with all parameters set to zero.

QImageIO::QImageIO (QIODevice * ioDevice, const char * format)

Constructs a QImageIO object with an I/O device and a format tag.

QImageIO::QImageIO (const char * fileName, const char * format)

Constructs a QImageIO object with a file name and a format tag.

QImageIO::˜QImageIO ()

Destroys the object an all related data.

void QImageIO::defineIOHandler (const char * format, const char * header, const char * flags, image_io_handler read_image, image_io_handler write_image) `[static]`

Defines a image IO handler for a specified image format. An image IO handler is responsible for reading and writing images.

Arguments:

- *format* is the name of the format.
- *header* is a regular expression that recognizes the image header.
- *flags* is "T" for text formats like PBM; generally you will want to use 0.
- *read_image* is a function to read an image of this format.
- *write_image* is a function to write an image of this format. Both read_image and write_image are of type image_io_handler, which is a function pointer.

Example:

```
void readGIF( QImageIO *image )
{
  // read the image, using the image->ioDevice()
}
void writeGIF( QImageIO *image )
{
  // write the image, using the image->ioDevice()
}
// add the GIF image handler
QImageIO::defineIOHandler( "GIF",
                           "^GIF[0-9][0-9][a-z]",
                           0,
                           readGIF,
                           writeGIF );
```

Prior to comparison with the regular expression, the file header is converted to change all 0 bytes into 1 bytes. This is done because 0 is such a common header byte yet regular expressions cannot match it.

For image formats supporting incremental display, such as sequences of animated frames, see the QImageFormatType class.

const char * QImageIO::description () const

Returns the image description string.

See also: setDescription().

const char * QImageIO::fileName () const

Returns the file name currently set.

See also: setFileName().

const char * QImageIO::format () const

Returns the image format string, or 0 if no format has been explicitly set.

const QImage & QImageIO::image () const

Returns the image currently set.

See also: setImage().

const char * QImageIO::imageFormat (QIODevice * d) `[static]`

Returns a string that specifies the image format of the image read from *d,* or null if the file cannot be read or if the format is not recognized.

const char * QImageIO::imageFormat (const char * fileName) `[static]`

Returns a string that specifies the image format of the file *fileName,* or null if the file cannot not be read or if the format is not recognized.

QStrList QImageIO::inputFormats () `[static]`

Returns a sorted list of image formats which are supported for image input.

QIODevice * QImageIO::ioDevice () const

Returns the IO device currently set.

See also: setIODevice().

QStrList QImageIO::outputFormats () `[static]`

Returns a sorted list of image formats which are supported for image output.

const char * QImageIO::parameters () const

Returns image parameters string.

See also: setParameters().

bool QImageIO::read ()

Reads an image into memory and returns TRUE if the image was successfully read.

Before reading an image, you must set an IO device or a file name. If both an IO device and a file name has been set, then the IO device will be used.

Setting the image file format string is optional.

Note that this function does *not* set the format used to read the image. If you need that information, use the imageFormat() static functions.

Example:

```
QImageIO iio;
QPixmap  pixmap;
iio.setFileName( "burger.bmp" );
if ( image.read() )                  // ok
    pixmap = iio.image();            // convert to pixmap
```

See also: setIODevice(), setFileName(), setFormat(), write() and QPixmap::load().

void QImageIO::setDescription (const char * description)

Sets the image description string for image handlers that support image descriptions.

Currently, no image format supported by Qt use the description string.

void QImageIO::setFileName (const char * fileName)

Sets the name of the file to read or write an image.

See also: setIODevice().

void QImageIO::setFormat (const char * format)

Sets the image format name of the image about to be read or written.

It is necessary to specify a format before writing an image.

It is not necessary to specify a format before reading an image. If not format has been set, Qt guesses the image format before reading it. If a format is set, but the image has another (valid) format, the image will not be read.

See also: read(), write() and format().

void QImageIO::setIODevice (QIODevice * ioDevice)

Sets the IO device to be used for reading or writing an image.

Setting the IO device allows images to be read/written to any block-oriented QIODevice.

If *ioDevice* is not null, this IO device will override file name settings.

See also: setFileName().

void QImageIO::setImage (const QImage & image)

Sets the image.

See also: image().

void QImageIO::setParameters (const char * parameters)

Sets the image parameters string for image handlers that require special parameters.

Although all image formats supported by Qt ignore the parameters string, it will be useful for future extentions or contributions (like JPEG).

void QImageIO::setStatus (int status)

Sets the image IO status. A non-zero value indicates an error, while 0 means that the IO operation was successful.

See also: status().

int QImageIO::status () const

Returns the image IO status. A non-zero value indicates an error, while 0 means that the IO operation was successful.

See also: setStatus().

bool QImageIO::write ()

Writes an image to an IO device and returns TRUE if the image was successfully written.

Before writing an image, you must set an IO device or a file name. If both an IO device and a file name has been set, then the IO device will be used.

The image will be written using the specified image format.

Example:

```
QImageIO iio;
QImage   im;
im = pixmap;                              // convert to image
iio.setImage( im );
iio.setFileName( "burger.bmp" );
iio.setFormat( "BMP" );
iio.write();                              // TRUE if ok
```

See also: setIODevice(), setFileName(), setFormat(), read() and QPixmap::save().

91.2.1. URL

Generated from http://www.troll.no/qt/qimageio.html.

92. QIntCache Class Reference

`#include <qintcache.h>`

Inherits QGCache.

92.1. Detailed Description

The QIntCache class is a template class that provides a cache based on `long` keys.

QIntCache is implemented as both a template and a macro class. Define a template instance QIntCache<X> to create a cache that operates on pointers to X, or X*.

A cache is a least recently used (LRU) list of cache items. Each cache item has a cost. The sum of item costs, totalCost(), will not exceed the maximum cache cost, maxCost(). If inserting a new item would cause the total cost to exceed the maximum cost, the least recently used items in the cache are deleted.

Apart from insert, by far the most important function is find() (which also exists as operatior[]). This function looks up an item, returns it, and by default marks it as being the least recently used item.

There are also methods to remove() an object from the cache and delete it, and to take() an object out of the cache without deleting it.

QIntCache, as it must, takes over ownership of the items you insert. That implies that it will delete the items at its discretion, and that you must *not* delete them. There is one exception, however: If the cost of an object you try to insert() is higher than the allowed total cost of the cache, insert() returns FALSE and in this case you must delete the object yourself.

In addition to totalCost() and maxCost()/setMaxCost(), there are access functions to obtain the number of items in the cache (count()), the size of the hash array (size(), set using the constructor) and whether the cache is empty (you guessed it: isEmpty()). Finally, there is a statistics(), which prints various cache statistics on `stdout.`

There is also a QIntCacheIterator which may be used to traverse the items in the cache in arbritrary order.

See also: QIntCacheIterator, QCache and Collection Classes

92.2. Member Function Documentation

QIntCache::QIntCache (const QIntCache<type> & c)

For internal use only.

QIntCache::QIntCache (int maxCost=100, int size=17)

Constructs an empty cache with maximum allowed total cost *maxCost* and optimized to contain a bit less than *size* items.

QIntCache::˜QIntCache ()

Destroys the object and all objects in it.

void QIntCache::clear () `[virtual]`

Removes all items from the cache, and deletes them.

All cache iterators that operate this on cache are reset.

See also: remove() and take().

Reimplemented from QCollection.

uint QIntCache::count () const `[virtual]`

Returns the number of items in the cache.

See also: totalCost().

Reimplemented from QCollection.

type * QIntCache::find (long k, bool ref=TRUE) const

Returns the item associated with \ k, or null if the key does not exist in the cache. If *ref* is TRUE (the default), the item is moved to the front of the LRU list.

If there are two or more items with equal keys, then the one that was inserted last is returned.

bool QIntCache::insert (long k, const type * d, long c=1, int p=0)

Inserts the item *d* into the cache with key *k,* cost `(default` 1) and priority *p* (default 0). Returns TRUE if it succeeds and FALSE if it fails.

The cache's size is limited, and if the total cost is too high, QIntCache will delete some of the least-used objects.

Warning: If this function returns FALSE, you must delete *d* yourself. Additionally, be very careful about using *d* after calling this function, as any other insertions into the cache, from anywhere in the application, or within Qt itself, could cause the object to be discarded from the cache, and the pointer to become invalid.

bool QIntCache::isEmpty () const

Returns TRUE if the cache is empty, or TRUE if there is at least one object in it.

int QIntCache::maxCost () const

Returns the maximum allowed total cost of the cache.

See also: setMaxCost() and totalCost().

QIntCache<type>& QIntCache::operator= (const QIntCache<type> & c)

For internal use only.

type * QIntCache::operator[] (long k) const

Returns the item associated with *k,* or null if *k* does not exist in the cache, and moves the item to the front of the LRU list.

If there are two or more items with equal keys, then the one that was inserted last is returned.

This is the same as find(k, TRUE).

See also: find().

bool QIntCache::remove (long k)

Deletes the item associated with *k,* and returns TRUE if the item was present in the cache or FALSE if it was not.

If there are two or more items with equal keys, then the one that was inserted last is is deleted.

All iterators that refer to the removed item are set to point to the next item in the cache's traversal order.

See also: take() and clear().

void QIntCache::setMaxCost (int m)

Sets the maximum allowed total cost of the cache to *m.* If the current total cost is above *m,* some items are deleted immediately.

See also: maxCost() and totalCost().

uint QIntCache::size () const

Returns the size of the hash array used to implement the cache. This should be a bit bigger than count() is likely to be.

type * QIntCache::take (long k)

Takes the item associated with *k* out of the cache without deleting it, and returns a pointer to the item taken out, or null if the key does not exist in the cache.

If there are two or more items with equal keys, then the one that was inserted last is taken.

All iterators that refer to the taken item are set to point to the next item in the cache's traversal order.

See also: remove() and clear().

int QIntCache::totalCost () const

Returns the total cost of the items in the cache. This is an integer in the range 0 to maxCost().

See also: setMaxCost().

92.2.1. URL

Generated from http://www.troll.no/qt/qintcache.html.

93. QIntCacheIterator Class Reference

`#include <qcache.h>`

Inherits QGCacheIterator.

93.1. Detailed Description

The QIntCacheIterator class provides an iterator for QIntCache collections.

Note that the traversal order is arbitrary, you are not guaranteed any particular order. If new objects are inserted into the cache while the iterator is active, the iterator may or may not see them.

Multiple iterators are completely independent, even when they operate on the same QIntCache. QIntCache updates all iterators that refer an item when that itam is removed.

QIntCacheIterator provides an operator++(), and an operator+=() to traverse the cache, current() and currentKey() to access the current cache item and its key, atFirst() atLast() which return TRUE if the iterator points to the first/last item in the cache, isEmpty() which returns TRUE if the cache is empty and finally count() which returns the number of items in the cache.

Note that atFirst() and atLast() refer to the iterator's arbitrary ordering, not to the cache's internal LRU list.

See also: QIntCache.

93.2. Member Function Documentation

QIntCacheIterator::QIntCacheIterator (const QIntCache<type> & cache)

Constructs an iterator for *cache*. The current iterator item is set to point on the first item in the *cache* (or rather, the first item is defined to be the item at which this constructor sets the iterator to point).

QIntCacheIterator::QIntCacheIterator (const QIntCacheIterator<type> & ci)

Constructs an iterator for the same cache as *ci*. The new iterator starts at the same item as ci.current(), but moves independently from there on.

QIntCacheIterator::operator type * () const

Cast operator. Returns a pointer to the current iterator item. Same as current().

uint QIntCacheIterator::count () const

Returns the number of items in the cache on which this iterator operates.

See also: isEmpty().

type * QIntCacheIterator::current () const

Returns a pointer to the current iterator item.

long QIntCacheIterator::currentKey () const

Returns the key for the current iterator item.

bool QIntCacheIterator::isEmpty () const

Returns TRUE if the cache is empty, i.e. count() == 0, otherwise FALSE.

See also: count().

type * QIntCacheIterator::operator() ()

Makes the succeeding item current and returns the original current item.

If the current iterator item was the last item in the cache or if it was null, null is returned.

type * QIntCacheIterator::operator++ ()

Prefix ++ makes the iterator point to the item just after *current()*, and makes that the new current item for the iterator. If current() was the last item, operator--() returns 0.

type * QIntCacheIterator::operator+= (uint jump)

Returns the item *jump* positions after the current item, or null if it is beyond the last item. Makes this the current item.

type * QIntCacheIterator::operator-- ()

Prefix -- makes the iterator point to the item just before *current()*, and makes that the new current item for the iterator. If current() was the first item, operator--() returns 0.

type * QIntCacheIterator::operator-= (uint jump)

Returns the item *jump* positions before the current item, or null if it is beyond the first item. Makes this the current item.

QIntCacheIterator<type>& QIntCacheIterator::operator= (const QIntCacheIterator<type> & ci)

Makes this an iterator for the same cache as *ci*. The new iterator starts at the same item as ci.current(), but moves independently thereafter.

type * QIntCacheIterator::toFirst ()

Sets the iterator to point to the first item in the cache and returns a pointer to the item.

Sets the iterator to null and returns null if if the cache is empty.

See also: toLast() and isEmpty().

type * QIntCacheIterator::toLast ()

Sets the iterator to point to the last item in the cache and returns a pointer to the item.

Sets the iterator to null and returns null if if the cache is empty.

See also: toLast() and isEmpty().

93.2.1. URL

Generated from http://www.troll.no/qt/qintcacheiterator.html.

94. QIntDict Class Reference

`#include <qintdict.h>`

Inherits QGDict.

94.1. Detailed Description

The QIntDict class is a template class that provides a dictionary based on `long` keys.

QIntDict is implemented as both a template and a macro class. Define a template instance QIntDict<X> to create a dictionary that operates on pointers to X, or X*.

A dictionary is a collection that associates an item with a key. The key is used for inserting and looking up an item. QIntDict has `long` keys.

The dictionary has very fast insertion and lookup.

Example:

```
#include <qintdict.h>
#include <stdio.h>
void main()
{
    QIntDict<char> dict;            // maps long ==> char*
    dict.insert( 33, "France" );
    dict.insert(  7, "Russia" );
    dict.insert( 49, "Norway" );
    printf( "%s\n", dict[49] );
    printf( "%s\n", dict[33] );
    printf( "%s\n", dict[7] );
    if ( !dict[39] )
        printf( "39 not defined\n" );
}
```

Program output:

```
Norway
France
Russia
39 not defined
```

The dictionary in our example maps `long` keys to `char*` items. QIntDict implements the [] operator to lookup an item.

A QIntDict can also be instantiated through a macro expansion, but this is necessary only for compilers that do not support templates. See the collection classes documentation for a general discussion on template-based versus macro-based collections.

QIntDict is implemented by QGDict as a hash array with a fixed number of entries. Each array entry points to a singly linked list of buckets, in which the dictionary items are stored.

When an item is inserted with a key, the key is converted (hashed) to an integer index into the hash array using the `mod` operation. The item is inserted before the first bucket in the list of buckets.

Looking up an item is normally very fast. The key is again hashed to an array index. Then QIntDict scans the list of buckets and returns the item found or null if the item was not found. You cannot insert null pointers into a dictionary.

The size of the hash array is very important. In order to get good performance, you should use a suitably large prime number. Suitable means equal to or larger than the maximum expected number of dictionary items.

Items with equal keys are allowed. When inserting two items with the same key, only the last inserted item will be visible (last in, first out) until it is removed.

Example:

```
#include <qintdict.h>
#include <stdio.h>
void main()
{
    QIntDict<char> dict;            // maps long ==> char*
    dict.insert( 7, "Russia" );
    dict.insert( 7, "USSR" );
    printf( "%s\n", dict[7] );
    dict.remove( 7 );               // Gorbie was here
    printf( "%s\n", dict[7] );
}
```

Program output:

```
USSR
Russia
```

The QIntDictIterator class can traverse the dictionary contents, but only in an arbitrary order. Multiple iterators may independently traverse the same dictionary.

Calling setAutoDelete(TRUE) for a dictionary tells it to delete items that are removed . The default is to not delete items when they are removed.

When inserting an item into a dictionary, only the pointer is copied, not the item itself. This is called a shallow copy. It is possible to make the dictionary copy all of the item's data (known as a deep copy) when an item is inserted. insert() calls the virtual function QCollection::newItem() for the item to be inserted. Inherit a dictionary and reimplement it if you want deep copies.

When removing a dictionary item, the virtual function QCollection::deleteItem() is called. QIntDict's default implementation is to delete the item if auto-deletion is enabled.

See also: QIntDictIterator, QDict and Collection Classes

94.2. Member Function Documentation

QIntDict::QIntDict (const QIntDict<type> & dict)

Constructs a copy of *dict.*

Each item in *dict* are inserted into this dictionary. Only the pointers are copied (shallow copy).

QIntDict::QIntDict (int size=17)

Constructs a dictionary using an internal hash array with the size *size.*

Setting *size* to a suitably large prime number (equal to or greater than the expected number of entries) makes the hash distribution better and hence the loopup faster.

QIntDict::~QIntDict ()

Removes all items from the dictionary and destroys it.

All iterators that access this dictionary will be reset.

See also: setAutoDelete().

void QIntDict::clear () `[virtual]`

Removes all items from the dictionary.

The removed items are deleted if auto-deletion is enabled.

All dictionary iterators that access this dictionary will be reset.

See also: remove(), take() and setAutoDelete().

Reimplemented from QCollection.

uint QIntDict::count () const `[virtual]`

Returns the number of items in the dictionary.

See also: isEmpty().

Reimplemented from QCollection.

type * QIntDict::find (long key) const

Returns the item associated with *key,* or null if the key does not exist in the dictionary.

This function uses an internal hashing algorithm to optimize lookup.

If there are two or more items with equal keys, then the last inserted of these will be found.

Equivalent to the [] operator.

See also: operator[].

void QIntDict::insert (long key, const type * item)

Inserts the *key* with the *item* into the dictionary.

The key does not have to be a unique dictionary key. If multiple items are inserted with the same key, only the last item will be visible.

Null items are not allowed.

See also: replace().

bool QIntDict::isEmpty () const

Returns TRUE if the dictionary is empty, i.e. count() == 0. Returns FALSE otherwise.

See also: count().

QIntDict<type> & QIntDict::operator= (const QIntDict<type> & dict)

Assigns *dict* to this dictionary and returns a reference to this dictionary.

This dictionary is first cleared, then each item in *dict* is inserted into this dictionary. Only the pointers are copied (shallow copy), unless newItem() has been reimplemented().

type * QIntDict::operator[] (long key) const

Returns the item associated with *key,* or null if the key does not exist in the dictionary.

This function uses an internal hashing algorithm to optimize lookup.

If there are two or more items with equal keys, then the last inserted of these will be found.

Equivalent to the find() function.

See also: find().

bool QIntDict::remove (long key)

Removes the item associated with *key* from the dictionary. Returns TRUE if successful, or FALSE if the key does not exist in the dictionary.

If there are two or more items with equal keys, then the last inserted of these will be removed.

The removed item is deleted if auto-deletion is enabled.

All dictionary iterators that refer to the removed item will be set to point to the next item in the dictionary traversing order.

See also: take(), clear() and setAutoDelete().

void QIntDict::replace (long key, const type * item)

Replaces an item which has a key equal to *key* with *item.*

If the item does not already exist, it will be inserted.

Null items are not allowed.

Equivalent to:

```
QIntDict<char> dict;
    ...
if ( dict.find(key) )
    dict.remove( key );
dict.insert( key, item );
```

If there are two or more items with equal keys, then the last inserted of these will be replaced.

See also: insert().

void QIntDict::resize (uint newsize)

Changes the size of the hashtable the *newsize.* The contents of the dictionary are preserved, but all iterators on the dictionary become invalid.

uint QIntDict::size () const

Returns the size of the internal hash array (as specified in the constructor).

See also: count().

void QIntDict::statistics () const

Debugging-only function that prints out the dictionary distribution using debug().

type * QIntDict::take (long key)

Takes the item associated with *key* out of the dictionary without deleting it (even if auto-deletion is enabled).

If there are two or more items with equal keys, then the last inserted of these will be taken.

Returns a pointer to the item taken out, or null if the key does not exist in the dictionary.

All dictionary iterators that refer to the taken item will be set to point to the next item in the dictionary travering order.

See also: remove(), clear() and setAutoDelete().

94.2.1. URL

Generated from http://www.troll.no/qt/qintdict.html.

95. QIntDictIterator Class Reference

```
#include <qintdict.h>
```

Inherits QGDictIterator.

95.1. Detailed Description

The QIntDictIterator class provides an iterator for QIntDict collections.

QIntDictIterator is implemented as both a template and a macro class. Define a template instance QIntDictIterator<X> to create a dictionary iterator that operates on QIntDict<X> (dictionary of X*).

Example:

```
#include <qintdict.h>
#include <stdio.h>
void main()
{
    QIntDict<char> dict;            // maps long ==> char*
    dict.insert( 33, "France" );
    dict.insert(  7, "Russia" );
    dict.insert( 49, "Norway" );
    QIntDictIterator<char> it( dict ); // iterator for dict
    while ( it.current() ) {
        printf( "%d -> %s\n", it.currentKey(), it.current() );
        ++it;
    }
}
```

Program output:

```
7 -> Russia
49 -> Norway
33 -> France
```

Note that the traversal order is arbitrary, you are not guaranteed the order above.

A QIntDictIterator can also be instantiated through a macro expansion, but this is necessary only for compilers that do not support templates. See the collection classes documentation for a general discussion on template-based versus macro-based collections.

Multiple iterators may independently traverse the same dictionary. A QIntDict knows about all iterators that are operating on the dictionary. When an item is removed from the dictionary, QIntDict update all iterators that are referring the removed item to point to the next item in the traversing order.

See also: QIntDict and Collection Classes

95.2. Member Function Documentation

QIntDictIterator::QIntDictIterator (const QIntDict<type> & dict)

Constructs an iterator for *dict*. The current iterator item is set to point on the first item in the *dict*.

QIntDictIterator::~QIntDictIterator ()

Destroys the iterator.

QIntDictIterator::operator type * () const

Cast operator. Returns a pointer to the current iterator item. Same as current().

uint QIntDictIterator::count () const

Returns the number of items in the dictionary this iterator operates on.

See also: isEmpty().

type * QIntDictIterator::current () const

Returns a pointer to the current iterator item.

long QIntDictIterator::currentKey () const

Returns the key for the current iterator item.

bool QIntDictIterator::isEmpty () const

Returns TRUE if the dictionary is empty, i.e. count() == 0. Returns FALSE otherwise.

See also: count().

type * QIntDictIterator::operator() ()

Makes the succeeding item current and returns the original current item.

If the current iterator item was the last item in the dictionary or if it was null, null is returned.

type * QIntDictIterator::operator++ ()

Prefix ++ makes the succeeding item current and returns the new current item.

If the current iterator item was the last item in the dictionary or if it was null, null is returned.

type * QIntDictIterator::operator+= (uint jump)

Sets the current item to the item *jump* positions after the current item, and returns a pointer to that item.

If that item is beyond the last item or if the dictionary is empty, it sets the current item to null and returns null.

type * QIntDictIterator::toFirst ()

Sets the current iterator item to point to the first item in the dictionary and returns a pointer to the item. If the dictionary is empty it sets the current item to null and returns null.

95.2.1. URL

Generated from http://www.troll.no/qt/qintdictiterator.html.

96. QIntValidator Class Reference

`#include <qvalidator.h>`

Inherits QValidator.

96.1. Detailed Description

The QIntValidator class provides range checking of integers.

QIntValidator provides a lower and an upper bound. It does not provide a fixup() function.

See also: QDoubleValidator.

96.2. Member Function Documentation

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

Creates a validator object which accepts all integers.

QIntValidator::QIntValidator (int bottom, int top, QWidget * parent, const char * name = 0)

Creates a validator object which accepts all integers from *bottom* up to and including *top*.

QIntValidator::˜QIntValidator ()

Deletes the validator and frees up any storage used.

int QIntValidator::bottom () const

Returns the lowest valid number according to this validator.

See also: top() and setRange().

void QIntValidator::setRange (int bottom, int top) `[virtual]`

Sets the validator to accept only number from *bottom* up to an including *top*.

int QIntValidator::top () const

Returns the highest valid number according to this validator.

See also: bottom() and setRange().

96.2.1. URL

Generated from http://www.troll.no/qt/qintvalidator.html.

97. QIODevice Class Reference

`#include <qiodevice.h>`

Inherited by QBuffer and QFile.

97.1. Detailed Description

The QIODevice class is the base class of I/O devices.

An I/O device represents a medium that one can read bytes from and write bytes to. The QIODevice class itself is not capable of reading or writing any data; it has virtual functions for doing so. These functions are implemented by the subclasses QFile and QBuffer.

There are two types of I/O devices; direct access and sequential access devices. Files can normally be accessed directly, except `stdin` etc., which must be processed sequentially. Buffers are always direct access devices.

The access mode of an I/O device can be either *raw* or *buffered.* QFile objects can be created using one of these. Raw access mode is more low level, while buffered access use smart buffering techniques. The raw access mode is best when I/O is block-operated using 4kB block size or greater. Buffered access works better when reading small portions of data at a time.

An I/O device operation can be executed in either *synchronous* or *asynchronous* mode. The I/O devices currently supported by Qt only execute synchronously.

The QDataStream and QTextStream provide binary and text operations on QIODevice objects.

QIODevice provides numerous pure virtual functions you need to implement when subclassing it. Here is a skeleton subclass:

```
class YourDevice : public QIODevice
{
public:
    YourDevice();
   ~YourDevice();
    bool open( int mode );
    void close();
    void flush();
    uint size() const;
    int  at() const;         // not a pure virtual function
    bool at( int );          // not a pure virtual function
    bool atEnd() const;      // not a pure virtual function
    int readBlock( char *data, uint len );
    int writeBlock( const char *data, uint len );
    int readLine( char *data, uint maxlen );
    int getch();
    int putch( int );
    int ungetch( int );
};
```

The three non-pure virtual functions can be ignored if your device is sequential (e.g. a tape device).

See also: QDataStream and QTextStream.

97.2. Member Function Documentation

QIODevice::QIODevice ()

Constructs an I/O device.

QIODevice::˜QIODevice () `[virtual]`

Destroys the I/O device.

bool QIODevice::at (int pos) `[virtual]`

Virtual function that sets the I/O device index to *pos*.

See also: size().

Reimplemented in QBuffer and QFile.

int QIODevice::at () const `[virtual]`

Virtual function that returns the current I/O device index.

This index is the data read/write head of the I/O device.

See also: size().

Reimplemented in QBuffer and QFile.

bool QIODevice::atEnd () const `[virtual]`

Virtual function that returns TRUE if the I/O device index is at the end of the input.

Reimplemented in QFile.

void QIODevice::close () `[virtual]`

Closes the I/O device.

This virtual function must be reimplemented by all subclasses.

See also: open().

Reimplemented in QBuffer and QFile.

int QIODevice::flags () const

Returns the current I/O device flags setting.

Flags consists of mode flags and state flags.

See also: mode() and state().

void QIODevice::flush () `[virtual]`

Flushes an open I/O device.

This virtual function must be reimplemented by all subclasses.

Reimplemented in QBuffer and QFile.

int QIODevice::getch () `[virtual]`

Reads a single byte/character from the I/O device.

Returns the byte/character read, or -1 if the end of the I/O device has been reached.

This virtual function must be reimplemented by all subclasses.

See also: putch() and ungetch().

Reimplemented in QBuffer and QFile.

bool QIODevice::isAsynchronous () const

Returns TRUE if the I/O device is a asynchronous device, otherwise FALSE.

This mode is currently not in use.

See also: isSynchronous().

bool QIODevice::isBuffered () const

Returns TRUE if the I/O device is a buffered (not raw) device, otherwise FALSE.

See also: isRaw().

bool QIODevice::isCombinedAccess () const

Returns TRUE if the I/O device is a combined access (both direct and sequential) device, otherwise FALSE.

This access method is currently not in use.

bool QIODevice::isDirectAccess () const

Returns TRUE if the I/O device is a direct access (not sequential) device, otherwise FALSE.

See also: isSequentialAccess().

bool QIODevice::isInactive () const

Returns TRUE if the I/O device state is 0, i.e. the device is not open.

See also: isOpen().

bool QIODevice::isOpen () const

Returns TRUE if the I/O device state has been opened, otherwise FALSE.

See also: isInactive().

bool QIODevice::isRaw () const

Returns TRUE if the I/O device is a raw (not buffered) device, otherwise FALSE.

See also: isBuffered().

bool QIODevice::isReadWrite () const

Returns TRUE if the I/O device was opened using `IO_ReadWrite` mode.

See also: isReadable() and isWritable().

bool QIODevice::isReadable () const

Returns TRUE if the I/O device was opened using `IO_ReadOnly` or `IO_ReadWrite` mode.

See also: isWritable() and isReadWrite().

bool QIODevice::isSequentialAccess () const

Returns TRUE if the I/O device is a sequential access (not direct) device, otherwise FALSE. Operations involving size() and at(int) are not valid on sequential devices.

See also: isDirectAccess().

bool QIODevice::isSynchronous () const

Returns TRUE if the I/O device is a synchronous device, otherwise FALSE.

See also: isAsynchronous().

bool QIODevice::isTranslated () const

Returns TRUE if the I/O device translates carriage-return and linefeed characters.

A QFile is translated if it is opened with the `IO_Translate` mode flag.

bool QIODevice::isWritable () const

Returns TRUE if the I/O device was opened using `IO_WriteOnly` or `IO_ReadWrite` mode.

See also: isReadable() and isReadWrite().

int QIODevice::mode () const

Returns bits OR'ed together that specify the current operation mode.

These are the flags that were given to the open() function.

The flags are: `IO_ReadOnly, IO_WriteOnly, IO_ReadWrite, IO_Append, IO_Truncate` and `IO_Translate.`

bool QIODevice::open (int mode) `[virtual]`

Opens the I/O device using the specified *mode.* Returns TRUE if successful, or FALSE if the device could not be opened.

The mode parameter *m* must be a combination of the following flags.

- `IO_Raw` specified raw (unbuffered) file access.
- `IO_ReadOnly` opens a file in read-only mode.
- `IO_WriteOnly` opens a file in write-only mode.
- `IO_ReadWrite` opens a file in read/write mode.
- `IO_Append` sets the file index to the end of the file.
- `IO_Truncate` truncates the file.
- `IO_Translate` enables carriage returns and linefeed translation for text files under MS-DOS, Window, OS/2 and Macintosh. Cannot be combined with `IO_Raw.`

This virtual function must be reimplemented by all subclasses.

See also: close().

Reimplemented in QBuffer and QFile.

int QIODevice::putch (int ch) `[virtual]`

Writes the character *ch* to the I/O device.

Returns *ch,* or -1 if some error occurred.

This virtual function must be reimplemented by all subclasses.

See also: getch() and ungetch().

Reimplemented in QBuffer and QFile.

int QIODevice::readBlock (char * data, uint len) `[virtual]`

Reads at most *len* bytes from the I/O device into *data* and returns the number of bytes actually read.

This virtual function must be reimplemented by all subclasses.

See also: writeBlock().

Reimplemented in QBuffer and QFile.

int QIODevice::readLine (char * data, uint maxlen) `[virtual]`

Reads a line of text, up to *maxlen* bytes including a terminating \0. If there is a newline at the end if the line, it is not stripped.

Returns the number of bytes read, or -1 in case of error.

This virtual function can be reimplemented much more efficiently by the most subclasses.

See also: readBlock() and QTextStream::readLine().

Reimplemented in QBuffer and QFile.

bool QIODevice::reset ()

Sets the device index to 0.

See also: at().

void QIODevice::resetStatus ()

Sets the I/O device status to `IO_Ok`.

See also: status().

void QIODevice::setFlags (int f) `[protected]`

For internal use only.

void QIODevice::setMode (int m) `[protected]`

For internal use only.

void QIODevice::setState (int s) `[protected]`

For internal use only.

void QIODevice::setStatus (int s) `[protected]`

For internal use only.

void QIODevice::setType (int t) `[protected]`

For internal use only.

uint QIODevice::size () const `[virtual]`

Virtual function that returns the size of the I/O device.

See also: at().

Reimplemented in QBuffer and QFile.

int QIODevice::state () const

Returns bits OR'ed together that specify the current state.

The flags are: `IO_Open`.

Subclasses may define more flags.

int QIODevice::status () const

Returns the I/O device status.

The I/O device status returns an error code. If open() returns FALSE or readBlock() or writeBlock() return -1, this function can be called to get the reason why the operation did not succeed.

The status codes are:

- `IO_Ok` The operation was successful.
- `IO_ReadError` Could not read from the device.
- `IO_WriteError` Could not write to the device.
- `IO_FatalError` A fatal unrecoverable error occurred.
- `IO_OpenError` Could not open the device.
- `IO_ConnectError` Could not connect to the device.
- `IO_AbortError` The operation was unexpectedly aborted.
- `IO_TimeOutError` The operation timed out.

See also: resetStatus().

int QIODevice::ungetch (int ch) `[virtual]`

Puts the character *ch* back into the I/O device and decrements the index if it is not zero.

This function is normally called to "undo" a getch() operation.

Returns *ch*, or -1 if some error occurred.

This virtual function must be reimplemented by all subclasses.

See also: getch() and putch().

Reimplemented in QBuffer and QFile.

int QIODevice::writeBlock (const char * data, uint len) `[virtual]`

Writes *len* bytes from *p* to the I/O device and returns the number of bytes actually written.

This virtual function must be reimplemented by all subclasses.

See also: readBlock().

Reimplemented in QBuffer and QFile.

97.2.1. URL

Generated from http://www.troll.no/qt/qiodevice.html.

98. QIODeviceSource Class Reference

`#include <qasyncio.h>`

Inherits QDataSource.

98.1. Detailed Description

A QIODeviceSource is a QDataSource that draws data from a QIODevice

This class encapsulates retrieving data from a QIODevice (such as a QFile).

98.2. Member Function Documentation

QIODeviceSource::QIODeviceSource (QIODevice * device, int buffer_size=4096)

Constructs a QIODeviceSource from a pointer to an QIODevice. The QIODevice *must* be dynamically allocated, becomes owned by the QIODeviceSource, and will be deleted when the QIODeviceSource destructs. *buffer_size* determines the size of buffering to use between asynchronous operations. The higher *buffer_size,* the more efficient but the less interleaved the operation will be with other processing.

QIODeviceSource::~QIODeviceSource ()

Destroys the QIODeviceSource, deleting the QIODevice from which it was constructed.

void QIODeviceSource::enableRewind (bool on) `[virtual]`

Enables rewinding. No special action is taken.

Reimplemented from QDataSource.

int QIODeviceSource::readyToSend () `[virtual]`

Ready until end-of-file.

Reimplemented from QDataSource.

bool QIODeviceSource::rewindable () const `[virtual]`

All QIODeviceSource's are rewindable.

Reimplemented from QDataSource.

void QIODeviceSource::sendTo (QDataSink * sink, int n) `[virtual]`

Reads and sends a block of data.

Reimplemented from QDataSource.

98.2.1. URL

Generated from http://www.troll.no/qt/qiodevicesource.html.

99. QKeyEvent Class Reference

`#include <qevent.h>`

Inherits QEvent.

99.1. Detailed Description

The QKeyEvent class contains parameters that describe a key event.

Key events occur when a key is pressed or released when a widget has keyboard input focus.

A key event contains a special accept flag which tells whether the receiver wants the key. You should call QKeyEvent::ignore() if the key press or release event is not handled by your widget.

The QWidget::setEnable() function can be used to enable or disable mouse and keyboard events for a widget.

The event handlers QWidget::keyPressEvent() and QWidget::keyReleaseEvent() receive key events.

See also: QFocusEvent and QWidget::grabKeyboard().

Examples:

table/table.cpp
picture/picture.cpp

99.2. Member Function Documentation

QKeyEvent::QKeyEvent (int type, int key, int ascii, int state)

Constructs a key event object.

The *type* parameter must be `Event_KeyPress` or `Event_KeyRelease.`

If *key* is 0, the event is not a result of a known key (e.g. it may be the result of a compose sequence or keyboard macro).

The accept flag is set to TRUE.

void QKeyEvent::accept ()

Sets the accept flag of the key event object.

Setting the accept parameter indicates that the receiver of the event wants the key event. Unwanted key events are sent to the parent widget.

The accept flag is set by default.

See also: ignore().

int QKeyEvent::ascii () const

Returns the ASCII code of the key that was pressed or released.

Examples:

picture/picture.cpp

void QKeyEvent::ignore ()

Clears the accept flag parameter of the key event object.

Clearing the accept parameter indicates that the event receiver does not want the key event. Unwanted key events are sent to the parent widget.

The accept flag is set by default.

See also: accept().

Examples:
table/table.cpp

bool QKeyEvent::isAccepted () const

Returns TRUE if the receiver of the event wants to keep the key.

int QKeyEvent::key () const

Returns the code of the key that was pressed or released.

The header file qkeycode.h lists the possible keyboard codes. These codes are independent of the underlying window system.

Key code 0 means that the event is not a result of a known key (e.g. it may be the result of a compose sequence or keyboard macro).

Examples:
table/table.cpp

int QKeyEvent::state () const

Returns the keyboard modifier flags.

The returned value is `ShiftButton, ControlButton` and `AltButton` OR'ed together.

99.2.1. URL

Generated from http://www.troll.no/qt/qkeyevent.html.

100. QLabel Class Reference

```
#include <qlabel.h>
```

Inherits QFrame.

100.1. Detailed Description

The QLabel widget displays a static text or pixmap.

A label is a static text or pixmap field.

It can have a frame (since QLabel inherits QFrame) and a "buddy" and an accelerator for moving keyboard focus to the buddy.

The contents of a label can be specified as a normal text, as a numeric value (which is internally converted to a text) or, as a pixmap. If the label is normal text and one of the letters is prefixed with '&', you can also specify a *buddy* for the label:

```
QLineEdit * phone = new QLineEdit( this, "phone number" );
QLabel * phoneLabel = new QLabel( phone, "&Phone", this );
```

In this example, keyboard focus is transferred to the label's buddy (the QLineEdit) when the user presses *Alt-P*. This is handy for many dialogs. You can also use the setBuddy() function to accomplish the same means.

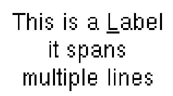
A label can be aligned in many different ways. The alignment setting specifies where to position the contents relative to the frame rectangle. See setAlignment() for a description of the alignment flags.

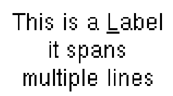
Enabling auto-resizing make the label resize itself whenever the contents change. The top left corner does not move.

This code sets up a sunken panel with a two-line text in the bottom right corner:

```
QLabel *label = new QLabel;
label->setFrameStyle( QFrame::Panel | QFrame::Sunken );
label->setText( "first line\nsecond line" );
label->setAlignment( AlignBottom | AlignRight );
```

Both lines are flush with the right side of the label.





See also: QLineEdit, QMovie and GUI Design Handbook: Label

Examples:

- tictac/tictac.cpp
- forever/forever.cpp
- movies/main.cpp
- xform/xform.cpp
- menu/menu.cpp
- pref/pref.cpp

progress/progress.cpp
cursor/cursor.cpp
layout/layout.cpp
qmag/qmag.cpp
showimg/showimg.cpp
widgets/widgets.cpp

100.2. Member Function Documentation

QLabel::QLabel (QWidget * buddy, const char * text, QWidget * parent, const char * name=0, WFlags f=0)

Constructs a label with an accelerator key.

The *parent, name* and *f* arguments are passed to the QFrame constructor. Note that the *parent* argument does *not* default to 0.

In a dialog, you might create two data entry widgets and a label for each, and set up the geometry so each label is just to the left of its data entry widget (its "buddy"), somewhat like this:

```
QLineEdit *name    = new QLineEdit( this );
QLabel    *name_l  = new QLabel( name, "&Name:", this );
QLineEdit *phone   = new QLineEdit( this );
QLabel    *phone_l = new QLabel( phone, "&Phone:", this );
// geometry management setup not shown
```

With the code above, the focus jumps to the Name field when the user presses Alt-N, and to the Phone field when the user presses Alt-P.

See also: setText() and setBuddy().

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

Constructs an empty label which is left-aligned, vertically centered, has an automatic margin and with manual resizing.

The *parent, name* and *f* arguments are passed to the QFrame constructor.

See also: setAlignment(), setFrameStyle(), setMargin() and setAutoResize().

QLabel::QLabel (const char * text, QWidget * parent=0, const char * name=0, WFlags f=0)

Constructs a label with a text. The label is left-aligned, vertically centered, has an automatic margin and with manual resizing.

The *parent, name* and *f* arguments are passed to the QFrame constructor.

See also: setAlignment(), setFrameStyle(), setMargin() and setAutoResize().

QLabel::~QLabel ()

Destroys the label.

int QLabel::alignment () const

Returns the alignment setting.

The default alignment is `AlignLeft | AlignVCenter | ExpandTabs` if the label doesn't have a buddy and `AlignLeft | AlignVCenter | ExpandTabs | ShowPrefix` if the label has a buddy.

See also: setAlignment().

bool QLabel::autoResize () const

Returns TRUE if auto-resizing is enabled, or FALSE if auto-resizing is disabled.

Auto-resizing is disabled by default.

See also: setAutoResize().

QWidget * QLabel::buddy () const

Returns the buddy of this label.

void QLabel::clear () `[slot]`

Clears the label. Equivalent with setText("").

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

Draws the label contents using the painter *p*.

Reimplemented from QFrame.

int QLabel::margin () const

Returns the margin of the label.

See also: setMargin().

QMovie* QLabel::movie () const

Returns the QMovie currently displaying in the label, or 0 if none has been set.

Examples:
 widgets/widgets.cpp

QPixmap * QLabel::pixmap () const

Returns the label pixmap.

See also: setPixmap().

void QLabel::setAlignment (int alignment)

Sets the alignment of the label contents and redraws itself.

The *alignment* is the bitwise OR of the following flags:

- `AlignLeft` aligns to the left border.
- `AlignRight` aligns to the right border.
- `AlignHCenter` aligns horizontally centered.
- `AlignTop` aligns to the top border.
- `AlignBottom` aligns to the bottom border.
- `AlignVCenter` aligns vertically centered
- `AlignCenter` (= `AlignHCenter` | AlignVCenter)
- `ExpandTabs` expands tabulators.

- `WordBreak` enables automatic word breaking.

If the label has a buddy, `ShowPrefix` is forced to TRUE.

See also: alignment(), setBuddy() and setText().

Examples:
cursor/cursor.cpp
layout/layout.cpp
qmag/qmag.cpp
widgets/widgets.cpp

void QLabel::setAutoResize (bool enable)

Enables auto-resizing if *enable* is TRUE, or disables it if *enable* is FALSE.

When auto-resizing is enabled, the label will resize itself whenever the contents change. The top left corner is not moved.

See also: autoResize() and adjustSize().

Examples:
forever/forever.cpp

void QLabel::setBuddy (QWidget * buddy)

Sets the buddy of this label to *buddy.*

When the user presses the accelerator key indicated by this label, the keyboard focus is transferred to the label's buddy.

See also: label() and setText().

Examples:
pref/pref.cpp
layout/layout.cpp

void QLabel::setMargin (int margin)

Sets the margin of the label to *margin* pixels.

The margin applies to the left edge if alignment() is `AlignLeft,` to the right edge if alignment() is `AlignRight,` to the top edge if alignment() is `AlignTop,` and to to the bottom edge if alignment() is `AlignBottom.`

If *margin* is negative (as it is by default), the label computes the margin as follows: If the frame width is zero, the effective margin becomes 0. If the frame style is greater than zero, the effective margin becomes half the width of the "x" character (of the widget's current font.

Setting a non-negative margin gives the specified margin in pixels.

See also: margin(), frameWidth() and font().

Examples:
movies/main.cpp
widgets/widgets.cpp

void QLabel::setMovie (const QMovie & movie) `[slot]`

Sets a QMovie to display in the label, or removes any existing movie if the given movie QMovie::isNull().

Any current pixmap or text label is cleared.

If the label has a buddy, the accelerator is disabled since the movie doesn't contain any suitable character.

Examples:
widgets/widgets.cpp

void QLabel::setNum (double num) `[slot]`

Sets the label contents to *num* (converts it to text) and redraws the contents.

If the label has a buddy, the accelerator is disabled since the number doesn't contain any suitable character.

The label resizes itself if auto-resizing is enabled.

See also: setAutoResize().

void QLabel::setNum (int num) `[slot]`

Sets the label contents to *num* (converts it to text) and redraws the contents.

If the label has a buddy, the accelerator is disabled since the number doesn't contain any suitable character.

The label resizes itself if auto-resizing is enabled. Nothing happens if *num* reads the same as the current label.

See also: setAutoResize().

void QLabel::setPixmap (const QPixmap & pixmap) `[slot]`

Sets the label contents to *pixmap* and redraws the contents.

If the label has a buddy, the accelerator is disabled since the pixmap doesn't contain any suitable character.

The label resizes itself if auto-resizing is enabled. Nothing happens if *pixmap* is the same as the current label.

See also: pixmap(), setText() and setAutoResize().

void QLabel::setText (const char * text) `[slot]`

Sets the label contents to *text,* updates the optional accelerator and redraws the contents.

The label resizes itself if auto-resizing is enabled. Nothing happens if *text* is the same as the current label.

See also: text(), setPixmap() and setAutoResize().

Examples:
forever/forever.cpp
xform/xform.cpp
pref/pref.cpp
cursor/cursor.cpp

layout/layout.cpp
qmag/qmag.cpp
widgets/widgets.cpp

QSize QLabel::sizeHint () const `[virtual]`

Returns a size which fits the contents of the label.

Bugs and limitations:

- Does not work well with the WordBreak flag

Examples:
pref/pref.cpp
layout/layout.cpp
widgets/widgets.cpp

Reimplemented from QWidget.

const char * QLabel::text () const

Returns the label text.

See also: setText().

100.2.1. URL

Generated from http://www.troll.no/qt/qlabel.html.

101. QLayout Class Reference

`#include <qlayout.h>`

Inherits QObject.

Inherited by QBoxLayout and QGridLayout.

101.1. Detailed Description

The QLayout class is the base class of geometry specifiers.

This is an abstract base class. The concrete layout managers QBoxLayout and QGridLayout inherit from this one and make QLayout's functionality avaialble in friendly APIs.

Most users of Q*Layout are likely to use some of the basic functions provided by QLayout, such as

- activate(), which compiles the layout into an internal representation and activates the result,
- setMenuBar(), which is necessary to manage a menu bar because of the special properties of menu bars, and
- freeze(), which allows you to freeze the widget's size and layout.

Geometry management stops when the layout manager is deleted.

To make a new layout manager, you need to implement the functions mainVerticalChain(), mainHorizontalChain() and initGM().

101.2. Member Function Documentation

QLayout::QLayout (QWidget * parent, int border, int autoBorder, const char * name) `[protected]`

Creates a new top-level QLayout with main widget *parent*. *parent* may not be 0.

border is the number of pixels between the edge of the widget and the managed children. *autoBorder* sets the value of defaultBorder(), which is interpreted by subclasses. If *autoBorder* is -1 the value of *border* is used.

name is the internal object name

Having several top-level layouts for the same widget will cause considerable confusion.

QLayout::QLayout (int autoBorder = -1, const char * name=0) `[protected]`

Constructs a new child QLayout, If *autoBorder* is -1, this QLayout inherits *parent's* defaultBorder(), otherwise *autoBorder* is used.

QLayout::˜QLayout () `[virtual]`

Deletes all children layouts. Geometry management stops when a toplevel layout is deleted.

bool QLayout::activate () `[virtual]`

Starts geometry management - analogous to show() for widgets. This function should only be called for top level layouts.

Examples:
  pref/pref.cpp
  layout/layout.cpp
  widgets/widgets.cpp

void QLayout::addChildLayout (QLayout * l) `[protected]`

This function is called from addLayout functions in subclasses, to add *l* layout as a sublayout.

QGManager * QLayout::basicManager () `[protected]`

Returns the QGManager for this layout. Returns 0 if this is a child layout which has not been inserted yet.

int QLayout::defaultBorder () const

Returns the default border for the geometry manager.

void QLayout::freeze (int w, int h)

Fixes the size of the main widget and distributes the available space to the child widgets. For widgets which should not be resizable, but where a QLayout subclass is used to set up the initial geometry.

A frozen layout cannot be unfrozen, the only sensible thing to do is to delete it.

The size is adjusted to a valid value. Thus freeze(0,0) fixes the widget to its minimum size.

void QLayout::freeze ()

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

This version of the method fixes the main widget at its minimum size. You can also achieve this with freeze(0, 0);.

QChain * QLayout::horChain (QLayout *) `[static protected]`

This function works around a dubious feature in the C++ language definition, to provide access to mainHorizontalChain().

void QLayout::initGM () `[virtual protected]`

Implement this function to do what's necessary to initialize chains, once the layout has a basicManager().

Reimplemented in QGridLayout and QBoxLayout.

QChain * QLayout::mainHorizontalChain () `[virtual protected]`

Implement this function to return the main horizontal chain.

Reimplemented in QGridLayout and QBoxLayout.

QChain * QLayout::mainVerticalChain () `[virtual protected]`

Implement this function to return the main vertical chain.

Reimplemented in QGridLayout and QBoxLayout.

QWidget * QLayout::mainWidget ()

Returns the main widget of this layout, or 0 if this layout is a sub-layout which is not yet inserted.

void QLayout::setMenuBar (QMenuBar * w)

Makes the geometry manager take account of the menu bar *w.* All child widgets are placed below the bottom edge of the menu bar.

A menu bar does its own geometry managing, never do addWidget() on a menu bar.

Examples:
widgets/widgets.cpp

QChain * QLayout::verChain (QLayout *) `[static protected]`

This function works around a dubious feature in the C++ language definition, to provide access to mainVerticalChain().

101.2.1. URL

Generated from http://www.troll.no/qt/qlayout.html.

102. QLCDNumber Class Reference

```
#include <qlcdnumber.h>
```

Inherits QFrame.

102.1. Detailed Description

The QLCDNumber widget displays a number with LCD-like digits.

It can display a number in just about any size, in decimal, hexadecimal, octal or binary notation, and is easy to connect to data sources via the display() slot, which is overloaded to take any of five argument types.

There are also slots to change the notation mode and decimal point mode.

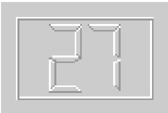
QLCDNumber emits the overflow() signal when it is asked to display something beyond its range. The range is set by setNumDigits() (but setSmallDecimalPoint() influences it too).

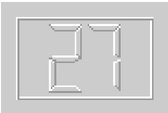
These digits and other symbols can be shown: 0/O, 1, 2, 3, 4, 5/S, 6, 7, 8, 9/g, minus, decimal point, A, B, C, D, E, F, h, H, L, o, P, r, u, U, Y, colon, degree sign (which is specified as single quote in the string) and space. QLCDNumber substitutes spaces for illegal characters.

It is impossible to retrieve the contents of a QLCDNumber object. If you need to, we recommend that you connect the signals which feed the display() slot to another slot as well and store the value there.

Warning: The current version of QLCDNumber does not work well with background pixmaps.

Incidentally, QLCDNumber is the very oldest part of Qt, tracing back to a BASIC program on the Sinclair Spectrum.





See also: QLabel and QFrame.

Examples:
dclock/dclock.cpp
xform/xform.cpp

102.2. Member Function Documentation

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

Constructs an LCD number, sets the number of digits to 5, the base to decimal, the decimal point mode to 'small' and the frame style to a raised box. The segmentStyle() is set to `Outline`.

The *parent* and *name* arguments are passed to the QFrame constructor.

See also: setNumDigits() and setSmallDecimalPoint().

QLCDNumber::QLCDNumber (uint numDigits, QWidget * parent=0, const char * name=0)

Constructs an LCD number, sets the number of digits to *numDigits,* the base to decimal, the decimal point mode to 'small' and the frame style to a raised box. The segmentStyle() is set to `Outline`.

The *parent* and *name* arguments are passed to the QFrame constructor.

See also: setNumDigits() and setSmallDecimalPoint().

QLCDNumber::˜QLCDNumber ()

Destroys the LCD number.

bool QLCDNumber::checkOverflow (double num) const

Returns TRUE if *num* is too big to be displayed in its entirety, FALSE otherwise.

See also: display(), numDigits() and smallDecimalPoint().

bool QLCDNumber::checkOverflow (int num) const

Returns TRUE if *num* is too big to be displayed in its entirety, FALSE otherwise.

See also: display(), numDigits() and smallDecimalPoint().

void QLCDNumber::display (int num) `[slot]`

Sets the contents of the display to *num.*

See also: setMode() and smallDecimalPoint().

Examples:
dclock/dclock.cpp
xform/xform.cpp

void QLCDNumber::display (const char * s) `[slot]`

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

This version of the function disregards mode() and smallDecimalPoint().

These digits and other symbols can be shown: 0/O, 1, 2, 3, 4, 5/S, 6, 7, 8, 9/g, minus, decimal point, A, B, C, D, E, F, h, H, L, o, P, r, u, U, Y, colon, degree sign (which is specified as single quote in the string) and space. QLCDNumber substitutes spaces for illegal characters.

void QLCDNumber::display (double num) `[slot]`

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

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

Draws the LCD number using painter *p.* This function is called from QFrame::paintEvent().

Reimplemented from QFrame.

int QLCDNumber::intValue () const

Returns the last value set by one of the display() slots rounded to the nearest integer.

If display(const char *) was the last called, 0 is returned.

See also: value() and display().

QLCDNumber::Mode QLCDNumber::mode() const

Returns the current display mode, which is one of `BIN, OCT, DEC` and `HEX.`

See also: setMode() and smallDecimalPoint().

int QLCDNumber::numDigits () const

Returns the current number of digits. If smallDecimalPoint() is FALSE, the decimal point occupies one digit position.

See also: setNumDigits() and smallDecimalPoint().

void QLCDNumber::overflow () `[signal]`

This signal is emitted whenever the QLCDNumber is asked to display a too large number or a too long string.

It is never emitted by setNumDigits().

void QLCDNumber::resizeEvent (QResizeEvent *) `[virtual protected]`

Handles resize Events for the QLCDNumber.

Reimplemented from QWidget.

QLCDNumber::SegmentStyle QLCDNumber::segmentStyle() const

Returns the style of the QLCDNumber.

See also: setSegmentStyle().

void QLCDNumber::setBinMode () `[slot]`

Calls setMode(BIN). Provided for convenience (e.g. for connecting buttons to this).

See also: setMode(), setHexMode(), setDecMode(), setOctMode() and mode().

void QLCDNumber::setDecMode () `[slot]`

Calls setMode(DEC). Provided for convenience (e.g. for connecting buttons to this).

See also: setMode(), setHexMode(), setOctMode(), setBinMode() and mode().

void QLCDNumber::setHexMode () `[slot]`

Calls setMode(HEX). Provided for convenience (e.g. for connecting buttons to this).

See also: setMode(), setDecMode(), setOctMode(), setBinMode() and mode().

void QLCDNumber::setMode (Mode m)

Sets the display mode to *m,* which must be one of `BIN`, `OCT`, `DEC` and `HEX`. All four modes can display both integers, floating-point numbers and strings (subject to character set limitations).

The display is set to "0".

Example:

```
myLcd.setMode( QLCDNumber::HEX );
```

See also: mode(), setHexMode(), setDecMode(), setOctMode(), setBinMode(), setSmallDecimalPoint() and display().

void QLCDNumber::setNumDigits (int numDigits)

Sets the number of digits shown to *numDigits,* which is forced into the [0-99] range.

If smallDecimalPoint() is TRUE, *numDigits* can be shown. If it is FALSE, only *numDigits-1* digits can be shown.

See also: numDigits(), setMode(), setSmallDecimalPoint() and overflow().

void QLCDNumber::setOctMode () `[slot]`

Calls setMode(OCT). Provided for convenience (e.g. for connecting buttons to this).

See also: setMode(), setHexMode(), setDecMode(), setBinMode() and mode().

void QLCDNumber::setSegmentStyle (SegmentStyle s)

Sets the style of the QLCDNumber to *s.*

- `Outline` gives raised segments filled with the background color.
- `Filled` gives raised segments filled with the foreground color.
- `Flat` gives flat segments filled with the foreground color.

`Outline` and `Filled` will additionally use QColorGroup::light() and QColorGroup::dark() for shadow effects.

See also: segmentStyle().

void QLCDNumber::setSmallDecimalPoint (bool b) `[slot]`

If *b* is TRUE, the decimal point is drawn between two digits. If *b* is FALSE, the decimal point is drawn in a digit position.

The inter-digit space is made slightly wider when the decimal point is drawn between the digits.

The display is set to "0.0".

See also: smallDecimalPoint() and setMode().

QSize QLCDNumber::sizeHint () const `[virtual]`

Returns a suitable size for this LCD number. The size hint is on the small side, so as to be usable as minimum size.

The size hint takes numDigits() and smallDecimalPoint() into consideration.

See also: setNumDigits(), setSmallDecimalPoint() and setMinimumSize().

Reimplemented from QWidget.

bool QLCDNumber::smallDecimalPoint () const

Returns TRUE if the decimal point is currently drawn between two digit positions, and FALSE if it is drawn in a digit position.

See also: setSmallDecimalPoint() and mode().

double QLCDNumber::value () const

Returns the last value set by one of the display() slots.

If display(const char *) was the last called, 0 is returned.

See also: intValue() and display().

102.2.1. URL

Generated from http://www.troll.no/qt/qlcdnumber.html.

103. QLineEdit Class Reference

#include <qlineedit.h>

Inherits QWidget.

103.1. Detailed Description

The QLineEdit widget is a simple line editor for inputting text.

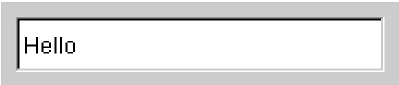
The default QLineEdit object has its own frame as specified by the Windows/Motif style guides, you can turn off the frame by calling setFrame(FALSE).

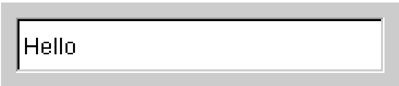
It draws the text using its own color group: colorGroup().text() on colorGroup().base(). The cursor and frame use other colors from same color group, of course.

QLineEdit can display the content of itself in three ways, depending on the current echo mode. The echo modes available are:

- `Normal` - display characters as they are entered. This is the default.
- `NoEcho` - do not display anything.
- `Password` - display asterisks instead of the characters actually entered.

The default key bindings are described in keyPressEvent(); they cannot be customized except by inheriting the class.





See also: QMultiLineEdit, QLabel, QComboBox, GUI Design Handbook: Field, Entry, and GUI Design Handbook: Field, Required.

Examples:
xform/xform.cpp
layout/layout.cpp
widgets/widgets.cpp

103.2. Member Function Documentation

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

Constructs a line editor with an empty edit buffer.

The cursor position is set to the start of the line, the maximum buffer size to 32767 characters, and the buffer contents to "".

The *parent* and *name* arguments are sent to the QWidget constructor.

QLineEdit::~QLineEdit ()

Destroys the line editor.

void QLineEdit::clear () `[slot]`

Syntax sugar for setText(""), provided to match no-argument signals.

void QLineEdit::clearValidator () `[slot]`

This slot is equivalent to setValidator(0).

int QLineEdit::cursorPosition () const

Returns the current cursor position for this line edit.

See also: setCursorPosition().

void QLineEdit::deselect () `[slot]`

Deselects all text (i.e. removes marking) and leaves the cursor at the current position.

QLineEdit::EchoMode QLineEdit::echoMode() const

Returns the current echo mode of the line edit.

See also: setEchoMode().

bool QLineEdit::event (QEvent * e) `[virtual protected]`

Don't use it if you don't mean it.

Reimplemented from QObject.

void QLineEdit::focusInEvent (QFocusEvent *) `[virtual protected]`

Handles the cursor blinking.

Reimplemented from QWidget.

void QLineEdit::focusOutEvent (QFocusEvent *) `[virtual protected]`

Handles the cursor blinking and selection copying.

Reimplemented from QWidget.

bool QLineEdit::frame () const

Returns TRUE if the line edit draws itself inside a frame, FALSE if it draws itself without any frame.

The default is to use a frame.

See also: setFrame().

bool QLineEdit::hasMarkedText () const `[protected]`

Returns TRUE if part of the text has been marked by the user (e.g. by clicking and dragging).

void QLineEdit::insert (const char * newText) `[slot]`

Removes any currently selected text, inserts *newText,* validates the result and if it is valid, sets it as the new contents of the line edit.

void QLineEdit::keyPressEvent (QKeyEvent * e) `[virtual protected]`

The key press event handler converts a key press to some line editor action.

If return or enter is pressed and the current text is valid (or if the validator can make the text valid), the signal returnPressed is emitted.

Here are the default key bindings:

- *Left Arrow* Move the cursor one character leftwards
- *Right Arrow* Move the cursor one character rightwards
- *Backspace* Delete the character to the left of the cursor
- *Home* Move the cursor to the beginning of the line
- *End* Move the cursor to the end of the line
- *Delete* Delete the character to the right of the cursor
- *Shift - Left Arrow* Mark text one character leftwards
- *Shift - Right Arrow* Mark text one character rightwards
- *Control-A* Move the cursor to the beginning of the line
- *Control-B* Move the cursor one character leftwards
- *Control-C* Copy the marked text to the clipboard.
- *Control-D* Delete the character to the right of the cursor
- *Control-E* Move the cursor to the end of the line
- *Control-F* Move the cursor one character rightwards
- *Control-H* Delete the character to the left of the cursor
- *Control-V* Paste the clipboard text into line edit.
- *Control-X* Cut the marked text, copy to clipboard.

All other keys with valid ASCII codes insert themselves into the line.

Reimplemented from QWidget.

void QLineEdit::leaveEvent (QEvent *) `[virtual protected]`

Handles selection copying.

Reimplemented from QWidget.

QString QLineEdit::markedText () const `[protected]`

Returns the text marked by the user (e.g. by clicking and dragging), or 0 if no text is marked.

See also: hasMarkedText().

int QLineEdit::maxLength () const

Returns the current maximum length of the text in the editor.

See also: setMaxLength().

void QLineEdit::mouseDoubleClickEvent (QMouseEvent *) `[virtual protected]`

Handles mouse double click events for this widget.

Reimplemented from QWidget.

void QLineEdit::mouseMoveEvent (QMouseEvent * e) `[virtual protected]`

Handles mouse move events for the line editor, primarily for marking text.

Reimplemented from QWidget.

void QLineEdit::mousePressEvent (QMouseEvent * e) `[virtual protected]`

Handles mouse press events for this widget.

Reimplemented from QWidget.

void QLineEdit::mouseReleaseEvent (QMouseEvent * e) `[virtual protected]`

Handles mouse release events for this widget.

Reimplemented from QWidget.

void QLineEdit::paintEvent (QPaintEvent * e) `[virtual protected]`

Handles paint events for the line editor.

Reimplemented from QWidget.

void QLineEdit::repaintArea (int from, int to) `[protected]`

Repaints all characters from *from* to *to*. If cursorPos is between from and to, ensures that cursorPos is visible.

void QLineEdit::resizeEvent (QResizeEvent *) `[virtual protected]`

Handles resize events for this widget.

Reimplemented from QWidget.

void QLineEdit::returnPressed () `[signal]`

This signal is emitted when the return or enter key is pressed.

void QLineEdit::selectAll () `[slot]`

Selects all text (i.e. marks it) and moves the cursor to the end. Useful when a default value has been inserted. If the user types before clicking on the widget the selected text will be erased.

void QLineEdit::setCursorPosition (int newPos)

Set the cursor position for this line edit to *newPos* and repaint accordingly.

See also: cursorPosition().

void QLineEdit::setEchoMode (EchoMode mode)

Sets the echo mode of the line edit widget.

The echo modes available are:

- `Normal` - display characters as they are entered. This is the default.
- `NoEcho` - do not display anything.
- `Password` - display asterisks instead of the characters actually entered.

It is always possible to cut and paste any marked text; only the widget's own display is affected.

See also: echoMode().

void QLineEdit::setEnabled (bool e) `[virtual]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QWidget.

void QLineEdit::setFont (const QFont & f) `[virtual]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QWidget.

void QLineEdit::setFrame (bool enable)

Sets the line edit to draw itself inside a two-pixel frame if *enable* is TRUE, and to draw itself without any frame if *enable* is FALSE.

The default is TRUE.

See also: frame() and QComboBox.

void QLineEdit::setMaxLength (int m)

Set the maximum length of the text in the editor. If the text is currently too long, it is chopped off at the limit. Any marked text will be unmarked. The cursor position is set to 0 and the first part of the string is shown.

See also: maxLength().

void QLineEdit::setPalette (const QPalette & p) `[virtual]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QWidget.

void QLineEdit::setSelection (int start, int length)

Sets the marked area of this line edit to start at *start* and be *length* characters long.

void QLineEdit::setText (const char * text) `[slot]`

Sets the line editor text to *text,* clears the selection and moves the cursor to the end of the line.

If necessary the text is truncated to fit maxLength().

See also: text().

Examples:
xform/xform.cpp

void QLineEdit::setValidator (QValidator * v)

Sets this line edit to accept input only as accepted by *v*.

If *v* == 0, remove the currently set input validator. The default is no input validator (ie. any input is accepted up to maxLength()).

See also: validator() and QValidator.

QSize QLineEdit::sizeHint () const `[virtual]`

Returns a size which fits the contents of the line edit.

The width returned tends to be enough for about 15-20 characters.

Examples:
layout/layout.cpp
widgets/widgets.cpp

Reimplemented from QWidget.

const char * QLineEdit::text () const

Returns a pointer to the text currently in the line.

If you need to store the text, you should make a copy of it. This can conveniently be done with a QString object:

```
QString s = lineEd->text();  // makes a copy and stores it in s
```

See also: setText().

Examples:
xform/xform.cpp

void QLineEdit::textChanged (const char *) `[signal]`

This signal is emitted every time the text has changed. The argument is the new text.

void QLineEdit::timerEvent (QTimerEvent * e) `[virtual protected]`

Not used.

Reimplemented from QObject.

bool QLineEdit::validateAndSet (const char * newText, int newPos, int newMarkAnchor, int newMarkDrag)

Validates and perhaps sets this line edit to contain *newText* with the cursor at position newPos, with marked text from *newMarkAnchor* to *newMarkDrag*. Returns TRUE if it changes the line edit and FALSE if it doesn't.

If *newText* contains more than one line is longer than maxLength(), validateAndSet() truncates it before testing its validity.

Repaints and emits textChanged() if appropriate.

QValidator * QLineEdit::validator () const

Returns a pointer to the current input validator, or 0 if no validator has been set.

103.2.1. URL

Generated from http://www.troll.no/qt/qlineedit.html.

104. QList Class Reference

`#include <qlist.h>`

Inherits QGList.

Inherited by QStrList.

104.1. Detailed Description

The QList class is a template class that provides doubly linked lists.

QList is implemented as both a template and a macro class. Define a template instance QList<X> to create a list that operates on pointers to X, or X*.

Example:

```
#include <qlist.h>
#include <qstring.h>
#include <stdio.h>
class Employee
{
public:
    Employee( const char *name, int salary ) { n=name; s=salary; }
    const char *name()   const               { return n; }
    int         salary() const               { return s; }
private:
    QString     n;
    int         s;
};
void main()
{
    QList<Employee> list;           // list of pointers to Employee
    list.setAutoDelete( TRUE );     // delete items when they are removed
    list.append( new Employee("Bill", 50000) );
    list.append( new Employee("Steve",80000) );
    list.append( new Employee("Ron",  60000) );
    Employee *emp;
    for ( emp=list.first(); emp != 0; emp=list.next() )
        printf( "%s earns %d\n", emp->name(), emp->salary() );
}
```

Program output:

```
Bill earns 50000
Steve earns 80000
Ron earns 60000
```

A list can also be instantiated through a macro expansion, but this is necessary only for compilers that do not support templates. See the collection classes documentation for a general discussion on template-based versus macro-based collections.

The list class is indexable and has a current index and a current item. The first item corresponds to index 0. The current index is -1 if the current item is null.

QList has several member functions for traversing the list, but using a QListIterator can be more practical. Multiple list iterators may traverse the same list, independent of each other and independent of the current list item.

In the example above, we make the call setAutoDelete(TRUE). Enabling auto-deletion tells the list to delete items that are removed from the list. The default is to not delete items when they are removed, but that would cause a memory leak in our example since we have no other references to the list items.

List items are stored as `void*` in an internal QLNode, which also holds pointers to the next and previous list items. The functions currentNode(), removeNode() and takeNode() operate directly on the QLNode, but they should be used with care.

When inserting an item into a list, only the pointer is copied, not the item itself. This is called a shallow copy. It is possible to make the list copy all of the item's data (known as a deep copy) when an item is inserted. insert(), inSort() and append() call the virtual function QCollection::newItem() for the item to be inserted. Inherit a list and reimplement it if you want deep copies.

When removing an item from a list, the virtual function QCollection::deleteItem() is called. QList's default implementation is to delete the item if auto-deletion is enabled.

The virtual function QGList::compareItems() can be reimplemented to compare two list items. This function is called from all list functions that need to compare list items, for instance remove(const type*). If you only want to deal with pointers, there are functions that compare pointers instead, for instance removeRef(const type*). These functions are somewhat faster than those that call compareItems().

The QStrList class in qstrlist.h is a list of `char*.` QStrList is a good example of a list that reimplements newItem(), deleteItem() and compareItems()

See also: QListIterator and Collection Classes

Examples:
grapher/grapher.cpp

104.2. Member Function Documentation

QList::QList ()

Constructs an empty list.

QList::QList (const QList<type> & list)

Constructs a copy of *list.*

Each item in *list* is appended to this list. Only the pointers are copied (shallow copy).

QList::~QList ()

Removes all items from the list and destroys the list.

All list iterators that access this list will be reset.

See also: setAutoDelete().

void QList::append (const type * item)

Inserts the *item* at the end of the list.

The inserted item becomes the current list item.

The *item* must not be a null pointer.

See also: insert() and current().

Examples:
grapher/grapher.cpp

int QList::at () const

Returns the index of the current list item. The returned value is -1 if the current item is null.

See also: current().

type * QList::at (uint index)

Returns a pointer to the item at position *index* in the list, or null if the index is out of range.

Sets the current list item to this item if *index* is valid. The valid range is `0 .. (count() - 1)` inclusive.

This function is very efficient. It starts scanning from the first item, last item or current item, whichever is closest to *index*.

See also: current().

void QList::clear () `[virtual]`

Removes all items from the list.

The removed items are deleted if auto-deletion is enabled.

All list iterators that access this list will be reset.

See also: remove(), take() and setAutoDelete().

Reimplemented from QCollection.

uint QList::contains (const type * item) const

Counts and returns the number of occurrences of *item* in the list.

The compareItems() function is called when looking for the *item* in the list. If compareItems() is not reimplemented, it is more efficient to call containsRef().

Does not affect the current list item.

See also: containsRef() and compareItems().

uint QList::containsRef (const type * item) const

Counts and returns the number of occurrences of *item* in the list.

Calling this function is must faster than contains(), because contains() compares *item* with each list item using compareItems(). This function only compares the pointers.

Does not affect the current list item.

See also: contains().

uint QList::count () const `[virtual]`

Returns the number of items in the list.

See also: isEmpty().

Examples:
grapher/grapher.cpp

Reimplemented from QCollection.

type * QList::current () const

Returns a pointer to the current list item. The current item may be null (implies that the current index is -1).

See also: at().

QLNode * QList::currentNode () const

Returns a pointer to the current list node.

The node can be kept and removed later using removeNode(). The advantage is that the item can be removed directly without searching the list.

Warning: Do not call this function unless you are an expert.

See also: removeNode(), takeNode() and current().

int QList::find (const type * item)

Finds the first occurrence of *item* in the list.

If the item is found, the list sets the current item to point to the found item and returns the index of this item. If the item is not found, the list sets the current item to null, the current index to -1 and returns -1.

The compareItems() function is called when searching for the item in the list. If compareItems() is not reimplemented, it is more efficient to call findRef().

See also: findNext(), findRef(), compareItems() and current().

int QList::findNext (const type * item)

Finds the next occurrence of *item* in the list, starting from the current list item.

If the item is found, the list sets the current item to point to the found item and returns the index of this item. If the item is not found, the list sets the current item to null, the current index to -1 and returns -1.

The compareItems() function is called when searching for the item in the list. If compareItems() is not reimplemented, it is more efficient to call findNextRef().

See also: find(), findNextRef(), compareItems() and current().

int QList::findNextRef (const type * item)

Finds the next occurrence of *item* in the list, starting from the current list item.

If the item is found, the list sets the current item to point to the found item and returns the index of this item. If the item is not found, the list sets the current item to null, the current index to -1 and returns -1.

Calling this function is must faster than findNext(), because findNext() compares *item* with each list item using compareItems(). This function only compares the pointers.

See also: findRef(), findNext() and current().

int QList::findRef (const type * item)

Finds the first occurrence of *item* in the list.

If the item is found, the list sets the current item to point to the found item and returns the index of this item. If the item is not found, the list sets the current item to null, the current index to -1 and returns -1.

Calling this function is must faster than find(), because find() compares *item* with each list item using compareItems(). This function only compares the pointers.

See also: findNextRef(), find() and current().

type * QList::first ()

Returns a pointer to the first item in the list and makes this the current list item, or null if the list is empty.

See also: getFirst(), last(), next(), prev() and current().

Examples:
grapher/grapher.cpp
showimg/showimg.cpp

type * QList::getFirst () const

Returns a pointer to the first item in the list, or null if the list is empty.

Does not affect the current list item.

See also: first() and getLast().

type * QList::getLast () const

Returns a pointer to the last item in the list, or null if the list is empty.

Does not affect the current list item.

See also: last() and getFirst().

void QList::inSort (const type * item)

Inserts the *item* at its sorted position in the list.

The sort order depends on the virtual QGList::compareItems() function. All items must be inserted with inSort() to maintain the sorting order.

The inserted item becomes the current list item.

The *item* must not be a null pointer.

See also: insert(), compareItems() and current().

bool QList::insert (uint index, const type * item)

Inserts the *item* at the position *index* in the list.

Returns TRUE if successful, or FALSE if *index* is out of range. The valid range is `0 .. count()` inclusive. The item is appended if *index* == count().

The inserted item becomes the current list item.

The *item* must not be a null pointer.

See also: append() and current().

bool QList::isEmpty () const

Returns TRUE if the list is empty, i.e. count() == 0. Returns FALSE otherwise.

See also: count().

type * QList::last ()

Returns a pointer to the last item in the list and makes this the current list item, or null if the list is empty.

See also: getLast(), first(), next(), prev() and current().

type * QList::next ()

Returns a pointer to the item succeeding the current item. Returns null if the current items is null or equal to the last item.

Makes the succeeding item current. If the current item before this function call was the last item, the current item will be set to null. If the current item was null, this function does nothing.

See also: first(), last(), prev() and current().

Examples:
grapher/grapher.cpp
showimg/showimg.cpp

QList<type> & QList::operator= (const QList<type> & list)

Assigns *list* to this list and returns a reference to this list.

This list is first cleared, then each item in *list* is appended to this list. Only the pointers are copied (shallow copy), unless newItem() has been reimplemented().

type * QList::prev ()

Returns a pointer to the item preceding the current item. Returns null if the current items is null or equal to the first item.

Makes the preceding item current. If the current item before this function call was the first item, the current item will be set to null. If the current item was null, this function does nothing.

See also: first(), last(), next() and current().

bool QList::remove ()

Removes the current list item.

Returns TRUE if successful, or FALSE if the current item is null.

The removed item is deleted if auto-deletion is enabled.

The item after the removed item becomes the new current list item if the removed item is not the last item in the list. If the last item is removed, the new last item becomes the current item. The current item is set to null if the list becomes empty.

All list iterators that refer to the removed item will be set to point to the new current item.

See also: take(), clear(), setAutoDelete(), current() and removeRef().

bool QList::remove (const type * item)

Removes the first occurence of *item* from the list.

Returns TRUE if successful, or FALSE if the item could not be found in the list.

The removed item is deleted if auto-deletion is enabled.

The compareItems() function is called when searching for the item in the list. If compareItems() is not reimplemented, it is more efficient to call removeRef().

The item after the removed item becomes the new current list item if the removed item is not the last item in the list. If the last item is removed, the new last item becomes the current item. The current item is set to null if the list becomes empty.

All list iterators that refer to the removed item will be set to point to the new current item.

See also: removeRef(), take(), clear(), setAutoDelete(), compareItems() and current().

bool QList::remove (uint index)

Removes the item at position *index* in the list.

Returns TRUE if successful, or FALSE if *index* is out of range. The valid range is `0 .. (count() - 1)` inclusive.

The removed item is deleted if auto-deletion is enabled.

The item after the removed item becomes the new current list item if the removed item is not the last item in the list. If the last item is removed, the new last item becomes the current item. The current item is set to null if the list becomes empty.

All list iterators that refer to the removed item will be set to point to the new current item.

See also: take(), clear(), setAutoDelete(), current() and removeRef().

bool QList::removeFirst ()

Removes the first item from the list. Returns TRUE if successful, or FALSE if the list is empty.

The removed item is deleted if auto-deletion is enabled.

The first item in the list becomes the new current list item. The current item is set to null if the list becomes empty.

All list iterators that refer to the removed item will be set to point to the new current item.

See also: removeLast(), setAutoDelete(), current() and remove().

bool QList::removeLast ()

Removes the last item from the list. Returns TRUE if successful, or FALSE if the list is empty.

The removed item is deleted if auto-deletion is enabled.

The last item in the list becomes the new current list item. The current item is set to null if the list becomes empty.

All list iterators that refer to the removed item will be set to point to the new current item.

See also: removeFirst(), setAutoDelete() and current().

void QList::removeNode (QLNode * node)

Removes the *node* from the list.

This node must exist in the list, otherwise the program may crash.

The removed item is deleted if auto-deletion is enabled.

The first item in the list will become the new current list item. The current item is set to null if the list becomes empty.

All list iterators that refer to the removed item will be set to point to the item succeeding this item, or the preceding item if the removed item was the last item.

Warning: Do not call this function unless you are an expert.

See also: takeNode(), currentNode(), remove() and removeRef().

bool QList::removeRef (const type * item)

Removes the first occurence of *item* from the list.

Returns TRUE if successful, or FALSE if the item cannot be found in the list.

The removed item is deleted if auto-deletion is enabled.

The list is scanned until the pointer *item* is found. It is removed if it is found.

Equivalent to:

```
if ( list.findRef(item) != -1 )
    list.remove();
```

The item after the removed item becomes the new current list item if the removed item is not the last item in the list. If the last item is removed, the new last item becomes the current item. The current item is set to null if the list becomes empty.

All list iterators that refer to the removed item will be set to point to the new current item.

See also: remove(), clear(), setAutoDelete() and current().

type * QList::take ()

Takes the current item out of the list without deleting it (even if auto-deletion is enabled). Returns a pointer to the item taken out of the list, or null if the current item is null.

The item after the taken item becomes the new current list item if the taken item is not the last item in the list. If the last item is taken, the new last item becomes the current item. The current item is set to null if the list becomes empty.

All list iterators that refer to the taken item will be set to point to the new current item.

See also: remove(), clear() and current().

type * QList::take (uint index)

Takes the item at position *index* out of the list without deleting it (even if auto-deletion is enabled).

Returns a pointer to the item taken out of the list, or null if the index is out of range. The valid range is `0 .. (count() - 1)` inclusive.

The item after the taken item becomes the new current list item if the taken item is not the last item in the list. If the last item is taken, the new last item becomes the current item. The current item is set to null if the list becomes empty.

All list iterators that refer to the taken item will be set to point to the new current item.

See also: remove(), clear() and current().

type * QList::takeNode (QLNode * node)

Takes the *node* out of the list without deleting its item (even if auto-deletion is enabled). Returns a pointer to the item taken out of the list.

This node must exist in the list, otherwise the program may crash.

The first item in the list becomes the new current list item.

All list iterators that refer to the taken item will be set to point to the item succeeding this item, or the preceding item if the taken item was the last item.

Warning: Do not call this function unless you are an expert.

See also: removeNode() and currentNode().

void QList::toVector (QGVector * vec) const

Stores all list items in the vector *vec.*

The vector must be have the same item type, otherwise the result will be undefined.

104.2.1. URL

Generated from http://www.troll.no/qt/qlist.html.

105. QListBox Class Reference

`#include <qlistbox.h>`

Inherits QTableView.

105.1. Detailed Description

The QListBox widget provides a single-column list of items that can be scrolled.

Each item in a QListBox contains a QListBoxItem. One of the items can be the current item. The highlighted() signal is emitted when the user highlights a new current item; selected() is emitted when the user double-clicks on an item or presses return when an item is highlighted.

If the user does not select anything, no signals are emitted and currentItem() returns -1.

A list box has `StrongFocus` as a default focusPolicy(), i.e. it can get keyboard focus both by tabbing and clicking.

New items may be inserted using either insertItem(), insertStrList() and inSort(). The list box is automatically updated to reflect the change; if you are going to insert a lot of data it may be worthwhile to wrap the insertion in calls to setAutoUpdate():

```
listBix->setAutoUpdate( FALSE );
for( i=1; i< hugeArray->count(); i++ )
    listBox->insertItem( hugeArray[i] );
listBox->setAutoUpdate( TRUE );
listBox->repaint();
```

Each change to insertItem() normally causes a screen update, and for a large change only a few of those updates are really necessary. Be careful to call repaint() when you re-enable updates, so the widget is completely repainted.

By default, vertical and horizontal scroll bars are added and removed as necessary. setAutoScrollBar() can be used to force a specific policy.

If you need to insert other types than texts and pixmaps, you must define new classes which inherit QListBoxItem.

Warning: The list box assumes ownership of all list box items and will delete them when they are not needed.

See also: QListView, QComboBox, QButtonGroup and GUI Design Handbook: List Box (two sections)

Examples:
widgets/widgets.cpp

105.2. Member Function Documentation

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

Constructs a list box. The arguments are passed directly to the QTableView constructor.

Note that the *f* argument is *not* table flags but rather widget flags.

QListBox::~QListBox ()

Destroys the list box. Deletes all list box items.

bool QListBox::autoBottomScrollBar () const

Returns TRUE if the list box has an automatic bottom scroll bar.

See also: setAutoBottomScrollBar() and autoScrollBar().

bool QListBox::autoScroll () const

Returns TRUE if auto-scrolling is enabled, otherwise FALSE.

See also: setAutoScroll and dragSelect().

bool QListBox::autoScrollBar () const

Returns TRUE if the list box has an automatic (vertical) scroll bar.

See also: setAutoScrollBar() and autoBottomScrollBar().

bool QListBox::autoUpdate () const

Returns TRUE if the list box updates itself automatically when items are inserted or removed.

The default setting is TRUE.

See also: setAutoUpdate().

bool QListBox::bottomScrollBar () const

Returns TRUE if the list box has a bottom scroll bar.

See also: setBottomScrollBar(), autoBottomScrollBar() and scrollBar().

int QListBox::cellHeight (int index = 0) `[virtual protected]`

Returns the height of the item at position *index* in pixels.

Reimplemented from QTableView.

void QListBox::centerCurrentItem ()

Scrolls the list box so the current (highlighted) item is centered in the list box.

See also: currentItem() and setTopItem().

void QListBox::changeItem (const QListBoxItem * lbi, int index)

Replaces the item at posistion *index* with *lbi.* If *index* is negative or too large, changeItem() does nothing.

See also: insertItem() and removeItem().

void QListBox::changeItem (const QPixmap & pixmap, int index)

Replaces the item at position *index* with *pixmap.*

The operation is ignored if *index* is out of range.

See also: insertItem() and removeItem().

void QListBox::changeItem (const char * text, int index)

Replaces the item at position *index* with *text.*

The operation is ignored if *index* is out of range.

See also: insertItem() and removeItem().

void QListBox::clear ()

Deletes all items in the list.

See also: removeItem() and setStrList().

void QListBox::clearList () `[protected]`

Deletes all items in the list. Protected function that does NOT update the list box.

void QListBox::clearSelection () `[slot]`

Deselects all items. Note that a single-selection listbox will automatically select its first item if it has keyboard focus.

uint QListBox::count () const

Returns the number of items in the list box.

int QListBox::currentItem () const

Returns the index of the current (highlighted) item of the list box, or -1 if no item has been selected.

See also: topItem().

bool QListBox::dragSelect () const

Returns TRUE if drag-selection is enabled, otherwise FALSE.

See also: setDragSelect() and autoScroll().

int QListBox::findItem (int yPos) const `[protected]`

Returns the index of the list box item at the vertical pixel-coordinate *yPos.*

See also: itemYPos().

void QListBox::focusInEvent (QFocusEvent *) `[virtual protected]`

Handles focus events. Repaints the current item (if not set, topItem() is made current).

See also: keyPressEvent() and focusOutEvent().

Reimplemented from QWidget.

void QListBox::focusOutEvent (QFocusEvent *) `[virtual protected]`

Handles focus out events. Repaints the current item, if set.

See also: keyPressEvent() and focusOutEvent().

Reimplemented from QWidget.

void QListBox::highlighted (const char *) `[signal]`

This signal is emitted when the user highlights a new current item and the new item is a string. The argument is the text of the new current item.

See also: selected(), currentItem() and selectionChanged().

void QListBox::highlighted (int index) `[signal]`

This signal is emitted when the user highlights a new current item. The argument is the index of the new item, which is already current.

See also: selected(), currentItem() and selectionChanged().

void QListBox::inSort (const QListBoxItem * lbi)

Inserts *lbi* at its sorted position in the list box.

All items must be inserted with inSort() to maintain the sorting order. inSort() treats any pixmap (or user-defined type) as lexicographically less than any string.

See also: insertItem().

void QListBox::inSort (const char * text)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

void QListBox::insertItem (const QListBoxItem * lbi, int index=-1)

Inserts the item *lbi* into the list at *index.*

If *index* is negative or larger than the number of items in the list box, *lbi* is inserted at the end of the list.

See also: insertStrList().

Examples:
widgets/widgets.cpp

void QListBox::insertItem (const QPixmap & pixmap, int index=-1)

Inserts *pixmap* into the list at *index.*

If *index* is negative, *pixmap* is inserted at the end of the list.

See also: insertStrList().

void QListBox::insertItem (const char * text, int index=-1)

Inserts *text* into the list at *index.*

If *index* is negative, *text* is inserted at the end of the list.

See also: insertStrList().

void QListBox::insertStrList (const QStrList * list, int index=-1)

Inserts the string list *list* into the list at item *index.*

If *index* is negative, *list* is inserted at the end of the list. If *index* is too large, the operation is ignored.

See also: insertItem() and inSort().

void QListBox::insertStrList (const char ** strings, int numStrings=-1, int index=-1)

Inserts the *numStrings* strings of the array *strings* into the list at item*index.*

If *index* is negative, insertStrList() inserts *strings* at the end of the list. If *index* is too large, the operation is ignored.

See also: insertItem() and inSort().

bool QListBox::isMultiSelection () const

Returns TRUE if the listbox is in multi-selection mode, and FALSE if it is in single-selection mode.

See also: setMultiSelection().

bool QListBox::isSelected (int i) const

Returns TRUE if item *i* is selected. Returns FALSE if it is not selected or if there is an error.

QListBoxItem * QListBox::item (int index) const `[protected]`

Returns a pointer to the item at position *index.*

int QListBox::itemHeight () const

Returns the standard item height (in pixels), or -1 if the list box has variable item height.

int QListBox::itemHeight (int index) const

Returns the height (in pixels) of item at *index.*

bool QListBox::itemVisible (int index) `[protected]`

Returns TRUE if the item at position *index* is at least partly visible.

bool QListBox::itemYPos (int index, int * yPos) const `[protected]`

Returns the vertical pixel-coordinate in **yPos,* of the list box item at position *index* in the list. Returns FALSE if the item is outside the visible area.

See also: findItem.

void QListBox::keyPressEvent (QKeyEvent * e) `[virtual protected]`

Handles key press events.

`Up` and `down` arrow keys make the highlighted item move and if necessary scroll the list box.

`Enter` makes the list box emit the selected() signal.

See also: selected() and setCurrentItem().

Reimplemented from QWidget.

long QListBox::maxItemWidth ()

For binary compatibility.

long QListBox::maxItemWidth () const

Returns the width in pixels of the widest item.

void QListBox::mouseDoubleClickEvent (QMouseEvent * e) `[virtual protected]`

Handles mouse double click events. Emits the selected() signal for the item that was double-clicked.

Reimplemented from QWidget.

void QListBox::mouseMoveEvent (QMouseEvent * e) `[virtual protected]`

Handles mouse move events. Scrolls the list box if auto-scroll is enabled.

See also: autoScroll().

Reimplemented from QWidget.

void QListBox::mousePressEvent (QMouseEvent * e) `[virtual protected]`

Handles mouse press events. Makes the clicked item the current item.

See also: currentItem().

Reimplemented from QWidget.

void QListBox::mouseReleaseEvent (QMouseEvent * e) `[virtual protected]`

Handles mouse release events.

Reimplemented from QWidget.

int QListBox::numItemsVisible () const

Returns the number of visible items. This may change at any time since the user may resize the widget.

See also: setFixedVisibleLines().

void QListBox::paintCell (QPainter * p, int row, int col) `[virtual protected]`

Repaints the cell at position *row* using *p*. The *col* argument is ignored, it is present because QTableView is more general. This function has the responsibility of showing focus and highlighting.

See also: QTableView::paintCell().

Reimplemented from QTableView.

const QPixmap * QListBox::pixmap (int index) const

Returns a pointer to the pixmap at position *index,* or 0 if there is no pixmap there.

See also: text().

void QListBox::removeItem (int index)

Removes the item at position *index.* If *index* is equal to currentItem(), a new item gets selected and the highlighted() signal is emitted.

See also: insertItem() and clear().

void QListBox::resizeEvent (QResizeEvent * e) `[virtual protected]`

Handles resize events. Updates internal parameters for the new list box size.

Reimplemented from QWidget.

bool QListBox::scrollBar () const

Returns TRUE if the list box has a (vertical) scroll bar.

See also: setScrollBar(), autoScrollBar() and bottomScrollBar().

void QListBox::selected (const char *) `[signal]`

This signal is emitted when the user double-clicks on an item or presses return while an item is highlighted, and the selected item is (or has) a string. The argument is the text of the selected item.

See also: highlighted() and selectionChanged().

void QListBox::selected (int index) `[signal]`

This signal is emitted when the user double-clicks on an item or presses return when an item is highlighted. The argument is the index of the selected item.

See also: highlighted() and selectionChanged().

void QListBox::selectionChanged () `[signal]`

This signal is emitted when the selection set of a multiple-choice listbox changes. If the user selects five items by drag-selecting, QListBox tries to emit just one selectionChanged() signal, so the signal can be connected to computationally expensive slots.

See also: selected() and currentItem().

void QListBox::setAutoBottomScrollBar (bool enable)

Enables an automatic bottom scroll bar if *enable* is TRUE, or disables it if *enable* is FALSE.

If it is enabled, then the list box will get a bottom scroll bar if the maximum list box item width exceeds the list box width.

The default setting is TRUE.

See also: autoBottomScrollBar(), setBottomScrollBar() and setAutoScrollBar().

void QListBox::setAutoScroll (bool enable)

Sets auto-scrolling if *enable* is TRUE, or disables it if *enable* is FALSE.

If auto-scrolling is enabled, the list box will scroll its contents when the user drags the mouse outside (below or above) the list box. Auto-scrolling only works if drag-selection is enabled.

The default setting is TRUE.

See also: autoScroll() and setDragSelect().

void QListBox::setAutoScrollBar (bool enable)

Enables an automatic (vertical) scroll bar if *enable* is TRUE, or disables it if *enable* is FALSE.

If it is enabled, then the list box will get a (vertical) scroll bar if the list box items exceed the list box height.

The default setting is TRUE.

See also: autoScrollBar(), setScrollBar() and setAutoBottomScrollBar().

void QListBox::setAutoUpdate (bool enable)

Specifies whether the list box should update itself automatically when items are inserted or removed.

Auto-update is enabled by default.

If *enable* is TRUE, the list box will update itself. If *enable* is FALSE, the list box will not update itself.

Warning: Do not leave the view in this state for a long time (i.e. between events). If the user interacts with the view when auto-update is off, strange things can happen.

See also: autoUpdate().

void QListBox::setBottomItem (int index)

Scrolls the list box so the item at position *index* in the list becomes the bottom row of the list box.

See also: setTopItem().

void QListBox::setBottomScrollBar (bool enable)

Enables a bottom scroll bar if *enable* is TRUE, or disables it if *enable* is FALSE.

The default setting is FALSE.

See also: bottomScrollBar(), setAutoBottomScrollBar() and setScrollBar().

void QListBox::setCurrentItem (int index)

Sets the highlighted item to the item at position *index* in the list. The highlighting is moved and the list box scrolled as necessary.

See also: currentItem().

void QListBox::setDragSelect (bool enable)

Sets drag-selection if *enable* is TRUE, or disables it if *enable* is FALSE.

If drag-selection is enabled, the list box will highlight new items when the user drags the mouse inside the list box.

The default setting is TRUE.

See also: drawSelect() and setAutoScroll().

void QListBox::setFixedVisibleLines (int lines)

Sets a fixed height for the widget, so that it shows the given number of lines of text for the current font size.

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

Reimplements QWidget::setFont() to update the list box line height and maxItemWidth().

Reimplemented from QWidget.

void QListBox::setMultiSelection (bool enable)

Sets the list box to multi-selection mode if *enable* is TRUE, and to single-selection mode if *enable* is FALSE.

Single- and multi-selections modes work the same, except that the highlighted() and selected() signals are emitted at different times.

See also: isMultiSelection().

void QListBox::setScrollBar (bool enable)

Enables a (vertical) scroll bar if *enable* is TRUE, or disables it if *enable* is FALSE.

The default setting is FALSE.

See also: scrollBar(), setAutoScrollBar() and setBottomScrollBar().

void QListBox::setSelected (int index, bool select)

Selects the item at position *index* if *select* is TRUE, or unselects it if *select* is FALSE. May also repaint the item.

If the listbox is a single-selection listbox and and *select* is TRUE, setCurrentItem will be called.

If the listbox is a single-selection listbox and and *select* is FALSE, clearSelection() will be called if *index* is the currently selected item.

See also: setMultiSelection(), setCurrentItem(), clearSelection() and currentItem().

void QListBox::setSmoothScrolling (bool enable)

Enables smooth list box scrolling if *enable* is TRUE, or disables it if *enable* is FALSE.

The default setting is TRUE.

See also: smoothScrolling().

void QListBox::setTopItem (int index)

Scrolls the list box so the item at position *index* in the list becomes the top row of the list box.

See also: topItem() and centerCurrentItem().

QSize QListBox::sizeHint () const `[virtual]`

Returns a size masedon maxItemWidth() and any value set by setFixedVisibleLines().

Reimplemented from QWidget.

bool QListBox::smoothScrolling () const

Returns TRUE if smooth list box scrolling is enabled, otherwise FALSE.

See also: setSmoothScrolling().

const char * QListBox::text (int index) const

Returns a pointer to the text at position *index,* or 0 if there is no text there.

See also: pixmap().

void QListBox::timerEvent (QTimerEvent *) `[virtual protected]`

Handles timer events. Does auto-scrolling.

Reimplemented from QObject.

void QListBox::toggleCurrentItem () `[protected]`

Toggles the selection status of currentItem() and repaints, if the listbox is a multi-selection listbox.

Does nothing if the listbox is a single-selection listbox.

See also: setMultiSelection().

int QListBox::topItem () const

Returns index of the item that is on the top line of the list box.

See also: setTopItem() and currentItem().

void QListBox::updateCellWidth () `[protected]`

Traverses the list and finds an item with the maximum width, and updates the internal list box structures accordingly.

void QListBox::updateItem (int index, bool erase = TRUE) `[protected]`

Repaints the item at position *index* in the list. Erases the line first if *erase* is TRUE.

105.2.1. URL

Generated from http://www.troll.no/qt/qlistbox.html.

106. QListBoxItem Class Reference

`#include <qlistbox.h>`

Inherited by QListBoxPixmap and QListBoxText.

106.1. Detailed Description

This is the base class of all list box items.

This class is the abstract base class of all list box items. If you need to insert customized items into a QListBox, you must inherit this class and reimplement paint(), height() and width().

The following shows how to define a list box item which shows a pixmap and a text:

```
class MyListBoxItem : public QListBoxItem
{
public:
    MyListBoxItem( const char *s, const QPixmap p )
        : QListBoxItem(), pm(p)
        { setText( s ); }
protected:
    virtual void paint( QPainter * );
    virtual int height( const QListBox * ) const;
    virtual int width( const QListBox * ) const;
    virtual const QPixmap *pixmap() { return &pm; }
private:
    QPixmap pm;
};
void MyListBoxItem::paint( QPainter *p )
{
    p->drawPixmap( 3, 0, pm );
    QFontMetrics fm = p->fontMetrics();
    int yPos;                       // vertical text position
    if ( pm.height() < fm.height() )
        yPos = fm.ascent() + fm.leading()/2;
    else
        yPos = pm.height()/2 - fm.height()/2 + fm.ascent();
    p->drawText( pm.width() + 5, yPos, text() );
}
int MyListBoxItem::height(const QListBox *lb ) const
{
    return QMAX( pm.height(), lb->fontMetrics().lineSpacing() + 1 );
}
int MyListBoxItem::width(const QListBox *lb ) const
{
    return pm.width() + lb->fontMetrics().width( text() ) + 6;
}
```

See also: QListBox.

106.2. Member Function Documentation

QListBoxItem::QListBoxItem ()

Constructs an empty list box item.

QListBoxItem::~QListBoxItem () `[virtual]`

Destroys the list box item.

int QListBoxItem::height (const QListBox *) const `[virtual]`

Implement this function to return the height of your item

See also: paint() and width().

Reimplemented in QListBoxPixmap and QListBoxText.

void QListBoxItem::paint (QPainter * p) `[virtual protected]`

Implement this function to draw your item.

See also: height() and width().

Reimplemented in QListBoxPixmap and QListBoxText.

const QPixmap * QListBoxItem::pixmap () const `[virtual]`

Returns the pixmap connected with the item, if any.

The default implementation of this function returns a null pointer.

Reimplemented in QListBoxPixmap.

void QListBoxItem::setText (const char * text) `[protected]`

Sets the text of the widget, which is used for sorting. The text is not shown unless explicitly drawn in paint().

See also: text().

const char * QListBoxItem::text () const `[virtual]`

Returns the text of the item, which is used for sorting.

See also: setText().

int QListBoxItem::width (const QListBox *) const `[virtual]`

Implement this function to return the width of your item

See also: paint() and height().

Reimplemented in QListBoxPixmap and QListBoxText.

106.2.1. URL

Generated from http://www.troll.no/qt/qlistboxitem.html.

107. QListBoxPixmap Class Reference

`#include <qlistbox.h>`

Inherits QListBoxItem.

107.1. Detailed Description

The QListBoxPixmap class provides list box items with a pixmap.

See also: QListBox and QListBoxItem.

107.2. Member Function Documentation

QListBoxPixmap::QListBoxPixmap (const QPixmap & pixmap)

Creates a new list box item showing the pixmap *pixmap*.

QListBoxPixmap::~QListBoxPixmap ()

Destroys the item.

int QListBoxPixmap::height (const QListBox *) const `[virtual protected]`

Returns the height of the pixmap.

See also: paint() and width().

Reimplemented from QListBoxItem.

void QListBoxPixmap::paint (QPainter * p) `[virtual protected]`

Draws the pixmap using painter *p*.

Reimplemented from QListBoxItem.

const QPixmap * QListBoxPixmap::pixmap () const `[virtual]`

Returns the pixmap connected with the item.

Reimplemented from QListBoxItem.

int QListBoxPixmap::width (const QListBox *) const `[virtual protected]`

Returns the width of the pixmap.

See also: paint() and height().

Reimplemented from QListBoxItem.

107.2.1. URL

Generated from http://www.troll.no/qt/qlistboxpixmap.html.

108. QListBoxText Class Reference

`#include <qlistbox.h>`

Inherits QListBoxItem.

108.1. Detailed Description

The QListBoxText class provides list box items with text.

The text is drawn in the widget's current font. If you need several different fonts, you have to make your own subclass of QListBoxItem.

See also: QListBox and QListBoxItem.

108.2. Member Function Documentation

QListBoxText::QListBoxText (const char * text = 0)

Constructs a list box item showing the text *text*.

QListBoxText::˜QListBoxText ()

Destroys the item.

int QListBoxText::height (const QListBox * lb) const `[virtual]`

Returns the height of a line of text.

See also: paint() and width().

Reimplemented from QListBoxItem.

void QListBoxText::paint (QPainter * p) `[virtual]`

Draws the text using painter *p*.

Reimplemented from QListBoxItem.

int QListBoxText::width (const QListBox * lb) const `[virtual]`

Returns the width of this line.

See also: paint() and height().

Reimplemented from QListBoxItem.

108.2.1. URL

Generated from http://www.troll.no/qt/qlistboxtext.html.

109. QListIterator Class Reference

`#include <qlist.h>`

Inherits QGListIterator.

109.1. Detailed Description

The QListIterator class provides an iterator for QList collections.

QListIterator is implemented as both a template and a macro class. Define a template instance QListIterator<X> to create a list iterator that operates on QList<X> (list of X*).

Example:

```
#include <qlist.h>
#include <qstring.h>
#include <stdio.h>
class Employee
{
public:
    Employee( const char *name, int salary ) { n=name; s=salary; }
    const char *name()   const               { return n; }
    int         salary() const               { return s; }
private:
    QString     n;
    int         s;
};
void main()
{
    QList<Employee> list;             // list of pointers to Employee
    list.setAutoDelete( TRUE );       // delete items when they are removed
    list.append( new Employee("Bill", 50000) );
    list.append( new Employee("Steve",80000) );
    list.append( new Employee("Ron",  60000) );
    QListIterator<Employee> it(list); // iterator for employee list
    for ( ; it.current(); ++it ) {
        Employee *emp = it.current();
        printf( "%s earns %d\n", emp->name(), emp->salary() );
    }
}
```

Program output:

```
Bill earns 50000
Steve earns 80000
Ron earns 60000
```

A QListIterator can also be instantiated through a macro expansion, but this is necessary only for compilers that do not support templates. See the collection classes documentation for a general discussion on template-based versus macro-based collections.

Although QList has member functions to traverse the doubly linked list structure, using a list iterator is a much more robust way of traversing the list, because multiple list iterators can operate on the same list, independent of each other and independent of the QList's current item. An iterator has its own current list item and can get the next and previous list items. It can only traverse the list, never modify it.

A QList knows about all list iterators that are operating on the list. When an item is removed from the list, the list update all iterators that are pointing the removed item to point to the new current list item.

Example:

```
#include <qlist.h>
#include <qstring.h>
#include <stdio.h>
class Employee
```

```
{
    ...     // same as above
};
void main()
{
    QList<Employee> list;              // list of pointers to Employee
    list.setAutoDelete( TRUE );        // delete items when they are removed
    list.append( new Employee("Bill", 50000) );
    list.append( new Employee("Steve",80000) );
    list.append( new Employee("Ron",  60000) );
    QListIterator<Employee> it(list);
    list.at( 1 );                      // current list item: "Steve"
    it.toLast();                       // it: "Ron"
    --it;                              // it: "Steve"
      // Now, both the list and the iterator are referring the same item
    list.remove();
    printf( "%s\n", it.current()->name() );
}
```

Program output:

```
Ron
```

See also: QList and collection classes

109.2. Member Function Documentation

QListIterator::QListIterator (const QList<type> & list)

Constructs an iterator for *list*. The current iterator item is set to point on the first item in the *list*.

QListIterator::~QListIterator ()

Destroys the iterator.

QListIterator::operator type * () const

Cast operator. Returns a pointer to the current iterator item. Same as current().

bool QListIterator::atFirst () const

Returns TRUE if the current iterator item is the first list item, otherwise FALSE.

See also: toFirst() and atLast().

bool QListIterator::atLast () const

Returns TRUE if the current iterator item is the last list item, otherwise FALSE.

See also: toLast() and atFirst().

uint QListIterator::count () const

Returns the number of items in the list this iterator operates on.

See also: isEmpty().

type * QListIterator::current () const

Returns a pointer to the current iterator item.

bool QListIterator::isEmpty () const

Returns TRUE if the list is empty, i.e. count() == 0, otherwise FALSE.

See also: count().

type * QListIterator::operator() ()

Makes the succeeding item current and returns the original current item.

If the current iterator item was the last item in the list or if it was null, null is returned.

type * QListIterator::operator++ ()

Prefix ++ makes the succeeding item current and returns the new current item.

If the current iterator item was the last item in the list or if it was null, null is returned.

type * QListIterator::operator+= (uint jump)

Sets the current item to the item *jump* positions after the current item, and returns a pointer to that item.

If that item is beyond the last item or if the dictionary is empty, it sets the current item to null and returns null.

type * QListIterator::operator-- ()

Prefix -- makes the preceding item current and returns the new current item.

If the current iterator item was the first item in the list or if it was null, null is returned.

type * QListIterator::operator-= (uint jump)

Returns the item *jump* positions before the current item, or null if it is beyond the first item. Makes this the current item.

QListIterator<type>& QListIterator::operator= (const QListIterator<type> & it)

Assignment. Makes a copy of the iterator *it* and returns a reference to this iterator.

type * QListIterator::toFirst ()

Sets the current iterator item to point to the first list item and returns a pointer to the item. Sets the current item to null and returns null if the list is empty.

See also: toLast() and atFirst().

type * QListIterator::toLast ()

Sets the current iterator item to point to the last list item and returns a pointer to the item. Sets the current item to null and returns null if the list is empty.

See also: toFirst() and atLast().

109.2.1. URL

Generated from http://www.troll.no/qt/qlistiterator.html.

110. QListView Class Reference

`#include <qlistview.h>`

Inherits QScrollView.

110.1. Detailed Description

The QListView class implements a list/tree view.

It can display and control a hierarchy of multi-column items, and provides the ability to add new items at run-time, let the user select one or many items, sort the list in increasing or decreasing order by any column, and so on.

The simplest mode of usage is to create a QListView, add some column headers using setColumn(), create one or more QListViewItem objects with the QListView as parent, set up the list view's geometry(), and show() it.

The main setup functions are

- addColumn() - adds a column, with text and perhaps width.
- setColumnWidthMode() - sets the column to be resized automatically or not.
- setMultiSelection() - decides whether one can select one or many objects in this list view. The default is FALSE (selecting one item unselects any other selected item).
- setAllColumnsShowFocus() - decides whether items should show keyboard focus using all columns, or just column 0. The default is to show focus using just column 0.
- setRootIsDecorated() - decides whether root items can be opened and closed by the user, and have open/close decoration to their left. The default is FALSE.
- setTreeStepSize() - decides the how many pixels an item's children are indented relative to their parent. The default is 20. This is mostly a matter of taste.
- setSorting() - decides whether the items should be sorted, whether it should be in ascending or descending order, and by what column it should be sorted.

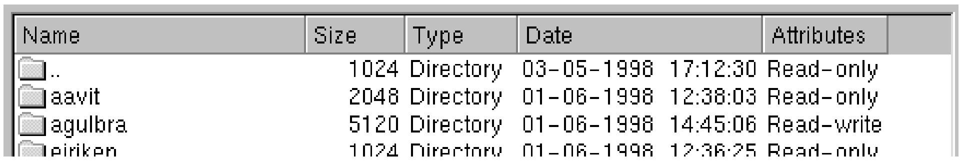
There are also several functions for mapping between items and coordinates. itemAt() returns the item at a position on-screen, itemRect() returns the rectangle an item occupies on the screen and itemPos() returns the position of any item (not on-screen, in the list view). firstChild() returns the item at the top of the view (not necessarily on-screen) so you can iterate over the items using either QListViewItem::itemBelow() or a combination of QListViewItem::firstChild() and QListViewItem::nextSibling().

Naturally, QListView provides a clear() function, as well as an explicit insertItem() for when QListViewItem's default insertion won't do.

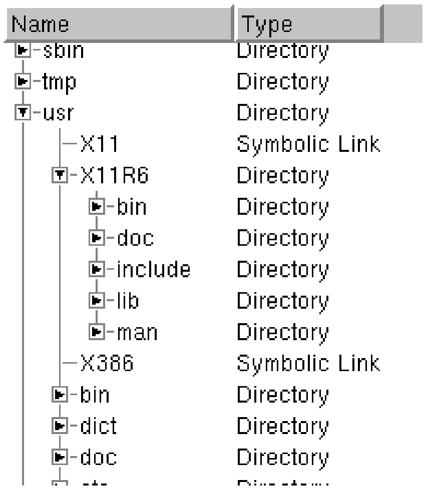
Since QListView offers multiple selection it has to display keyboard focus and selection state separately. Therefore there are functions both to set the selection state of an item, setSelected(), and to select which item displays keyboard focus, setCurrentItem().

QListView emits two groups of signals: One group signals changes in selection/focus state and one signals selection. The first group consists of selectionChanged(), applicable to all list views, and selectionChanged(QListViewItem *), applicable only to single-selection list view, and currentChanged(QListViewItem *). The second group consists of doubleClicked(QListViewItem *), returnPressed(QListViewItem *) and rightButtonClicked(QListViewItem *, const QPoint&, int).

In Motif style, QListView deviates fairly strongly from the look and feel of the Motif hierarchical tree view. This is done mostly to provide a usable keyboard interface and to make the list view look better with a white background.



Windows style, flat (from QFileDialog)



Motif style, hierarchial (from the dirview/dirview.cpp example).

Examples:
dirview/dirview.cpp

110.2. Member Function Documentation

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

Creates a new empty list view, with *parent* as a parent and *name* as object name.

QListView::˜QListView ()

Deletes the list view and all items in it, and frees all allocated resources.

int QListView::addColumn (const char * label, int width = -1) `[virtual]`

Adds a new column at the right end of the widget, with the header *label,* and returns the index of the column.

If *width* is negative, the new column will have WidthMode Maximum, otherwise it will be Manual at *width* pixels wide.

See also: setColumnText(), setColumnWidth() and setColumnWidthMode().

bool QListView::allColumnsShowFocus () const

Returns TRUE if the items in this list view indicate focus and selection state using all of their columns, else FALSE.

See also: setAllColumnsShowFocus().

int QListView::childCount () const

Returns the current number of parentless QListViewItem objects in this QListView, like QListViewItem::childCount() returns the number of child items for a QListViewItem.

See also: QListViewItem::childCount().

void QListView::clear () `[virtual]`

Remove and delete all the items in this list view, and trigger an update.

See also: triggerUpdate().

int QListView::columnAlignment (int column) const

const char* QListView::columnText (int c) const

Returns the text for the heading of column *c*.

int QListView::columnWidth (int c) const

Returns the width of the heading of column *c*.

QListView::WidthMode QListView::columnWidthMode(int c) const

Returns the currently set WidthMode for column *c*.

See also: setColumnWidthMode().

void QListView::currentChanged (QListViewItem *) `[signal]`

This signal is emitted whenever the current item has changed (normally after the screen update). The current item is the item responsible for indicating keyboard focus.

The argument is the newly current item.

See also: setCurrentItem() and currentItem().

QListViewItem * QListView::currentItem () const

Returns a pointer to the currently highlighted item, or 0 if there isn't any.

See also: setCurrentItem().

void QListView::doubleClicked (QListViewItem *) `[signal]`

This signal is emitted whenever an item is double-clicked. It's emitted on the second button press, not the second button release.

void QListView::drawContentsOffset (QPainter * p, int ox, int oy, int cx, int cy, int cw, int ch) `[virtual protected]`

Calls QListViewItem::paintCell() and/or QListViewItem::paintBranches() for all list view items that require repainting. See the documentation for those functions for details.

Reimplemented from QScrollView.

void QListView::ensureItemVisible (const QListViewItem * i)

Ensures that *i* is makde visible, scrolling the list view vertically as required.

See also: itemRect() and QSCrollView::ensureVisible().

bool QListView::eventFilter (QObject * o, QEvent * e) `[virtual]`

Redirects events for the viewport to mousePressEvent(), keyPressEvent() and friends.

Reimplemented from QObject.

QListViewItem * QListView::firstChild () const

Returns the first item in this QListView. You can use its firstChild() and nextSibling() functions to traverse the entire tree of items.

Returns 0 if there is no first item.

See also: itemAt(), itemBelow() and itemAbove().

void QListView::focusInEvent (QFocusEvent *) `[virtual protected]`

Handles focus in events on behalf of viewport(). Since viewport() is this widget's focus proxy by default, you can think of this function as handling this widget's focus in events.

See also: setFocusPolicy(), setFocusProxy() and focusOutEvent().

Reimplemented from QWidget.

void QListView::focusOutEvent (QFocusEvent *) `[virtual protected]`

Handles focus out events on behalf of viewport(). Since viewport() is this widget's focus proxy by default, you can think of this function as handling this widget's focus in events.

See also: setFocusPolicy(), setFocusProxy() and focusInEvent().

Reimplemented from QWidget.

QHeader * QListView::header () const

Returns a pointer to the QHeader object that manages this list view's columns. Please don't modify the header behind the list view's back.

See also: setHeader().

void QListView::insertItem (QListViewItem * i) `[virtual]`

Inserts a top-level QListViewItem into this list view. You generally do not need to call this; the QListViewItem constructor does it for you.

bool QListView::isMultiSelection () const

Returns TRUE if this list view is in multi-selection mode and FALSE if it is in single-selection mode.

See also: setMultiSelection().

bool QListView::isOpen (QListViewItem * item) const

Identical to *item->isOpen()*. Provided for completeness.

See also: setOpen().

bool QListView::isSelected (QListViewItem * i) const

Returns i->isSelected().

Provided only because QListView provides setSelected() and trolls are neat creatures and like neat, orthogonal interfaces.

QListViewItem * QListView::itemAt (const QPoint & screenPos) const

Returns a pointer to the QListViewItem at *screenPos*. Note that *screenPos* is in the coordinate system of viewport(), not in the listview's own, much larger, coordinate system.

itemAt() returns 0 if there is no such item.

See also: itemPos() and itemRect().

int QListView::itemMargin () const

Returns the advisory item margin which list items may use.

See also: QListViewItem::paintCell() and setItemMargin().

int QListView::itemPos (const QListViewItem * item)

Returns the y coordinate of *item* in the list view's coordinate system. This functions is normally much slower than itemAt(), but it works for all items, while itemAt() normally works only for items on the screen.

See also: itemAt() and itemRect().

QRect QListView::itemRect (const QListViewItem * i) const

Returns the rectangle on the screen *i* occupies in viewport()'s coordinates, or an invalid rectangle if *i* is a null pointer or is not currently visible.

The rectangle returned does not include any children of the rectangle (ie. it uses QListViewItem::height() rather than QListViewItem::totalHeight()). If you want the rectangle including children, you can use something like this code:

```
QRect r( listView->itemRect( item ) );
r.setHeight( (QCOORD)(QMIN( item->totalHeight(),
                            listView->viewport->height() - r.y() ) ) )
```

Note the way it avoids too-high rectangles. totalHeight() can be much larger than the window system's coordinate system allows.

itemRect() is comparatively slow. It's best to call it only for items that are probably on-screen.

void QListView::keyPressEvent (QKeyEvent * e) `[virtual protected]`

Handles key press events on behalf of viewport(). Since viewport() is this widget's focus proxy by default, you can think of this function as handling this widget's keyboard input.

Reimplemented from QWidget.

void QListView::mouseDoubleClickEvent (QMouseEvent * e) `[virtual protected]`

Processes mouse double-click events on behalf of the viewed widget; eventFilter() calls this function. Note that the coordinates in *e* is in the coordinate system of viewport().

Reimplemented from QWidget.

void QListView::mouseMoveEvent (QMouseEvent * e) `[virtual protected]`

Processes mouse move events on behalf of the viewed widget; eventFilter() calls this function. Note that the coordinates in *e* is in the coordinate system of viewport().

Reimplemented from QWidget.

void QListView::mousePressEvent (QMouseEvent * e) `[virtual protected]`

Processes mouse move events on behalf of the viewed widget; eventFilter() calls this function. Note that the coordinates in *e* is in the coordinate system of viewport().

Reimplemented from QWidget.

void QListView::mouseReleaseEvent (QMouseEvent * e) `[virtual protected]`

Processes mouse move events on behalf of the viewed widget; eventFilter() calls this function. Note that the coordinates in *e* is in the coordinate system of viewport().

Reimplemented from QWidget.

void QListView::paintEmptyArea (QPainter * p, const QRect & rect) `[virtual protected]`

Paints *rect* so that it looks like empty background using painter p. *rect* is is widget coordinates, ready to be fed to *p*.

The default function fills *rect* with colorGroup().base().

void QListView::repaintItem (const QListViewItem * item) const

Repaints *item* on the screen, if *item* is currently visible. Takes care to avoid multiple repaints.

void QListView::resizeEvent (QResizeEvent * e) `[virtual protected]`

Ensures that the header is correctly sized and positioned.

Reimplemented from QWidget.

void QListView::returnPressed (QListViewItem *) `[signal]`

This signal is emitted when enter or return is pressed. The argument is currentItem().

void QListView::rightButtonClicked (QListViewItem *, const QPoint &, int) `[signal]`

This signal is emitted when the right button is clicked (ie. when it's released). The arguments are the relevant QListViewItem (may be 0), the point in global coordinates and the relevant column.

void QListView::rightButtonPressed (QListViewItem *, const QPoint &, int) `[signal]`

This signal is emitted when the right button is pressed. Then arguments are the relevant QListViewItem (may be 0), the point in global coordinates and the relevant column.

bool QListView::rootIsDecorated () const

Returns TRUE if root items can be opened and closed by the user, FALSE if not.

void QListView::selectionChanged () `[signal]`

This signal is emitted whenever the set of selected items has changed (normally before the screen update). It is available both in single-selection and multi-selection mode, but is most meaningful in multi-selection mode.

See also: setSelected() and QListViewItem::setSelected().

void QListView::selectionChanged (QListViewItem *) `[signal]`

This signal is emitted whenever the selected item has changed in single-selection mode (normally after the screen update). The argument is the newly selected item.

There is another signal which is more useful in multi-selection mode.

See also: setSelected(), QListViewItem::setSelected() and currentChanged().

void QListView::setAllColumnsShowFocus (bool enable) `[virtual]`

Sets this list view to assume that the items show focus and selection state using all of their columns if *enable* is TRUE, or that they show it just using column 0 if *enable* is FALSE.

The default is FALSE.

Setting this to TRUE if it isn't necessary can cause noticeable flicker.

See also: allColumnsShowFocus().

void QListView::setColumnAlignment (int column, int align) `[virtual]`

void QListView::setColumnText (int column, const char * label) `[virtual]`

Sets the heading text of column *column* to *label*. The leftmost colum is number 0.

void QListView::setColumnWidth (int column, int w) `[virtual]`

Sets the width of column *column* to *w* pixels. Note that if the column has a WidthMode other than Manual, this width setting may be subsequently overridden. The leftmost colum is number 0.

void QListView::setColumnWidthMode (int c, WidthMode mode) `[virtual]`

Sets column `to` behave according to *mode,* which is one of:

- `Manual` - the column width does not change automatically
- `Maximum` - the column is automatically sized according to the widths of all items in the column.

See also: QListViewItem::width().

Bugs and limitations:

- doesn't shrink back yet when items shrink or close

void QListView::setCurrentItem (QListViewItem * i) `[virtual]`

Sets *i* to be the current highlighted item and repaints appropriately. This highlighted item is used for keyboard navigation and focus indication; it doesn't mean anything else.

See also: currentItem().

void QListView::setFont (const QFont & f) `[virtual]`

Reimplemented to let the list view items update themselves. *f* is the new font.

Reimplemented from QWidget.

void QListView::setItemMargin (int m) `[virtual]`

Sets the advisory item margin which list items may use to *m.*

The item margin defaults to one pixels and is the margin between the item's edges and the area where it draws its contents. QListViewItem::paintFocus() draws in the margin.

See also: QListViewItem::paintCell().

void QListView::setMultiSelection (bool enable) `[virtual]`

Sets the list view to multi-selection mode if *enable* is TRUE, and to single-selection mode if *enable* is FALSE.

See also: isMultiSelection().

void QListView::setOpen (QListViewItem * item, bool open) `[virtual]`

Sets *item* to be open if *open* is TRUE and \item is expandable, and to be closed if *open* is FALSE. Repaints accordingly.

Does nothing if *item* is not expandable.

See also: QListViewItem::setOpen() and QListViewItem::setExpandable().

void QListView::setPalette (const QPalette & p) `[virtual]`

Reimplemented to let the list view items update themselves. *p* is the new palette.

Reimplemented from QWidget.

void QListView::setRootIsDecorated (bool enable) `[virtual]`

Sets this list view to show open/close signs on root items if *enable* is TRUE, and to not show such signs if *enable* is FALSE.

Open/close signs is a little + or - in windows style, an arrow in Motif style.

void QListView::setSelected (QListViewItem * item, bool selected) `[virtual]`

Sets *item* to be selected if *selected* is TRUE, and to be not selected if *selected* is FALSE.

If the list view is in single-selection mode and *selected* is TRUE, the present selected item is unselected and made current. Unlike QListViewItem::setSelected(), this function updates the list view as necessary and emits the selectionChanged() signals.

See also: isSelected(), setMultiSelection(), isMultiSelection() and setCurrentItem().

void QListView::setSorting (int column, bool ascending = TRUE) `[virtual]`

Set the list view to be sorted by *column* and to be sorted in ascending order if *ascending* is TRUE or descending order if it is FALSE.

If *column* is -1, sorting is disabled.

void QListView::setStyle (GUIStyle s) `[virtual]`

Reimplemented to let the list view items update themselves. *s* is the new GUI style.

Reimplemented from QWidget.

void QListView::setTreeStepSize (int l) `[virtual]`

Sets the the number of pixels a child is offset from its parent, in a tree view to *l*. The default is 20.

See also: treeStepSize().

void QListView::show () `[virtual]`

Reimplemented to set the correct background mode and viewed area size.

Reimplemented from QWidget.

QSize QListView::sizeHint () const `[virtual]`

Returns a size suitable for this scroll view. This is as wide as mostly upon QHeader's sizeHint() recommends and tall enough for perhaps 10 items.

Reimplemented from QWidget.

int QListView::treeStepSize () const

Returns the number of pixels a child is offset from its parent. This number has meaning only for tree views. The default is 20.

See also: setTreeStepSize().

void QListView::triggerUpdate () `[slot]`

Triggers a size, geometry and contentual update during the next iteration of the event loop. Cleverly makes sure that there'll be just one update, to avoid flicker.

void QListView::updateContents () `[protected slot]`

Updates the sizes of the viewport, header, scrollbars and so on. Don't call this directly; call triggerUpdates() instead.

110.2.1. URL

Generated from http://www.troll.no/qt/qlistview.html.

111. QListViewItem Class Reference

`#include <qlistview.h>`

Inherited by QCheckListItem.

111.1. Detailed Description

The QListViewItem class implements a list view item.

A list viev item is a multi-column object capable of displaying itself. Its design has the following main goals:

- Work quickly and well for *large* sets of data.
- Be easy to use in the simple case.

The simplest way to use QListViewItem is to construct one with a few constant strings. This creates an item which is a child of *parent,* with two fixed-content strings, and discards the pointer to it:

```
(void) new QListViewItem( parent, "first column", "second column" );
```

This object will be deleted when *parent* is deleted, as for QObjects.

The parent is either another QListViewItem or a QListView. If the parent is a QListView, this item is a top-level item within that QListView. If the parent is another QListViewItem, this item becomes a child of the parent item.

If you keep the pointer, you can set or change the texts using setText(), add pixmaps using setPixmap(), change its mode using setSelectable(), setSelected(), setOpen() and setExpandable(), change its height using setHeight(), and do much tree traversal. The set* functions in QListView also affect QListViewItem, of course.

You can traverse the tree as if it were a doubly linked list using itemAbove() and itemBelow(); they return pointers to the items directly above and below this item on the screen (even if none of the three are actually visible at the moment).

You can also traverse it as a tree, using parent(), firstChild() and nextSibling(). This code does something to each of an item's children:

```
QListViewItem * myChild = myItem->firstChild();
while( myChild ) {
    doSomething( myChild );
    myChild = myChild->nextSibling();
}
```

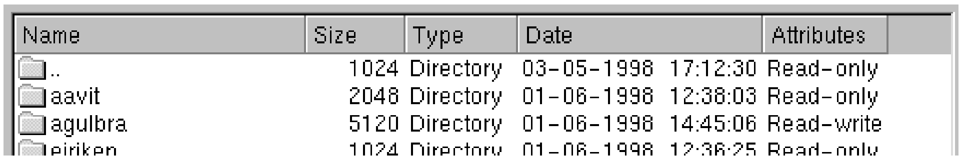
Note that the order of the children will change when the sorting order changes, and is undefined if the items are not visible. You can however call enforceSortOrder() at any time, and QListView will always call it before it needs to show an item.

Many programs will need to reimplement QListViewItem. The most commonly reimplemented functions are:

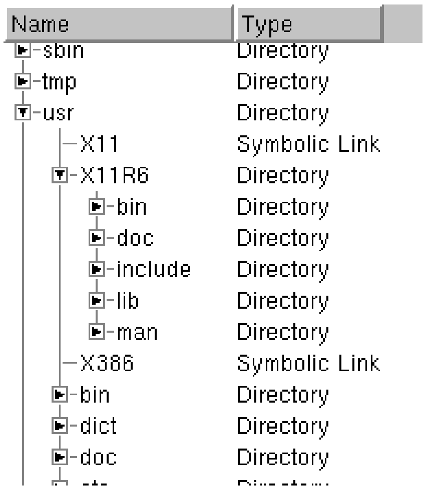
- text() returns the text in a column. Many subclasses will compute that on the fly.
- key() is used for sorting. The default key() simply calls text(), but judicious use of key can be used to sort by e.g. date (as QFileDialog does).
- setup() is called before showing the item, and whenever e.g. the font changes.
- activate() is called whenever the user clicks on the item or presses space when the item is the currently highlighted item.

Some subclasses call setExpandable(TRUE) even when they have no children, and populate themselves when setup() is called. The dirview/dirview.cpp example program uses precisely this technique to start up quickly: The files and subdirectories in a directory aren't entered into the tree until they need to.

This example shows a number of root items in a QListView. These items are actually subclassed from QListViewItem: The file size, type etc. are computed on the fly.



The next example shows a fraction of the dirview example. Again, the Directory/Symbolic Link column is computed on the fly. None of the items are root items; the *usr* item is a child of the root and the *X11* item is a child of the *usr* item.



Examples:
dirview/dirview.cpp

111.2. Member Function Documentation

QListViewItem::QListViewItem (QListView * parent)

Create a new list view item in the QListView *parent* and first in the parent's list of children.

QListViewItem::QListViewItem (QListView * parent, QListViewItem * after)

Constructs an empty list view item which is a child of *parent* and is after *after* in the parent's list of children.

QListViewItem::QListViewItem (QListView * parent, QListViewItem * after, const char * label1, const char * label2 = 0, const char * label3 = 0, const char * label4 = 0, const char * label5 = 0, const char * label6 = 0, const char * label7 = 0, const char * label8 = 0)

Creates a new list view item in the QListView *parent,* after item *after,* with at most 8 constant strings as contents.

Note that the order is changed according to QListViewItem::key() unless the list view's sorting is disabled using QListView::setSorting(-1).

QListViewItem::QListViewItem (QListView * parent, const char * label1, const char * label2 = 0, const char * label3 = 0, const char * label4 = 0, const char * label5 = 0, const char * label6 = 0, const char * label7 = 0, const char * label8 = 0)

Creates a new list view item in the QListView *parent, parent,* with at most 8 constant strings as contents.

```
(void)new QListViewItem( lv, "/", "Root directory" );
```

QListViewItem::QListViewItem (QListViewItem * parent)

Create a new list view item which is a child of *parent* and first in the parent's list of children.

QListViewItem::QListViewItem (QListViewItem * parent, QListViewItem * after)

Constructs an empty list view item which is a child of *parent* and is after *after* in the parent's list of children.

QListViewItem::QListViewItem (QListViewItem * parent, QListViewItem * after, const char * label1, const char * label2 = 0, const char * label3 = 0, const char * label4 = 0, const char * label5 = 0, const char * label6 = 0, const char * label7 = 0, const char * label8 = 0)

Creates a new list view item that's a child of the QListViewItem *parent,* after item *after,* with at most 8 constant strings as contents.

Note that the order is changed according to QListViewItem::key() unless the list view's sorting is disabled using QListView::setSorting(-1).

QListViewItem::QListViewItem (QListViewItem * parent, const char * label1, const char * label2 = 0, const char * label3 = 0, const char * label4 = 0, const char * label5 = 0, const char * label6 = 0, const char * label7 = 0, const char * label8 = 0)

Creates a new list view item that's a child of the QListViewItem *parent,* with at most 8 constant strings as contents. Possible example in a threaded news or e-mail reader:

```
(void)new QListViewItem( parentMessage, author, subject );
```

QListViewItem::˜QListViewItem () `[virtual]`

Deletes this item and all children of it, freeing up all allocated resources.

void QListViewItem::activate () `[virtual protected]`

This virtual function is called whenever the user clicks on this item or presses Space on it. The default implementation does nothing.

Reimplemented in QCheckListItem.

int QListViewItem::childCount () const

Returns the current number of children of this item.

int QListViewItem::depth () const

Returns the depth of this item.

void QListViewItem::enforceSortOrder () const `[virtual protected]`

Makes sure that this object's children are sorted appropriately.

This only works if every item in the chain from the root item to this item is sorted appropriately.

See also: sortChildItems().

QListViewItem* QListViewItem::firstChild () const

Returns a pointer to the first (top) child of this item.

Note that the children are not guaranteed to be sorted properly. QListView and QListViewItem try to postpone or avoid sorting to the greatest degree possible, in order to keep the user interface snappy.

See also: nextSibling().

int QListViewItem::height () const

Returns the height of this item in pixels. This does not include the height of any children; totalHeight() returns that.

void QListViewItem::insertItem (QListViewItem * newChild) `[virtual]`

Inserts *newChild* into its list of children. Called by the constructor of *newChild.*

void QListViewItem::invalidateHeight () `[virtual]`

Invalidates the cached total height of this item including all open children.

See also: setHeight(), ownHeight() and totalHeight().

bool QListViewItem::isExpandable ()

{ return expnadable; }

Returns TRUE if this item is expandable even when it has no children.

bool QListViewItem::isOpen () const

Returns TRUE if this list view item has children *and* they are potentially visible, or FALSE if the item has no children or they are hidden.

See also: setOpen().

bool QListViewItem::isSelectable () const

Returns TRUE if the item is selectable (as it is by default) and FALSE if it isn't.

See also: setSelectable().

bool QListViewItem::isSelected () const

Returns TRUE if this item is selected, or FALSE if it is not.

See also: setSelection(), selectionChanged() and QListViewItem::setSelected().

QListViewItem * QListViewItem::itemAbove ()

Returns a pointer to the item immediately above this item on the screen. This is usually the item's closest older sibling, but may also be its parent or its next older sibling's youngest child, or something else if anyoftheabove->height() returns 0.

This function assumes that all parents of this item are open (ie. that this item is visible, or can be made visible by scrolling).

See also: itemBelow() and itemRect().

QListViewItem * QListViewItem::itemBelow ()

Returns a pointer to the item immediately below this item on the screen. This is usually the item's eldest child, but may also be its next younger sibling, its parent's next younger sibling, granparent's etc., or something else if anyoftheabove->height() returns 0.

This function assumes that all parents of this item are open (ie. that this item is visible, or can be made visible by scrolling).

See also: itemAbove() and itemRect().

const char * QListViewItem::key (int column, bool ascending) const `[virtual]`

Returns a key that can be used for sorting by column *column.* The default implementation returns text(). Derived classes may also incorporate the order indicated by *ascending* into this key, although this is not recommended.

QListViewItem immediately copies the return value of this function, so it's safe to return a pointer to a static variable.

You can use this function to sort by non-alphabetic data. This code excerpt sort by file modification date, for example

```
if ( column == 3 ) {
    QDateTime epoch( QDate( 1980, 1, 1 ) );
    tmpString.sprintf( "%08d", epoch.secsTo( myFile.lastModified() ) );
} else {
    // ....
}
return tmpString;
```

See also: sortChildItems().

QListView * QListViewItem::listView () const

Returns a pointer to the listview containing this item.

QListViewItem* QListViewItem::nextSibling () const

Returns a pointer to the next sibling (below this one) of this item.

Note that the siblings are not guaranteed to be sorted properly. QListView and QListViewItem try to postpone or avoid sorting to the greatest degree possible, in order to keep the user interface snappy.

See also: firstChild().

void QListViewItem::paintBranches (QPainter * p, const QColorGroup & cg, int w, int y, int h, GUIStyle s) `[virtual]`

Paints a set of branches from this item to (some of) its children.

p is set up with clipping and translation so that you can draw only in the rectangle you need to; *cg* is the color group to use, 0,top is the top left corner of the update rectangle, w-1,top is the top right corner, 0,bottom-1 is the bottom left corner and the bottom right corner is left as an excercise for the reader.

The update rectangle is in an undefined state when this function is called; this function must draw on *all* of the pixels.

See also: paintCell() and QListView::drawContentsOffset().

Reimplemented in QCheckListItem.

void QListViewItem::paintCell (QPainter * p, const QColorGroup & cg, int column, int width, int align) `[virtual]`

This virtual function paints the contents of one column of one item.

p is a QPainter open on the relevant paint device. *pa* is translated so 0, 0 is the top left pixel in the cell and *width-1,* height()-1 is the bottom right pixel *in* the cell. The other properties of *p* (pen, brush etc) are undefined. *cg* is the color group to use. *column* is the logical column number within the item that is to be painted; 0 is the column which may contain a tree.

This function may use QListView::itemMargin() for readability spacing on the left and right sides of information such as text, and should honour isSelected() and QListView::allColumnsShowFocus().

If you reimplement this function, you should also reimplement width().

The rectangle to be painted is in an undefined state when this function is called, so you *must* draw on all the pixels.

See also: paintBranches() and QListView::drawContentsOffset().

Reimplemented in QCheckListItem.

void QListViewItem::paintFocus (QPainter * p, const QColorGroup & cg, const QRect & r) `[virtual]`

Paints a focus indication on the rectangle *r* using painter *p* and colors *cg.*

p is already clipped.

See also: paintCell(), paintBranches() and QListView::setAllColumnsShowFocus().

QListViewItem* QListViewItem::parent () const

Returns a pointer to the parent of this item.

See also: firstChild() and nextSibling().

const QPixmap * QListViewItem::pixmap (int column) const `[virtual]`

Returns a pointer to the pixmap for *column,* or a null pointer if there is no pixmap for *column.*

See also: setText() and setPixmap().

void QListViewItem::removeItem (QListViewItem * tbg) `[virtual]`

Removes *tbg* from this object's list of children. This is normally called by tbg's destructor.

void QListViewItem::repaint () const

Repaints this item on the screen, if it is currently visible.

void QListViewItem::setExpandable (bool enable) `[virtual]`

Sets this item to be expandable even if it has no children if *enable* is TRUE, and to be expandable only if it has children if *enable* is FALSE (the default).

The dirview example uses this in the canonical fashion: It checks whether the directory is empty in setup() and calls setExpandable(TRUE) if not, and in setOpen() it reads the contents of the directory and inserts items accordingly. This strategy means that dirview can display the entire file system without reading very much at start-up.

Note that root items are not expandable by the user unless QListView::setRootIsDecorated() is set to TRUE.

See also: setSelectable().

Examples:
dirview/dirview.cpp

void QListViewItem::setHeight (int height) `[virtual protected]`

Sets this item's own height to *height* pixels. This implictly changes totalHeight() too.

Note that e.g. a font change causes this height to be overwitten unless you reimplement setup().

See also: ownHeight(), totalHeight() and isOpen();.

void QListViewItem::setOpen (bool o) `[virtual]`

Sets this item to be open (its children are visible) if *o* is TRUE, and to be closed (its children are not visible) if *o* is FALSE.

Also does some bookeeping.

See also: ownHeight() and totalHeight().

Examples:
dirview/dirview.cpp

void QListViewItem::setPixmap (int column, const QPixmap & pm) `[virtual]`

Sets the pixmap in column *column* to *pm,* if *pm* is non-null and *column* is non-negative.

See also: pixmap() and setText().

void QListViewItem::setSelectable (bool enable) `[virtual]`

Sets this items to be selectable if *enable* is TRUE (the default) or not to be selectable if *enable* is FALSE.

The user is not able to select a non-selectable item using either the keyboard or mouse. The application programmer still can, of course.

See also: isSelectable().

void QListViewItem::setSelected (bool s) `[virtual]`

Sets this item to be selected *s* is TRUE, and to not be selected if *o* is FALSE.

This function does not maintain any invariants or repaint anything - QListView::setSelected() does that.

See also: ownHeight() and totalHeight().

void QListViewItem::setText (int column, const char * text) `[virtual]`

Sets the text in column *column* to *text,* if *column* is a valid column number and *text* is non-null.

If *text()* has been reimplemented, this function may be a no-op.

See also: text() and key().

void QListViewItem::setup () `[virtual]`

This virtual function is called before the first time QListView needs to know the height or any other graphical attribute of this object, and whenever the font, GUI style or colors of the list view change.

The default calls widthChanged() and sets the item's height.

Examples:
dirview/dirview.cpp

Reimplemented in QCheckListItem.

void QListViewItem::sortChildItems (int column, bool ascending) `[virtual]`

Sorts the children of this item by the return values of key(*column, ascending),* in ascending order if *ascending* is TRUE and in descending order of *descending* is FALSE.

Asks some of the children to sort their children. (QListView and QListViewItem ensure that all on-screen objects are properly sorted, but may avoid or defer sorting other objects in order to be more responsive.)

See also: key().

const char * QListViewItem::text (int column) const `[virtual]`

Returns the text in column *column,* or else 0.

The returned pointer must be copied or used at once so that reimplementations of this function are at liberty to e.g. return a pointer into a static buffer.

See also: key() and paintCell().

Reimplemented in QCheckListItem.

int QListViewItem::totalHeight () const

Returns the total height of this object, including any visible children. This height is recomputed lazily and cached for as long as possible.

setOwnHeight() can be used to set the item's own height, setOpen() to show or hide its children, and invalidateHeight() to invalidate the cached height.

See also: height().

int QListViewItem::width (const QFontMetrics & fm, const QListView * lv, int c) const `[virtual]`

Returns the number of pixels of width required to draw column *c* of listview *lv,* using the metrics *fm* without cropping. The list view containing this item may use this information, depending on the QListView::WidthMode settings for the column.

The default implementation returns the width of the bounding rectangle of the text of column *c.*

See also: listView(), widthChanged(), QListView::setColumnWidthMode() and QListView::itemMargin().

void QListViewItem::widthChanged (int c=-1) const

Call this function when the value of width() may have changed for column *c.* Normally, you should call this if text(c) changes. Passing -1 for *c* indicates all columns may have changed. For efficiency, you should do this if more than one call to widthChanged() is required.

See also: width().

111.2.1. URL

Generated from http://www.troll.no/qt/qlistviewitem.html.

112. QLNode Class Reference

```
#include <qglist.h>
```

112.1. Detailed Description

The QLNode class is an internal class for the QList template collection.

QLNode is a doubly linked list node; it has three pointers:

1. Pointer to the previous node.
2. Pointer to the next node.
3. Pointer to the actual data.

Sometimes it might be practical to have direct access to the list nodes in a QList, but it is seldom required.

Warning: Be very careful if you want to access the list nodes. The heap can easily get corrupted if you make a mistake.

See also: QList::currentNode(), QList::removeNode() and QList::takeNode().

112.2. Member Function Documentation

GCI QLNode::getData ()

Returns a pointer `(void*)` to the actual data in the list node.

112.2.1. URL

Generated from http://www.troll.no/qt/qlnode.html.

113. QMainWindow Class Reference

`#include <qmainwindow.h>`

Inherits QWidget.

113.1. Detailed Description

The QMainWindow class provides a typical application window, with a menu bar, some tool bars and a status bar.

In addition, you need the large central widget, which you supply and tell QMainWindow about using setCentralWidget(), and perhaps a few tool bars, which you can add using addToolBar().

The central widget is not touched by QMainWindow. QMainWindow manages its geometry, and that is all. For example, the application/application.cpp exmple (an editor) sets a QMultiLineEdit to be the central widget.

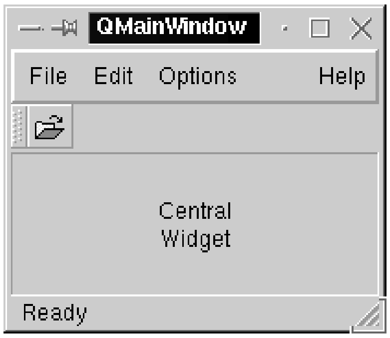
QMainWindow automatically detects the creation of a menu bar or status bar if you specify the QMainWindow as parent, or you can use the provided menuBar() and statusBar() functions. menuBar() and statusBar() create a suitable widget if one doesn't exist, and updates the window's layout to make space.

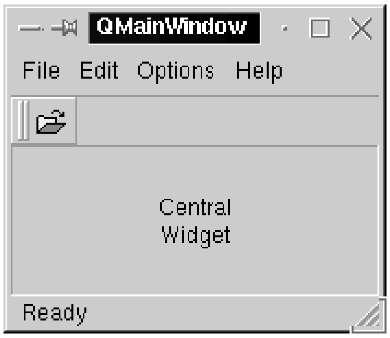
QMainWindow also provides a QToolTipGroup connected to the status bar. toolTipGroup() provides access to the QToolTipGroup, but there is no way to set the tool tip group.

By default, QMainWindow only allows toolbars above the central widget. You can use setDockEnabled() to allow toolbars in other docks (a *dock* is a place where toolbars can stay). Currently, only `Top, Left, Right` and `Bottom` are meaningful.

Several functions let you change the appearance of a QMainWindow globally: setRightJustification() determines whether QMainWindow should ensure that the toolbars fill the available space, setUsesBigPixmaps() determines whether QToolButton (and other classes) should draw small or large pixmaps (see QIconSet for more about that).

The current release of QMainWindow does not provide draggable toolbars. This feature is planned for inclusion in one of the next releases.





See also: QToolBar, QStatusBar, QMenuBar, QToolTipGroup and QDialog.

Examples:
application/application.cpp

113.2. Member Function Documentation

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

Constructs an empty main window.

QMainWindow::˜QMainWindow ()

Destroys the object and frees any allocated resources.

void QMainWindow::addToolBar (QToolBar * toolBar, const char * label, ToolBarDock edge = Top, bool nl = FALSE)

Adds *toolbar* to this the *edge* window of this window.

QWidget * QMainWindow::centralWidget () const

Returns a pointer to the main child of this main widget. The main child is the big widget around which the tool bars are arranged.

See also: setCentralWidget().

bool QMainWindow::event (QEvent * e) `[virtual protected]`

Monitors events to ensure layout is updated.

Reimplemented from QObject.

bool QMainWindow::eventFilter (QObject * o, QEvent * e) `[virtual]`

Monitors events to ensure layout is updated.

Reimplemented from QObject.

bool QMainWindow::isDockEnabled (ToolBarDock dock) const

Returns TRUE if *dock* is enabled, or FALSE if it is not.

See also: setDockEnabled().

QMenuBar * QMainWindow::menuBar () const

Returns the menu bar for this window. If there isn't any, menuBar() creates an empty menu bar on the fly.

See also: statusBar().

Examples:
application/application.cpp

void QMainWindow::paintEvent (QPaintEvent *) `[virtual protected]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QWidget.

void QMainWindow::pixmapSizeChanged (bool) `[signal]`

This signal is called whenever the setUsesBigPixmaps() is called with a value which is different from the current setting. All relevant widgets must connect to this signal.

void QMainWindow::removeToolBar (QToolBar * toolBar)

Removes *toolBar* from this main window, if *toolBar* is non-null and known by this main window.

bool QMainWindow::rightJustification () const

Returns TRUE if this main windows right-justifies its toolbars, and FALSE if it uses a ragged right edge.

The default is to use a ragged right edge.

("Right edge" sometimes means "bottom edge".)

See also: setRightJustification().

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

Sets the central widget for this window to *w*. The centail widget is the one around which the toolbars etc. are arranged.

Examples:
application/application.cpp

void QMainWindow::setDockEnabled (ToolBarDock dock, bool enable)

Sets *dock* to be available if *enable* is TRUE, and not available if *enable* is FALSE.

The user can drag a toolbar to any enabled dock.

void QMainWindow::setRightJustification (bool enable) `[slot]`

Sets this main window to expand its toolbars to fill all available space if *enable* is TRUE, and to give the toolbars just the space they need if *enable* is FALSE.

The default is FALSE.

See also: rightJustification();.

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

Sets up the geometry management of this window. Called automatically when needed, so you should never need to call this.

void QMainWindow::setUsesBigPixmaps (bool enable) `[slot]`

Sets tool buttons in this main windows to use big pixmaps if *enable* is TRUE, and small pixmaps if *enable* is FALSE.

The default is FALSE.

Tool buttons and other interested widgets are responsible for reading the correct state on startup, and for connecting to this widget's pixmapSizeChanged() signal.

See also: QToolButton::setUsesBigPixmap().

void QMainWindow::show () `[virtual]`

Reimplemented for internal reasons; the API is not affected.

Examples:
application/application.cpp

Reimplemented from QWidget.

QStatusBar * QMainWindow::statusBar () const

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

See also: setMenuBar(), statusBar() and toolTipGroup().

Examples:
application/application.cpp

QToolTipGroup * QMainWindow::toolTipGroup () const

Returns the tool tip group for this window. If there isn't any, menuBar() creates an empty tool tip group on the fly.

See also: menuBar() and statusBar().

bool QMainWindow::usesBigPixmaps () const

Returns the state last set by setUsesBigPixmaps(). The initial state is FALSE.

See also: setUsesBigPixmaps();.

113.2.1. URL

Generated from http://www.troll.no/qt/qmainwindow.html.

114. QMenuBar Class Reference

```
#include <qmenubar.h>
```

Inherits QFrame and QMenuData.

114.1. Detailed Description

The QMenuBar class provides a horizontal menu bar.

It automatically sets its own geometry to the top of the parent widget and changes appropriately it when the parent widget is resized.

menu/menu.cpp is a typical example of QMenuBar and QPopupMenu use.

See also: QPopupMenu and GUI Design Handbook: Menu Bar

Examples:

grapher/grapher.cpp
xform/xform.cpp
menu/menu.cpp
progress/progress.cpp
layout/layout.cpp
showimg/showimg.cpp
widgets/widgets.cpp

114.2. Member Function Documentation

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

Creates 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 handy in such cases.

See also: highlighted() and QMenuData::insertItem().

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

Called from QFrame::paintEvent().

Reimplemented from QFrame.

bool QMenuBar::eventFilter (QObject * object, QEvent * event) `[virtual]`

This function is used to adjust the menu bar's geometry to the parent widget's. Note that this is *not* part of the public interface - the function is `public` only because QObject::eventFilter() is.

Reimplemented from QObject.

void QMenuBar::fontChange (const QFont &) `[virtual protected]`

For internal use only.

Reimplemented from QWidget.

int QMenuBar::heightForWidth (int max_width) const

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

void QMenuBar::hide () `[virtual]`

Reimplements QWidget::hide() in order to deselect any selected item.

Examples:
grapher/grapher.cpp

Reimplemented from QWidget.

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 handy in such cases.

See also: activated() and QMenuData::insertItem().

void QMenuBar::keyPressEvent (QKeyEvent * e) `[virtual protected]`

Handles key press events for the menu bar.

Reimplemented from QWidget.

void QMenuBar::leaveEvent (QEvent * e) `[virtual protected]`

Handles leave events for the menu bar.

Reimplemented from QWidget.

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.

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.

void QMenuBar::mouseMoveEvent (QMouseEvent * e) `[virtual protected]`

Handles mouse move events for the menu bar.

Reimplemented from QWidget.

void QMenuBar::mousePressEvent (QMouseEvent * e) `[virtual protected]`

Handles mouse press events for the menu bar.

Reimplemented from QWidget.

void QMenuBar::mouseReleaseEvent (QMouseEvent * e) `[virtual protected]`

Handles mouse release events for the menu bar.

Reimplemented from QWidget.

void QMenuBar::resizeEvent (QResizeEvent *) `[virtual protected]`

Handles resize events for the menu bar.

Reimplemented from QWidget.

QMenuBar::Separator QMenuBar::separator() const

Returns the currently set separator usage

void QMenuBar::setSeparator (Separator when)

When a menubar is used above an unframed widget, it may look better with a separating line when displayed with WindowsStyle.

This function sets the usage of such a separator to appear either QMenuBar::Never, or QMenuBar::InWindowsStyle.

The default is QMenuBar::Never.

Examples:
layout/layout.cpp
widgets/widgets.cpp

void QMenuBar::show () `[virtual]`

Reimplements QWidget::show() in order to set up the correct keyboard accelerators and raise itself to the top of the widget stack.

Examples:

grapher/grapher.cpp

Reimplemented from QWidget.

void QMenuBar::updateItem (int) `[virtual]`

For internal use only.

Reimplemented from QMenuData.

114.2.1. URL

Generated from http://www.troll.no/qt/qmenubar.html.

115. QMenuData Class Reference

`#include <qmenudata.h>`

Inherited by QMenuBar and QPopupMenu.

115.1. 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 a 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 can be accessed through identifiers.

See also: QAccel.

115.2. Member Function Documentation

QMenuData::QMenuData ()

Constructs an empty list.

QMenuData::~QMenuData () `[virtual]`

Removes all menu items and disconnects any signals that have been connected.

int 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(), QAccel and qkeycode.h.

void QMenuData::changeItem (const QPixmap & pixmap, const char * text, int id)

Changes the pixmap and text of the menu item *id.*

See also: pixmap().

void QMenuData::changeItem (const QPixmap & pixmap, int id)

Changes the pixmap of the menu item *id.*

See also: pixmap().

void QMenuData::changeItem (const char * text, int id)

Changes the text of the menu item *id.*

See also: text().

void QMenuData::clear ()

Removes all menu items.

See also: removeItem() and removeItemAt().

bool QMenuData::connectItem (int id, const QObject * receiver, const char * member)

Connects a menu item to a receiver and a slot or signal.

The receiver's slot/signal is activated when the menu item is activated.

uint QMenuData::count () const

Returns the number of items in the menu.

bool QMenuData::disconnectItem (int id, const QObject * receiver, const char * member)

Disconnects a receiver/member from a menu item.

All connections are removed when the menu data object is destroyed.

QMenuItem * QMenuData::findItem (int id) const

Returns a pointer to the menu item with identifier *id,* or 0 if there is no item with such an identifier.

See also: indexOf().

QMenuItem * QMenuData::findItem (int id, QMenuData ** parent) const

Returns a pointer to the menu item with identifier *id,* or 0 if there is no item with such an identifier, and changes *parent* to point to the parent of the return value.

See also: indexOf().

QMenuItem * QMenuData::findPopup (QPopupMenu * popup, int * index = 0) `[protected]`

Internal function that finds the menu item where *popup* is located, storing its index at *index* if *index* is not NULL.

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() and indexOf().

int QMenuData::indexOf (int id) const

Returns the index of the menu item with identifier *id,* or -1 if there is no item with such an identifier.

See also: idAt() and findItem().

int QMenuData::insertItem (const QPixmap & pixmap, QPopupMenu * popup, int id=-1, int index=-1)

Inserts a menu item with a pixmap and a sub menu. Returns the menu item identifier.

The *popup* must be deleted by the programmer. It is not deleted when this menu item is removed or when the menu is deleted. Note that all popups are automatically deleted when the application terminates. (The QApplication destructor destroys all remaining widgets.)

The menu item is assigned the identifier *id* or an automatically generated identifier. It works as follows: If *id* >= 0, this identifier is assigned. If *id* == -1 (default), the identifier is set equal to the menu item's real index (see below). If *id* is any other negative integer, for instance -2, a unique identifier (negative integer <= -2) is generated.

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

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

See also: removeItem(), changeItem(), setAccel() and connectItem().

int QMenuData::insertItem (const QPixmap & pixmap, const QObject * receiver, const char * member, int accel, int id, int index = -1)

Inserts a menu item with a pixmap, an accelerator key, an id and an optional index and connects it to an object/slot.

The menu item is assigned the identifier *id* or an automatically generated identifier. It works as follows: If *id* >= 0, this identifier is assigned. If *id* == -1 (default), the identifier is set equal to the menu item's real index (see below). If *id* is any other negative integer, for instance -2, a unique identifier (negative integer <= -2) is generated.

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 accelerators only work for QPopupMenu items that live in a menu bar. For stand-alone popup menus, use an independent QAccel object.

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

See also: removeItem(), changeItem(), setAccel(), connectItem(), QAccel and qkeycode.h.

int QMenuData::insertItem (const QPixmap & pixmap, const QObject * receiver, const char * member, int accel=0)

Inserts a menu item with a pixmap and an optional accelerator key, and connects it to an object/slot.

Returns a unique menu item identifier (negative integer <= -2).

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

See also: removeItem(), changeItem(), setAccel(), connectItem(), QAccel and qkeycode.h.

int QMenuData::insertItem (const QPixmap & pixmap, const char * text, QPopupMenu * popup, int id=-1, int index=-1)

Inserts a menu item with a pixmap, a text and a sub menu. Returns the menu item identifier.

The *popup* must be deleted by the programmer. It is not deleted when this menu item is removed or when the menu is deleted. Note that all popups are automatically deleted when the application terminates. (The QApplication destructor destroys all remaining widgets.)

The menu item is assigned the identifier *id* or an automatically generated identifier. It works as follows: If *id* >= 0, this identifier is assigned. If *id* == -1 (default), the identifier is set equal to the menu item's real index (see below). If *id* is any other negative integer, for instance -2, a unique identifier (negative integer <= -2) is generated.

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

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

See also: removeItem(), changeItem(), setAccel() and connectItem().

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

Inserts a menu item with a pixmap, a text, an accelerator key, an id and an optional index and connects it to an object/slot.

The menu item is assigned the identifier *id* or an automatically generated identifier. It works as follows: If *id* >= 0, this identifier is assigned. If *id* == -1 (default), the identifier is set equal to the menu item's real index (see below). If *id* is any other negative integer, for instance -2, a unique identifier (negative integer <= -2) is generated.

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 accelerators only work for QPopupMenu items that live in a menu bar. For stand-alone popup menus, use an independent QAccel object.

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

See also: removeItem(), changeItem(), setAccel(), connectItem(), QAccel and qkeycode.h.

Examples:

qtimage/qtimage.cpp
grapher/grapher.cpp
xform/xform.cpp
application/application.cpp
menu/menu.cpp
progress/progress.cpp
layout/layout.cpp
showimg/showimg.cpp
widgets/widgets.cpp

int QMenuData::insertItem (const QPixmap & pixmap, const char * text, const QObject * receiver, const char * member, int accel=0)

Inserts a menu item with a pixmap, a text and an optional accelerator key, and connects it to an object/slot. The pixmap will be displayed to the left of the text in the item.

Returns a unique menu item identifier (negative integer <= -2).

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

See also: removeItem(), changeItem(), setAccel(), connectItem(), QAccel and qkeycode.h.

int QMenuData::insertItem (const QPixmap & pixmap, const char * text, int id=-1, int index=-1)

Inserts a menu item with a pixmap and a text. Returns the menu item identifier.

The menu item is assigned the identifier *id* or an automatically generated identifier. It works as follows: If *id* >= 0, this identifier is assigned. If *id* == -1 (default), the identifier is set equal to the menu item's real index (see below). If *id* is any other negative integer, for instance -2, a unique identifier (negative integer <= -2) is generated.

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

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

See also: removeItem(), changeItem(), setAccel() and connectItem().

int QMenuData::insertItem (const QPixmap & pixmap, int id=-1, int index=-1)

Inserts a menu item with a pixmap. Returns the menu item identifier.

The menu item is assigned the identifier *id* or an automatically generated identifier. It works as follows: If *id* >= 0, this identifier is assigned. If *id* == -1 (default), the identifier is set equal to the menu item's real index (see below). If *id* is any other negative integer, for instance -2, a unique identifier (negative integer <= -2) is generated.

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

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

See also: removeItem(), changeItem(), setAccel() and connectItem().

int QMenuData::insertItem (const char * text, QPopupMenu * popup, int id=-1, int index=-1)

Inserts a menu item with a text and a sub menu. Returns the menu item identifier.

The *popup* must be deleted by the programmer. It is not deleted when this menu item is removed or when the menu is deleted. Note that all popups are automatically deleted when the application terminates. (The QApplication destructor destroys all remaining widgets.)

The menu item is assigned the identifier *id* or an automatically generated identifier. It works as follows: If *id* >= 0, this identifier is assigned. If *id* == -1 (default), the identifier is set equal to the menu item's real index (see below). If *id* is any other negative integer, for instance -2, a unique identifier (negative integer <= -2) is generated.

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

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

See also: removeItem(), changeItem(), setAccel() and connectItem().

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

Inserts a menu item with a text, an accelerator key, an id and an optional index and connects it to an object/slot.

The menu item is assigned the identifier *id* or an automatically generated identifier. It works as follows: If *id* >= 0, this identifier is assigned. If *id* == -1 (default), the identifier is set equal to the menu item's real index (see below). If *id* is any other negative integer, for instance -2, a unique identifier (negative integer <= -2) is generated.

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 accelerators only work for QPopupMenu items that live in a menu bar. For stand-alone popup menus, use an independent QAccel object.

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

See also: removeItem(), changeItem(), setAccel(), connectItem(), QAccel and qkeycode.h.

int QMenuData::insertItem (const char * text, const QObject * receiver, const char * member, int accel=0)

Inserts a menu item with a text and an optional accelerator key, and connects it to an object/slot.

Returns a unique menu item identifier (negative integer <= -2).

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 );
```

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

Note that accelerators only work for QPopupMenu items that live in a menu bar. For stand-alone popup menus, use an independent QAccel object.

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

See also: removeItem(), changeItem(), setAccel(), connectItem(), QAccel and qkeycode.h.

int QMenuData::insertItem (const char * text, int id=-1, int index=-1)

Inserts a menu item with a text. Returns the menu item identifier.

The menu item is assigned the identifier *id* or an automatically generated identifier. It works as follows: If *id* >= 0, this identifier is assigned. If *id* == -1 (default), the identifier is set equal to the menu item's real index (see below). If *id* is any other negative integer, for instance -2, a unique identifier (negative integer <= -2) is generated.

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

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

See also: removeItem(), changeItem(), setAccel() and connectItem().

void 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 menubar, a separator is spacing, so the rest of the items (just "Help", normally) are drawn right-justified. In a Windows menubar, all separator are ignored (to comply with the Windows style guide).

Examples:
grapher/grapher.cpp
application/application.cpp
menu/menu.cpp
progress/progress.cpp
widgets/widgets.cpp

bool QMenuData::isItemChecked (int id) const

Returns TRUE if the menu item has been checked, otherwise FALSE.

See also: setItemChecked().

bool QMenuData::isItemEnabled (int id) const

Returns TRUE if the item with identifier *id* is enabled or FALSE if it is disabled.

See also: setItemEnabled().

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() and text().

void QMenuData::removeItem (int id)

Removes the menu item which has the identifier *id.*

See also: removeItemAt() and clear().

void QMenuData::removeItemAt (int index)

Removes the menu item at position *index.*

See also: removeItem() and clear().

void QMenuData::setAccel (int key, int id)

Defines an accelerator key for the menu item *id.*

An accelerator key consists of a key code and a combination of the modifiers `SHIFT, CTRL` and `ALT` (OR'ed or added). The header file qkeycode.h contains a list of key codes.

Defining an accelerator key generates a text which is added to the menu item, for instance, `CTRL + Key_O` generates "Ctrl+O". The text is formatted differently for different platforms.

Note that accelerators only work for QPopupMenu items that live in a menu bar. For stand-alone popup menus, 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 );
fileMenu->insertItem( "Quit", 69 );         // add "Quit" item
fileMenu->setAccel( CTRL + ALT + Key_Delete, 69 );
mainMenu->insertItem( "File", fileMenu );   // add the file menu
```

You can also specify the accelerator in the insertItem() function.

See also: accel(), insertItem(), QAccel and qkeycode.h.

void QMenuData::setId (int index, int id)

Sets the menu identifier of the item at *index* to *id.*

If index is out of range the operation is ignored.

See also: idAt().

void QMenuData::setItemChecked (int id, bool check)

Checks the menu item with id *id* if *check* is TRUE, or unchecks it if *check* is FALSE, and calls setCheckable(TRUE) if necessary.

See also: isItemChecked().

Examples:
grapher/grapher.cpp

progress/progress.cpp

void QMenuData::setItemEnabled (int id, bool enable)

Enables the menu item with identifier *id* if *enable* is TRUE, or disables the item if *enable* is FALSE.

See also: isItemEnabled().

Examples:
menu/menu.cpp
progress/progress.cpp
widgets/widgets.cpp

const char * QMenuData::text (int id) const

Returns the text that has been set for menu item *id,* or 0 if no text has been set.

See also: changeItem() and pixmap().

void QMenuData::updateItem (int) `[virtual]`

Virtual function; notifies subclasses about an item that has been changed.

Reimplemented in QMenuBar.

115.2.1. URL

Generated from http://www.troll.no/qt/qmenudata.html.

116. QMessageBox Class Reference

`#include <qmessagebox.h>`

Inherits QDialog.

116.1. Detailed Description

Displays a brief message, an icon, and some buttons.

A message box is a modal dialog that displays an icon, a 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 errors or ask the user how to fix an error, or
- `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, sometimes Dismiss for Motif applications) for a simple messages, or two or even three buttons for questions.

Here 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 perhaps do:

```
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 recommends using the application name as window 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. Since 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, and if there are unsaved data the user probably should be asked what to do, for example like this:

```
switch( QMessageBox::information( this, "Application name here",
                                  "The document contains unsaved work\n"
                                  "Do you want to save it before exiting?",
                                  "&Save", "&Don't Save", "&Cancel",
                                  0,      // Enter == button 0
                                  2 ) ) { // Escape == button 2
case 0: // Save clicked, Alt-S or Enter pressed.
    // save
    break;
case 1: // Don't Save clicked or Alt-D pressed
    // don't save but exit
    break;
case 2: // Cancel clicked, Alt-C or Escape pressed
    // don't exit
    break;
}
```

Again, the application name is used as window caption, as Microsoft recommends. The Escape button cancels the entire Exit operation, and Enter/Return saves the document and exits.

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 here",
                              "Could not connect to the <mumble> server.\n"
                              "This program can't function correctly "
                              "without the server.\n\n",
                              "Try again", "Quit", 0,
                              0, 1 );
case 0: // Try again or Enter
    // try again
    break;
case 1: // Quit or Escape
    // exit
    break;
}
```

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 hardcoded button texts:

```
switch( QMessageBox::warning( this, "Application name here",
                              "Could not save the 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 or Enter
    // try again
    break;
case QMessageBox::Abort: // Abort or Cancel
    // 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 occured. Please call ") +
                       "technical support at 123456789 and report these\n"+
                       "numbers:\n\n" + errorDetails +
                       "\n\n<Application> will now exit." );
```

QMessageBox provides a very simple About box, which displays an appropriate icon and the string you give it:

```
 QMessageBox::about( this, "About <Application>",
                     "<Application> is a <one-paragraph blurb>\n\n"
                     "Copyright 1951-1997 Such-and-such.  "
                     "<License words here.>\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 make a QMessageBox from scratch and set custom button texts:

```
QMessageBox mb( "Application name here",
                "Saving the file will overwrite the old file on 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, "Don't Save" );
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, by the static member functions information(), warning() and critical(), and there is a function called standardIcon() which 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 or Return be equivalent with clicking this button. Normally used with Ok, Yes or similar.
- `Escape` - makes pressing Escape be equivalent with this button. Normally used with Abort, Cancel or similar.

The text(), icon() and iconPixmap() functions provide access to the current text and pixmap of a message box, and setText(), setIcon() and setIconPixmap() lets you change it. The difference between setIcon() and setIconPixmap() is that the former accepts a QMessageBox::Icon and can it be used to set standard icons while 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, Isys on error messages, and GUI Design Handbook: Message Box.

Examples:
showimg/showimg.cpp

116.2. Member Function Documentation

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

Constructs a message box with no text and a button with the text "OK".

If *parent* is 0, then 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 char * caption, const char * text, Icon icon, int button0, int button1, int button2, QWidget * parent=0, const char * name=0, bool modal=TRUE, WFlags f=0)

Constructs a message box with a *caption*, a *text*, an *icon* and up to three buttons.

The *icon* must be one of:

- `QMessageBox::NoIcon`
- `QMessageBox::Information`
- `QMessageBox::Warning`
- `QMessageBox::Critical`

Each button can have one of the following values:

- `QMessageBox::Ok`
- `QMessageBox::Cancel`
- `QMessageBox::Yes`
- `QMessageBox::No`
- `QMessageBox::Abort`
- `QMessageBox::Retry`
- `QMessageBox::Ignore`

One of the buttons can be combined with the `QMessageBox::Default` flag to make a default button.

One of the buttons can be combined with the `QMessageBox::Escape` flag to make an escape option. Hitting the Esc key on the keyboard has the same effect as clicking this button with the mouse.

Example:

```
QMessageBox mb( "Hardware failure",
                "Disk 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, then 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: setCaption(), setText() and setIcon().

QMessageBox::˜QMessageBox ()

Destroys the message box.

void QMessageBox::about (QWidget * parent, const char * caption, const char * text) `[static]`

Displays a simple about box with window caption *caption* and body text *text.*

about() looks for a suitable icon for the box 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 too 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() and QApplication::mainWidget().

Examples:
menu/menu.cpp

void QMessageBox::aboutQt (QWidget * parent, const char * caption=0) `[static]`

Displays a simple message box about Qt, with window caption *caption* and optionally centered over *parent.*

This is neat for inclusion into the Help menu. See the menu.cpp example.

Examples:
trivial/trivial.cpp
menu/menu.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.

const char * QMessageBox::buttonText (int button) const

Returns the text of the message box button *button,* or null if the message box does not contain the button.

Example:

```
QMessageBox mb( QMessageBox::Ok, QMessageBox::Cancel, 0 );
mb.buttonText( QMessageBox::Cancel );  // returns "Cancel"
mb.buttonText( QMessageBox::Ignore );  // returns 0
```

See also: setButtonText().

int QMessageBox::critical (QWidget * parent, const char * caption, const char * text, const char * button0Text = "OK", const char * button1Text = 0, const char * button2Text = 0, int defaultButtonNumber = 0, int escapeButtonNumber = -1) `[static]`

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 must be present, *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 be equivalent with clicking the relevant button.

If *parent* is 0, then 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() and warning().

int QMessageBox::critical (QWidget * parent, const char * caption, const char * text, int button0, int button1, int button2=0) `[static]`

Opens a critical message box with a caption, a text and up to three buttons. Returns the identifier of the button that was clicked.

If *parent* is 0, then 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() and warning().

QMessageBox::Icon QMessageBox::icon() const

Returns the icon of the message box.

The return value is one of the following:

- `QMessageBox::NoIcon`
- `QMessageBox::Information`
- `QMessageBox::Warning`
- `QMessageBox::Critical`

See also: setIcon() and iconPixmap().

const QPixmap * QMessageBox::iconPixmap () const

Returns the icon pixmap of the message box.

Example:

```
QMessageBox mb(...);
mb.setIcon( QMessageBox::Warning );
mb.iconPixmap();    // returns the warning icon pixmap
```

See also: setIconPixmap() and icon().

int QMessageBox::information (QWidget * parent, const char * caption, const char * text, const char * button0Text = "OK", const char * button1Text = 0, const char * button2Text = 0, int defaultButtonNumber = 0, int escapeButtonNumber = -1) `[static]`

Displays an information 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 must be present, *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 be equivalent with clicking the relevant button.

If *parent* is 0, then 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() and critical().

int QMessageBox::information (QWidget * parent, const char * caption, const char * text, int button0, int button1=0, int button2=0) `[static]`

Opens an information message box with a caption, a text and up to three buttons. Returns the identifier of the button that was clicked.

If *parent* is 0, then 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() and critical().

Examples:
picture/picture.cpp
widgets/widgets.cpp

void QMessageBox::keyPressEvent (QKeyEvent * e) `[virtual protected]`

Handles key press events for the message box.

Reimplemented from QWidget.

void QMessageBox::resizeEvent (QResizeEvent *) `[virtual protected]`

Handles resize events for the message box.

Reimplemented from QWidget.

void QMessageBox::setButtonText (int button, const char * 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 quietly ignored.

Example:

```
QMessageBox mb( QMessageBox::Ok, QMessageBox::Cancel, 0 );
mb.setButtonText( QMessageBox::Ok, "All Right" );
mb.setButtonText( QMessageBox::Yes, "Yo" );   // ignored
```

See also: buttonText().

void QMessageBox::setIcon (Icon icon)

Sets the icon of the message box to *icon,* which is a predefined icon:

- `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 setIconPixmap() function.

See also: icon(), setIconPixmap() and iconPixmap().

void QMessageBox::setIconPixmap (const QPixmap & pixmap)

Sets the icon of the message box to a custom *pixmap.* Note that it's often hard to draw one pixmap which looks appropriate in both Motif and Windoes GUI styles. You may want to draw two.

See also: iconPixmap() and setIcon().

void QMessageBox::setStyle (GUIStyle s) `[virtual]`

Reimplemented for implementational reasons.

Reimplemented from QWidget.

void QMessageBox::setText (const char * text)

Sets the message box text to be displayed.

See also: text().

QPixmap QMessageBox::standardIcon (Icon icon, GUIStyle style) `[static]`

Returns the pixmap used for a standard icon. This allows the pixmaps to be used in more complex message boxes.

const char * QMessageBox::text () const

Returns the message box text currently set, or null if no text has been set.

See also: setText().

int QMessageBox::warning (QWidget * parent, const char * caption, const char * text, const char * button0Text = "OK", const char * button1Text = 0, const char * button2Text = 0, int defaultButtonNumber = 0, int escapeButtonNumber = -1) `[static]`

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 must be present, *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 be equivalent with clicking the relevant button.

If *parent* is 0, then 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() and critical().

int QMessageBox::warning (QWidget * parent, const char * caption, const char * text, int button0, int button1, int button2=0) `[static]`

Opens a warning message box with a caption, a text and up to three buttons. Returns the identifier of the button that was clicked.

If *parent* is 0, then 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() and critical().

Examples:
showimg/showimg.cpp

116.2.1. URL

Generated from http://www.troll.no/qt/qmessagebox.html.

117. QMouseEvent Class Reference

`#include <qevent.h>`

Inherits QEvent.

117.1. Detailed Description

The QMouseEvent class contains parameters that describe a mouse event.

Mouse events occur when a mouse button is pressed or released inside a widget, or when the mouse cursor is moved.

Mouse move events will only occur when some mouse button is pressed down, unless mouse tracking has been enabled.

Qt makes an automatic mouse grab when a mouse button is pressed inside a widget, and the widget will continue to receive mouse events until the last mouse button is released.

The QWidget::setEnable() function can be used to enable or disable mouse and keyboard events for a widget.

The QCursor widget has static functions for reading and setting the position of the mouse cursor.

The event handlers QWidget::mousePressEvent(), QWidget::mouseReleaseEvent(), QWidget::mouseDoubleClickEvent() and QWidget::mouseMoveEvent() receive mouse events.

See also: QWidget::setMouseTracking() and QWidget::grabMouse().

Examples:

qtimage/qtimage.cpp
table/table.cpp
life/life.cpp
dclock/dclock.cpp
connect/connect.cpp
trivial/trivial.cpp
tooltip/tooltip.cpp
hello/hello.cpp
movies/main.cpp
qmag/qmag.cpp
showimg/showimg.cpp
biff/biff.cpp
widgets/widgets.cpp

117.2. Member Function Documentation

QMouseEvent::QMouseEvent (int type, const QPoint & pos, int button, int state)

Constructs a mouse event object.

The type parameter must be `Event_MouseButtonPress, Event_MouseButtonRelease, Event_MouseButtonDblClick` or `Event_MouseMove.`

int QMouseEvent::button () const

Returns the button that caused the event.

Possible return values are `LeftButton, RightButton, MidButton` and `NoButton.`

Note that the returned value is always `NoButton` (0) for mouse move events.

See also: state().

Examples:
dclock/dclock.cpp
movies/main.cpp
widgets/widgets.cpp

const QPoint & QMouseEvent::globalPos () const

Returns the global position of the mouse pointer at the time of the event

See also: globalX() and globalY().

int QMouseEvent::globalX () const

Returns the global X position of the mouse pointer at the time of the event

See also: globalY() and globalPos().

int QMouseEvent::globalY () const

Returns the global Y position of the mouse pointer at the time of the event

See also: globalX() and globalPos().

const QPoint & QMouseEvent::pos () const

Returns the position of the mouse pointer, relative to the widget that received the event.

See also: x() and y().

Examples:
qtimage/qtimage.cpp
connect/connect.cpp
tooltip/tooltip.cpp
hello/hello.cpp
qmag/qmag.cpp

int QMouseEvent::state () const

Returns the button state (a combination of mouse buttons and keyboard modifiers), i.e. what buttons and keys were being held depressed immediately before the event was generated.

Note that this means that for `Event_MouseButtonPress` and `Event_MouseButtonDblClick`, the flag for the button() itself will not be set in the state; while for `Event_MouseButtonRelease`, it will.

The returned value is `LeftButton`, `RightButton`, `MidButton`, `ShiftButton`, `ControlButton` and `AltButton` OR'ed together.

See also: button().

Examples:
movies/main.cpp
widgets/widgets.cpp

int QMouseEvent::x () const

Returns the X position of the mouse pointer, relative to the widget that received the event.

See also: y() and pos().

int QMouseEvent::y () const

Returns the Y position of the mouse pointer, relative to the widget that received the event.

See also: x() and pos().

117.2.1. URL

Generated from http://www.troll.no/qt/qmouseevent.html.

118. QMoveEvent Class Reference

`#include <qevent.h>`

Inherits QEvent.

118.1. Detailed Description

The QMoveEvent class contains event parameters for move events.

Move events are sent to widgets that have been moved to a new position relative to their parent.

The event handler QWidget::moveEvent() receives move events.

See also: QWidget::move() and QWidget::setGeometry().

118.2. Member Function Documentation

QMoveEvent::QMoveEvent (const QPoint & pos, const QPoint & oldPos)

Constructs a move event with the new and old widget positions.

const QPoint & QMoveEvent::oldPos () const

Returns the old position of the widget.

const QPoint & QMoveEvent::pos () const

Returns the new position of the widget, which is the same as QWidget::pos().

118.2.1. URL

Generated from http://www.troll.no/qt/qmoveevent.html.

119. QMovie Class Reference

```
#include <qmovie.h>
```

119.1. Detailed Description

Incrementally loads an animation or image, signalling as it progresses.

A QMovie provides a QPixmap as the framePixmap(), and connections can be made via connectResize() and connectUpdate() to receive notification of size and pixmap changes. All decoding is driven by the normal event processing mechanisms. The simplest way to display a QMovie, is to use a QLabel and QLabel::setMovie().

The movie begins playing as soon as the QMovie is created (actually, once control returns to the event loop). When the last frame in the movie has been played, it may loop back to the start if such looping is defined in the input source.

QMovie objects are explicitly shared. This means that a QMovie copied from another QMovie will be displaying the same frame at all times. If one shared movie pauses, all pause. To make *independent* movies, they must be constructed separately.

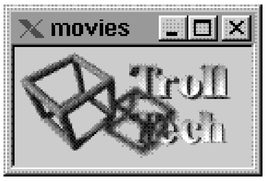
The set of data formats supported by QMovie is determined by the decoder factories which have been installed, and the format of the input is determined as the input is decoded.

In Qt 1.30, the decoder factory interface is not available for adding support for new formats. Only GIF support is installed. The GIF decoder supports interlaced images, transparency, looping, image-restore disposal, local color maps, and background colors. The Netscape looping extension is obeyed.

Archives of animated GIFs and tools for building them can be found at Yahoo!.

We are required to state: The Graphics Interchange Format(c) is the Copyright property of CompuServe Incorporated. GIF(sm) is a Service Mark property of CompuServe Incorporated.

Warning: Unisys has changed its position regarding GIF. If you are in a country where Unisys holds a patent on LZW compression and/or decompression, Unisys may require a license from you. These countries include Canada, Japan, the USA, France, Germany, Italy and the UK. There is more information on Unisys web site: Overview of Unisys' position. GIF support may be removed in a future version of Qt. We recommend using the PNG format, which is available in the Qt Image IO Extension package.



See also: QLabel::setMovie().

Examples:
- movies/main.cpp
- widgets/widgets.cpp

119.2. Member Function Documentation

QMovie::QMovie ()

Creates a null QMovie. The only interesting thing to do to such a movie is to assign another movie to it.

See also: isNull().

QMovie::QMovie (QDataSource * src, int bufsize=1024)

Creates a QMovie which reads an image sequence from the given QDataSource. The source must be allocated dynamically, as it becomes owned by the QMovie, and will be destroyed when the movie is destroyed. The movie starts playing as soon as event processing continues.

The *bufsize* argument sets the maximum amount of data the movie will transfer from the data source per event loop. The lower this value, the better interleaved the movie playback will be with other event processing, but the slower the overall processing.

QMovie::QMovie (const QMovie & movie)

Constructs a movie that uses the same data as another movie. QMovies use explicit sharing, so operations on the copy will effect the same operations on the original.

QMovie::QMovie (QByteArray data, int bufsize=1024)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Creates a QMovie which reads an image sequence from given data.

QMovie::QMovie (const char * fileName, int bufsize=1024)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Creates a QMovie which reads an image sequence from the named file.

QMovie::~QMovie ()

Destroys the QMovie. If this is the last reference to the data of the movie, that will also be destroyed.

const QColor& QMovie::backgroundColor () const

Returns the background color of the movie set by setBackgroundColor().

void QMovie::connectResize (QObject * receiver, const char * member)

Connects the given member, of type

```
void member(const QSize&)
```

such that it is signalled when the movie changes size.

Note that due to the explicit sharing of QMovie objects, these connections persist until they are explicitly disconnected with disconnectResize(), or until *every* shared copy of the movie is deleted.

void QMovie::connectStatus (QObject * receiver, const char * member)

Connects the given member, of type

```
void member(int)
```

such that it is signalled when the movie changes status. The status code are negative for errors and positive for information, and they are currently:

- `QMovie::SourceEmpty` - signalled if the input cannot be read.
- `QMovie::UnrecognizedFormat` - signalled if the input data is unrecognized.
- `QMovie::Paused` - signalled when the movie is paused by a call to paused(), or by after stepping pauses.
- `QMovie::EndOfFrame` - signalled at end-of-frame, after any update and Paused signals.
- `QMovie::EndOfLoop` - signalled at end-of-loop, after any update signals, EndOfFrame, but before EndOfMovie.
- `QMovie::EndOfMovie` - signalled when the movie completes and is not about to loop.

More status messages may be added in the future, so a general test for error would test for negative.

Note that due to the explicit sharing of QMovie objects, these connections persist until they are explicitly disconnected with disconnectStatus(), or until *every* shared copy of the movie is deleted.

Examples:
widgets/widgets.cpp

void QMovie::connectUpdate (QObject * receiver, const char * member)

Connects the given member, of type

```
void member(const QRect&)
```

such that it is signalled when an area of the framePixmap() has changed since the previous frame.

Note that due to the explicit sharing of QMovie objects, these connections persist until they are explicitly disconnected with disconnectUpdate(), or until *every* shared copy of the movie is deleted.

Examples:
widgets/widgets.cpp

void QMovie::disconnectResize (QObject * receiver, const char * member)

Disconnects the given member, or all members if member is zero, previously connected by connectResize().

void QMovie::disconnectStatus (QObject * receiver, const char * member=0)

Disconnects the given member, or all members if member is zero, previously connected by connectStatus().

void QMovie::disconnectUpdate (QObject * receiver, const char * member)

Disconnects the given member, or all members if member is zero, previously connected by connectUpdate().

bool QMovie::finished () const

Returns TRUE if the image is no longer playing - this happens when all loops of all frames is complete.

int QMovie::frameNumber () const

Returns the number of times EndOfFrame has been emitted since the start of the current loop of the movie. Thus, before any EndOfFrame has been emitted, the value will be 0, within slots processing the first signal, frameNumber() will be 1, and so on.

const QPixmap& QMovie::framePixmap () const

Returns the current frame of the movie. It is not generally useful to keep a copy of this pixmap. Better to keep a copy of the QMovie and get the framePixmap() only when needed for drawing.

Examples:
widgets/widgets.cpp

const QRect& QMovie::getValidRect () const

Returns the area of the pixmap for which pixels have been generated.

bool QMovie::isNull () const

Returns TRUE if the movie is null.

QMovie& QMovie::operator= (const QMovie & movie)

Makes this movie use the same data as another movie. QMovies use explicit sharing.

void QMovie::pause ()

Pauses the progress of the animation.

See also: unpause().

bool QMovie::paused () const

Returns TRUE if the image is paused.

void QMovie::restart ()

Rewinds the movie to the beginning. If the movie has not been paused, it begins playing again.

bool QMovie::running () const

Returns TRUE if the image is not single-stepping, not paused, and not finished.

void QMovie::setBackgroundColor (const QColor & c)

Set the background color of the pixmap. If the background color isValid(), the pixmap will never have a mask, as the background color will be used in transparent regions of the image.

See also: backgroundColor().

void QMovie::setSpeed (int percent)

Sets the speed-up factor of the movie. This is a percentage of the speed dictated by the input data format. The default is 100 percent.

Examples:
widgets/widgets.cpp

int QMovie::speed () const

Returns the speed-up factor of the movie. The default is 100 percent.

See also: setSpeed().

void QMovie::step ()

Steps forward 1 frame, then pauses.

void QMovie::step (int steps)

Steps forward, showing the given number of frames, then pauses.

int QMovie::steps () const

Returns the number of steps remaining after a call to step(), 0 if paused, or a negative value if the movie is running normally or is finished.

void QMovie::unpause ()

Unpauses the progress of the animation.

See also: pause().

119.2.1. URL

Generated from http://www.troll.no/qt/qmovie.html.

120. QMultiLineEdit Class Reference

#include <qmultilinedit.h>

Inherits QTableView.

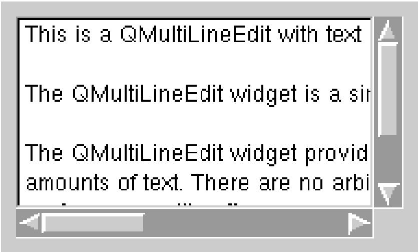
120.1. Detailed Description

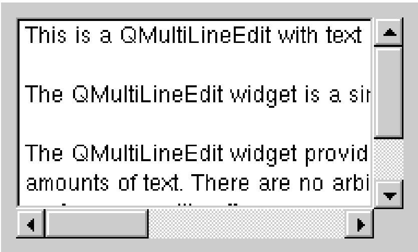
The QMultiLineEdit widget is a simple editor for inputting text.

The QMultiLineEdit widget provides multiple line text input and display. It is intended for moderate amounts of text. There are no arbitrary limitations, but if you try to handle megabytes of data, performance will suffer.

This widget can be used to display text by calling setReadOnly(TRUE)

The default key bindings are described in keyPressEvent(); they cannot be customized except by inheriting the class.





Examples:

mainlyMotif/editor.cpp
mainlyQt/editor.cpp
application/application.cpp
pref/pref.cpp
layout/layout.cpp
mainlyXt/editor.cpp
widgets/widgets.cpp

120.2. Member Function Documentation

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

Creates a new, empty, QMultiLineEdit.

QMultiLineEdit::~QMultiLineEdit ()

Destroys the QMultiLineEdit.

void QMultiLineEdit::append (const char * s) `[slot]`

Appends *s* to the text.

bool QMultiLineEdit::atBeginning () const

Returns TRUE if the cursor is placed at the beginning of the text.

bool QMultiLineEdit::atEnd () const

Returns TRUE if the cursor is placed at the end of the text.

bool QMultiLineEdit::autoUpdate () const

Returns TRUE if the view updates itself automatically whenever it is changed in some way.

See also: setAutoUpdate().

void QMultiLineEdit::backspace () `[virtual protected]`

Deletes the character on the left side of the text cursor and moves the cursor one position to the left. If a text has been marked by the user (e.g. by clicking and dragging) the cursor is put at the beginning of the marked text and the marked text is removed.

See also: del().

void QMultiLineEdit::clear () `[slot]`

Removes all text.

void QMultiLineEdit::copyText () `[slot]`

Copies the marked text to the clipboard.

void QMultiLineEdit::cursorDown (bool mark=FALSE) `[virtual protected]`

Moves the cursor one line down. If *mark* is TRUE, the text is marked.

See also: cursorDown(), cursorLeft() and cursorRight().

void QMultiLineEdit::cursorLeft (bool mark=FALSE, bool wrap = TRUE) `[virtual protected]`

Moves the cursor one character to the left. If *mark* is TRUE, the text is marked. If *wrap* is TRUE, the cursor moves to the end of the previous line if it is placed at the beginning of the current line.

See also: cursorRight(), cursorUp() and cursorDown().

QPoint QMultiLineEdit::cursorPoint () const `[protected]`

Returns the top center point where the cursor is drawn.

void QMultiLineEdit::cursorPosition (int * line, int * col) const

Sets *line* to the current line and *col* to the current character position within that line.

See also: setCursorPosition().

void QMultiLineEdit::cursorRight (bool mark=FALSE, bool wrap = TRUE) `[virtual protected]`

Moves the cursor one character to the right. If *mark* is TRUE, the text is marked. If *wrap* is TRUE, the cursor moves to the beginning of the next line if it is placed at the end of the current line.

See also: cursorLeft(), cursorUp() and cursorDown().

void QMultiLineEdit::cursorUp (bool mark=FALSE) `[virtual protected]`

Moves the cursor up one line. If *mark* is TRUE, the text is marked.

See also: cursorDown(), cursorLeft() and cursorRight().

void QMultiLineEdit::cut () `[slot]`

Copies the selected text to the clipboard and deletes the selected text.

void QMultiLineEdit::del () `[virtual protected]`

Deletes the character on the right side of the text cursor. If a text has been marked by the user (e.g. by clicking and dragging) the cursor is put at the beginning of the marked text and the marked text is removed.

See also: backspace().

void QMultiLineEdit::deselect () `[slot]`

Deselects all text (i.e. removes marking) and leaves the cursor at the current position.

void QMultiLineEdit::end (bool mark=FALSE) `[virtual protected]`

Moves the text cursor to the right end of the line. If mark is TRUE text is marked towards the last position. If it is FALSE and the cursor is moved, all marked text is unmarked.

See also: home().

void QMultiLineEdit::focusInEvent (QFocusEvent *) `[virtual protected]`

Starts the cursor blinking.

Reimplemented from QWidget.

void QMultiLineEdit::focusOutEvent (QFocusEvent *) `[virtual protected]`

stops the cursor blinking.

Reimplemented from QWidget.

void QMultiLineEdit::getCursorPosition (int * line, int * col)

Sets *line* to the current line and *col* to the current character position within that line.

OBSOLETE - Use cursorPosition() instead.

bool QMultiLineEdit::getMarkedRegion (int * line1, int * col1, int * line2, int * col2) const `[protected]`

If there is marked text, sets *line1, col1, line2* and *col2* to the start and end of the marked region and returns TRUE. Returns FALSE if there is no marked text.

QString * QMultiLineEdit::getString (int line) const `[protected]`

Returns a pointer to the text at line *line.*

bool QMultiLineEdit::hasMarkedText () const `[protected]`

Returns TRUE if there is marked text.

void QMultiLineEdit::home (bool mark=FALSE) `[virtual protected]`

Moves the text cursor to the left end of the line. If *mark* is TRUE, text is marked towards the first position. If it is FALSE and the cursor is moved, all marked text is unmarked.

See also: end().

void QMultiLineEdit::insertAt (const char * txt, int line, int col) `[virtual]`

Inserts *txt* at line number *line,* after character number *col* in the line. If *txt* contains newline characters, new lines are inserted.

The cursor position is adjusted. If the insertion position is equal to the cursor poition, the cursor is placed after the end of the new text.

void QMultiLineEdit::insertChar (char c) `[virtual protected]`

Inserts *c* at the current cursor position.

void QMultiLineEdit::insertLine (const char * txt, int line = -1) `[virtual]`

Inserts *txt* at line number *line.* If *line* is less than zero, or larger than the number of rows, the new text is put at the end. If *txt* contains newline characters, several lines are inserted.

The cursor position is not changed.

bool QMultiLineEdit::isOverwriteMode () const

Returns TRUE if this multi line edit is in overwrite mode, i.e. if characters typed replace characters in the editor.

See also: setOverwriteMode().

bool QMultiLineEdit::isReadOnly () const

Returns FALSE if this multi line edit accepts text input. Scrolling and cursor movements are accepted in any case.

See also: setReadOnly() and QWidget::isEnabled().

void QMultiLineEdit::keyPressEvent (QKeyEvent * e) `[virtual protected]`

The key press event handler converts a key press to some line editor action.

Here are the default key bindings when isReadOnly() is FALSE:

- *Left Arrow* Move the cursor one character leftwards
- *Right Arrow* Move the cursor one character rightwards
- *Up Arrow* Move the cursor one line upwards
- *Down Arrow* Move the cursor one line downwards
- *Page Up* Move the cursor one page upwards
- *Page Down* Move the cursor one page downwards
- *Backspace* Delete the character to the left of the cursor
- *Home* Move the cursor to the beginning of the line
- *End* Move the cursor to the end of the line
- *Delete* Delete the character to the right of the cursor
- *Shift - Left Arrow* Mark text one character leftwards
- *Shift - Right Arrow* Mark text one character rightwards
- *Control-A* Move the cursor to the beginning of the line
- *Control-B* Move the cursor one character leftwards
- *Control-C* Copy the marked text to the clipboard.
- *Control-D* Delete the character to the right of the cursor
- *Control-E* Move the cursor to the end of the line
- *Control-F* Move the cursor one character rightwards
- *Control-H* Delete the character to the left of the cursor
- *Control-K* Delete to end of line
- *Control-N* Move the cursor one line downwards
- *Control-P* Move the cursor one line upwards
- *Control-V* Paste the clipboard text into line edit.
- *Control-X* Cut the marked text, copy to clipboard. All other keys with valid ASCII codes insert themselves into the line.

Here are the default key bindings when isReadOnly() is TRUE:

- *Left Arrow* Scrolls the table rightwards
- *Right Arrow* Scrolls the table rightwards
- *Up Arrow* Scrolls the table one line downwards
- *Down Arrow* Scrolls the table one line upwards
- *Page Up* Scrolls the table one page downwards
- *Page Down* Scrolls the table one page upwards

Reimplemented from QWidget.

void QMultiLineEdit::killLine () `[virtual protected]`

Deletes text from the current cursor position to the end of the line.

void QMultiLineEdit::leaveEvent (QEvent *) `[virtual protected]`

Handles auto-copy of selection (X11 only).

Reimplemented from QWidget.

int QMultiLineEdit::lineLength (int line) const `[protected]`

Returns the number of characters at line number *line*.

QString QMultiLineEdit::markedText () const `[protected]`

Returns a copy of the marked text.

void QMultiLineEdit::mouseDoubleClickEvent (QMouseEvent * m) `[virtual protected]`

Handles double click events.

Reimplemented from QWidget.

void QMultiLineEdit::mouseMoveEvent (QMouseEvent * e) `[virtual protected]`

Handles mouse move events.

Reimplemented from QWidget.

void QMultiLineEdit::mousePressEvent (QMouseEvent * m) `[virtual protected]`

Handles mouse press events.

Reimplemented from QWidget.

void QMultiLineEdit::mouseReleaseEvent (QMouseEvent *) `[virtual protected]`

Handles mouse release events.

Reimplemented from QWidget.

void QMultiLineEdit::newLine () `[virtual protected]`

Makes a line break at the current cursor position.

int QMultiLineEdit::numLines () const

Returns the number of lines in the editor. The count includes any empty lines at top and bottom, so for an empty editor this method will return 1.

void QMultiLineEdit::pageDown (bool mark=FALSE) `[virtual protected]`

Moves the cursor one page down. If *mark* is TRUE, the text is marked.

void QMultiLineEdit::pageUp (bool mark=FALSE) `[virtual protected]`

Moves the cursor one page up. If *mark* is TRUE, the text is marked.

void QMultiLineEdit::paintCell (QPainter * painter, int row, int) `[virtual protected]`

Implements the basic drawing logic.

Reimplemented from QTableView.

void QMultiLineEdit::paste () `[slot]`

Copies text from the clipboard onto the current cursor position. Any marked text is first deleted.

void QMultiLineEdit::removeLine (int line) `[virtual]`

Deletes the line at line number *line.* If *line* is less than zero, or larger than the number of lines, no line is deleted.

void QMultiLineEdit::returnPressed () `[signal]`

This signal is emitted when the user presses the return or enter key. It is not emitted if isReadOnly() is TRUE.

See also: textChanged().

void QMultiLineEdit::selectAll () `[slot]`

Selects all text without moving the cursor.

void QMultiLineEdit::setAutoUpdate (bool enable)

Sets the auto-update option of multi-line editor to *enable.*

If *enable* is TRUE (this is the default) then the editor updates itself automatically whenever it has changed in some way (generally, when text has been inserted or deleted).

If *enable* is FALSE, the view does NOT repaint itself, or update its internal state variables itself when it is changed. This can be useful to avoid flicker during large changes, and is singularly useless otherwise: Disable auto-update, do the changes, re-enable auto-update, and call repaint().

Warning: Do not leave the view in this state for a long time (i.e. between events). If, for example, the user interacts with the view when auto-update is off, strange things can happen.

Setting auto-update to TRUE does not repaint the view, you must call repaint() to do this.

See also: autoUpdate() and repaint().

void QMultiLineEdit::setCursorPosition (int line, int col, bool mark = FALSE)

Sets the cursor position to character number *col* in line number *line.* The parameters are adjusted to lie within the legal range.

If *mark* is FALSE, the selection is cleared. otherwise it is extended

See also: cursorPosition().

void QMultiLineEdit::setFixedVisibleLines (int lines)

Sets the fixed height of the QMultiLineEdit so that *lines* text lines are visible given the current font.

See also: setFixedHeight().

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

Reimplements QWidget::setFont() to update the list box line height.

Reimplemented from QWidget.

void QMultiLineEdit::setOverwriteMode (bool on) `[slot]`

Sets overwrite mode if *on* is TRUE. Overwrite mode means that characters typed replace characters in the editor.

See also: isOverwriteMode().

void QMultiLineEdit::setReadOnly (bool on) `[slot]`

If *on* is FALSE, this multi line edit accepts text input. Scrolling and cursor movements are accepted in any case.

See also: inputEnabled() and QWidget::setEnabled().

void QMultiLineEdit::setText (const char * s) `[slot]`

Sets the text to *s,* removing old text, if any.

Examples:
mainlyMotif/editor.cpp
mainlyQt/editor.cpp
mainlyXt/editor.cpp
widgets/widgets.cpp

QString QMultiLineEdit::text () const

Returns a copy of the whole text. If the multi line edit contains no text, the empty string is returned.

void QMultiLineEdit::textChanged () `[signal]`

This signal is emitted when the text is changed by an event or by a slot. Note that the signal is not emitted when you call a non-slot function such as insertLine().

See also: returnPressed().

const char * QMultiLineEdit::textLine (int line) const

Returns the text at line number *line,* or 0 if *line* is invalid.

int QMultiLineEdit::textWidth (QString * s) `[protected]`

Returns the width in pixels of the string *s.* NOTE: only appropriate for whole lines.

int QMultiLineEdit::textWidth (int line) `[protected]`

Returns the width in pixels of the text at line *line.*

void QMultiLineEdit::timerEvent (QTimerEvent * t) `[virtual protected]`

Cursor blinking, drag scrolling.

Reimplemented from QObject.

120.2.1. URL

Generated from http://www.troll.no/qt/qmultilineedit.html.

121. QNPInstance Class Reference

`#include <qnp.h>`

Inherits QObject.

121.1. Detailed Description

This class is defined in the **Qt NSPlugin Extension**, which can be found in the `qt/extensions` directory. It is not included in the main Qt API.

a QObject that is a Web-browser plugin

Deriving from QNPInstance creates an object that represents a single <EMBED> tag in an HTML document.

The QNPInstance is responsible for creating an appropriate window if required (not all plugins have windows), and for interacting with the input/output facilities intrinsic to plugins.

Note that there is *absolutely no garrantee* as to the order in which functions are called. Sometimes the browser will call setWindow() first, at other times, newStreamCreated() will be called first (assuming the <EMBED> tag has a SRC parameter).

No GUI functionality of Qt may be used until the first call to setWindow(). This includes any use of QPaintDevice (ie. QPixmap, QWidget, and all subclasses), QApplication, anything related to QPainter (QBrush, etc.), fonts, QMovie, QToolTip, etc. Useful classes which specifically *can* be used are QImage, QFile, and QBuffer.

By structuring your plugin so that the task of the QNPInstance is to gather data, while the task of the QNPWidget is to provide a graphical interface to that data, this restriction can be easily accommodated.

Examples:
qtimage/qtimage.cpp
grapher/grapher.cpp
trivial/trivial.cpp

121.2. Member Function Documentation

QNPInstance::QNPInstance () `[protected]`

Creates a QNPInstance. Can only be called from within a derived class created within QNPlugin::newInstance().

QNPInstance::~QNPInstance ()

Called when the plugin instance is about to disappear.

const char* QNPInstance::arg (const char * name) const

Returns the value of the named arguments, or 0 if no argument with that name appears in the <EMBED> tag of this instance. If the argument appears, but has no value assigned, the empty string is returned. In summary:

- **<EMBED ...>** - arg("FOO") == 0
- **<EMBED FOO ...>** - arg("FOO") == ""
- **<EMBED FOO=BAR ...>** - arg("FOO") == "BAR"

int QNPInstance::argc () const

Returns the number of arguments to the instance. Note that you should not normally rely on the ordering of arguments, and also note that the SGML specification does not permit multiple arguments with the same name.

See also: arg().

const char* QNPInstance::argn (int i) const

Returns the name of the *i*th argument. See notes of argc().

const char* QNPInstance::argv (int i) const

Returns the value of the *i*th argument. See notes of argc().

void* QNPInstance::getJavaPeer () const

Returns the Java object associated with the plug-in instance, an object of the plug-in's Java class, or 0 if the plug-in does not have a Java class, Java is disabled, or an error occurred .

The return value is actually a `jref` we use `void*` so as to avoid burdening plugins which do not require Java.

See also: QNPlugin::getJavaClass(), QNPlugin::getJavaEnv(), getJavaPeer() and Netscape: NPN_GetJavaPeer

void QNPInstance::getURL (const char * url, const char * window=0)

Requests that the given URL be retrieved and sent to the named window. See Netscape's JavaScript documentation for an explanation of window names.

See also: Netscape: NPN_GetURL method

Examples:
grapher/grapher.cpp

QNPInstance::InstanceMode QNPInstance::mode() const

Returns the mode of the plugin.

QNPStream* QNPInstance::newStream (const char * mimetype, const char * window, bool as_file=FALSE)

Requests the creation of a new data stream *from* the plug-in. This function is not tested. It is an interface to the NPN_NewStream function of the Netscape Plugin API.

See also: Netscape: NPN_NewStream method

bool QNPInstance::newStreamCreated (QNPStream *, StreamMode & smode) `[virtual]`

This function is called when a new stream has been created. The instance should return TRUE if it accepts the processing of the stream. If the instance requires the stream as a file, it should set *smode* to AsFileOnly, in which case the data will be delivered some time later to the streamAsFile() function. Otherwise, the data will be delivered in chunks to the write() function which must consume at least as much data as was returned by the most recent call to writeReady().

Note that the AsFileOnly method is not supported by Netscape 2.0 and MS-Explorer 3.0. A future version of Qt will hide this problem.

See also: Netscape: NPP_NewStream method

QNPWidget* QNPInstance::newWindow () `[virtual]`

Called at most once, at some time after the QNPInstance is created. If the plugin requires a window, this function should return a derived class of QNPWidget that provides the required interface.

void QNPInstance::postURL (const char * url, const char * window, uint len, const char * buf, bool file)

This function is not tested. It is an interface to the NPN_PostURL function of the Netscape Plugin API.

See also: Netscape: NPN_PostURL method

void QNPInstance::print (QPainter *) `[virtual]`

Print the instance embedded in a page.

This function is not tested. It is an encapsulation of the NPP_Print function of the Netscape Plugin API.

See also: Netscape: NPP_Print method

bool QNPInstance::printFullPage () `[virtual]`

Print the instance full-page. By default, this returns FALSE, causing the browser to call the (embedded) print() function instead.

This function is not tested. It is an encapsulation of the NPP_Print function of the Netscape Plugin API.

See also: Netscape: NPP_Print method

void QNPInstance::status (const char * msg)

Sets the status message in the browser containing this instance.

See also: Netscape: NPN_Status method

void QNPInstance::streamAsFile (QNPStream *, const char *) `[virtual]`

Called when a stream is delivered as a single file rather than as chunks. This may be simpler for a plugin to deal with, but precludes any incremental behaviour.

See also: newStreamCreated(), newStream(), Note, that, the, AsFileOnly, method, is, not, supported, by, Netscape, 2.0, MS-Explorer, 3.0., See, also: and Netscape: NPP_StreamAsFile method

void QNPInstance::streamDestroyed (QNPStream *) `[virtual]`

Called when a stream is destroyed. At this this, the stream may be complete() and okay(). If it is not okay(), then an error has occurred. If it is okay(), but not complete(), then the user has cancelled the transmission - do not give an error message in this case.

See also: Netscape: NPP_DestroyStream method

const char* QNPInstance::userAgent () const

Returns the user agent (browser name) containing this instance.

See also: Netscape: NPN_UserAgent method

QNPWidget* QNPInstance::widget ()

Returns the plugin window created at newWindow(), if any.

Examples:
grapher/grapher.cpp

int QNPInstance::write (QNPStream *, int offset, int len, void * buffer) `[virtual]`

Called when incoming data is available for processing by the instance. The instance *must* consume at least the amount that it returned in the most recent call to writeReady(), but it may consume up to the amount given by *len. buffer* is the data available for consumption. The *offset* argument is merely an informational value indicating the total amount of data that has been consumed in prior calls.

This function should return the amount of data actually consumed.

See also: Netscape: NPP_NewStream method

int QNPInstance::writeReady (QNPStream *) `[virtual]`

Called to inquire the minimum amount of data the instance is willing to receive from the given stream.

The default returns a very large value.

See also: Netscape: NPP_WriteReady method

121.2.1. URL

Generated from http://www.troll.no/qt/qnpinstance.html.

122. QNPlugin Class Reference

`#include <qnp.h>`

122.1. Detailed Description

This class is defined in the **Qt NSPlugin Extension**, which can be found in the `qt/extensions` directory. It is not included in the main Qt API.

The plugin central factory.

This class is the heart of the plugin. One instance of this object is created when the plugin is *first* needed, by calling QNPlugin::create(), which must be implemented in your plugin code to return some derived class of QNPlugin. The one QNPlugin object creates all instances for a single running Web-browser process.

Additionally, if Qt is linked to the plugin as a dynamic library, only one instance of QApplication will exist *across all plugins that have been made with Qt.* So, your plugin should tread lightly on global settings - do not for example, use QApplication::setFont() - that will change the font in every widget of every Qt-based plugin currently loaded!.

Examples:

qtimage/qtimage.cpp
grapher/grapher.cpp
trivial/trivial.cpp

122.2. Member Function Documentation

QNPlugin::QNPlugin () `[protected]`

Creates a QNPlugin. This may only be used by the constructor derived class returned by plugin's implementation of the QNPlugin::create() function.

QNPlugin::~QNPlugin () `[virtual]`

Destroys the QNPlugin. This is called by the plugin binding code just before the plugin is about to be unloaded from memory. If newWindow() has been called, a QApplication will still exist at this time, but will be deleted shortly after before the plugin is deleted.

QNPlugin* QNPlugin::actual () `[static]`

Returns the plugin most recently returns by QNPlugin::create().

QNPlugin* QNPlugin::create () `[static]`

This must be implemented by your plugin code. It should return a derived class of QNPlugin.

void* QNPlugin::getJavaClass () `[virtual]`

Override to return a reference to the Java class that represents the plugin. The default returns 0, indicating no class.

If you override this class, you must also override QNPlugin::unuseJavaClass().

The return value is actually a `jref` we use `void*` so as to avoid burdening plugins which do not require Java.

See also: getJavaEnv(), QNPInstance::getJavaPeer() and Netscape: NPP_GetJavaClass

void* QNPlugin::getJavaEnv () const

Returns a pointer to the Java execution environment, or 0 if Java is disabled or an error occurred.

The return value is actually a `JRIEnv*` we use `void*` so as to avoid burdening plugins which do not require Java.

See also: getJavaClass(), QNPInstance::getJavaPeer() and Netscape: NPN_GetJavaEnv

const char* QNPlugin::getMIMEDescription () const `[virtual]`

Override this to return the MIME description of the data formats supported by your plugin. The format of this string is described by the following example:

```
const char* getMIMEDescription() const
{
    return "image/x-png:png:PNG Image;"
           "image/png:png:PNG Image;"
           "image/x-portable-bitmap:pbm:PBM Image;"
           "image/x-portable-graymap:pgm:PGM Image;"
           "image/x-portable-pixmap:ppm:PPM Image;"
           "image/bmp:bmp:BMP Image;"
           "image/x-ms-bmp:bmp:BMP Image;"
           "image/x-xpixmap:xpm:XPM Image;"
           "image/xpm:xpm:XPM Image";
}
```

const char* QNPlugin::getPluginDescriptionString () const `[virtual]`

Returns a plain-text description of the plugin.

const char* QNPlugin::getPluginNameString () const `[virtual]`

Returns the plain-text name of the plugin.

See also: Netscape: NPP_GetValue method

void QNPlugin::getVersionInfo (int & plugin_major, int & plugin_minor, int & browser_major, int & browser_minor)

Returns the version information - the version of the plugin API, and the version of the browser.

See also: Netscape: NPN_Version method

QNPInstance* QNPlugin::newInstance () `[virtual]`

Override this to return an appropriate derived class of QNPInstance.

void QNPlugin::unuseJavaClass () `[virtual]`

This function is called when the plugin is shutting down, with *jc* set to the value returned earlier by getJavaClass(). The function should *unuse* the Java class and return 0.

See also: Netscape: Associating a Class with your Plug-in

122.2.1. URL

Generated from http://www.troll.no/qt/qnplugin.html.

123. QNPStream Class Reference

```
#include <qnp.h>
```

123.1. Detailed Description

This class is defined in the **Qt NSPlugin Extension**, which can be found in the `qt/extensions` directory. It is not included in the main Qt API.

A stream of data provided to a QNPInstance by the browser.

Note that this is neither a QTextStream or a QDataStream.

See also: QNPInstance::write() and QNPInstance::newStreamCreated().

Examples:
qtimage/qtimage.cpp
grapher/grapher.cpp

123.2. Member Function Documentation

QNPStream::QNPStream (QNPInstance * in,const char * mt, _NPStream * st, bool se)

Creates a stream. Plugins should not call this, but rather QNPInstance::newStream() if a stream is required.

QNPStream::~QNPStream ()

Destroys the stream.

bool QNPStream::complete () const

Returns TRUE if the stream has received all the data from the source.

uint QNPStream::end () const

Returns the length of the stream (???).

QNPInstance* QNPStream::instance ()

Returns the QNPInstance for which this stream was created.

uint QNPStream::lastModified () const

Returns the time when the source of the stream was last modified.

bool QNPStream::okay () const

Returns TRUE if no errors have occurred on the stream.

void QNPStream::requestRead (int offset, uint length)

Requests the given section of the stream be sent to the QNPInstance::write() function of the instance() of this stream.

See also: Netscape: NPN_RequestRead method

bool QNPStream::seekable () const

Returns TRUE if the stream is seekable.

void QNPStream::setComplete (bool y)

For internal use only.

void QNPStream::setOkay (bool y)

For internal use only.

const char* QNPStream::type () const

Returns the MIME type of the stream.

const char* QNPStream::url () const

Returns the URL from which the stream was created.

int QNPStream::write (int len, void * buffer)

Writes data *to* the stream.

See also: Netscape: NPN_Write method

123.2.1. URL

Generated from http://www.troll.no/qt/qnpstream.html.

124. QNPWidget Class Reference

`#include <qnp.h>`

Inherits QWidget.

124.1. Detailed Description

This class is defined in the **Qt NSPlugin Extension**, which can be found in the `qt/extensions` directory. It is not included in the main Qt API.

A QWidget that is a Web-browser plugin window

Derive from QNPWidget to create a widget that can be used as a Browser plugin window, or create one and add child widgets. Instances of QNPWidget may only be created when QNPInstance::newWindow() is called by the browser.

A common way to develop a plugin widget is to develop it as a stand-alone application window, then make it a *child* of a plugin widget to use it as a browser plugin. The technique is:

```
class MyPluginWindow : public QNPWidget {
    QWidget* child;
public:
    MyPluginWindow()
    {
        // Some widget that is normally used as a top-level widget
        child = new MyIndependentlyDevelopedWidget();
        // Use the background color of the web page
        child->setBackgroundColor( backgroundColor() );
        // Fill the plugin widget
        child->setGeometry( 0, 0, width(), height() );
    }
    void resizeEvent(QResizeEvent*)
    {
        // Fill the plugin widget
        child->resize(size());
    }
};
```

The default implementation is an empty window.

Examples:
 qtimage/qtimage.cpp
 grapher/grapher.cpp
 trivial/trivial.cpp

124.2. Member Function Documentation

QNPWidget::QNPWidget ()

Creates a QNPWidget.

QNPWidget::~QNPWidget ()

Destroys the window. This will be called by the plugin binding code when the window is no longer required. The Web-browser will delete windows when they leave the page. The bindings will change the QWidget::winId() of the window when the window is resized, but this should not affect normal widget behaviour.

void QNPWidget::enterInstance () `[virtual]`

Called when the mouse enters the plugin window. Default does nothing.

QNPInstance* QNPWidget::instance ()

Returns the instance for which this widget is the window.

void QNPWidget::leaveInstance () `[virtual]`

Called when the mouse leaves the plugin window. Default does nothing.

void QNPWidget::setWindow (bool delold)

For internal use only.

void QNPWidget::unsetWindow ()

For internal use only.

124.2.1. URL

Generated from http://www.troll.no/qt/qnpwidget.html.

125. QObject Class Reference

`#include <qobject.h>`

Inherited by QAccel, QApplication, QClipboard, QDataPump, QDragManager, QDragObject, QFileIconProvider, QGManager, QLayout, QNPInstance, QSenderObject, QSignal, QSignalMapper, QSocketNotifier, QTimer, QToolTipGroup, QValidator and QWidget.

125.1. Detailed Description

The QObject class is the base class of all Qt objects that can deal with signals, slots and events.

Qt provides a very powerful mechanism for seamless object communication; signal/slot connections. The signal/slot mechanism is an advanced way of making traditional callback routines.

Example:

```
//
// The Mandelbrot class uses a QTimer to calculate the mandelbrot
// set one scanline at a time without blocking the CPU.
// It inherits QObject to use signals and slots.
// Calling start() starts the calculation. The done() signal is
// emitted when it has finished.
// Note that this example is not complete. Feel free to complete it.
//
class Mandelbrot : public QObject
{
    Q_OBJECT                                // required for signals/slots
public:
    Mandelbrot( QObject *parent=0, const char *name );
    ...
public slots:
    void    start();
signals:
    void    done();
private slots:
    void    calculate();
private:
    QTimer  timer;
    ...
};
//
// Constructs and initializes a Mandelbrot object.
//
Mandelbrot::Mandelbrot( QObject *parent=0, const char *name )
    : QObject( parent, name )
{
    connect( &timer, SIGNAL(timeout()), SLOT(calculate()) );
    ...
}
//
// Starts the calculation task. The internal calculate() slot
// will be activated every 10 milliseconds.
//
void Mandelbrot::start()
{
    if ( !timer.isActive() )                // not already running
        timer.start( 10 );                  // timeout every 10 ms
}
//
// Calculates one scanline at a time.
// Emits the done() signal when finished.
//
void Mandelbrot::calculate()
```

```
{
    ...                         // perform the calculation for a scanline
    if ( finished ) {           // no more scanlines
        timer.stop();
        emit done();
    }
}
```

When an object has changed in some way that might be interesting for the outside world, it emits a signal to tell whoever is listening. All slots that are connected to this signal will be activated (called). It is even possible to connect a signal directly to another signal. (This will emit the second signal immediately whenever the first is emitted.)

There is no limitation on how many slots that can be connected to a signal. The slots will be activated in the order they were connected to the signal.

Notice that the `Q_OBJECT` macro is mandatory for any object that implement signals or slots. You also need to run the moc program (Meta Object Compiler) on the source file.

The signal/slot mechanism allows objects to easily reused, because the object that emits a signal does not need to know what the signals are connected to.

All Qt widgets inherit QObject and use signals and slots. A QScrollBar, for example, emits valueChanged() whenever the scroll bar value changes.

Meta objects are useful for doing more than connecting signals to slots. They also allow the programmer to obtain information about the class to which an object is instantiated from (see isA() and inherits()) or to produce a list of child objects that inherit a particular class (see queryList()).

Examples:
  widgets/widgets.cpp

125.2. Member Function Documentation

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

Constructs an object with the parent object *parent* and a *name.*

The parent of an object may be viewed as the object's owner. For instance, a dialog box is the parent of the "ok" and "cancel" buttons inside it.

The destructor of a parent object destroys all child objects.

Setting *parent* to 0 constructs an object with no parent. If the object is a widget, it will become a top-level window.

The object name is a text that can be used to identify this QObject. It is not very useful in the current version of Qt, but it will become increasingly important in the future.

The queryList() function searches the object tree for objects that matches a particular object name.

See also: parent(), name() and queryList().

QObject::~QObject () `[virtual]`

Destroys the object and all its children objects.

All signals to and from the object are automatically disconnected.

Warning: *All* child objects are deleted. If any of these objects are on the stack or global, your program will sooner or later crash. We do not recommend holding pointers to child objects from outside the parent. If you still do, the QObject::destroyed() signal gives you an opportunity to detect when an object is destroyed.

bool QObject::activate_filters (QEvent * e) `[protected]`

For internal use only.

void QObject::activate_signal (const char * signal) `[protected]`

For internal use only.

void QObject::activate_signal (const char * signal, const char *) `[protected]`

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

void QObject::activate_signal (const char * signal, int) `[protected]`

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

void QObject::activate_signal (const char * signal, long) `[protected]`

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

void QObject::activate_signal (const char * signal, short) `[protected]`

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

void QObject::badSuperclassWarning (const char * className, const char * superclassName) `[static protected]`

Internal function, called from initMetaObject(). Used to emit a warning when a class containing the macro Q_OBJECT inherits from a class that does not contain it.

void QObject::blockSignals (bool block)

Blocks signals if *block* is TRUE, or unblocks signals if *block* is FALSE.

Emitted signals disappear into hyperspace if signals are blocked.

Examples:
xform/xform.cpp

bool QObject::checkConnectArgs (const char * signal, const QObject * receiver, const char * member) `[virtual protected]`

Returns TRUE if the *signal* and the *member* arguments are compatible, otherwise FALSE.

Warning: We recommend that you do not reimplement this function but use the default implementation.

const QObjectList * QObject::children () const

Returns a list of child objects, or 0 if this object has no children.

The QObjectList class is defined in the qobjcoll.h header file.

The latest child added is the first object in the list and the first child is added is the last object in the list.

Note that the list order might change when widget children are raised or lowered. A widget that is raised becomes the last object in the list. A widget that is lowered becomes the first object in the list.

See also: queryList(), parent(), insertChild() and removeChild().

const char * QObject::className () const `[virtual]`

Returns the class name of this object.

This function is generated by the Meta Object Compiler.

Warning: This function will return an invalid name if the class definition lacks the `Q_OBJECT` macro.

See also: name(), inherits(), isA() and isWidgetType().

Examples:
widgets/widgets.cpp

bool QObject::connect (const QObject * sender, const char * signal, const QObject * receiver, const char * member) `[static]`

Connects *signal* from the *sender* object to *member* in object *receiver*.

You must use the SIGNAL() and SLOT() macros when specifying the *signal* and the *member*.

Example:

```
QLabel     *label  = new QLabel;
QScrollBar *scroll = new QScrollBar;
QObject::connect( scroll, SIGNAL(valueChanged(int)),
                  label,  SLOT(setNum(int)) );
```

This example connects the scroll bar's valueChanged() signal to the label's setNum() slot. It makes the label always display the current scroll bar value.

A signal can even be connected to another signal, i.e. *member* is a SIGNAL().

```
class MyWidget : public QWidget
{
public:
    MyWidget();
...
signals:
    void aSignal(int);
...
private:
...
    QPushButton *aButton;
};
MyWidget::MyWidget()
{
    aButton = new QPushButton( this );
    connect( aButton, SIGNAL(clicked()), SIGNAL(aSignal()) );
}
```

In its constructor, MyWidget creates a private button and connects the clicked() signal to relay clicked() to the outside world. You can achieve the same effect by connecting the clicked() signal to a private slot and emitting aSignal() in this slot, but that takes a few lines of extra code and is not quite as clear, of course.

A signal can be connected to many slots/signals. Many signals can be connected to one slot.

If a signal is connected to several slots, the slots are activated in arbitrary order when the signal is emitted.

See also: disconnect().

Examples:
- tictac/tictac.cpp
- grapher/grapher.cpp
- drawdemo/drawdemo.cpp
- hello/hello.cpp
- movies/main.cpp
- xform/xform.cpp
- aclock/aclock.cpp
- menu/menu.cpp
- pref/pref.cpp
- qmag/qmag.cpp
- showimg/showimg.cpp
- widgets/widgets.cpp

bool QObject::connect (const QObject * sender, const char * signal, const char * member) const

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Connects *signal* from the *sender* object to *member* in this object.

Equivalent to: `QObject::connect(sender, signal, this, member)`.

See also: disconnect().

void QObject::connectNotify (const char * signal) `[virtual protected]`

This virtual function is called when something has been connected to *signal* in this object.

Warning: This function violates the object-oriented principle of modularity. However, it might be useful when you need to perform expensive initialization only if something is connected to a signal.

See also: connect() and disconnectNotify().

Reimplemented in QClipboard.

void QObject::destroyed () `[signal]`

This signal is emitted immediately before the object is destroyed.

All the objects's children are destroyed immediately after this signal is emitted.

Examples:
- xform/xform.cpp

bool QObject::disconnect (const QObject * sender, const char * signal, const QObject * receiver, const char * member) `[static]`

Disconnects *signal* in object *sender* from *member* in object *receiver*.

A signal-slot connection is removed when either of the objects involved are destroyed.

disconnect() is typically used in three ways, as the following examples show.

1 Disconnect everything connected to an object's signals:

```
disconnect( myObject );
```

2 Disconnect everything connected to a signal:

```
disconnect( myObject, SIGNAL(mySignal()) );
```

3 Disconnect a specific receiver.

```
disconnect( myObject, 0, myReceiver, 0 );
```

0 may be used as a wildcard in three of the four arguments, meaning "any signal", "any receiving object" or "any slot in the receiving object" respectively.

The *sender* has no default and may never be 0. (You cannot disconnect signals from more than one object.)

If *signal* is 0, it disconnects *receiver* and *member* from any signal. If not, only the specified signal is disconnected.

If *receiver* is 0, it disconnects anything connected to *signal*. If not, slots in objects other than *receiver* are not disconnected.

If *member* is 0, it disconnects anything that is connected to *receiver*. If not, only slots named *member* will be disconnected, and all other slots are left alone. The *member* must be 0 if *receiver* is left out, so you cannot disconnect a specifically-named slot on all objects.

See also: connect().

bool QObject::disconnect (const QObject * receiver, const char * member=0)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Disconnects all signals in this object from *member* of *receiver*.

A signal-slot connection is removed when either of the objects involved are destroyed.

bool QObject::disconnect (const char * signal=0, const QObject * receiver=0, const char * member=0)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Disconnects *signal* from *member* of *receiver*.

A signal-slot connection is removed when either of the objects involved are destroyed.

void QObject::disconnectNotify (const char * signal) `[virtual protected]`

This virtual function is called when something has been disconnected from *signal* in this object.

Warning: This function violates the object-oriented principle of modularity. However, it might be useful for optimizing access to expensive resources.

See also: disconnect() and connectNotify().

void QObject::dumpObjectInfo ()

Dumps information about signal connections etc. for this object to the debug output.

This function is useful for debugging.

void QObject::dumpObjectTree ()

Dumps a tree of children to the debug output.

This function is useful for debugging.

bool QObject::event (QEvent * e) `[virtual]`

This virtual function receives events to an object and should return TRUE if the event was recognized and processed.

The event() function can be reimplemented to customize the behavior of an object.

See also: installEventFilter(), timerEvent(), QApplication::sendEvent(), QApplication::postEvent() and QWidget::event().

Reimplemented in QToolBar, QClipboard, QTimer, QSocketNotifier, QLineEdit, QWidget, QMainWindow, QWidgetStack and QSplitter.

bool QObject::eventFilter (QObject *, QEvent *) `[virtual]`

Filters events if this object has been installed as an event filter for another object.

The reimplementation of this virtual function must return TRUE if the event should be stopped, or FALSE if the event should be dispatched normally.

See also: installEventFilter().

Reimplemented in QSpinBox, QListView, QMainWindow, QFileDialog, QAccel, QScrollView, QMenuBar, QGManager and QComboBox.

bool QObject::highPriority () const

Returns TRUE if the object is a high priority object, or FALSE if it is a standard priority object.

High priority objects are placed first in list of children, on the assumption that they will be referenced very often.

bool QObject::inherits (const char * clname) const

Returns TRUE if this object is an instance of a class that inherits *clname,* and *clname* inherits QObject.

(A class is considered to inherit itself.)

Example:

```
QTimer *t = new QTimer;            // QTimer inherits QObject
t->inherits("QTimer");             // returns TRUE
t->inherits("QObject");            // returns TRUE
t->inherits("QButton");            // returns FALSE
```

```
QScrollBar * s = new QScrollBar;    // inherits QWidget and QRangeControl
s->inherits( "QWidget" );           // returns TRUE
s->inherits( "QRangeControl" );     // returns FALSE
```

See also: isA() and metaObject().

void QObject::initMetaObject () `[virtual protected]`

Initializes the meta object of this object. This method is automatically executed on demand from the QObject constructor.

See also: metaObject().

void QObject::insertChild (QObject * obj)

Inserts an object *obj* into the list of child objects.

Warning: This function cannot be used to make a widget a child widget of another. Child widgets can be created only by setting the parent widget in the constructor or by calling QWidget::recreate().

See also: removeChild() and QWidget::recreate().

void QObject::installEventFilter (const QObject * obj)

Installs an event filter object for this object.

An event filter is an object that receives all events that are sent to this object. The filter can either stop the event or forward it to this object. The event filter object receives events via the eventFilter() function. The eventFilter() function must return TRUE if the event should be stopped, or FALSE if the event should be dispatched normally.

If multiple event filters are installed for a single object, the filter that was installed last will be activated first.

Example:

```
#include <qwidget.h>
class MyWidget : public QWidget
{
public:
    MyWidget::MyWidget( QWidget *parent=0, const char *name=0 );
protected:
    bool  eventFilter( QObject *, QEvent * );
};
MyWidget::MyWidget( QWidget *parent, const char *name )
    : QWidget( parent, name )
{
    if ( parent )                           // has a child widget
        parent->installEventFilter( this ); // then install filter
}
bool MyWidget::eventFilter( QObject *, QEvent *e )
{
    if ( e->type() == Event_KeyPress ) {    // key press
        QKeyEvent *k = (QKeyEvent*)e;
        debug( "Ate key press %d", k->key() );
        return TRUE;                        // eat event
    }
    return FALSE;                           // standard event processing
}
```

The QAccel class was implemented using this technique.

Warning: If you delete the receiver object in your eventFilter() function, be sure to return TRUE. If you return FALSE, Qt sends the event to the deleted object and the program will crash.

See also: removeEventFilter(), eventFilter() and event().

bool QObject::isA (const char * clname) const

Returns TRUE if this object is an instance of a specified class, otherwise FALSE.

Example:

```
QTimer *t = new QTimer;              // QTimer inherits QObject
t->isA("QTimer");                    // returns TRUE
t->isA("QObject");                   // returns FALSE
```

See also: inherits() and metaObject().

bool QObject::isWidgetType () const

Returns TRUE if the object is a widget, or FALSE if not.

Calling this function is equivalent to calling inherits("QWidget"), except that it is much faster.

void QObject::killTimer (int id)

Kills the timer with the identifier *id.*

The timer identifier is returned by startTimer() when a timer event is started.

See also: timerEvent(), startTimer() and killTimers().

Examples:
dclock/dclock.cpp
grapher/grapher.cpp

void QObject::killTimers ()

Kills all timers that this object has started.

Using this function makes it harder to subclass your class (it kills timers started by subclasses as well as those started by you), so it is generally better to use killTimer().

See also: timerEvent(), startTimer() and killTimer().

Examples:
qmag/qmag.cpp

QMetaObject * QObject::metaObject () const `[virtual]`

Returns a pointer to the meta object of this object.

A meta object contains information about a class that inherits QObject: class name, super class name, signals and slots. Every class that contains the `Q_OBJECT` macro will also have a meta object.

The meta object information is required by the signal/slot connection mechanism. The functions isA() and inherits() also make use of the meta object.

The meta object is created by the initMetaObject() function, which is generated by the meta object compiler and called from the QObject constructor.

const char * QObject::name () const

Returns the name of this object, or 0 if the object does not have a name.

The object name is set by the constructor or by the setName() function. The object name is not very useful in the current version of Qt, but will become increasingly important in the future.

The queryList() function searches the object tree for objects that matches a particular object name.

See also: setName(), className() and queryList().

Examples:
widgets/widgets.cpp

const char * QObject::name (const char * defaultName) const

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Returns the name of this object, or *defaultName* if the object does not have a name.

This version of name() is particularly useful for debug() calls such as:

```
debug( "MyClass::setPrecision(): (%s) unable to set precision to %f",
        name( "unnamed" ), newPrecision );
```

name("unnamed") is used here so that printf() (used in debug()) will not be asked to output a null pointer.

QObject * QObject::parent () const

Returns a pointer to the parent object.

See also: children().

QObjectList * QObject::queryList (const char * inheritsClass = 0, const char * objName = 0, bool regexpMatch = TRUE, bool recursiveSearch = TRUE)

Returns a list of child objects found by a query.

The query is specified by:

Arguments:

- *inheritsClass* is the name of the base class that an object should inherit. Any class will be matched if *inheritsClass* is 0.
- *objName* is the object name to search for. Any object name will be matched if *objName* is 0.
- *regexpMatch* specifies whether *objName* is a regular expression (default) or not.
- *recursiveSearch* must be `TRUE` (default) if you want to search the entire object tree, or `FALSE` if you want the search to traverse just the 1st level child objects of this object. Example:

```
//
// Sets a Courier 24 point fonts for all children in myWidget that
// inherit QButton (i.e. QPushButton, QCheckBox, QRadioButton).
//
QObjectList *list = myWidget->queryList( "QButton" );
QObjectListIt it( *list );          // iterate over the buttons
QFont newFont( "Courier", 24 );
QObject * obj;
while ( (obj=it.current()) != 0 ) { // for each found object...
    ++it;
    ((QButton*)obj)->setFont( newFont );
}
delete list;                        // delete the list, not the objects
```

The QObjectList class is defined in the qobjcoll.h header file.

Warning: Delete the list away as soon you have finished using it. You can get in serious trouble if you for instance try to access an object that has been deleted.

See also: children(), parent(), inherits(), name() and QRegExp.

QConnectionList * QObject::receivers (const char * signal) const `[protected]`

Returns a list of objects/slot pairs that are connected to the signal, or 0 if nothing is connected to it.

This function is for internal use.

void QObject::removeChild (QObject * obj)

Removes the child object *obj* from the list of children.

Warning: This function will not remove a child widget from the screen. It will only remove it from the parent widget's list of children.

See also: insertChild() and QWidget::recreate().

void QObject::removeEventFilter (const QObject * obj)

Removes an event filter object *obj* from this object. The request is ignored if such an event filter has not been installed.

All event filters for this object are automatically removed when this object is destroyed.

It is always safe to remove an event filter, even during event filter activation (i.e. from the eventFilter() function).

See also: installEventFilter(), eventFilter() and event().

const QObject * QObject::sender () `[protected]`

Returns a pointer to the object that sent the last signal received by this object.

Warning: This function violates the object-oriented principle of modularity, However, getting access to the sender might be practical when many signals are connected to a single slot. The sender is undefined if the slot is called as a normal C++ function.

void QObject::setName (const char * name)

Sets the name of this object to *name*. The default name is the one assigned by the constructor.

The object name is not very useful in the current version of Qt, but it will become increasingly important in the future.

The queryList() function searches the object tree for objects that matches a particular object name.

See also: name(), className() and queryList().

bool QObject::signalsBlocked () const

Returns TRUE if signals are blocked, or FALSE if signals are not blocked.

Signals are not blocked by default.

See also: blockSignals().

int QObject::startTimer (int interval)

Starts a timer and returns a timer identifier, or returns zero if it could not start a timer.

A timer event will occur every *interval* milliseconds until killTimer() or killTimers() is called. If *interval* is 0, then timer event occurs once every time there are no more window system events to process.

The virtual timerEvent() function is called with the QTimerEvent event parameter class when a timer event occurs. Reimplement this function to get timer events.

If multiple timers are running, the QTimerEvent::timerId() can be used to find out which timer was activated.

Example:

```
class MyObject : public QObject
{
public:
    MyObject( QObject *parent=0, const char *name=0 );
protected:
    void  timerEvent( QTimerEvent * );
};
MyObject::MyObject( QObject *parent, const char *name )
    : QObject( parent, name )
{
    startTimer( 50 );                      // 50 millisec timer
    startTimer( 1000 );                    // 1 second timer
    startTimer( 60000 );                   // 1 minute timer
}
void MyObject::timerEvent( QTimerEvent *e )
{
    debug( "timer event, id=%d", e->timerId() );
}
```

There is practically no upper limit for the interval value (more than one year). The accuracy depends on the underlying operating system. Windows 95 has 55 millisecond (18.2 times per second) accuracy; other systems that we have tested (UNIX X11, Windows NT and OS/2) can handle 1 millisecond intervals.

The QTimer class provides a high-level programming interface with one-shot timers and timer signals instead of events.

See also: timerEvent(), killTimer() and killTimers().

Examples:

forever/forever.cpp
qmag/qmag.cpp
biff/biff.cpp

void QObject::timerEvent (QTimerEvent *) `[virtual protected]`

This event handler can be reimplemented in a subclass to receive timer events for the object.

The default implementation does nothing.

QTimer provides a higher-level interface to the timer functionality, and also more general information about timers.

See also: startTimer(), killTimer(), killTimers() and event().

Reimplemented in QListBox, QMultiLineEdit, QPopupMenu, QScrollBar and QLineEdit.

const char * QObject::tr (const char * text) const

Returns a translated version of *text,* or *text* if there is no appropriate translated version.

125.3. Related Functions

void * qt_find_obj_child (QObject * parent, const char * type, const char * name)

Returns a pointer to the child named *name* of QObject *parent* which inherits type *type.*

Returns 0 if there is no such child.

The CHILD macro (defined in qwindowdefs.h) does all this, and also casts the returt type to *type* *.

Examples:

```
QListBox * c = CHILD(myWidget,QListBox,"listboxname");
if ( c )
    c->insertItem( "another string" );
```

```
QListBox * c = (QListBox *)::qt_find_obj_child(myWidget,QListBox,
                                               "listboxname");
if ( c )
    c->insertItem( "another string" );
```

125.3.1. URL

Generated from http://www.troll.no/qt/qobject.html.

126. QPaintDevice Class Reference

```
#include <qpaintdevice.h>
```

Inherited by QPicture, QPixmap, QPrinter and QWidget.

126.1. Detailed Description

The base class of objects that can be painted.

A paint device is an abstraction of a two-dimensional space that can be drawn using a QPainter. The drawing capabilities are implemented by the subclasses: QWidget, QPixmap, QPicture and QPrinter.

The default coordinate system of a paint device has its origin located at the top left position. X increases to the right and Y increases downwards. The unit is one pixel. There are several ways to set up a user-defined coordinate system using the painter, for example by QPainter::setWorldMatrix().

Example (draw on a paint device):

```
void MyWidget::paintEvent( QPaintEvent * )
{
    QPainter p;                             // our painter
    p.begin( this );                        // start painting widget
    p.setPen( red );                        // blue outline
    p.setBrush( yellow );                   // yellow fill
    p.drawEllipse( 10,20, 100,100 );        // 100x100 ellipse at 10,20
    p.end();                                // painting done
}
```

The bit block transfer is an extremely useful operation for copying pixels from one paint device to another (or to itself). It is implemented as the global function bitBlt().

Example (scroll widget contents 10 pixels to the right):

```
bitBlt( myWidget, 10,0, myWidget );
```

Warning: Qt requires that a QApplication object must exist before any paint devices can be created. Paint devices access window system resources, and these resources are not initialized before an application object is created.

126.2. Member Function Documentation

QPaintDevice::QPaintDevice (uint devflags) `[protected]`

Constructs a paint device with internal flags *devflags.* This constructor can only be invoked from subclasses of QPaintDevice.

QPaintDevice::~QPaintDevice () `[virtual]`

Destroys the paint device and frees window system resources.

bool QPaintDevice::cmd (int, QPainter *, QPDevCmdParam *) `[virtual protected]`

Internal virtual function that interprets drawing commands from the painter.

Implemented by subclasses that have no direct support for drawing graphics (external paint devices, for example QPicture).

Reimplemented in QPicture and QPrinter.

int QPaintDevice::devType () const

Returns the device type identifier: `PDT_WIDGET`, `PDT_PIXMAP`, `PDT_PRINTER`, `PDT_PICTURE` or `PDT_UNDEF`.

int QPaintDevice::fontInf (QFont *, int) const `[virtual protected]`

Internal virtual function. Reserved for future use.

int QPaintDevice::fontMet (QFont *, int, const char * = 0, int = 0) const `[virtual protected]`

Internal virtual function. Reserved for future use.

HANDLE QPaintDevice::handle () const

Returns the window system handle of the paint device, for low-level access. *Using this function is not portable.*

The HANDLE type varies with platform; see qpaintdevice.h and qwindowdefs.h for details.

See also: x11Display().

bool QPaintDevice::isExtDev () const

Returns TRUE if the device is a so-called external paint device.

External paint devices cannot be bitBlt()'ed from. QPicture and QPrinter are external paint devices.

int QPaintDevice::metric (int) const `[virtual protected]`

Internal virtual function that returns paint device metrics.

Please use the QPaintDeviceMetrics class instead.

Reimplemented in QPixmap, QWidget, QPicture and QPrinter.

bool QPaintDevice::paintingActive () const

Returns TRUE if the device is being painted, i.e. someone has called QPainter::begin() and not yet QPainter::end() for this device.

See also: QPainter::isActive().

Display * QPaintDevice::x11Display () const

Returns a pointer to the X display (X11 only). *Using this function is not portable.*

See also: handle().

126.3. Related Functions

void bitBlt (QPaintDevice * dst, int dx, int dy, const QPaintDevice * src, int sx, int sy, int sw, int sh, RasterOp rop, bool ignoreMask)

This function copies a block of pixels from one paint device to another (bitBlt means bit block transfer).

Arguments:

- *dst* is the paint device to copy to.
- *dx* and *dy* is the position to copy to.
- *src* is the paint device to copy from.
- *sx* and *sy* is the position to copy from.
- *sw* and *sh* is the width and height of the block to be copied.
- *rop* defines the raster operation to be used when copying. If *sw* is 0 or *sh* is 0, then bitBlt will do nothing.

If *sw* is negative, then bitBlt calculates `sw = src->width - sx.` If *sh* is negative, then bitBlt calculates `sh = src->height - sy.`

The *rop* argument can be one of:

- `CopyROP:` dst = src.
- `OrROP:` dst = dst OR src.
- `XorROP:` dst = dst XOR src.
- `EraseROP:` dst = (NOT src) AND dst
- `NotCopyROP:` dst = NOT src
- `NotOrROP:` dst = (NOT src) OR dst
- `NotXorROP:` dst = (NOT src) XOR dst
- `NotEraseROP:` dst = src AND dst
- `NotROP:` dst = NOT dst

The *ignoreMask* argument (default FALSE) applies where *src* is a QPixmap with a mask. If *ignoreMask* is TRUE, bitBlt ignores the pixmap's mask.

BitBlt has two restrictions:

1 The *src* device must be QWidget or QPixmap. You cannot copy pixels from a picture or a printer (external device).

2 The *src* device may not have pixel depth greater than *dst*. You cannot copy from an 8 bit pixmap to a 1 bit pixmap.

Examples:
desktop/desktop.cpp
hello/hello.cpp
xform/xform.cpp
showimg/showimg.cpp
biff/biff.cpp

void bitBlt (QPaintDevice * dst, const QPoint & dp, const QPaintDevice * src, const QRect & sr, RasterOp rop)

Overloaded bitBlt() with the destination point *dp* and source rectangle *sr*.

126.3.1. URL

Generated from http://www.troll.no/qt/qpaintdevice.html.

127. QPaintDeviceMetrics Class Reference

```
#include <qpaintdevicemetrics.h>
```

127.1. Detailed Description

The QPaintDeviceMetrics class provides information about a paint device.

Sometimes it is necessary to obtain information about the physical size of a paint device when drawing graphics.

Example:

```
QPaintDeviceMetrics pdm( myWidget );
float aspect = (float)pdm.widthMM / (float)pdm.heightMM();
```

Examples:

application/application.cpp

127.2. Member Function Documentation

QPaintDeviceMetrics::QPaintDeviceMetrics (const QPaintDevice * pd)

Constructs a metric for the paint device *pd*.

int QPaintDeviceMetrics::depth () const

Returns the bit depth (number of bit planes) of the paint device.

int QPaintDeviceMetrics::height () const

Returns the height of the paint device, in default coordinate system units (e.g. pixels for QPixmap and QWidget).

Examples:

application/application.cpp

int QPaintDeviceMetrics::heightMM () const

Returns the height of the paint device, measured in millimeters.

int QPaintDeviceMetrics::numColors () const

Returns the number of different colors available for the paint device.

int QPaintDeviceMetrics::width () const

Returns the width of the paint device, in default coordinate system units (e.g. pixels for QPixmap and QWidget).

Examples:

application/application.cpp

int QPaintDeviceMetrics::widthMM () const

Returns the width of the paint device, measured in millimeters.

127.2.1. URL

Generated from http://www.troll.no/qt/qpaintdevicemetrics.html.

128. QPainter Class Reference

```
#include <qpainter.h>
```

128.1. Detailed Description

The QPainter class paints on paint devices.

The painter provides efficient graphics rendering on any QPaintDevice object. QPainter can draw everything from simple lines to complex shapes like pies and chords. It can also draw aligned text and pixmaps.

Graphics can be transformed using view transformation, world transformation or a combination of these two. View transformation is a window/viewport transformation with translation and scaling. World transformation is a full 2D transformation including rotation and shearing.

The typical use of a painter is:

1 Construct a painter.

2 Set a pen, a brush etc.

3 Draw.

4 Destroy the painter.

This example uses a convenience constructor that calls begin(), and relies on the destructor to call end():

```
void MyWidget::paintEvent()
{
    QPainter paint( this );                 // start painting widget
    paint.setPen( blue );                   // set blue pen
    paint.drawText( rect(),                 // draw a text, centered
                    AlignCenter,            //   in the widget
                    "The Text" );
}
```

You can also use the begin() and end() functions to begin and end painting explicitly:

```
void MyWidget::paintEvent()
{
    QPainter paint;
    paint.begin( this );                    // start painting widget
    paint.setPen( blue );                   // set blue pen
    paint.drawText( rect(),                 // draw a text, centered
                    AlignCenter,            //   in the widget
                    "The Text" );
    paint.end();                            // painting done
}
```

This is useful since it is not possible to have two painters active on the same paint device at a time.

QPainter is almost never used outside paintEvent(). Any widget *must* be able to repaint itself at any time via paintEvent(), therefore it's almost always best to design the widget so that it does all the painting in paintEvent() and use either QWidget::update() or QWidget::repaint() force a paint event as necessary.

Note that both painters and some paint devices have attributes such as current font, current foreground colors and so on.

QPainter::begin() copies these attributes from the paint device, and changing a paint device's attributes will have effect only the next time a painter is opened on it.

Warning: QPainter::begin() resets all attributes to their default values, from the device, thus setting fonts, brushes, etc, before begin() will have *no* effect.

See also: QPaintDevice, QWidget and QPixmap.

Examples:
- qtimage/qtimage.cpp
- tictac/tictac.cpp
- table/table.cpp
- life/life.cpp
- grapher/grapher.cpp
- forever/forever.cpp
- desktop/desktop.cpp
- connect/connect.cpp
- trivial/trivial.cpp
- tooltip/tooltip.cpp
- drawdemo/drawdemo.cpp
- hello/hello.cpp
- movies/main.cpp
- picture/picture.cpp
- xform/xform.cpp
- application/application.cpp
- aclock/aclock.cpp
- progress/progress.cpp
- qmag/qmag.cpp
- showimg/showimg.cpp

128.2. Member Function Documentation

QPainter::QPainter ()

Constructs a painter.

Notice that all painter settings (setPen,setBrush etc.) are reset to default values when begin() is called.

See also: begin() and end().

QPainter::QPainter (const QPaintDevice * pd)

Constructs a painter that begins painting the paint device *pd* immediately.

This constructor is convenient for short-lived painters, e.g. in a paint event and should be used only once. The constructor calls begin() for you and the QPainter destructor automatically calls end().

Example using begin() and end():

```
void MyWidget::paintEvent( QPaintEvent * )
{
    QPainter p( this );
    p.drawLine( ... );      // drawing code
}
```

Example using this constructor:

```
void MyWidget::paintEvent( QPaintEvent * )
{
    QPainter p( this );
    p.drawLine( ... );      // drawing code
}
```

See also: begin() and end().

QPainter::QPainter (const QPaintDevice * pd, const QWidget * copyAttributes)

Constructs a painter that begins painting the paint device *pd* immediately, with the default arguments taken from *copyAttributes.*

See also: begin().

QPainter::~QPainter ()

Destroys the painter.

If you called begin() but not end(), the destructor outputs a warning message. Note that there is no need to call end() if you used one of the constructors which takes a paint device argument.

const QColor & QPainter::backgroundColor () const

Returns the background color currently set.

See also: setBackgroundColor().

BGMode QPainter::backgroundMode () const

Returns the background mode currently set.

See also: setBackgroundMode().

bool QPainter::begin (const QPaintDevice * pd)

Begins painting the paint device *pd* and returns TRUE if successful, or FALSE if it cannot begin painting. Call end() when you have finished painting.

On the X Window System, paint commands are buffered and may not appear on the screen immediately. The flush() function flushes the buffer.

As an alternative to calling begin() and end(), you can use the QPainter constructor that takes a paint device argument. This is for short-lived painters, for example in paint events.

This function initializes all painter settings:

- The font is set to the default application font, or to the widget's font if *pd* is a widget.
- The pen is set to QPen(black,0,SolidLine), or to QPen(widget->foreground(), 0,SolidLine) if *pd* is a widget.
- The brush is set to QBrush(NoBrush).
- The background color is set to white, or to the widget's background color if *pd* is a widget.
- The background mode is set to `TransparentMode`.
- The raster operation is set to `CopyROP`.
- The brush origin is set to (0,0).
- The view transformation setting is turned off.
- The window is set to the paint device rectangle; (0,0,pd->width(),pd->height()).
- The viewport is set to the paint device rectangle; (0,0,pd->width(),pd->height()).
- The world transformation setting is turned off.
- The world matrix is set to the identify matrix.
- Clipping is disabled.
- The clip region is set to an empty region.

Warning: A paint device can only be painted by one painter at a time.

See also: end(), flush(), setFont(), setPen(), setBrush(), setBackgroundColor(), setBackgroundMode(), setRasterOp(), setBrushOrigin(), setViewXForm(), setWindow(), setViewport(), setWorldXForm(), setWorldMatrix() and setClipRegion().

Examples:
desktop/desktop.cpp
hello/hello.cpp
picture/picture.cpp
application/application.cpp

bool QPainter::begin (const QPaintDevice * pd, const QWidget * copyAttributes)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

This version opens the painter on a paint device *pd* and sets the initial pen, background color and font from *copyAttributes*. This is equivalent with:

```
QPainter p;
p.begin( pd );
p.setPen( copyAttributes->foregroundColor() );
p.setBackgroundColor( copyAttributes->backgroundColor() );
p.setFont( copyAttributes->font() );
```

This begin function is convenient for double buffering. When you draw in a pixmap instead of directly in a widget (to later bitBlt the pixmap into the widget) you will need to set the widgets's font etc. This function does exactly that.

Example:

```
void MyWidget::paintEvent( QPaintEvent * )
{
    QPixmap pm(rect());
    QPainter p;
    p.begin(&pm, this);
    // ... potential flickering paint operation ...
    p.end();
    bitBlt(this, 0, 0, &pm);
}
```

See also: end().

QRect QPainter::boundingRect (int x, int y, int w, int h, int tf, const char * str, int len = -1, char ** internal=0)

Returns the bounding rectangle of the aligned text that would be printed with the corresponding drawText() function (the first *len* characters from *str*). The drawing, and hence the bounding rectangle, is constrained to the rectangle *(x,y,w,h)*.

The *tf* text formatting is the bitwise OR of the following flags:

- `AlignLeft` aligns to the left border.
- `AlignRight` aligns to the right border.
- `AlignHCenter` aligns horizontally centered.
- `AlignTop` aligns to the top border.
- `AlignBottom` aligns to the bottom border.
- `AlignVCenter` aligns vertically centered
- `AlignCenter` (= `AlignHCenter | AlignVCenter`)
- `SingleLine` ignores newline characters in the text.

- `ExpandTabs` expands tabulators.
- `ShowPrefix` displays "&x" as "x" underlined.
- `WordBreak` breaks the text to fit the rectangle.

These flags are defined in qwindowdefs.h.

See also: drawText() and fontMetrics().

QRect QPainter::boundingRect (const QRect & r, int tf,const char * str, int len = -1, char ** i=0)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

const QBrush & QPainter::brush () const

Returns the current painter brush.

See also: QPainter::setBrush().

const QPoint & QPainter::brushOrigin () const

Returns the brush origin currently set.

See also: setBrushOrigin().

void QPainter::cleanup () `[static]`

Internal function that cleans up the painter.

const QRegion & QPainter::clipRegion () const

Returns the clip region currently set. Note that the clip region is given in physical device coordinates and *not* subject to any coordinate transformation.

See also: setClipRegion(), setClipRect() and setClipping().

QPaintDevice * QPainter::device () const

Returns the paint device currently active for this painter, or null if begin() has not been called.

See also: QPaintDevice::paintingActive().

void QPainter::drawArc (int x, int y, int w, int h, int a, int alen)

Draws an arc defined by the rectangle *(x,y,w,h),* the start angle *a* and the arc length *alen.*

The angles *a* and *alen* are 1/16th of a degree, i.e. a full circle equals 5760 (16*360). Positive values of *a* and *alen* mean counter-clockwise while negative values mean clockwise direction. Zero degrees is at the 3'o clock position.

Example:

```
QPainter p;
p.begin( myWidget );
p.drawArc( 10,10, 70,100, 100*16, 160*16 ); // draws a "(" arc
p.end();
```

See also: drawPie() and drawChord().

void QPainter::drawArc (const QRect & r, int a, int alen)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

void QPainter::drawChord (int x, int y, int w, int h, int a, int alen)

Draws a chord defined by the rectangle *(x,y,w,h),* the start angle *a* and the arc length *alen.*

The chord is filled with the current brush.

The angles *a* and *alen* are 1/16th of a degree, i.e. a full circle equals 5760 (16*360). Positive values of *a* and *alen* mean counter-clockwise while negative values mean clockwise direction. Zero degrees is at the 3'o clock position.

See also: drawArc() and drawPie().

void QPainter::drawChord (const QRect & r, int a, int alen)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

void QPainter::drawEllipse (int x, int y, int w, int h)

Draws an ellipse with center at *(x+w/2,y+h/2)* and size *(w,h).*

Examples:
drawdemo/drawdemo.cpp
picture/picture.cpp

void QPainter::drawEllipse (const QRect & r)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

void QPainter::drawImage (int x, int y, const QImage & image, int sx=0, int sy=0, int sw=-1, int sh=-1)

Draws at (*x, y)* the *sw* by *sh* area of pixels from (*sx, sy)* in *image.*

This function simply converts *image* to a QPixmap and draws it.

See also: drawPixmap() and QPixmap::convertFromImage().

void QPainter::drawImage (const QPoint &, const QImage &)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

void QPainter::drawImage (const QPoint &, const QImage &, const QRect & sr)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

void QPainter::drawLine (int x1, int y1, int x2, int y2)

Draws a line from *(x1,y2)* to *(x2,y2)*. Both endpoints are drawn.

See also: moveTo() and lineTo().

Examples:

table/table.cpp
grapher/grapher.cpp
connect/connect.cpp
aclock/aclock.cpp
progress/progress.cpp

void QPainter::drawLine (const QPoint & p1, const QPoint & p2)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

void QPainter::drawLineSegments (const QPointArray & a, int index=0, int nlines=-1)

Draws *nlines* separate lines from points defined in *a,* starting at a[*index]*. If *nlines* is -1 all points until the end of the array are used (i.e. (a.size()-index)/2 lines are drawn).

Draws the 1st line from *a[index]* to *a[index+1]*. Draws the 2nd line from *a[index+2]* to *a[index+3]* etc.

See also: drawPolyline() and drawPolygon().

void QPainter::drawPicture (const QPicture & pic)

Replays the picture *pic.*

This function does exactly the same as QPicture::play().

Examples:

picture/picture.cpp

void QPainter::drawPie (int x, int y, int w, int h, int a, int alen)

Draws a pie defined by the rectangle *(x,y,w,h),* the start angle *a* and the arc length *alen.*

The pie is filled with the current brush.

The angles *a* and *alen* are 1/16th of a degree, i.e. a full circle equals 5760 (16*360). Positive values of *a* and *alen* mean counter-clockwise while negative values mean clockwise direction. Zero degrees is at the 3'o clock position.

See also: drawArc() and drawPie().

Examples:

grapher/grapher.cpp
drawdemo/drawdemo.cpp

void QPainter::drawPie (const QRect & r, int a, int alen)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

void QPainter::drawPixmap (int x, int y, const QPixmap & pixmap, int sx=0, int sy=0, int sw=-1, int sh=-1)

Draws a pixmap at *(x,y)* by copying a part of the pixmap into the paint device.

Arguments:

- *(x,y)* specify the point in the paint device.
- *(sx,sy)* specify an offset in the pixmap.
- *(sw,sh)* specify the area of the area of the pixmap to be copied. The value -1 means to the right/bottom of the pixmap. The pixmap is clipped if a mask has been set.

See also: bitBlt() and QPixmap::setMask().

Examples:
qtimage/qtimage.cpp
grapher/grapher.cpp
movies/main.cpp
picture/picture.cpp
qmag/qmag.cpp
showimg/showimg.cpp

void QPainter::drawPixmap (const QPoint & p, const QPixmap & pm)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

This version of the call draws the entire pixmap.

void QPainter::drawPixmap (const QPoint & p, const QPixmap & pm, const QRect & sr)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

void QPainter::drawPoint (int x, int y)

Draws/plots a single point at *(x,y)* using the current pen.

Examples:
desktop/desktop.cpp
connect/connect.cpp

void QPainter::drawPoint (const QPoint & p)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

void QPainter::drawPoints (const QPointArray & a, int index=0, int npoints=-1)

Draws/plots an array of points using the current pen. The *index* and *npoints* arguments allow a subsequence of the array to be drawn.

void QPainter::drawPolygon (const QPointArray & a, bool winding=FALSE, int index=0, int npoints=-1)

Draws the polygon defined by the *npoints* points in *a* starting at *a[index].*

If *npoints* is -1 all points until the end of the array are used (i.e. a.size()-index line segments define the polygon).

The first point is always connected to the last point.

The polygon is filled with the current brush. If *winding* is TRUE, the polygon is filled using the winding fill algorithm. If *winding* is FALSE, the polygon is filled using the even-odd (alternative) fill algorithm.

See also: drawLineSegments() and drawPolyline().

Examples:
desktop/desktop.cpp
picture/picture.cpp
aclock/aclock.cpp

void QPainter::drawPolyline (const QPointArray & a, int index=0, int npoints=-1)

Draws the polyline defined by the *npoints* points in *a* starting at *a[index].*

If *npoints* is -1 all points until the end of the array are used (i.e. a.size()-index-1 line segments are drawn).

See also: drawLineSegments() and drawPolygon().

Examples:
drawdemo/drawdemo.cpp

void QPainter::drawQuadBezier (const QPointArray & a, int index=0)

Draws a cubic Bezier curve defined by the control points in *a,* starting at *a[index].*

a must have 4 points or more. The control point *a[index+4]* and beyond are ignored.

void QPainter::drawRect (int x, int y, int w, int h)

Draws a rectangle with upper left corner at *(x,y)* and with width *w* and height *h.*

See also: drawRoundRect().

Examples:
table/table.cpp
grapher/grapher.cpp
forever/forever.cpp
trivial/trivial.cpp
tooltip/tooltip.cpp
drawdemo/drawdemo.cpp
picture/picture.cpp

void QPainter::drawRect (const QRect & r)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

void QPainter::drawRoundRect (int x, int y, int w, int h, int xRnd, int yRnd)

Draws a rectangle with round corners at *(x,y),* with width *w* and height *h.*

The *xRnd* and *yRnd* arguments specify how rounded the corners should be. 0 is angled corners, 99 is maximum roundedness.

The width and height include both lines.

See also: drawRect().

Examples:
drawdemo/drawdemo.cpp

void QPainter::drawRoundRect (const QRect & r, int xRnd, int yRnd)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

void QPainter::drawText (int x, int y, const char * str, int len = -1)

Draws at most *len* characters from *str* at position *(x,y).*

(x,y) is the base line position. Note that the meaning of *y* is not the same for the two drawText() varieties.

Examples:
table/table.cpp
grapher/grapher.cpp
desktop/desktop.cpp
trivial/trivial.cpp
drawdemo/drawdemo.cpp
hello/hello.cpp
movies/main.cpp
picture/picture.cpp
application/application.cpp
progress/progress.cpp

void QPainter::drawText (int x, int y, int w, int h, int tf, const char * str, int len = -1, QRect * brect=0, char ** internal=0)

Draws at most *len* characters from *str* in the rectangle *(x,y,w,h).*

Note that the meaning of *y* is not the same for the two drawText() varieties.

This function draws formatted text. The *tf* text formatting is the bitwise OR of the following flags:

- `AlignLeft` aligns to the left border.
- `AlignRight` aligns to the right border.
- `AlignHCenter` aligns horizontally centered.
- `AlignTop` aligns to the top border.
- `AlignBottom` aligns to the bottom border.
- `AlignVCenter` aligns vertically centered
- `AlignCenter` (= `AlignHCenter` | AlignVCenter)
- `SingleLine` ignores newline characters in the text.
- `DontClip` never clips the text to the rectangle.
- `ExpandTabs` expands tabulators.

- `ShowPrefix` displays "&x" as "x" underlined.
- `WordBreak` breaks the text to fit the rectangle.
- `GrayText` grays out the text.

Horizontal alignment defaults to AlignLeft and vertical alignment defaults to AlignTop.

If several of the horizontal or several of the vertical alignment flags are set, the resulting alignment is undefined.

If ExpandTabs is set and no tab stops or tab array have been set tabs will expand to the closest reasonable tab stop based on the current font. For fixed pitch (fixed width) fonts you are guaranteed that each tab stop will be at a multiple of eight of the width of the characters in the font.

brect (if non-null) is set to the actual bounding rectangle of the output. *internal* is, yes, internal.

These flags are defined in qwindowdefs.h.

See also: boundingRect().

void QPainter::drawText (const QPoint & p, const char * s, int len = -1)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

void QPainter::drawText (const QRect & r, int tf, const char * str, int len = -1, QRect * br=0, char ** i=0)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

void QPainter::drawTiledPixmap (int x, int y, int w, int h, const QPixmap & pixmap, int sx=0, int sy=0)

Draws a tiled *pixmap* in the specified rectangle.

Arguments:

- *(x,y,w,h)* is the rectangle to be filled.
- *(sx,sy)* spefify an offset in the pixmap. The pixmap is clipped if a mask has been set.

Calling drawTiledPixmap() is similar to calling drawPixmap() several times to fill (tile) an area with a pixmap.

See also: drawPixmap().

void QPainter::drawTiledPixmap (const QRect & r, const QPixmap & pm)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

void QPainter::drawTiledPixmap (const QRect & r, const QPixmap & pm, const QPoint & sp)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

void QPainter::drawWinFocusRect (int x, int y, int w, int h)

Draws a Windows focus rectangle with upper left corner at *(x,y)* and with width *w* and height *h*.

This function draws a stippled XOR rectangle that is used to indicate keyboard focus (when the GUI style is `WindowStyle)`.

Warning: This function draws nothing if the coordinate system has been rotated or sheared.

See also: drawRect() and QApplication::style().

void QPainter::drawWinFocusRect (int x, int y, int w, int h, const QColor & bgColor)

Draws a Windows focus rectangle with upper left corner at *(x,y)* and with width *w* and height *h* using a pen color that contrasts with *bgColor*.

This function draws a stippled rectangle (XOR is not used) that is used to indicate keyboard focus (when the GUI style is `WindowStyle)`.

The pen color used to draw the rectangle is either white or black depending on the grayness of *bgColor* (see QColor::gray()).

Warning: This function draws nothing if the coordinate system has been rotated or sheared.

See also: drawRect() and QApplication::style().

void QPainter::drawWinFocusRect (const QRect & r)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

void QPainter::drawWinFocusRect (const QRect & r, const QColor & bgColor)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

bool QPainter::end ()

Ends painting. Any resources used while painting are released.

See also: begin().

Examples:
- desktop/desktop.cpp
- hello/hello.cpp
- picture/picture.cpp
- application/application.cpp

void QPainter::eraseRect (int x, int y, int w, int h)

Erases the area inside *(x,y,w,h)*. Equivalent to `fillRect( x, y, w, h, backgroundColor() )`

Examples:
- life/life.cpp
- showimg/showimg.cpp

void QPainter::eraseRect (const QRect & r)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

void QPainter::fillRect (int x, int y, int w, int h, const QBrush & brush)

Fills the rectangle *(x,y,w,h)* with the *brush.*

You can specify a QColor as *brush,* since there is a QBrush constructor that takes a QColor argument and creates a solid pattern brush.

See also: drawRect().

Examples:
progress/progress.cpp

void QPainter::fillRect (const QRect & r, const QBrush & brush)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

void QPainter::flush ()

Flushes any buffered drawing operations.

const QFont & QPainter::font () const

Returns the current painter font.

See also: setFont() and QFont.

QFontInfo QPainter::fontInfo () const

Returns the font info for the painter. Font info can only be obtained when the painter is active.

See also: fontMetrics() and isActive().

QFontMetrics QPainter::fontMetrics () const

Returns the font metrics for the painter. Font metrics can only be obtained when the painter is active.

See also: fontInfo() and isActive().

Examples:
desktop/desktop.cpp
drawdemo/drawdemo.cpp
movies/main.cpp
application/application.cpp

HANDLE QPainter::handle () const

Returns the platform-dependent handle used for drawing.

bool QPainter::hasClipping () const

Returns TRUE if clipping has been set, otherwise FALSE.

See also: setClipping().

bool QPainter::hasViewXForm () const

Returns TRUE if view transformation is enabled, otherwise FALSE.

See also: setViewXForm() and xForm().

bool QPainter::hasWorldXForm () const

Returns TRUE if world transformation is enabled, otherwise FALSE.

See also: setWorldXForm().

void QPainter::initialize () `[static]`

Internal function that initializes the painter.

bool QPainter::isActive () const

Returns the TRUE if the painter is active painting, i.e. begin() has been called and end() has not yet been called.

See also: QPaintDevice::paintingActive().

Examples:
desktop/desktop.cpp

void QPainter::lineTo (int x, int y)

Draws a line from the current point to *(x,y)* and sets this to the new current point. Both endpoints are are drawn.

See also: moveTo() and drawLine().

void QPainter::lineTo (const QPoint & p)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

void QPainter::moveTo (int x, int y)

Sets the current point.

See also: lineTo() and drawLine().

void QPainter::moveTo (const QPoint & p)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

const QPen & QPainter::pen () const

Returns the current pen for the painter.

See also: setPen().

Examples:
progress/progress.cpp

RasterOp QPainter::rasterOp () const

Returns the raster operation currently set.

See also: setRasterOp().

void QPainter::redirect (QPaintDevice * pdev, QPaintDevice * replacement) `[static]`

Redirects all paint command for a paint device *pdev* to another paint device *replacement.*

A redirected paint device is reset if *replacement* is 0.

The following example redirects painting of a widget to a pixmap:

```
QPixmap pm( myWidget->width(), myWidget->height() );
pm.fill( myWidget->backgroundColor() );
QPainter::redirect( myWidget, &pm );
myWidget->repaint( FALSE );
QPainter::redirect( myWidget, 0 );
```

void QPainter::resetXForm ()

Resets any transformations that were made using translate(), scale(), shear(), rotate(), setWorldMatrix(), setViewport() and setWindow()

See also: worldMatrix(), viewPort() and window().

void QPainter::restore ()

Restores the current painter state (pops a saved state off the stack).

See also: save().

void QPainter::rotate (float a)

Rotates the coordinate system *a* degrees.

See also: translate(), scale(), shear(), resetXForm(), setWorldMatrix() and xForm().

void QPainter::save ()

Saves the current painter state (pushes the state onto a stack).

A save() must have a corresponding restore().

See also: restore().

void QPainter::scale (float sx, float sy)

Scales the coordinate system by *(sx,sy).*

See also: translate(), shear(), rotate(), resetXForm(), setWorldMatrix() and xForm().

void QPainter::setBackgroundColor (const QColor & c)

Sets the background color of the painter to *c.*

The background color is the color that is filled in when drawing opaque text, stippled lines and bitmaps. The background color has no effect when transparent background mode is set.

See also: backgroundColor() and setBackgroundMode().

void QPainter::setBackgroundMode (BGMode m)

Sets the background mode of the painter to *m,* which must be one of:

- `TransparentMode` (default)
- `OpaqueMode`

Transparent mode draws stippled lines and text without setting the background pixels. Opaque mode fills these space with the current background color.

In order to draw a bitmap or pixmap transparently, you must use QPixmap::setMask().

See also: backgroundMode() and setBackgroundColor().

Examples:
picture/picture.cpp

void QPainter::setBrush (BrushStyle style)

Sets a new painter brush with black color and the specified *style.*

See also: brush() and QBrush.

void QPainter::setBrush (const QBrush & brush)

Sets a new painter brush.

The brush defines how to fill shapes.

See also: brush().

Examples:
grapher/grapher.cpp
forever/forever.cpp
desktop/desktop.cpp
tooltip/tooltip.cpp
drawdemo/drawdemo.cpp
picture/picture.cpp
aclock/aclock.cpp

void QPainter::setBrush (const QColor & color)

Sets a new painter brush with the style `SolidPattern` and the specified *color.*

See also: brush() and QBrush.

void QPainter::setBrushOrigin (int x, int y)

Sets the brush origin to *(x,y)*.

The brush origin specifies the (0,0) coordinate of the painter's brush. This setting is only necessary for pattern brushes or pixmap brushes.

See also: brushOrigin().

void QPainter::setBrushOrigin (const QPoint & p)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

void QPainter::setClipRect (int x, int y, int w, int h)

Sets the clip region to *(x,y,w,h)* and enables clipping.

Note that the clip rectangle is given in physical device coordinates and *not* subject to any coordinate transformation.

See also: setClipRegion(), clipRegion() and setClipping().

Examples:

qtimage/qtimage.cpp
grapher/grapher.cpp
trivial/trivial.cpp
progress/progress.cpp
showimg/showimg.cpp

void QPainter::setClipRect (const QRect & r)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

void QPainter::setClipRegion (const QRegion & rgn)

Sets the clip region to *rgn* and enables clipping.

Note that the clip region is given in physical device coordinates and *not* subject to any coordinate transformation.

See also: setClipRect(), clipRegion() and setClipping().

void QPainter::setClipping (bool enable)

Enables clipping if *enable* is TRUE, or disables clipping if *enable* is FALSE.

See also: hasClipping(), setClipRect() and setClipRegion().

void QPainter::setFont (const QFont & font)

Sets a new painter font.

This font is used by all subsequent drawText() functions. The text color is the same as the pen color.

See also: font() and drawText().

Examples:
grapher/grapher.cpp
drawdemo/drawdemo.cpp
hello/hello.cpp
movies/main.cpp
picture/picture.cpp
application/application.cpp

void QPainter::setPen (PenStyle style)

Sets a new painter pen with style `style`, width 0 and black color.

See also: pen() and QPen.

void QPainter::setPen (const QColor & color)

Sets a new painter pen with style `SolidLine`, width 0 and the specified *color*.

See also: pen() and QPen.

void QPainter::setPen (const QPen & pen)

Sets a new painter pen.

The pen defines how to draw lines and outlines, and it also defines the text color.

See also: pen().

Examples:
table/table.cpp
grapher/grapher.cpp
forever/forever.cpp
desktop/desktop.cpp
connect/connect.cpp
drawdemo/drawdemo.cpp
hello/hello.cpp
movies/main.cpp
progress/progress.cpp

void QPainter::setRasterOp (RasterOp r)

Sets the raster operation to *r*.

The *r* parameter must be one of:

- `CopyROP:` dst = src.
- `OrROP:` dst = dst OR src.
- `XorROP:` dst = dst XOR src.
- `EraseROP:` dst = (NOT src) AND dst
- `NotCopyROP:` dst = NOT src
- `NotOrROP:` dst = (NOT src) OR dst
- `NotXorROP:` dst = (NOT src) XOR dst
- `NotEraseROP:` dst = src AND dst
- `NotROP:` dst = NOT dst

See also: rasterOp().

void QPainter::setTabArray (int * ta)

Set an array containing the tab stops.

Tab stops are used when drawing formatted text with `ExpandTabs` set.

The last tab stop must be 0 (terminates the array).

Notice that setting a tab array overrides any fixed tabulator stop that is set using setTabStops().

See also: tabArray(), setTabStops(), drawText() and fontMetrics().

void QPainter::setTabStops (int ts)

Set the number of pixels per tab stop to a fixed number.

Tab stops are used when drawing formatted text with `ExpandTabs` set. This fixed tab stop value has lower precedence than tab array settings.

See also: tabStops(), setTabArray(), drawText() and fontMetrics().

void QPainter::setViewXForm (bool enable)

Enables view transformations if *enable* is TRUE, or disables view transformations if *enable* is FALSE.

See also: hasViewXForm(), setWindow(), setViewport(), setWorldMatrix(), setWorldXForm() and xForm().

void QPainter::setViewport (int x, int y, int w, int h)

Sets the viewport rectangle view transformation for the painter and enables view transformation.

The viewport rectangle is part of the view transformation. The viewport specifies the device coordinate system.

The viewport and the window are initially set to *(0,0,width,height),* where *(width,height)* is the pixel size of the paint device.

You can use this method to normalize the coordinate system of the painter when drawing on a part of a paint device. The following example will draw a line from the top left to the bottom right corner of a page, excluding margins:

```
QPrinter page;
int margin, pageWidth, pageHeight;
...
QPainter p( page );
p.setViewPort( margin, margin, pageWidth - margin, pageHeight - margin );
p.drawLine( 0, 0, pageWidth - 2*margin, pageHeight - 2*margin );
```

The setViewPort() method is often used in conjunction with setWindow(), as in this example:

```
QPainter p( myWidget );
p.setWindow( 0, 0, 1000, 2000 );
p.setViewport( 100,100, 200,200 );
p.drawPoint( 500, 500 );                // draws pixel at (150,125)
```

The preceding example sets up a transformation that maps the logical coordinates (0,0,1000,2000) into a (200,200) rectangle at (100,100).

View transformations can be combined with world transformations. World transformations are applied after the view transformations.

See also: viewport(), setWindow(), setViewXForm(), setWorldMatrix(), setWorldXForm() and xForm().

void QPainter::setViewport (const QRect & r)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

void QPainter::setWindow (int x, int y, int w, int h)

Sets the window rectangle view transformation for the painter and enables view transformation.

The window rectangle is part of the view transformation. The window specifies the logical coordinate system.

The window and the viewport are initially set to *(0,0,width,height),* where *(width,height)* is the pixel size of the paint device.

You can use this method to normalize the coordinate system of the painter. The following example will draw a vertical line, from top to bottom, at the center of a pixmap, independent of the size of the pixmap:

```
int width, height;
...
QPixmap icon( width, height );
QPainter p( icon );
p.setWindow( 0, 0, 100, 100 );
p.drawLine( 50, 0, 50, 100 );            // draw center line
```

The setWindow() method is often used in conjunction with setViewport(), as in this example:

```
QPainter p( myWidget );
p.setWindow( 0, 0, 1000, 2000 );
p.setViewport( 100,100, 200,200 );
p.drawPoint( 500, 500 );                 // draws pixel at (150,125)
```

The preceding example sets up a transformation that maps the logical coordinates (0,0,1000,2000) into a (200,200) rectangle at (100,100).

View transformations can be combined with world transformations. World transformations are applied after the view transformations.

See also: window(), setViewport(), setViewXForm(), setWorldMatrix() and setWorldXForm().

Examples:
forever/forever.cpp
drawdemo/drawdemo.cpp

void QPainter::setWindow (const QRect & r)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

void QPainter::setWorldMatrix (const QWMatrix & m, bool combine=FALSE)

Sets the world transformation matrix to *m* and enables world transformation.

If *combine* is TRUE, then *m* is combined with the current transformation matrix, otherwise *m* will replace the current transformation matrix.

World transformations are applied after the view transformations (i.e. window and viewport).

If the matrix set is the identity matrix (m11 and m22 are 1.0 and the rest are 0.0), this function calls setWorldXForm(FALSE).

The following functions can transform the coordinate system without using a QWMatrix:

- translate()
- scale()
- shear()
- rotate()

They operate on the painter's world matrix and are implemented like this:

```
void QPainter::rotate( float a )
{
    wxmat.rotate( a );
    setWorldMatrix( wxmat );
}
```

See the QWMatrix documentation for a general discussion on coordinate system transformations.

See also: worldMatrix(), setWorldXForm(), setWindow(), setViewport(), setViewXForm() and xForm().

Examples:
drawdemo/drawdemo.cpp
aclock/aclock.cpp

void QPainter::setWorldXForm (bool enable)

Enables world transformations if *enable* is TRUE, or disables world transformations if *enable* is FALSE.

See also: setWorldMatrix(), setWindow(), setViewport(), setViewXForm() and xForm().

void QPainter::shear (float sh, float sv)

Shears the coordinate system *(sh,sv)*.

See also: translate(), scale(), rotate(), resetXForm(), setWorldMatrix() and xForm().

int * QPainter::tabArray () const

Returns the tab stop array currently set.

See also: setTabArray().

int QPainter::tabStops () const

Returns the tab stop setting.

See also: setTabStops().

void QPainter::translate (float dx, float dy)

Translates the coordinate system by *(dx,dy)*.

For example, the following code draws a single vertical line 20 pixels high.

```
void MyWidget::paintEvent()
{
    QPainter paint( this );
    paint.drawLine(10,0,10,20);
    paint.translate(100.0,100.0);
    paint.drawLine(-90,-80,-90,-70);
}
```

See also: scale(), shear(), rotate(), resetXForm(), setWorldMatrix() and xForm().

QRect QPainter::viewport () const

Returns the viewport rectangle.

See also: setViewport() and setViewXForm().

QRect QPainter::window () const

Returns the window rectangle.

See also: setWindow() and setViewXForm().

const QWMatrix & QPainter::worldMatrix () const

Returns the world transformation matrix.

See also: setWorldMatrix().

QPoint QPainter::xForm (const QPoint & pv) const

Returns the point *pv* transformed from user coordinates to device coordinates.

See also: xFormDev() and QWMatrix::xForm().

QPointArray QPainter::xForm (const QPointArray & av) const

Returns the point array *av* transformed from user coordinates to device coordinates.

See also: xFormDev() and QWMatrix::xForm().

QPointArray QPainter::xForm (const QPointArray & av, int index, int npoints) const

Returns the point array *av* transformed from user coordinates to device coordinates. The *index* is the first point in the array and *npoints* denotes the number of points to be transformed. If *npoints* is negative, all points from *av[index]* until the last point in the array are transformed.

The returned point array consists of the number of points that were transformed.

Example:

```
QPointArray a(10);
QPointArray b;
b = painter.xForm(a,2,4);   // b.size() == 4
b = painter.xForm(a,2,-1);  // b.size() == 8
```

See also: xFormDev() and QWMatrix::xForm().

QRect QPainter::xForm (const QRect & rv) const

Returns the rectangle *rv* transformed from user coordinates to device coordinates.

If world transformation is enabled and rotation or shearing has been specified, then the bounding rectangle is returned.

See also: xFormDev() and QWMatrix::xForm().

QPoint QPainter::xFormDev (const QPoint & pd) const

Returns the point *pv* transformed from device coordinates to user coordinates.

See also: xForm() and QWMatrix::xForm().

QPointArray QPainter::xFormDev (const QPointArray & ad) const

Returns the point array *av* transformed from device coordinates to user coordinates.

See also: xForm() and QWMatrix::xForm().

QPointArray QPainter::xFormDev (const QPointArray & ad, int index, int npoints) const

Returns the point array *ad* transformed from device coordinates to user coordinates. The *index* is the first point in the array and *npoints* denotes the number of points to be transformed. If *npoints* is negative, all points from *av[index]* until the last point in the array are transformed.

The returned point array consists of the number of points that were transformed.

Example:

```
QPointArray a(10);
QPointArray b;
b = painter.xFormDev(a,1,3);        // b.size() == 3
b = painter.xFormDev(a,1,-1);       // b.size() == 9
```

See also: xForm() and QWMatrix::xForm().

QRect QPainter::xFormDev (const QRect & rd) const

Returns the rectangle *rv* transformed from device coordinates to user coordinates.

If world transformation is enabled and rotation or shearing is used, then the bounding rectangle is returned.

See also: xForm() and QWMatrix::xForm().

128.3. Related Functions

void qDrawShadeLine (QPainter * p, int x1, int y1, int x2, int y2, const QColorGroup & g, bool sunken, int lineWidth, int midLineWidth)

Draws a horizontal *(y1 == y2)* or vertical *(x1 == x2)* shaded line using the painter *p.*

Nothing is drawn if *y1* != y2 and *x1* != x2 (i.e. the line is neither horizontal nor vertical).

The color group argument *g* specifies the shading colors (light, dark and middle colors).

The line appears sunken if *sunken* is TRUE, or raised if *sunken* is FALSE.

The *lineWidth* argument specifies the line width for each of the lines. It is not the total line width.

The *midLineWidth* argument specifies the width of a middle line drawn in the QColorGroup::mid() color.

If you want to use a QFrame widget instead, you can make it display a shaded line, for example `QFrame::setFrameStyle( QFrame::HLine | QFrame::Sunken )`.

See also: qDrawShadeRect() and qDrawShadePanel().

void qDrawShadeRect (QPainter * p, int x, int y, int w, int h, const QColorGroup & g, bool sunken, int lineWidth, int midLineWidth, const QBrush * fill)

Draws a shaded rectangle/box given by *(x,y,w,h)* using the painter *p.*

The color group argument *g* specifies the shading colors (light, dark and middle colors).

The rectangle appears sunken if *sunken* is TRUE, or raised if *sunken* is FALSE.

The *lineWidth* argument specifies the line width for each of the lines. It is not the total line width.

The *midLineWidth* argument specifies the width of a middle line drawn in the QColorGroup::mid() color.

The rectangle interior is filled with the **fill* brush unless *fill* is null.

If you want to use a QFrame widget instead, you can make it display a shaded rectangle, for example `QFrame::setFrameStyle( QFrame::Box | QFrame::Raised )`.

See also: qDrawShadeLine(), qDrawShadePanel() and qDrawPlainRect().

void qDrawShadePanel (QPainter * p, int x, int y, int w, int h, const QColorGroup & g, bool sunken, int lineWidth, const QBrush * fill)

Draws a shaded panel given by *(x,y,w,h)* using the painter *p*.

The color group argument *g* specifies the shading colors (light, dark and middle colors).

The panel appears sunken if *sunken* is TRUE, or raised if *sunken* is FALSE.

The *lineWidth* argument specifies the line width.

The panel interior is filled with the **fill* brush unless *fill* is null.

If you want to use a QFrame widget instead, you can make it display a shaded panel, for example `QFrame::setFrameStyle( QFrame::Panel | QFrame::Sunken )`.

See also: qDrawWinPanel(), qDrawShadeLine() and qDrawShadeRect().

Examples:
- tictac/tictac.cpp
- life/life.cpp

void qDrawWinButton (QPainter * p, int x, int y, int w, int h, const QColorGroup & g, bool sunken, const QBrush * fill)

Draws a Windows-style button given by *(x,y,w,h)* using the painter *p*.

The color group argument *g* specifies the shading colors (light, dark and middle colors).

The button appears sunken if *sunken* is TRUE, or raised if *sunken* is FALSE.

The line width is 2 pixels.

The button interior is filled with the **fill* brush unless *fill* is null.

See also: qDrawWinPanel().

void qDrawWinPanel (QPainter * p, int x, int y, int w, int h, const QColorGroup & g, bool sunken, const QBrush * fill)

Draws a Windows-style panel given by *(x,y,w,h)* using the painter *p*.

The color group argument *g* specifies the shading colors.

The panel appears sunken if *sunken* is TRUE, or raised if *sunken* is FALSE.

The line width is 2 pixels.

The button interior is filled with the **fill* brush unless *fill* is null.

If you want to use a QFrame widget instead, you can make it display a shaded panel, for example `QFrame::setFrameStyle( QFrame::WinPanel | QFrame::Raised )`.

See also: qDrawShadePanel() and qDrawWinButton().

void qDrawPlainRect (QPainter * p, int x, int y, int w, int h, const QColor & c, int lineWidth, const QBrush * fill)

Draws a plain rectangle given by *(x,y,w,h)* using the painter *p*.

The color argument *c* specifies the line color.

The *lineWidth* argument specifies the line width.

The rectangle interior is filled with the **fill* brush unless *fill* is null.

If you want to use a QFrame widget instead, you can make it display a shaded rectangle, for example `QFrame::setFrameStyle( QFrame::Box | QFrame::Plain )`.

See also: qDrawShadeRect().

128.3.1. URL

Generated from http://www.troll.no/qt/qpainter.html.

129. QPaintEvent Class Reference

`#include <qevent.h>`

Inherits QEvent.

129.1. Detailed Description

The QPaintEvent class contains event parameters for paint events.

Paint events are sent to widgets that need to update themselves, for instance when a part of a widget is exposed because an overlying widget is moved away.

The event handler QWidget::paintEvent() receives paint events.

See also: QPainter, QWidget::update() and QWidget::repaint().

Examples:

- qtimage/qtimage.cpp
- life/life.cpp
- grapher/grapher.cpp
- forever/forever.cpp
- desktop/desktop.cpp
- connect/connect.cpp
- trivial/trivial.cpp
- tooltip/tooltip.cpp
- drawdemo/drawdemo.cpp
- hello/hello.cpp
- picture/picture.cpp
- xform/xform.cpp
- aclock/aclock.cpp
- progress/progress.cpp
- qmag/qmag.cpp
- showimg/showimg.cpp
- biff/biff.cpp

129.2. Member Function Documentation

QPaintEvent::QPaintEvent (const QRect & paintRect)

Constructs a paint event object with the rectangle that should be updated.

const QRect & QPaintEvent::rect () const

Returns the rectangle that should be updated.

Examples:

- qtimage/qtimage.cpp
- life/life.cpp
- grapher/grapher.cpp
- trivial/trivial.cpp
- progress/progress.cpp
- showimg/showimg.cpp

129.2.1. URL

Generated from http://www.troll.no/qt/qpaintevent.html.

130. QPalette Class Reference

```
#include <qpalette.h>
```

130.1. Detailed Description

The QPalette class contains color groups for each widget state.

A palette consists of three color groups: a *normal,* a *disabled* and an *active* color group. All widgets contain a palette, and all the widgets in Qt use their palette to draw themselves. This makes the user interface consistent and easily configurable.

If you make a new widget you are strongly advised to use the colors in the palette rather than hard-coding specific colors.

The *active* group is used for the widget in focus. Normally it contains the same colors as *normal* so as not to overwhelm the user with bright and flashing colors, but if you need to you can change it.

The *disabled* group is used for widgets that are currently inactive or not usable.

The *normal* color group is used in all other cases.

See also: QApplication::setPalette(), QWidget::setPalette(), QColorGroup and QColor.

Examples:
    xform/xform.cpp
    widgets/widgets.cpp

130.2. Member Function Documentation

QPalette::QPalette ()

Constructs a palette that consists of color groups with only black colors.

QPalette::QPalette (const QColor & background)

Constructs a palette from the *background* color. The other colors are automatically calculated, based on this color.

QPalette::QPalette (const QColorGroup & normal, const QColorGroup & disabled, const QColorGroup & active)

Constructs a palette that consists of the three color groups *normal, disabled* and *active.*

QPalette::QPalette (const QPalette & p)

Constructs a palette that is a shallow copy of *p.*

See also: copy().

QPalette::~QPalette ()

Destroys the palette.

const QColorGroup & QPalette::active () const

Returns the active color group of this palette.

See also: QColorGroup, normal(), disabled() and setActive().

QPalette QPalette::copy () const

Returns a deep copy of the palette.

const QColorGroup & QPalette::disabled () const

Returns the disabled color group of this palette.

See also: QColorGroup, normal(), active() and setDisabled().

bool QPalette::isCopyOf (const QPalette & p)

Returns TRUE if this palette and *p* are copies of each other, ie. one of them was created as a copy of the other and neither was subsequently modified. This is much stricter than equality.

See also: operator= and operator==.

const QColorGroup & QPalette::normal () const

Returns the normal color group of this palette.

See also: QColorGroup, disabled(), active() and setNormal().

bool QPalette::operator!= (const QPalette & p) const

Returns TRUE if this palette is different from *p,* or FALSE if they are equal.

QPalette & QPalette::operator= (const QPalette & p)

Assigns *p* to this palette and returns a reference to this palette. Note that a *shallow* copy of *p* is used.

See also: copy().

bool QPalette::operator== (const QPalette & p) const

Returns TRUE if this palette is equal to *p,* or FALSE if they are different.

int QPalette::serialNumber () const

Returns a number that uniquely identifies this QPalette object. The serial number is very useful for caching.

See also: QPixmap and QPixmapCache.

void QPalette::setActive (const QColorGroup & g)

Sets the `active` color group to *g.*

See also: active().

void QPalette::setDisabled (const QColorGroup & g)

Sets the `disabled` color group to *g.*

See also: disabled().

void QPalette::setNormal (const QColorGroup & g)

Sets the `normal` color group to *g*.

See also: normal().

130.3. Related Functions

QDataStream & operator<< (QDataStream & s, const QPalette & p)

Writes a palette to the stream and returns a reference to the stream.

Serialization format:

1 QColorGroup normal

2 QColorGroup disabled

3 QColorGroup active The color groups are serialized in the listed order.

QDataStream & operator>> (QDataStream & s, QPalette & p)

Reads a palette from the stream and returns a reference to the stream.

130.3.1. URL

Generated from http://www.troll.no/qt/qpalette.html.

131. QPen Class Reference

```
#include <qpen.h>
```

131.1. Detailed Description

The QPen class defines how a QPainter should draw lines and outlines of shapes.

A pen has a style, a width and a color.

The pen style defines the line type. The default pen style is `SolidPen.` Setting the style to `NoPen` tells the painter to not draw lines or outlines.

The pen width defines the line width. The default line width is 0, which draws a 1-pixel line very fast, but with lower presicion than with a line width of 1. Setting the line width to 1 or more draws lines that are precise, but drawing is slower.

The pen color defines the color of lines and text. The default line color is black. The QColor documentation lists predefined colors.

Use the QBrush class for specifying fill styles.

Example:

```
QPainter painter;
QPen     pen( red, 2 );             // red solid line, 2 pixel width
painter.begin( &anyPaintDevice );   // paint something
painter.setPen( pen );              // set the red, fat pen
painter.drawRect( 40,30, 200,100 ); // draw rectangle
painter.setPen( blue );             // set blue pen, 0 pixel width
painter.drawLine( 40,30, 240,130 ); // draw diagonal in rectangle
painter.end();                      // painting done
```

See the setStyle() function for a complete list of pen styles.

See also: QPainter and QPainter::setPen().

Examples:

tictac/tictac.cpp
desktop/desktop.cpp
progress/progress.cpp

131.2. Member Function Documentation

QPen::QPen ()

Constructs a default black solid line pen with 0 width.

QPen::QPen (PenStyle style)

Constructs a pen black with 0 width and a specified style.

See also: setStyle().

QPen::QPen (const QColor & color, uint width=0, PenStyle style=SolidLine)

Constructs a pen with a specified color, width and style.

See also: setWidth(), setStyle() and setColor().

QPen::QPen (const QPen & p)

Constructs a pen which is a copy of *p.*

QPen::~QPen ()

Destroys the pen.

const QColor & QPen::color () const

Returns the pen color.

See also: setColor().

bool QPen::operator!= (const QPen & p) const

Returns TRUE if the pen is different from *p,* or FALSE if the pens are equal.

Two pens are different if they have different styles, widths or colors.

See also: operator==().

QPen & QPen::operator= (const QPen & p)

Assigns *c* to this pen and returns a reference to this pen.

bool QPen::operator== (const QPen & p) const

Returns TRUE if the pen is equal to *p,* or FALSE if the pens are different.

Two pens are equal if they have equal styles, widths and colors.

See also: operator!=().

void QPen::setColor (const QColor & c)

Sets the pen color to *c.*

See also: color().

Examples:
progress/progress.cpp

void QPen::setStyle (PenStyle s)

Sets the pen style to *s.*

The pen styles are:

- `NoPen` no outline is drawn.
- `SolidLine` solid line (default).
- `DashLine` - - - (dashes) line.
- `DotLine` * * * (dots) line.
- `DashDotLine` - * - * line.
- `DashDotDotLine` - ** - ** line.

See also: style().

void QPen::setWidth (uint w)

Sets the pen width to *w*.

See also: width().

Examples:
progress/progress.cpp

PenStyle QPen::style () const

Returns the pen style.

See also: setStyle().

uint QPen::width () const

Returns the pen width.

See also: setWidth().

131.3. Related Functions

QDataStream & operator<< (QDataStream & s, const QPen & p)

Writes a pen to the stream and returns a reference to the stream.

The serialization format is:

1 The pen style (UINT8)

2 The pen width (UINT8)

3 The pen color (QColor)

QDataStream & operator>> (QDataStream & s, QPen & p)

Reads a pen from the stream and returns a reference to the stream.

131.3.1. URL

Generated from http://www.troll.no/qt/qpen.html.

132. QPicture Class Reference

`#include <qpicture.h>`

Inherits QPaintDevice.

132.1. Detailed Description

The QPicture class is a paint device that records and replays QPainter commands.

A picture serializes painter commands to an IO device in a platform-independent format. A picture created under Windows can be read on a Sun SPARC.

Pictures are called meta-files on some platforms.

Qt pictures use a proprietary binary format. Unlike native picture (meta-file) formats on many window systems, Qt pictures have no limitations regarding the contents. Everything that can be painted can also be stored in a picture (fonts, pixmaps, regions, transformed graphics etc.)

Example of how to record a picture:

```
QPicture  pic;
QPainter  p;
p.begin( &pic );                             // paint in picture
p.drawEllipse( 10,20, 80,70 );               // draw an ellipse
p.end();                                     // painting done
pic.save( "drawing.pic" );                   // save picture
```

Example of how to replay a picture:

```
QPicture  pic;
pic.load( "drawing.pic" );                   // load picture
QPainter  p;
p.begin( &myWidget );                        // paint in myWidget
p.drawPicture( pic );                        // draw the picture
p.end();                                     // painting done
```

Examples:
picture/picture.cpp

132.2. Member Function Documentation

QPicture::QPicture ()

Constructs an empty picture.

QPicture::~QPicture ()

Destroys the picture.

bool QPicture::cmd (int c, QPainter *, QPDevCmdParam * p) `[virtual protected]`

For internal use only.

Reimplemented from QPaintDevice.

const char * QPicture::data () const

Returns a pointer to the picture data. The returned pointer is null if the picture contains no data.

See also: size() and isNull().

bool QPicture::isNull () const

Returns TRUE if the picture contains no data, otherwise FALSE.

bool QPicture::load (const char * fileName)

Loads a picture from the file specified by *fileName* and returns TRUE if successful, otherwise FALSE.

See also: save().

Examples:
picture/picture.cpp

int QPicture::metric (int m) const `[virtual protected]`

Internal implementation of the virtual QPaintDevice::metric() function.

Use the QPaintDeviceMetrics class instead.

A picture has the following hard coded values: width=640, height=480. widthMM=236, heightMM=176, numcolors=16777216 and depth=24.

Reimplemented from QPaintDevice.

bool QPicture::play (QPainter * painter)

Replays the picture using *painter* and returns TRUE if successful, or FALSE if the internal picture data is inconsistent.

This function does exactly the same as QPainter::drawPicture().

bool QPicture::save (const char * fileName)

Saves a picture to the file specified by *fileName* and returns TRUE if successful, otherwise FALSE.

See also: load().

Examples:
picture/picture.cpp

void QPicture::setData (const char * data, uint size)

Sets the picture data directly from *data* and *size.* This function copies the input data.

See also: data() and size().

uint QPicture::size () const

Returns the size of the picture data.

See also: data().

132.2.1. URL

Generated from http://www.troll.no/qt/qpicture.html.

133. QPixmap Class Reference

`#include <qpixmap.h>`

Inherits QPaintDevice.

Inherited by QBitmap.

133.1. Detailed Description

The QPixmap class is an off-screen pixel-based paint device.

It is one of the two classes Qt provides for dealing with images, the other being QImage. QPixmap is designed and optimized for drawing; QImage is designed and optimized for I/O and for direct pixel access/manipulation. There are (slow) functions to convert between QImage and QPixmp; convertToImage() and convertFromImage().

One common use of the QPixmap class is to enable smooth updating of widgets. Whenever something complex needs to be drawn, you can use a pixmap to obtain flicker-free drawing, like this:

1. Create a pixmap with the same size as the widget.
2. Fill the pixmap with the widget background color.
3. Paint the pixmap.
4. bitBlt() the pixmap contents onto the widget.

Pixel data in a pixmap is internal and managed by the underlying window system. Pixels can only be accessed through QPainter functions, through bitBlt(), and by converting the QPixmap to a QImage.

You can display a QPixmap on the screen easily using e.g. QLabel::setPixmap(), and all the button classes support pixmap use.

There are also convenience functions to get and set single pixels and to load and save the entire pixmap; these work by converting the pixmap to a QImage internally.

The QPixmap class is optimized by the use of implicit sharing, so it is very efficient to pass QPixmap objects as arguments.

See also: QBitmap, QImage, QImageIO and Shared Classes

Examples:

qtimage/qtimage.cpp
grapher/grapher.cpp
desktop/desktop.cpp
hello/hello.cpp
movies/main.cpp
picture/picture.cpp
xform/xform.cpp
application/application.cpp
menu/menu.cpp
qmag/qmag.cpp
widgets/widgets.cpp

133.2. Member Function Documentation

QPixmap::QPixmap ()

Constructs a null pixmap.

See also: isNull().

QPixmap::QPixmap (const QPixmap & pixmap)

Constructs a pixmap which is a copy of *pixmap.*

QPixmap::QPixmap (const char * fileName, const char * format, int conversion_flags)

Constructs a pixmap from the file *fileName.* If the file does not exist, or is of an unknown format, the pixmap becomes a null pixmap.

The parameters are passed on to load().

See also: isNull(), load(), loadFromData(), save() and imageFormat().

QPixmap::QPixmap (const char * fileName, const char * format=0, ColorMode mode=Auto)

Constructs a pixmap from the file *fileName.* If the file does not exist, or is of an unknown format, the pixmap becomes a null pixmap.

The parameters are passed on to load().

See also: isNull(), load(), loadFromData(), save() and imageFormat().

QPixmap::QPixmap (const char * xpm[])

Constructs a pixmap from *xpm,* which must be a valid XPM image.

QPixmap::QPixmap (int w, int h, const uchar * bits, bool isXbitmap) `[protected]`

Constructs a monochrome pixmap which is initialized with the data in *bits.* This constructor is protected and used by the QBitmap class.

QPixmap::QPixmap (int w, int h, int depth=-1)

Constructs a pixmap with *w* width, *h* height and of *depth* bits per pixels.

The contents of the pixmap is uninitialized.

The *depth* can be either 1 (monochrome) or the depth of the current video mode. If *depth* is negative, then the hardware depth of the current video mode will be used.

If either *width* or *height* is zero, a null pixmap is constructed.

See also: isNull().

QPixmap::QPixmap (const QSize & size, int depth=-1)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

QPixmap::˜QPixmap ()

Destroys the pixmap.

bool QPixmap::convertFromImage (const QImage & img, int conversion_flags)

Converts an image and sets this pixmap. Returns TRUE if successful.

The *conversion_flags* argument is a bitwise-OR from the following choices. The options marked *(default)* are the choice if no other choice from the list is included (they are zero):

Color/Mono preference (ignored for QBitmap)

- `AutoColor` (default) - If the *image* has depth 1 and contains only black and white pixels, then the pixmap becomes monochrome.
- `ColorOnly` - The pixmap is dithered/converted to the native display depth.
- `MonoOnly` - The pixmap becomes monochrome. If necessary, it is dithered using the chosen dithering algorithm.

Dithering mode preference, for RGB channels

- `DiffuseDither` (default) - a high quality dither
- `OrderedDither` - a faster more ordered dither
- `ThresholdDither` - no dithering, closest color is used

Dithering mode preference, for alpha channel

- `DiffuseAlphaDither` - a high quality dither
- `OrderedAlphaDither` - a faster more ordered dither
- `ThresholdAlphaDither` (default) - no dithering

Color matching versus dithering preference

- `PreferDither` - always dither 32-bit images when the image is being converted to 8-bits. This is the default when converting to a pixmap.
- `AvoidDither` - only dither 32-bit images if the image has more than 256 colours and it is being converted to 8-bits. This is the default when an image is converted for the purpose of saving to a file.

Passing 0 for *conversion_flags* gives all the default options.

Note that even though a QPixmap with depth 1 behaves much like a QBitmap, isQBitmap() returns FALSE.

See also: convertToImage(), isQBitmap(), QImage::convertDepth(), defaultDepth() and hasAlphaBuffer().

Bugs and limitations:

- Does not support 2 or 4 bit display hardware.

Examples:
qtimage/qtimage.cpp

bool QPixmap::convertFromImage (const QImage & img, ColorMode mode=Auto)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

QImage QPixmap::convertToImage () const

Converts the pixmap to an image. Returns a null image if the operation failed.

If the pixmap has 1 bit depth, the returned image will also be 1 bits deep. If the pixmap has 2-8 bit depth, the returned image has 8 bit depth. If the pixmap has greater than 8 bit depth, the returned image has 32 bit depth.

See also: convertFromImage().

Bugs and limitations:

- Does not support 2 or 4 bit display hardware.
- Alpha masks on monochrome images are ignored.

Examples:
qmag/qmag.cpp

QBitmap QPixmap::createHeuristicMask (bool clipTight = TRUE) const

Creates and returns a heuristic mask for this pixmap. It works by selecting a color from one of the corners, then chipping away pixels of that color, starting at all the edges.

The mask may not be perfect but should be reasonable, so you can do things like:

```
pm->setMask( pm->createHeuristicMask() );
```

This function is slow because it involves transformation to a QImage, non-trivial computations and a transformation back to QBitmap.

See also: QImage::createHeuristicMask().

int QPixmap::defaultDepth () `[static]`

Returns the default pixmap depth, i.e. the depth a pixmap gets if -1 is specified.

See also: depth().

Examples:
showimg/showimg.cpp

QPixmap::Optimization QPixmap::defaultOptimization() `[static]`

Returns the default pixmap optimization setting.

See also: setDefaultOptimization(), setOptimization() and optimization().

int QPixmap::depth () const

Returns the depth of the image.

The pixmap depth is also called bits per pixel (bpp) or bit planes of a pixmap. A null pixmap has depth 0.

See also: defaultDepth(), isNull() and QImage::convertDepth().

void QPixmap::detach () `[virtual]`

Special-purpose function that detaches the pixmap from shared pixmap data.

A pixmap is automatically detached by Qt whenever its contents is about to change. This is done in all QPixmap member functions that modify the pixmap (fill(), resize(), convertFromImage(), load() etc.), in bitBlt() for the destination pixmap and in QPainter::begin() on a pixmap.

It is possible to modify a pixmap without letting Qt know. You can first obtain the system-dependent handle and then call system-specific functions (for instance BitBlt under Windows) that modifies the pixmap contents. In this case, you can call detach() to cut the pixmap loose from other pixmaps that share data with this one.

detach() returns immediately if there is just a single reference or if the pixmap has not been initialized yet.

void QPixmap::fill (const QColor & fillColor=white)

Fills the pixmap with the color *fillColor*.

Examples:
grapher/grapher.cpp
desktop/desktop.cpp
hello/hello.cpp
xform/xform.cpp

void QPixmap::fill (const QWidget * widget, const QPoint & ofs)

Fills the pixmap with the widget's background color or pixmap. If the background is empty, nothing is done.

The *ofs* point is an offset in the widget.

The point *ofs* is a point in the widget's coordinate system. The pixmap's top left pixel will be mapped to the point *ofs* in the widget. This is significant if the widget has a background pixmap, otherwise the pixmap will simply be filled with the background color of the widget.

Example:

```
void CuteWidget::paintEvent( QPaintEvent *e )
{
  QRect ur = e->rect();               // rectangle to update
  QPixmap  pix( ur.size() );          // Pixmap for double-buffering
  pix.fill( this, ur.topLeft() );     // fill with widget background
  QPainter p( &pix );
  p.translate( -ur.x(), -ur.y() );    // use widget coordinate system
                                      // when drawing on pixmap
  //    ... draw on pixmap ...
  p.end();
  bitBlt( this, ur.topLeft(), &pix );
}
```

void QPixmap::fill (const QWidget * widget, int xofs, int yofs)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

QPixmap QPixmap::grabWindow (WId window, int x=0, int y=0, int w=-1, int h=-1) `[static]`

Grabs the contents of a window and makes a pixmap out of it. Returns the pixmap.

The argments *(x,y)* specify the offset in the window, while *(w,h)* specify the width and height of the area to be copied.

If *w* is negative, the function copies everything to the right border of the window. If *h* is negative, the function copies everything to the bottom of the window.

Note that grabWindows() grabs pixels from the screen, not from the window. This means that If there is another window partially or entirely over the one you grab, you get pixels from the overlying window too.

The reason we use a window identifier and not a QWidget is to enable grabbing of windows that are not part of the application.

Warning: Grabbing an area outside the window, or screen, is not safe in general. This depends on the underlying window system.

int QPixmap::height () const

Returns the height of the pixmap.

See also: width(), size() and rect().

Examples:
qtimage/qtimage.cpp
desktop/desktop.cpp
movies/main.cpp
xform/xform.cpp

const char * QPixmap::imageFormat (const char * fileName) `[static]`

Returns a string that specifies the image format of the file *fileName,* or null if the file cannot be read or if the format cannot be recognized.

The QImageIO documentation lists the supported image formats.

See also: load() and save().

bool QPixmap::isNull () const

Returns TRUE if it is a null pixmap.

A null pixmap has zero width, zero height and no contents. You cannot draw in a null pixmap or bitBlt() anything to it.

Resizing an existing pixmap to (0,0) makes a pixmap into a null pixmap.

See also: resize().

Examples:
qmag/qmag.cpp

bool QPixmap::isQBitmap () const

Returns TRUE if this is a QBitmap, otherwise FALSE.

bool QPixmap::load (const char * fileName, const char * format, int conversion_flags)

Loads a pixmap from the file *fileName.* Returns TRUE if successful, or FALSE if the pixmap could not be loaded.

If *format* is specified, the loader attempts to read the pixmap using the specified format. If *format* is not specified (default), the loader reads a few bytes from the header to guess the file format.

See the convertFromImage() documentation for a description of the *conversion_flags* argument.

The QImageIO documentation lists the supported image formats and explains how to add extra formats.

See also: loadFromData(), save(), imageFormat(), QImage::load() and QImageIO.

Examples:
picture/picture.cpp
xform/xform.cpp
widgets/widgets.cpp

bool QPixmap::load (const char * fileName, const char * format=0, ColorMode mode=Auto)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

bool QPixmap::loadFromData (const uchar * buf, uint len, const char * format, int conversion_flags)

Loads a pixmap from the binary data in *buf* (*len* bytes). Returns TRUE if successful, or FALSE if the pixmap could not be loaded.

If *format* is specified, the loader attempts to read the pixmap using the specified format. If *format* is not specified (default), the loader reads a few bytes from the header to guess the file format.

See the convertFromImage() documentation for a description of the *conversion_flags* argument.

The QImageIO documentation lists the supported image formats and explains how to add extra formats.

See also: load(), save(), imageFormat(), QImage::loadFromData() and QImageIO.

bool QPixmap::loadFromData (QByteArray buf, const char * format=0, int conversion_flags=0)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

bool QPixmap::loadFromData (const uchar * buf, uint len, const char * format=0, ColorMode mode=Auto)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

const QBitmap * QPixmap::mask () const

Returns the mask bitmap, or null if no mask has been set.

See also: setMask() and QBitmap.

int QPixmap::metric (int m) const `[virtual protected]`

Internal implementation of the virtual QPaintDevice::metric() function.

Use the QPaintDeviceMetrics class instead.

Reimplemented from QPaintDevice.

QPixmap & QPixmap::operator= (const QImage & image)

Converts the image *image* to a pixmap that is assigned to this pixmap. Returns a reference to the pixmap.

See also: convertFromImage().

QPixmap & QPixmap::operator= (const QPixmap & pixmap)

Assigns the pixmap *pixmap* to this pixmap and returns a reference to this pixmap.

QPixmap::Optimization QPixmap::optimization() const

Returns the optimization setting for this pixmap.

The default optimization setting is `QPixmap::NormalOptim.` You may change this settings in two ways:

- Call setDefaultOptimization() to set the default optimization for all new pixmaps.
- Call setOptimization() to set a the optimization for individual pixmaps.

See also: setOptimization(), setDefaultOptimization() and defaultOptimization().

QRect QPixmap::rect () const

Returns the enclosing rectangle (0,0,width(),height()) of the pixmap.

See also: width(), height() and size().

Examples:
xform/xform.cpp

void QPixmap::resize (int w, int h)

Resizes the pixmap to *w* width and *h* height. If either *w* or *h* is less than 1, the pixmap becomes a null pixmap.

If both *w* and *h* are greater than 0, a valid pixmap is created. New pixels will be uninitialized (random) if the pixmap is expanded.

Examples:
grapher/grapher.cpp
desktop/desktop.cpp

void QPixmap::resize (const QSize & size)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

bool QPixmap::save (const char * fileName, const char * format) const

Saves the pixmap to the file *fileName,* using the image file format *format.* Returns TRUE if successful, or FALSE if the pixmap could not be saved.

See also: load(), loadFromData(), imageFormat(), QImage::save() and QImageIO.

Examples:
qmag/qmag.cpp

bool QPixmap::selfMask () const

Returns TRUE if the pixmap's mask is identical to the pixmap itself.

See also: mask().

int QPixmap::serialNumber () const

Returns a number that uniquely identifies this QPixmap object. The serial number is very useful for caching.

See also: QPixmapCache.

void QPixmap::setDefaultOptimization (Optimization optimization) `[static]`

Sets the default pixmap optimization.

All *new* pixmaps that are created will use this default optimization. You may also set optimization for individual pixmaps using the setOptimization() function.

The initial default optimization setting is `QPixmap::Normal.`

See also: defaultOptimization(), setOptimization() and optimization().

void QPixmap::setMask (const QBitmap & mask)

Sets a mask bitmap.

The *mask* bitmap defines the clip mask for this pixmap. Every pixel in *mask* corresponds to a pixel in this pixmap. Pixel value 1 means opaque and pixel value 0 means transparent. The mask must have the same size as this pixmap.

Setting a null mask resets the mask,

See also: mask(), createHeuristicMask() and QBitmap.

void QPixmap::setOptimization (Optimization optimization)

Sets pixmap drawing optimization for this pixmap.

The optimization setting affects pixmap operations, in particular drawing of transparent pixmaps (bitBlt() a pixmap with a mask set) and pixmap transformations (the xForm() function).

Pixmap optimization involves keeping intermediate results in a cache buffer and use the data in the cache to speed up bitBlt() and xForm(). The cost is more memory consumption, up to twice as much as an unoptimized pixmap.

The *optimization* parameter can be:

- `QPixmap::NoOptim,` avoid optimization. Little or no caching is done. Use this setting if memory is scarce and the speed of pixmap operations is not critical to your application.
- `QPixmap::NormalOptim,` normal optimization to make pixmap drawing faster. This option is the default and is suitable for most purposes.
- `QPixmap::BestOptim,` heavily optimized pixmap drawing. Use this option for pixmap drawn extremely frequently.

Use the setDefaultOptimization() to change the default optimization for all new pixmaps.

See also: optimization(), setDefaultOptimization() and defaultOptimization().

QSize QPixmap::size () const

Returns the size of the pixmap.

See also: width(), height() and rect().

Examples:
qtimage/qtimage.cpp
movies/main.cpp

QWMatrix QPixmap::trueMatrix (const QWMatrix & matrix, int w, int h) [static]

Returns the actual matrix used for transforming a pixmap with *w* width and *h* height.

When transforming a pixmap with xForm(), the transformation matrix is internally adjusted to compensate for unwanted translation, i.e. xForm() returns the smallest pixmap containing all transformed points of the original pixmap.

This function returns the modified matrix, which maps points correctly from the original pixmap into the new pixmap.

See also: xForm() and QWMatrix.

int QPixmap::width () const

Returns the width of the pixmap.

See also: height(), size() and rect().

Examples:
qtimage/qtimage.cpp
desktop/desktop.cpp
movies/main.cpp
xform/xform.cpp

QPixmap QPixmap::xForm (const QWMatrix & matrix) const

Returns a copy of the pixmap that is transformed using *matrix.*

Qt uses this function to implement rotated text on window systems that do not support such complex features.

Example of how to manually draw a rotated text at (100,200) in a widget:

```
char    *str = "Trolls R Qt";        // text to be drawn
QFont    f( "Charter", 24 );         // use Charter 24pt font
QPixmap  pm( 8, 8 );
QPainter p;
QRect    r;                          // text bounding rectangle
QPoint   bl;                         // text baselink position
p.begin( &pm );                      // first get the bounding
p.setFont( f );                      //   text rectangle
r = p.fontMetrics().boundingRect(str);
bl = -r.topLeft();                   // get baseline position
p.end();
pm.resize( r.size() );               // resize to fit the text
pm.fill( white );                    // fills pm with white
p.begin( &pm );                      // begin painting pm
p.setFont( f );                      // set the font
p.setPen( blue );                    // set blue text color
p.drawText( bl, str );               // draw the text
p.end();                             // painting done
QWMatrix m;                          // transformation matrix
m.rotate( -33.4 );                   // rotate coordinate system
QPixmap rp = pm.xForm( m );          // rp is rotated pixmap
QWMatrix t = QPixmap::trueMatrix( m, pm.width(), pm.height() );
int x, y;
t.map( bl.x(),bl.y(), &x,&y );       // get pm's baseline pos in rp
```

```
bitBlt( myWidget, 100-x, 200-y,     // blt rp into a widget
        &rp, 0, 0, -1, -1 );
```

This example outlines how Qt implements rotated text under X11. The font calculation is the most tedious part. The rotation itself is only 3 lines of code.

If you want to draw rotated text, you do not have to implement all the code above. The code below does exactly the same thing as the example above, except that it uses a QPainter.

```
char    *str = "Trolls R Qt";       // text to be drawn
QFont    f( "Charter", 24 );        // use Charter 24pt font
QPainter p;
p.begin( myWidget );
p.translate( 100, 200 );            // translates coord system
p.rotate( -33.4 );                  // rotates it counterclockwise
p.setFont( f );
p.drawText( 0, 0, str );
p.end();
```

See also: trueMatrix(), QWMatrix and QPainter::setWorldMatrix().

Bugs and limitations:

- 2 and 4 bits pixmaps are not supported.

Examples:

qtimage/qtimage.cpp
desktop/desktop.cpp
movies/main.cpp
xform/xform.cpp
qmag/qmag.cpp

133.3. Related Functions

QDataStream & operator<< (QDataStream & s, const QPixmap & pixmap)

Writes a pixmap to the stream as a BMP image.

See also: QPixmap::save().

QDataStream & operator>> (QDataStream & s, QPixmap & pixmap)

Reads a pixmap from the stream.

See also: QPixmap::load().

133.3.1. URL

Generated from http://www.troll.no/qt/qpixmap.html.

134. QPixmapCache Class Reference

```
#include <qpixmapcache.h>
```

134.1. Detailed Description

The QPixmapCache class provides an application-global cache for pixmaps.

This class is a tool for optimized drawing with QPixmaps. Here follows an example. The QRadioButton widget has a non-trivial visual representation. In the drawButton() function, we do not draw the radio button directly. Instead, we first check the global pixmap cache if a pixmap called "$qt_radio_s_" exists. The `s` is a numerical value that specifies the the radio button state. If a pixmap is found, we bitBlt() it onto the widget and return. Otherwise, we create a new pixmap, draw the radio button in the pixmap and finally insert the pixmap in the global pixmap cache, using the key above. The bitBlt() is 10 times faster than drawing the radio button. All radio buttons in the program share the cached pixmap since QPixmapCache is application-global.

QPixmapCache contains no member data, only static functions to access the global pixmap cache. It creates an internal QCache for caching the pixmaps.

The cache associates a pixmap with a normal string (key). If two pixmaps are inserted into the cache using equal keys, then the last pixmap will hide the first pixmap. The QDict and QCache classes do exactly the same.

The cache becomes full when the total size of all pixmaps in the cache exceeds the cache limit. The initial cache limit is 1024 KByte (1 MByte). A pixmap takes roughly width*height*depth/8 bytes of memory.

See the QCache documentation for a more details about the cache mechanism.

134.2. Member Function Documentation

int QPixmapCache::cacheLimit () `[static]`

Returns the cache limit (in kilobytes).

The default setting is 1024 kilobytes.

See also: setCacheLimit().

void QPixmapCache::clear () `[static]`

Removes all pixmaps from the cache.

QPixmap * QPixmapCache::find (const char * key) `[static]`

Returns the pixmap associated with *key* in the cache, or null if there is no such pixmap.

NOTE: if valid, you should copy the pixmap immediately (this is quick since QPixmaps are implicitly shared), because subsequent insertions into the cache could cause the pointer to become invalid. For this reason, we recommend you use find(const char*, QPixmap&) instead.

Example:

```
QPixmap* pp;
QPixmap p;
if ( (pp=QPixmapCache::find("my_previous_copy", pm)) ) {
    p = *pp;
} else {
```

```
        p.load("bigimage.gif");
        QPixmapCache::insert("my_previous_copy", new QPixmap(p));
    }
    painter->drawPixmap(0, 0, p);
```

bool QPixmapCache::find (const char * key, QPixmap & pm) `[static]`

Sets *pm* to the cached pixmap associated with *key* in the cache and returns TRUE. If FALSE is returned, no cached copy was found, and *pm* is unchanged.

Example:

```
    QPixmap p;
    if ( !QPixmapCache::find("my_previous_copy", pm) ) {
        pm.load("bigimage.gif");
        QPixmapCache::insert("my_previous_copy", pm);
    }
    painter->drawPixmap(0, 0, p);
```

bool QPixmapCache::insert (const char * key, QPixmap * pm) `[static]`

Inserts the pixmap *pm* associated with *key* into the cache. Returns TRUE if successful, or FALSE if the pixmap is too big for the cache.

NOTE: If this function returns FALSE, you must delete *pm* yourself. Additionally, be very careful about using *pm* after calling this function, as any other insertions into the cache, from anywhere in the application, or within Qt itself, could cause the pixmap to be discarded from the cache, and the pointer to become invalid.

Due to these dangers, we strongly recommend that you use insert(const char*, const QPixmap&) instead.

void QPixmapCache::insert (const char * key, const QPixmap & pm) `[static]`

Inserts a copy of the pixmap *pm* associated with *key* into the cache. Returns TRUE if successful, or FALSE if the pixmap is too big for the cache.

All pixmaps inserted by the Qt library have a key starting with "$qt..". Use something else for you pixmaps.

When a pixmap is inserted and the cache is about to exceed its limit, it removes pixmaps until there is enough room for the pixmap to be inserted.

The oldest pixmaps (least recently accessed in the cache) are deleted when more space is needed.

See also: setCacheLimit().

void QPixmapCache::setCacheLimit (int n) `[static]`

Sets the cache limit to *n* kilobytes.

The default setting is 1024 kilobytes.

See also: cacheLimit().

134.2.1. URL

Generated from http://www.troll.no/qt/qpixmapcache.html.

135. QPoint Class Reference

```
#include <qpoint.h>
```

135.1. Detailed Description

The QPoint class defines a point in the plane.

A point is specified by an x coordinate and a y coordinate.

The coordinate type is QCOORD (defined in qwindowdefs.h as `short`). The minimum value of QCOORD is QCOORD_MIN (-32768) and the maximum value is QCOORD_MAX (32767).

We have defined many operator functions that make arithmetic on points simple and intuitive.

Example:

```
QPoint p(  1, 2 );
QPoint q( -8, 5 );
QPoint r(  9, 7 );
QPoint x = 2*p + (q-r)*5.5 - (r+p/1.5);
```

See also: QSize and QRect.

Examples:

tictac/tictac.cpp
table/table.cpp
forever/forever.cpp
connect/connect.cpp
drawdemo/drawdemo.cpp
xform/xform.cpp
aclock/aclock.cpp

135.2. Member Function Documentation

QPoint::QPoint ()

Constructs a point with undefined x and y values.

QPoint::QPoint (int xpos, int ypos)

Constructs a point with the x value *xpos* and y value *ypos.*

bool QPoint::isNull () const

Returns TRUE if both the x value and the y value are 0.

QPoint & QPoint::operator*= (double c)

Multiplies both x and y with *c,* and return a reference to this point.

Example:

```
QPoint p( -1, 4 );
p *= 2.5;                  // p becomes (-3,10)
```

Note that the result is truncated.

QPoint & QPoint::operator*= (int c)

Multiplies both x and y with *c,* and return a reference to this point.

Example:

```
QPoint p( -1, 4 );
p *= 2;                    // p becomes (-2,8)
```

QPoint & QPoint::operator+= (const QPoint & p)

Adds *p* to the point and returns a reference to this point.

Example:

```
QPoint p(  3, 7 );
QPoint q( -1, 4 );
p += q;                    // p becomes (2,11)
```

QPoint & QPoint::operator-= (const QPoint & p)

Subtracts *p* from the point and returns a reference to this point.

Example:

```
QPoint p(  3, 7 );
QPoint q( -1, 4 );
p -= q;                    // p becomes (4,3)
```

QPoint & QPoint::operator/= (double c)

Divides both x and y by *c,* and return a reference to this point.

Example:

```
QPoint p( -3, 10 );
p /= 2.5;                  // p becomes (-1,4)
```

Note that the result is truncated.

QPoint & QPoint::operator/= (int c)

Divides both x and y by *c,* and return a reference to this point.

Example:

```
QPoint p( -2, 8 );
p /= 2;                    // p becomes (-1,4)
```

QCOORD & QPoint::rx ()

Returns a reference to the x coordinate of the point.

Using a reference makes it possible to directly manipulate x.

Example:

```
QPoint p( 1, 2 );
p.rx()--;                  // p becomes (0,2)
```

See also: ry().

QCOORD & QPoint::ry ()

Returns a reference to the y coordinate of the point.

Using a reference makes it possible to directly manipulate y.

Example:

```
QPoint p( 1, 2 );
p.ry()++;                   // p becomes (1,3)
```

See also: rx().

void QPoint::setX (int x)

Sets the x coordinate of the point to *x*.

See also: setY().

void QPoint::setY (int y)

Sets the y coordinate of the point to *y*.

See also: setX().

int QPoint::x () const

Returns the x coordinate of the point.

See also: y().

Examples:
table/table.cpp
drawdemo/drawdemo.cpp
aclock/aclock.cpp

int QPoint::y () const

Returns the y coordinate of the point.

See also: x().

Examples:
table/table.cpp
drawdemo/drawdemo.cpp
aclock/aclock.cpp

135.3. Related Functions

bool operator== (const QPoint & p1, const QPoint & p2)

Returns TRUE if *p1* and *p2* are equal, or FALSE if they are different.

bool operator!= (const QPoint & p1, const QPoint & p2)

Returns TRUE if *p1* and *p2* are different, or FALSE if they are equal.

QPoint operator+ (const QPoint & p1, const QPoint & p2)

Returns the sum of *p1* and *p2;* each component is added separately.

QPoint operator- (const QPoint & p1, const QPoint & p2)

Returns *p2* subtracted from *p1;* each component is subtracted separately.

QPoint operator* (const QPoint & p, int c)

Multiplies both of *p's* components by *c* and returns the result.

QPoint operator* (int c, const QPoint & p)

Multiplies both of *p's* components by *c* and returns the result.

QPoint operator* (const QPoint & p, double c)

Multiplies both of *p's* components by *c* and returns the result.

QPoint operator* (double c, const QPoint & p)

Multiplies both of *p's* components by *c* and returns the result.

QPoint operator- (const QPoint & p)

Returns *p* where x and y have opposite signs.

QPoint operator/ (const QPoint & p, int c)

Divides both of *p's* components by *c* and returns the result.

QPoint operator/ (const QPoint & p, double c)

Divides both of *p's* components by *c* and returns the result.

Note that the result is truncated.

QDataStream & operator<< (QDataStream & s, const QPoint & p)

Writes a QPoint to the stream and returns a reference to the stream.

Serialization format: [x (INT16), y (INT16)].

QDataStream & operator>> (QDataStream & s, QPoint & p)

Reads a QPoint from the stream and returns a reference to the stream.

135.3.1. URL

Generated from http://www.troll.no/qt/qpoint.html.

136. QPointArray Class Reference

`#include <qpointarray.h>`

Inherits QArray.

136.1. Detailed Description

The QPointArray class provides an array of points.

QPointArray is used by the QPainter to draw line segments, polylines, polygons and Bezier curves.

The QPointArray is not an array of QPoint, instead it contains a platform dependent point type to make the QPainter functions more efficient (no conversion needed). On the other hand, QPointArray has member functions that operate on QPoint to make programming easier.

Note that since this class is a QArray, it is explicitly shared and works with shallow copies by default.

Examples:
desktop/desktop.cpp
drawdemo/drawdemo.cpp
picture/picture.cpp
aclock/aclock.cpp

136.2. Member Function Documentation

QPointArray::QPointArray ()

Constructs a null point array.

See also: isNull().

QPointArray::QPointArray (const QRect & r, bool closed=FALSE)

Constructs a point array from the rectangle *r*.

If *closed* is FALSE, then the point array will contain the following four points (in the listed order):

1 r.topLeft()

2 r.topRight()

3 r.bottomRight()

4 r.bottomLeft()

If *closed* is TRUE, then a fifth point is set to r.topLeft() to close the point array.

QPointArray::QPointArray (const QPointArray & a)

Constructs a shallow copy of the point array *a*.

See also: copy().

QPointArray::QPointArray (int nPoints, const QCOORD * points)

Constructs a point array with *nPoints* points, taken from the *points* array.

Equivalent to setPoints(nPoints,points).

QPointArray::QPointArray (int size)

Constructs a point array with room for *size* points. Makes a null array if *size* == 0.

See also: resize() and isNull().

QPoint QPointArray::at (uint index) const

Returns the point at position *index* in the array.

See also: operator[].

QRect QPointArray::boundingRect () const

Returns the bounding rectangle of the points in the array, or QRect(0,0,0,0) if the array is empty.

QPointArray QPointArray::copy () const

Creates a deep copy of the array.

bool QPointArray::fill (const QPoint & p, int size = -1)

Fills the point array with the point *p*. If *size* is specified as different from -1, then the array will be resized before filled.

Returns TRUE if successful, or FALSE if the memory cannot be allocated (only when *size* != -1).

See also: resize().

void QPointArray::makeArc (int x, int y, int w, int h, int a1, int a2)

Sets the points of the array to those describing an arc of an ellipse with size *w* by *h* and position (*x, y*), starting from angle \a1, spanning *a2*. Angles are specified in 16ths of a degree, i.e. a full circle equals 5760 (16*360). Positive values mean counter-clockwise while negative values mean clockwise direction. Zero degrees is at the 3'o clock position.

void QPointArray::makeEllipse (int xx, int yy, int w, int h)

Sets the points of the array to those describing an ellipse with size *w* by *h* and position (*x, y*).

QPointArray & QPointArray::operator= (const QPointArray & a)

Assigns a shallow copy of *a* to this point array and returns a reference to this point array.

Equivalent to assign(a).

See also: copy().

QPoint QPointArray::operator[] (int i) const

Returns the point at position *index* in the array.

QPoint QPointArray::operator[] (uint i) const

Returns the point at position *index* in the array.

QPointVal QPointArray::operator[] (int index)

Returns a reference to the point at position *index* in the array.

QPointVal QPointArray::operator[] (uint index)

Returns a reference to the point at position *index* in the array.

QPoint QPointArray::point (uint index) const

Returns the point at position *index* in the array.

void QPointArray::point (uint index, int * x, int * y) const

Returns the point at position *index* in the array in **x* and **y*.

Examples:
drawdemo/drawdemo.cpp

bool QPointArray::putPoints (int index, int nPoints, const QCOORD * points)

Copies *nPoints* points from the *points* array into this point array. Will resize this point array if `index+nPoints` exceeds the size of the array.

Returns TRUE if successful, or FALSE if the array could not be resized.

Example:

```
QPointArray a( 1 );
a[0] = QPoint( 1, 2 );
static QCOORD points[] = { 3,4, 5,6 };
a.putPoints( 1, 2, points );
```

The example code creates an array with three points: (1,2), (3,4) and (5,6).

This function differs from setPoints() in that it does not resize the array unless the array size is exceeded.

See also: resize() and setPoints().

bool QPointArray::putPoints (int index, int nPoints, int firstx, int firsty, ...)

Copies *nPoints* points from the variable argument list into this point array. Will resize this point array if `index+nPoints` exceeds the size of the array.

Returns TRUE if successful, or FALSE if the array could not be resized.

Example:

```
QPointArray a( 1 );
a[0] = QPoint( 1, 2 );
a.putPoints( 1, 2, 3,4, 5,6 );
```

The example code creates an array with two points (1,2), (3,4) and (5,6).

This function differs from setPoints() because it does not resize the array unless the array size is exceeded.

See also: resize() and setPoints().

QPointArray QPointArray::quadBezier () const

Returns the Bezier points for the four control points in this array.

Examples:
drawdemo/drawdemo.cpp

void QPointArray::setPoint (uint i, const QPoint & p)

Equivalent to setPoint(i, p.x(), p.y()).

void QPointArray::setPoint (uint index, int x, int y)

Sets the point at position *index* in the array to *(x,y)*.

Examples:
drawdemo/drawdemo.cpp

bool QPointArray::setPoints (int nPoints, const QCOORD * points)

Resizes the array to *nPoints* and sets the points in the array to the values taken from *points*.

Returns TRUE if successful, or FALSE if the array could not be resized.

Example:

```
static QCOORD points[] = { 1,2, 3,4 };
QPointArray a;
a.setPoints( 2, points );
```

The example code creates an array with two points (1,2) and (3,4).

See also: resize() and putPoints().

Examples:
aclock/aclock.cpp

bool QPointArray::setPoints (int nPoints, int firstx, int firsty, ...)

Resizes the array to *nPoints* and sets the points in the array to the values taken from the variable argument list.

Returns TRUE if successful, or FALSE if the array could not be resized.

Example:

```
QPointArray a;
a.setPoints( 2, 1,2, 3,4 );
```

The example code creates an array with two points (1,2) and (3,4).

See also: resize() and putPoints().

void QPointArray::translate (int dx, int dy)

Translates all points in the array *(dx,dy)*.

136.3. Related Functions

QDataStream & operator<< (QDataStream & s, const QPointArray & a)

Writes a point array to the stream and returns a reference to the stream.

The serialization format is:

1 The array size (UINT32)

2 The array points (QPoint)

QDataStream & operator>> (QDataStream & s, QPointArray & a)

Reads a point array from the stream and returns a reference to the stream.

136.3.1. URL

Generated from http://www.troll.no/qt/qpointarray.html.

137. QPointVal Class Reference

```
#include <qpointarray.h>
```

137.1. Detailed Description

The QPointVal class is an internal class, used with QPointArray.

The QPointVal is required by the indexing [] operator on point arrays. It is probably a bad idea to use it in any other context.

137.2. Member Function Documentation

QPointVal::QPointVal (QPointData * ptr)

Constructs a reference to an element in a QPointArray. This is what QPointArray::operator[] contructs its return value with.

QPointVal::operator QPoint () const

Returns the value of the point referenced by this QPointVal.

QPointVal& QPointVal::operator+= (const QPoint & point)

Vector-adds a given point to the value referenced by this QPointVal.

QPointVal& QPointVal::operator-= (const QPoint & point)

Vector-subtracts a given point to the value referenced by this QPointVal.

QPointVal& QPointVal::operator= (const QPoint & point)

Assigns a given point to the value referenced by this QPointVal.

int QPointVal::x () const

Returns the X coordinate of the point referenced by this QPointVal.

int QPointVal::y () const

Returns the Y coordinate of the point referenced by this QPointVal.

137.2.1. URL

Generated from http://www.troll.no/qt/qpointval.html.

138. QPopupMenu Class Reference

```
#include <qpopupmenu.h>
```

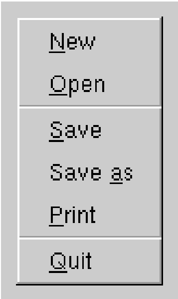
Inherits QMenuData and QTableView.

138.1. Detailed Description

The QPopupMenu class provides a popup menu widget.

The popup widget is different from other widgets in the way it relates to the parent widget.

menu/menu.cpp is a typical example of QMenuBar and QPopupMenu use.



See also: QMenuBar and GUI Design Handbook: Menu, Drop-Down and Pop-Up

Examples:

qtimage/qtimage.cpp
grapher/grapher.cpp
xform/xform.cpp
application/application.cpp
menu/menu.cpp
progress/progress.cpp
layout/layout.cpp
showimg/showimg.cpp
widgets/widgets.cpp

138.2. Member Function Documentation

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

Constructs a popup menu with a null parent and a widget name.

A popup menu must be a top level widget, i.e. parent must be 0. This argument is present merely for API uniformity.

QPopupMenu::˜QPopupMenu ()

Destroys the popup menu.

void QPopupMenu::aboutToShow () `[signal]`

This signal is emited 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: setItemEnabled(), setItemChecked(), insertItem() and removeItem().

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 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 handy in such cases.

See also: highlighted() and QMenuData::insertItem().

void QPopupMenu::activatedRedirect (int id) `[signal]`

For internal use only.

int QPopupMenu::cellHeight (int row) `[virtual protected]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QTableView.

int QPopupMenu::cellWidth (int col) `[virtual protected]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QTableView.

int QPopupMenu::exec ()

Execute this popup synchronously.

Similar to the above function, but the position of the popup is not set, so you must choose an appropriate position. The function move the popup if it is partially off-screen.

More 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)));
```

int QPopupMenu::exec (const QPoint & pos, int indexAtPoint = 0)

Execute 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, 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)));
```

See also: popup().

void QPopupMenu::hide () `[virtual]`

Reimplements QWidget::hide() for internal purposes.

Reimplemented from QWidget.

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 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 handy in such cases.

See also: activated() and QMenuData::insertItem().

void QPopupMenu::highlightedRedirect (int id) `[signal]`

For internal use only.

bool QPopupMenu::isCheckable () const

Returns whether display of check marks by the menu items is enabled.

See also: setCheckable() and QMenuData::setItemChecked().

void QPopupMenu::keyPressEvent (QKeyEvent * e) `[virtual protected]`

Handles key press events for the popup menu.

Reimplemented from QWidget.

void QPopupMenu::mouseMoveEvent (QMouseEvent * e) `[virtual protected]`

Handles mouse move events for the popup menu.

Reimplemented from QWidget.

void QPopupMenu::mousePressEvent (QMouseEvent * e) `[virtual protected]`

Handles mouse press events for the popup menu.

Reimplemented from QWidget.

void QPopupMenu::mouseReleaseEvent (QMouseEvent * e) `[virtual protected]`

Handles mouse release events for the popup menu.

Reimplemented from QWidget.

void QPopupMenu::paintCell (QPainter * p, int row, int col) `[virtual protected]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QTableView.

void QPopupMenu::paintEvent (QPaintEvent * e) `[virtual protected]`

Handles paint events for the popup menu.

Reimplemented from QWidget.

void QPopupMenu::popup (const QPoint & pos, int indexAtPoint = 0)

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().

Examples:
qtimage/qtimage.cpp

void QPopupMenu::setActiveItem (int i)

Sets the currently active item to *i* and repaints as necessary.

void QPopupMenu::setCheckable (bool enable)

Enables or disables display of check marks by the menu items.

Notice that checking is always enabled when in windows-style.

See also: isCheckable() and QMenuData::setItemChecked().

Examples:
grapher/grapher.cpp
progress/progress.cpp

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

Reimplements QWidget::setFont() to be able to refresh the popup menu when its font changes.

Reimplemented from QWidget.

void QPopupMenu::show () `[virtual]`

Reimplements QWidget::show() for internal purposes.

Reimplemented from QWidget.

void QPopupMenu::timerEvent (QTimerEvent * e) `[virtual protected]`

Handles timer events for the popup menu.

Reimplemented from QObject.

138.2.1. URL

Generated from http://www.troll.no/qt/qpopupmenu.html.

139. QPrintDialog Class Reference

`#include <qprintdialog.h>`

Inherits QDialog.

139.1. Detailed Description

The QPrintDialog class provides a dialog for specifying print-out details.

It encompasses both the sort of details needed for doing a simple print-out and some print configuration setup.

At present, the only easy way to use the class is through the static function getPrinterSetup(). You can however also call the global QPrintDialot::getPrinterConfigure(), or subclass in order to extend one of the group boxes.

Note that in 1.40 the printer dialog is a little too high for comfortable use on a small-screen machine. This will be improved on in 2.0.

<img width="479" height="540" src="printerdialog.gif">
The printer dialog, on a large screen, in Motif style.

139.2. Member Function Documentation

QPrintDialog::QPrintDialog (QPrinter * prn, QWidget * parent=0, const char * name=0)

Creates a new modal printer dialog that configures *prn* and is a child of *parent* named *name.*

QPrintDialog::~QPrintDialog ()

Destroys the object and frees any allocated resources. Does not delete the associated QPrinter object.

bool QPrintDialog::getPrinterSetup (QPrinter * p) `[static]`

Display a dialog and allow the user to configure the QPrinter *p.* Returns TRUE if the user clicks OK or presses Enter, FALSE if the user clicks Cancel or presses Escape.

getPrinterSetup() remembers the settings and provides the same settings the next time the dialog is shown.

QPrinter * QPrintDialog::printer () const

Returns a pointer to the printer this dialog configures, or 0 if this dialog does not operate on any printer.

void QPrintDialog::setPrinter (QPrinter * p, bool pickUpSettings = FALSE)

Sets this dialog to configure *p,* or no printer if *p* is FALSE. If *pickUpSettings* is TRUE, the dialog reads most of its settings from *printer.* If *pickUpSettings* is FALSE (the default) the dialog keeps its old settings.

139.2.1. URL

Generated from http://www.troll.no/qt/qprintdialog.html.

140. QPrinter Class Reference

`#include <qprinter.h>`

Inherits QPaintDevice.

140.1. Detailed Description

The QPrinter class is a paint device that prints graphics on a printer.

All window systems that Qt supports, except X11, have built-in printer drivers. For X11, Qt provides PostScript (tm) printing.

Clipping and even-odd filling are not supported.

Font support is limited to Times, Helvetica, Courier and Symbol, and font metrics for these four fonts can be inaccurate.

Drawing graphics on a printer is almost identical to drawing graphics in a widget or a pixmap. The only difference is that the programmer must think about dividing the document into pages and handling abort commands.

The default coordinate system of a printer is a 72 dpi (dots per inch) system with (0,0) at the upper left corner, with increasing values to the right and downwards. This causes printer output to be roughly the same size as screen output on most screens. You can easily change the coordinate system using QPainter::setViewport(). QPainter::setWindow() and/or QPainter::setWorldMatrix().

The newPage() function should be called to finish the current page and start printing a new page.

If the user decides to abort printing, aborted() will return TRUE. The QPrinter class handles abortion automatically, but the programmer should from time to time check the aborted() flag and stop painting if the print job has been aborted.

Example (a complete application):

```
#include <qapplication.h>
#include <qpainter.h>
#include <qprinter.h>
#include <qprintdialog.h>
int main( int argc, char **argv )
{
    QApplication a( argc, argv );
    QPrinter prt;
    if ( QPrintDialog::getPrinterSetup( &prt ) ) {
        QPainter p;
        p.begin( &prt );
        p.rotate( 55 );
        p.setFont( QFont("times", 144, QFont::Bold) );
        p.drawText( 80,30, "Hello, world!" );
        p.end();
    }
    return 0;
}
```

Examples:
    drawdemo/drawdemo.cpp
    application/application.cpp

140.2. Member Function Documentation

QPrinter::QPrinter ()

Constructs a printer paint device.

QPrinter::~QPrinter ()

Destroys the printer paint device and cleans up.

bool QPrinter::abort ()

Aborts the print job. Returns TRUE if successful, otherwise FALSE.

See also: aborted().

bool QPrinter::aborted () const

Returns TRUE is the printer job was aborted, otherwise FALSE.

See also: abort().

bool QPrinter::cmd (int c, QPainter * paint, QPDevCmdParam * p) `[virtual protected]`

For internal use only.

Reimplemented from QPaintDevice.

QPrinter::ColorMode QPrinter::colorMode() const

Returns the current color mode. The default color more is `Color.`

See also: setColorMode().

const char * QPrinter::creator () const

Returns the creator name.

See also: setCreator().

const char * QPrinter::docName () const

Returns the document name.

See also: setDocName().

int QPrinter::fromPage () const

Returns the from-page setting. The default value is 0.

The programmer is responsible for reading this setting and print accordingly.

See also: setFromTo() and toPage().

int QPrinter::maxPage () const

Returns the max-page setting. The default value is 0.

See also: minPage() and setMinMax().

int QPrinter::metric (int m) const `[virtual protected]`

Internal implementation of the virtual QPaintDevice::metric() function.

Use the QPaintDeviceMetrics class instead.

Reimplemented from QPaintDevice.

int QPrinter::minPage () const

Returns the min-page setting. The default value is 0.

See also: maxPage() and setMinMax().

bool QPrinter::newPage ()

Advances to a new page on the printer. Returns TRUE if successful, otherwise FALSE.

int QPrinter::numCopies () const

Returns the number of copies to be printed. The default value is 1.

See also: setNumCopies().

Orientation QPrinter::orientation () const

Returns the orientation setting. The default value is `QPrinter::Portrait.`

See also: setOrientation().

const char * QPrinter::outputFileName () const

Returns the name of the output file. There is no default file name.

See also: setOutputFileName() and setOutputToFile().

bool QPrinter::outputToFile () const

Returns TRUE if the output should be written to a file, or FALSE if the output should be sent directly to the printer. The default setting is FALSE.

This function is currently only supported under X11.

See also: setOutputToFile() and setOutputFileName().

QPrinter::PageOrder QPrinter::pageOrder() const

Returns the current page order.

The default page order is *FirstPageFirst.*

PageSize QPrinter::pageSize () const

Returns the printer page size. The default value is `QPrinter::A4.`

See also: setPageSize().

const char * QPrinter::printProgram () const

Returns the name of the program that sends the print output to the printer.

The default print program is "lpr" under X11. This function returns 0 for all other window systems.

See also: setPrintProgram().

const char * QPrinter::printerName () const

Returns the printer name. This value is initially set to the name of the default printer.

See also: setPrinterName().

void QPrinter::setColorMode (ColorMode newColorMode)

Sets the printer's color mode to *newColorMode,* which can be one of `Color` (the default) and `GrayScale.`

A future version of Qt will modify its printing accordingly. At present, QPrinter behaves as if `Color` is selected.

See also: colorMode().

void QPrinter::setCreator (const char * creator)

Sets the creator name.

Calling this function only has effect for the X11 version of Qt. The creator name is the name of the application that created the document. If no creator name is specified, then the creator will be set to "Qt".

See also: creator().

void QPrinter::setDocName (const char * name)

Sets the document name.

void QPrinter::setFromTo (int fromPage, int toPage)

Sets the from page and to page.

The from-page and to-page settings specify what pages to print.

See also: fromPage(), toPage(), setMinMax() and setup().

void QPrinter::setMinMax (int minPage, int maxPage)

Sets the min page and max page.

The min-page and max-page restrict the from-page and to-page settings. When the printer setup dialog comes up, the user cannot select from and to that are outsize the range specified by min and max pages.

See also: minPage(), maxPage(), setFromTo() and setup().

void QPrinter::setNumCopies (int numCopies)

Sets the number of pages to be printed.

The printer driver reads this setting and prints the specified number of copies.

See also: numCopies() and setup().

void QPrinter::setOrientation (Orientation orientation)

Sets the print orientation.

The orientation can be either `QPrinter::Portrait` or `QPrinter::Landscape.`

The printer driver reads this setting and prints using the specified orientation.

See also: orientation().

void QPrinter::setOutputFileName (const char * fileName)

Sets the name of the output file.

Setting a null name (0 or "") disables output to a file, i.e. calls setOutputToFile(FALSE); Setting non-null name enables output to a file, i.e. calls setOutputToFile(TRUE).

This function is currently only supported under X11.

See also: outputFileName() and setOutputToFile().

void QPrinter::setOutputToFile (bool enable)

Specifies whether the output should be written to a file or sent directly to the printer.

Will output to a file if *enable* is TRUE, or will output directly to the printer if *enable* is FALSE.

This function is currently only supported under X11.

See also: outputToFile() and setOutputFileName().

void QPrinter::setPageOrder (PageOrder newPageOrder)

Sets the page order to *newPageOrder.*

The page order can be `QPrinter::FirstPageFirst` or `QPrinter::LastPageFirst.` The application programmer is responsible for reading the page order and printing accordingly.

void QPrinter::setPageSize (PageSize newPageSize)

Sets the printer page size to *newPageSize.*

The page size can be one of

- `QPrinter::A0` (841 x 1189 mm)
- `QPrinter::A1` (594 x 841 mm)
- `QPrinter::A2` (420 x 594 mm)
- `QPrinter::A3` (297 x 420 mm)
- `QPrinter::A4` (210x297 mm, 8.26x11.7 inches)
- `QPrinter::A5` (148 x 210 mm)
- `QPrinter::A6` (105 x 148 mm)
- `QPrinter::A7` (74 x 105 mm)
- `QPrinter::A8` (52 x 74 mm)
- `QPrinter::A9` (37 x 52 mm)
- `QPrinter::B0` (1030 x 1456 mm)
- `QPrinter::B1` (728 x 1030 mm)
- `QPrinter::B10` (32 x 45 mm)

- `QPrinter::B2` (515 x 728 mm)
- `QPrinter::B3` (364 x 515 mm)
- `QPrinter::B4` (257 x 364 mm)
- `QPrinter::B5` (182x257 mm, 7.17x10.13 inches)
- `QPrinter::B6` (128 x 182 mm)
- `QPrinter::B7` (91 x 128 mm)
- `QPrinter::B8` (64 x 91 mm)
- `QPrinter::B9` (45 x 64 mm)
- `QPrinter::C5E` (163 x 229 mm)
- `QPrinter::Comm10E` (105 x 241 mm, US Common #10 Envelope)
- `QPrinter::DLE` (110 x 220 mm)
- `QPrinter::Executive` (7.5x10 inches, 191x254 mm)
- `QPrinter::Folio` (210 x 330 mm)
- `QPrinter::Ledger` (432 x 279 mm)
- `QPrinter::Legal` (8.5x14 inches, 216x356 mm)
- `QPrinter::Letter` (8.5x11 inches, 216x279 mm)
- `QPrinter::Tabloid` (279 x 432 mm)

See also: pageSize().

void QPrinter::setPrintProgram (const char * printProg)

Sets the name of the program that should do the print job.

If an output file has been defined, the printer driver will print to the output file instead of directly to the printer.

On X11, this function sets the program to call with the PostScript output. On other platforms, it has no effect.

See also: printProgram().

void QPrinter::setPrinterName (const char * name)

Sets the printer name.

The default printer will be used if no printer name is set.

Under X11, the PRINTER environment variable defines the default printer. Under any other window system, the window system defines the default printer.

See also: printerName().

bool QPrinter::setup (QWidget * parent = 0)

Opens a printer setup dialog and asks the user to specify what printer to use and miscellaneous printer settings.

Now obsoleted by QPrintDialog::getPrinterSetup().

Returns TRUE if the user pressed "Ok" to print, or FALSE if the user cancelled the operation.

Examples:
drawdemo/drawdemo.cpp

int QPrinter::toPage () const

Returns the to-page setting. The default value is 0.

The programmer is responsible for reading this setting and print accordingly.

See also: setFromTo() and fromPage().

140.2.1. URL

Generated from http://www.troll.no/qt/qprinter.html.

141. QProgressBar Class Reference

```
#include <qprogressbar.h>
```

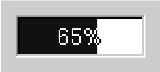
Inherits QFrame.

141.1. Detailed Description

The QProgressBar widget provides a horizontal progress bar.

A progress bar is used to give the user an indication of progress of an operation. To reassure them that the application has not crashed.

QProgressBar only implements the basic progress display, while QProgressDialog provides a fuller encapsulation.



See also: QProgressDialog and GUI Design Handbook: Progress Indicator

141.2. Member Function Documentation

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

Constructs a progress bar.

The total number of steps is set to 100 by default.

parent, name and *f* are sent to the QFrame::QFrame() constructor.

See also: setTotalSteps().

QProgressBar::QProgressBar (int totalSteps, QWidget * parent=0, const char * name=0, WFlags f=0)

Constructs a progress bar.

totalSteps is the total number of steps in the operation of which this progress bar shows the progress. For example, if the operation is to examine 50 files, this value would be 50, then before examining the first file, call setProgress(0), and after examining the last file call setProgress(50).

parent, name and *f* are sent to the QFrame::QFrame() constructor.

See also: setTotalSteps() and setProgress().

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

Handles paint events for the progress bar. In WindowsStyle, colorGroup().text() and QApplication::winStyleHighlightColor() are used. In MotifStyle, colorGroup().base() is also used.

Reimplemented from QFrame.

int QProgressBar::progress () const

Returns the current amount of progress, or -1 if the progress counting has not started.

See also: setProgress().

void QProgressBar::reset () `[slot]`

Reset the progress bar. The progress bar 'rewinds'.

bool QProgressBar::setIndicator (QString & indicator, int progress, int totalSteps) `[virtual protected]`

This method is called to generate the text displayed in the center of the progress bar.

The progress may be negative, indicating that the bar is in the "reset" state before any progress is set.

The default implementation it is the percentage of completion or blank in the reset state.

This method should return FALSE if the string is unchanged since the last call to the method, to allow efficient repainting of the progress bar.

void QProgressBar::setProgress (int progress) `[slot]`

Sets the current amount of progress made to *progress* units of the total number of steps.

See also: progress() and totalSteps().

void QProgressBar::setTotalSteps (int totalSteps) `[slot]`

Sets the total number of steps to *totalSteps*.

See also: totalSteps().

QSize QProgressBar::sizeHint () const `[virtual]`

Returns a size which fits the contents of the progress bar.

Reimplemented from QWidget.

int QProgressBar::totalSteps () const

Returns the total number of steps.

See also: setTotalSteps().

141.2.1. URL

Generated from http://www.troll.no/qt/qprogressbar.html.

142. QProgressDialog Class Reference

#include <qprogressdialog.h>

Inherits QSemiModal.

142.1. Detailed Description

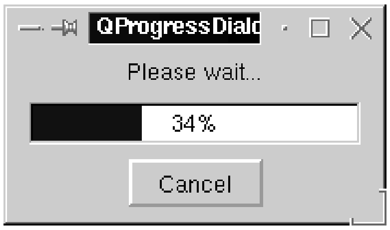
Provides feedback on the progress of a slow operation.

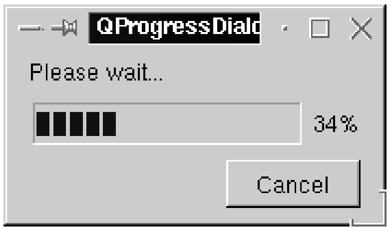
A progress dialog is used to give the user an indication of how long an operation is going to take to perform, and to reassure them that the application has not frozen.

A potential problem with progress dialogs is that it is difficult to know when to use them, as 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 3 seconds.

Example:

```
QProgressDialog progress( "Copying files...", "Abort Copy", numFiles, this );
for (int i=0; i<numFiles; i++) {
    progress.setProgress( i );
    if ( progress.wasCancelled() )
        break;
    ... // copy one file
}
progress.setProgress( numFiles );
```





See also: QDialog, QProgressBar and GUI Design Handbook: Progress Indicator

Examples:

progress/progress.cpp

142.2. 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.

parent, name, modal, and *f* are sent to the QSemiModal::QSemiModal() 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 it is TRUE, the dialog ensures events are processed when needed.

See also: setLabelText(), setLabel(), setCancelButtonText(), setCancelButton() and setTotalSteps().

QProgressDialog::QProgressDialog (const char * labelText, const char * cancelButtonText, int totalSteps, QWidget * creator=0, const char * name=0, bool modal=FALSE, WFlags f=0)

Constructs a progress dialog.

Arguments:

- *labelText* is text telling the user what is progressing.
- *cancelButtonText* is the text on the cancel button, or 0 if no cancel button is to be shown.
- *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, then before examining the first file, call setProgress(0), and after examining the last file call setProgress(50).
- *name, modal,* and *f* are sent to the QSemiModal::QSemiModal() 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 it is TRUE, the dialog ensures events are processed when needed.

See also: setLabelText(), setLabel(), setCancelButtonText(), setCancelButton() and setTotalSteps().

QProgressDialog::~QProgressDialog ()

Destroys the progress dialog.

void QProgressDialog::cancel () `[slot]`

Reset 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().

int QProgressDialog::minimumDuration () const `[slot]`

Returns the currently set minimum duration for the QProgressDialog

See also: setMinimumDuration().

int QProgressDialog::progress () const

Returns the current amount of progress, or -1 if the progress counting has not started.

See also: setProgress().

void QProgressDialog::reset () `[slot]`

Reset the progress dialog. The progress dialog becomes hidden.

void QProgressDialog::resizeEvent (QResizeEvent *) `[virtual protected]`

Handles resize events for the progress dialog, sizing the label, dialog, and cancellation button.

Reimplemented from QWidget.

void QProgressDialog::setBar (QProgressBar * bar)

Sets the progress bar widget. The progress dialog resizes to fit. The progress bar becomes owned by the progress dialog and will be deleted when necessary.

void QProgressDialog::setCancelButton (QPushButton * cancelButton)

Sets the cancellation button. The button 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: setCancelButtonText().

void QProgressDialog::setCancelButtonText (const char * cancelButtonText) `[slot]`

Sets the cancellation button text.

See also: setCancelButton().

void QProgressDialog::setLabel (QLabel * label)

Sets the 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: setLabelText().

Examples:
progress/progress.cpp

void QProgressDialog::setLabelText (const char * text) `[slot]`

Sets the label text. The progress dialog resizes to fit.

See also: setLabel().

void QProgressDialog::setMinimumDuration (int ms) `[slot]`

The dialog will not appear if the anticipated duration of the progressing task is less than *ms* milliseconds.

If set to 0 the dialog is always shown as soon as any progress is set.

void QProgressDialog::setProgress (int progress) `[slot]`

Sets the current amount of progress made to *prog* units of the total number of steps. For the progress dialog to work correctly, you must at least call this with the parameter 0 initially, then later with QProgressDialog::totalSteps(), and you may call it 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 to your code. For example, don't use a QProgressDialog inside a paintEvent()!

See also: progress().

Examples:
progress/progress.cpp

void QProgressDialog::setTotalSteps (int totalSteps) `[slot]`

Sets the total number of steps.

See also: totalSteps() and QProgressBar::setTotalSteps().

QSize QProgressDialog::sizeHint () const `[virtual]`

Returns a size which fits the contents of the progress dialog. The progress dialog resizes itself as required, so this should not be needed in user code.

Reimplemented from QWidget.

void QProgressDialog::styleChange (GUIStyle s) `[virtual protected]`

Ensures layout conforms to style of GUI.

Reimplemented from QWidget.

int QProgressDialog::totalSteps () const

Returns the total number of steps.

See also: setTotalSteps() and QProgressBar::totalSteps().

Examples:
progress/progress.cpp

bool QProgressDialog::wasCancelled () const

Returns the TRUE if the dialog was cancelled, otherwise FALSE.

See also: setProgress(), cancel() and cancelled().

Examples:
progress/progress.cpp

142.2.1. URL

Generated from http://www.troll.no/qt/qprogressdialog.html.

143. QPtrDict Class Reference

`#include <qptrdict.h>`

Inherits QGDict.

143.1. Detailed Description

The QPtrDict class is a template class that provides a dictionary based on `void*` keys.

QPtrDict is implemented as both a template and a macro class. Define a template instance QPtrDict<X> to create a dictionary that operates on pointers to X, or X*.

A dictionary is a collection that associates an item with a key. The key is used for inserting and looking up an item. QPtrDict has `void*` keys.

The dictionary has very fast insertion and lookup.

Example:

```
#include <qptrdict.h>
#include <stdio.h>
void main()
{
    int *a = new int[12];
    int *b = new int[10];
    int *c = new int[18];
    int *d = new int[13];
    QPtrDict<char> dict;                // maps void* -> char*
    dict.insert( a, "a is int[12]" );  // descripe pointers
    dict.insert( b, "b is int[10]" );
    dict.insert( c, "c is int[18]" );
    printf( "%s\n", dict[a] );         // print descriptions
    printf( "%s\n", dict[b] );
    printf( "%s\n", dict[c] );
    if ( !dict[d] )
        printf( "d not in dictionary\n" );
}
```

Program output:

```
a is int[12]
b is int[10]
c is int[18]
d not in dictionary
```

The dictionary in our example maps `int*` keys to `char*` items. QPtrDict implements the [] operator to lookup an item.

A QPtrDict can also be instantiated through a macro expansion, but this is necessary only for compilers that do not support templates. See the collection classes documentation for a general discussion on template-based versus macro-based collections.

QPtrDict is implemented by QGDict as a hash array with a fixed number of entries. Each array entry points to a singly linked list of buckets, in which the dictionary items are stored.

When an item is inserted with a key, the key is converted (hashed) to an integer index into the hash array using the `mod` operation. The item is inserted before the first bucket in the list of buckets.

Looking up an item is normally very fast. The key is again hashed to an array index. Then QPtrDict scans the list of buckets and returns the item found or null if the item was not found. You cannot insert null pointers into a dictionary.

The size of the hash array is very important. In order to get good performance, you should use a suitably large prime number. Suitable means equal to or larger than the maximum expected number of dictionary items.

Items with equal keys are allowed. When inserting two items with the same key, only the last inserted item will be visible (last in, first out) until it is removed.

Example:

```
#include <qptrdict.h>
#include <stdio.h>
void main()
{
    QPtrDict<char> dict;            // maps char* ==> char*
    double *ptr = new double[28];
    dict.insert( ptr, "first" );
    dict.insert( ptr, "second" );
    printf( "%s\n", dict[ptr] );
    dict.remove( ptr );
    printf( "%s\n", dict[ptr] );
}
```

Program output:

```
second
first
```

The QPtrDictIterator class can traverse the dictionary contents, but only in an arbitrary order. Multiple iterators may independently traverse the same dictionary.

Calling setAutoDelete(TRUE) for a dictionary tells it to delete items that are removed . The default is to not delete items when they are removed.

When inserting an item into a dictionary, only the pointer is copied, not the item itself. This is called a shallow copy. It is possible to make the dictionary copy all of the item's data (known as a deep copy) when an item is inserted. insert() calls the virtual function QCollection::newItem() for the item to be inserted. Inherit a dictionary and reimplement it if you want deep copies.

When removing a dictionary item, the virtual function QCollection::deleteItem() is called. QPtrDict's default implementation is to delete the item if auto-deletion is enabled.

See also: QPtrDictIterator, QDict and Collection Classes

143.2. Member Function Documentation

QPtrDict::QPtrDict (const QPtrDict<type> & dict)

Constructs a copy of *dict.*

Each item in *dict* are inserted into this dictionary. Only the pointers are copied (shallow copy).

QPtrDict::QPtrDict (int size=17)

Constructs a dictionary using an internal hash array with the size *size.*

Setting *size* to a suitably large prime number (equal to or greater than the expected number of entries) makes the hash distribution better and hence the loopup faster.

QPtrDict::˜QPtrDict ()

Removes all items from the dictionary and destroys it.

All iterators that access this dictionary will be reset.

See also: setAutoDelete().

void QPtrDict::clear () `[virtual]`

Removes all items from the dictionary.

The removed items are deleted if auto-deletion is enabled.

All dictionary iterators that access this dictionary will be reset.

See also: remove(), take() and setAutoDelete().

Reimplemented from QCollection.

uint QPtrDict::count () const `[virtual]`

Returns the number of items in the dictionary.

See also: isEmpty().

Reimplemented from QCollection.

type * QPtrDict::find (void * key) const

Returns the item associated with *key,* or null if the key does not exist in the dictionary.

This function uses an internal hashing algorithm to optimize lookup.

If there are two or more items with equal keys, then the last inserted of these will be found.

Equivalent to the [] operator.

See also: operator[].

void QPtrDict::insert (void * key, const type * item)

Inserts the *key* with the *item* into the dictionary.

The key does not have to be a unique dictionary key. If multiple items are inserted with the same key, only the last item will be visible.

Null items are not allowed.

See also: replace().

bool QPtrDict::isEmpty () const

Returns TRUE if the dictionary is empty, i.e. count() == 0. Returns FALSE otherwise.

See also: count().

QPtrDict<type> & QPtrDict::operator= (const QPtrDict<type> & dict)

Assigns *dict* to this dictionary and returns a reference to this dictionary.

This dictionary is first cleared, then each item in *dict* is inserted into this dictionary. Only the pointers are copied (shallow copy), unless newItem() has been reimplemented().

type * QPtrDict::operator[] (void * key) const

Returns the item associated with *key,* or null if the key does not exist in the dictionary.

This function uses an internal hashing algorithm to optimize lookup.

If there are two or more items with equal keys, then the last inserted of these will be found.

Equivalent to the find() function.

See also: find().

bool QPtrDict::remove (void * key)

Removes the item associated with *key* from the dictionary. Returns TRUE if successful, or FALSE if the key does not exist in the dictionary.

If there are two or more items with equal keys, then the last inserted of these will be removed.

The removed item is deleted if auto-deletion is enabled.

All dictionary iterators that refer to the removed item will be set to point to the next item in the dictionary traversing order.

See also: take(), clear() and setAutoDelete().

void QPtrDict::replace (void * key, const type * item)

Replaces an item which has a key equal to *key* with *item.*

If the item does not already exist, it will be inserted.

Null items are not allowed.

Equivalent to:

```
QPtrDict<char> dict;
    ...
if ( dict.find(key) )
    dict.remove( key );
dict.insert( key, item );
```

If there are two or more items with equal keys, then the last inserted of these will be replaced.

See also: insert().

void QPtrDict::resize (uint newsize)

Changes the size of the hashtable the *newsize.* The contents of the dictionary are preserved, but all iterators on the dictionary become invalid.

uint QPtrDict::size () const

Returns the size of the internal hash array (as specified in the constructor).

See also: count().

void QPtrDict::statistics () const

Debugging-only function that prints out the dictionary distribution using debug().

type * QPtrDict::take (void * key)

Takes the item associated with *key* out of the dictionary without deleting it (even if auto-deletion is enabled).

If there are two or more items with equal keys, then the last inserted of these will be taken.

Returns a pointer to the item taken out, or null if the key does not exist in the dictionary.

All dictionary iterators that refer to the taken item will be set to point to the next item in the dictionary travering order.

See also: remove(), clear() and setAutoDelete().

143.2.1. URL

Generated from http://www.troll.no/qt/qptrdict.html.

144. QPtrDictIterator Class Reference

`#include <qptrdict.h>`

Inherits QGDictIterator.

144.1. Detailed Description

The QPtrDictIterator class provides an iterator for QPtrDict collections.

QPtrDictIterator is implemented as both a template and a macro class. Define a template instance QPtrDictIterator<X> to create a dictionary iterator that operates on QPtrDict<X> (dictionary of X*).

Example:

```
#include <qptrdict.h>
#include <stdio.h>
void main()
{
    int *a = new int[12];
    int *b = new int[10];
    int *c = new int[18];
    int *d = new int[13];
    QPtrDict<char> dict;               // maps void* -> char*
    dict.insert( a, "a is int[12]" );  // descripe pointers
    dict.insert( b, "b is int[10]" );
    dict.insert( c, "c is int[18]" );
    QPtrDictIterator<char> it( dict ); // iterator for dict
    while ( it.current() ) {
        printf( "%x -> %s\n", it.currentKey(), it.current() );
        ++it;
    }
}
```

Program output:

```
804a788 -> a is int[12]
804a7f0 -> c is int[18]
804a7c0 -> b is int[10]
```

Note that the traversal order is arbitrary, you are not guaranteed the order above.

A QPtrDictIterator can also be instantiated through a macro expansion, but this is necessary only for compilers that do not support templates. See the collection classes documentation for a general discussion on template-based versus macro-based collections.

Multiple iterators may independently traverse the same dictionary. A QPtrDict knows about all iterators that are operating on the dictionary. When an item is removed from the dictionary, QPtrDict update all iterators that are referring the removed item to point to the next item in the traversing order.

See also: QPtrDict and Collection Classes

144.2. Member Function Documentation

QPtrDictIterator::QPtrDictIterator (const QPtrDict<type> & dict)

Constructs an iterator for *dict.* The current iterator item is set to point on the first item in the *dict.*

QPtrDictIterator::~QPtrDictIterator ()

Destroys the iterator.

QPtrDictIterator::operator type * () const

Cast operator. Returns a pointer to the current iterator item. Same as current().

uint QPtrDictIterator::count () const

Returns the number of items in the dictionary this iterator operates on.

See also: isEmpty().

type * QPtrDictIterator::current () const

Returns a pointer to the current iterator item.

void * QPtrDictIterator::currentKey () const

Returns the key for the current iterator item.

bool QPtrDictIterator::isEmpty () const

Returns TRUE if the dictionary is empty, i.e. count() == 0. Returns FALSE otherwise.

See also: count().

type * QPtrDictIterator::operator() ()

Makes the succeeding item current and returns the original current item.

If the current iterator item was the last item in the dictionary or if it was null, null is returned.

type * QPtrDictIterator::operator++ ()

Prefix ++ makes the succeeding item current and returns the new current item.

If the current iterator item was the last item in the dictionary or if it was null, null is returned.

type * QPtrDictIterator::operator+= (uint jump)

Sets the current item to the item *jump* positions after the current item, and returns a pointer to that item.

If that item is beyond the last item or if the dictionary is empty, it sets the current item to null and returns null.

type * QPtrDictIterator::toFirst ()

Sets the current iterator item to point to the first item in the dictionary and returns a pointer to the item. If the dictionary is empty it sets the current item to null and returns null.

144.2.1. URL

Generated from http://www.troll.no/qt/qptrdictiterator.html.

145. QPushButton Class Reference

```
#include <qpushbutton.h>
```

Inherits QButton.

145.1. Detailed Description

The QPushButton widget provides a push button with a text or pixmap label.

A default push button in a dialog emits the clicked signal if the user presses the Enter key.

A push button has `TabFocus` as a default focusPolicy(), i.e. it can get keyboard focus by tabbing but not by clicking.

See also: QRadioButton, QToolButton and GUI Design Handbook: Push Button

Examples:

tictac/tictac.cpp
drawdemo/drawdemo.cpp
xform/xform.cpp
layout/layout.cpp
qmag/qmag.cpp
widgets/widgets.cpp

145.2. Member Function Documentation

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

Constructs a push button with no text.

The *parent* and *name* arguments are sent to the QWidget constructor.

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

Constructs a push button with a text.

The *parent* and *name* arguments are sent to the QWidget constructor.

bool QPushButton::autoDefault () const

Returns TRUE if the button is an auto-default button.

See also: setAutoDefault().

void QPushButton::drawButton (QPainter * paint) `[virtual protected]`

Draws the push button, except its label.

See also: drawButtonLabel().

Reimplemented from QButton.

void QPushButton::drawButtonLabel (QPainter * paint) `[virtual protected]`

Draws the push button label.

See also: drawButton().

Reimplemented from QButton.

void QPushButton::focusInEvent (QFocusEvent * e) `[virtual protected]`

Handles focus in events for the push button.

Reimplemented from QWidget.

bool QPushButton::isDefault () const

Returns TRUE if the button is default.

See also: setDefault().

bool QPushButton::isMenuButton () const

Returns TRUE if this button indicates to the user that pressing it will pop up a menu, and FALSE otherwise. The default is FALSE.

See also: setIsMenuButton().

void QPushButton::move (const QPoint & p)

Reimplements QWidget::move() for internal purposes.

void QPushButton::move (int x, int y)

Reimplements QWidget::move() for internal purposes.

Examples:
drawdemo/drawdemo.cpp

void QPushButton::resize (const QSize & s)

Reimplements QWidget::resize() for internal purposes.

void QPushButton::resize (int w, int h)

Reimplements QWidget::resize() for internal purposes.

Examples:
drawdemo/drawdemo.cpp

void QPushButton::setAutoDefault (bool enable)

Sets the push buttons to an auto-default button if *enable* is TRUE, or to a normal button if *enable* is FALSE.

An auto-default button becomes the default push button automatically when it receives the keyboard input focus.

See also: autoDefault() and setDefault().

void QPushButton::setDefault (bool enable)

Sets the button to be the default button if *enable* is TRUE, or to be a normal button if *enable* is FALSE.

A default push button in a dialog emits the QButton::clicked() signal if the user presses the Enter key. Only one push button in the dialog can be default.

Default push buttons are only allowed in dialogs.

See also: isDefault(), setAutoDefault() and QDialog.

Examples:
xform/xform.cpp

void QPushButton::setGeometry (const QRect & r)

Reimplements QWidget::setGeometry() for internal purposes.

void QPushButton::setGeometry (int x, int y, int w, int h) `[virtual]`

Reimplements QWidget::setGeometry() for internal purposes.

Examples:
xform/xform.cpp
qmag/qmag.cpp

Reimplemented from QWidget.

void QPushButton::setIsMenuButton (bool enable)

Tells this button to draw a menu indication triangle if *enable* is TRUE, and to not draw one if *enable* is FALSE (the default).

setIsMenuButton() does not cause the button to do anything other than draw the menu indication.

See also: isMenuButton().

void QPushButton::setOn (bool enable) `[slot]`

Switches a toggle button on if *enable* is TRUE or off if *enable* is FALSE.

See also: isOn(), toggle(), toggled() and isToggleButton().

Examples:
qmag/qmag.cpp

void QPushButton::setToggleButton (bool enable)

Makes the push button a toggle button if *enable* is TRUE, or a normal push button if *enable* is FALSE.

Toggle buttons have an on/off state similar to check boxes. A push button is initially not a toggle button.

See also: setOn(), toggle(), isToggleButton() and toggled().

Examples:
qmag/qmag.cpp

QSize QPushButton::sizeHint () const `[virtual]`

Returns a size which fits the contents of the push button.

Examples:

layout/layout.cpp
widgets/widgets.cpp

Reimplemented from QWidget.

void QPushButton::toggle () `[slot]`

Toggles the state of a toggle button.

See also: isOn(), setOn(), toggled() and isToggleButton().

145.2.1. URL

Generated from http://www.troll.no/qt/qpushbutton.html.

146. QQueue Class Reference

```
#include <qqueue.h>
```

146.1. Detailed Description

The QQueue class is a template class that provides a queue.

QQueue is implemented as both a template and a macro class. Define a template instance QQueue<X> to create a queue that operates on pointers to X, or X*.

A stack is a collection that can be used to enqueue and dequeue items.

See also: Collection Classes

146.2. Member Function Documentation

QQueue::QQueue ()

Creates an empty queue.

QQueue::QQueue (const QQueue<type> & queue)

Creates a queue from another.

Only the pointers are copied.

QQueue::˜QQueue ()

Destroys the list. Items in the list are destroyed if autoDelete() is TRUE.

QQueue::operator type* () const

Returns a reference to the first item in the queue. The queue is not changed.

type* QQueue::current () const

Returns a reference to the first item in the queue. The queue is not changed.

type* QQueue::dequeue ()

Removes the front item from the queue. Last in, last out.

void QQueue::enqueue (const type * d)

Adds an item to the queue. Last in, last out.

type* QQueue::head () const

Returns a reference to the first item in the queue. The queue is not changed.

bool QQueue::isEmpty () const

Returns TRUE if there are no items to be dequeued

QQueue<type>& QQueue::operator= (const QQueue<type> & queue)

Assigns *queue* to this queue and returns a reference to this queue.

This queue is first cleared, then each item in *queue* is enqueued to this queue. Only the pointers are copied (shallow copy).

bool QQueue::remove ()

Removes the front item from the queue. Last in, last out. The item is deleted if autoDelete() is TRUE. Returns TRUE if there was an item to remove.

146.2.1. URL

Generated from http://www.troll.no/qt/qqueue.html.

147. QRadioButton Class Reference

`#include <qradiobutton.h>`

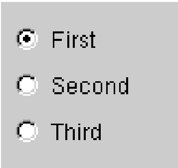
Inherits QButton.

147.1. Detailed Description

The QRadioButton widget provides a radio button with a text label.

QRadioButton and QCheckBox are both toggle buttons, that is, they can be switched on (checked) or off (unchecked). Unlike check boxes, radio buttons are normally organized in groups where only one radio button can be switched on at a time.

The QButtonGroup widget is very useful for defining groups of radio buttons.



See also: QPushButton, QToolButton and GUI Design Handbook: Radio Button

Examples:

drawdemo/drawdemo.cpp
xform/xform.cpp
pref/pref.cpp
widgets/widgets.cpp

147.2. Member Function Documentation

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

Constructs a radio button with no text.

The *parent* and *name* arguments are sent to the QWidget constructor.

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

Constructs a radio button with a text.

The *parent* and *name* arguments are sent to the QWidget constructor.

void QRadioButton::drawButton (QPainter * paint) `[virtual protected]`

Draws the radio button, but not the button label.

See also: drawButtonLabel().

Reimplemented from QButton.

void QRadioButton::drawButtonLabel (QPainter * p) `[virtual protected]`

Draws the radio button label.

See also: drawButton().

Reimplemented from QButton.

bool QRadioButton::hitButton (const QPoint & pos) const `[virtual protected]`

Reimplements QButton::hitButton(). This function is implemented to prevent a radio button that is on from being switched off.

Reimplemented from QButton.

bool QRadioButton::isChecked () const

Returns TRUE if the radio button is checked, or FALSE if it is not checked.

See also: setChecked().

void QRadioButton::keyPressEvent (QKeyEvent * e) `[virtual protected]`

Obsolete; to be removed in Qt 2.0.

Reimplemented from QWidget.

void QRadioButton::mouseReleaseEvent (QMouseEvent * e) `[virtual protected]`

Obsolete; to be removed in Qt 2.0.

Reimplemented from QWidget.

void QRadioButton::setChecked (bool check)

Checks the radio button if *check* is TRUE, or unchecks it if *check* is FALSE.

Calling this function does not affect other radio buttons unless a radio button group has been defined using the QButtonGroup widget.

See also: isChecked().

Examples:
drawdemo/drawdemo.cpp
widgets/widgets.cpp

QSize QRadioButton::sizeHint () const `[virtual]`

Returns a size which fits the contents of the radio button.

Examples:
widgets/widgets.cpp

Reimplemented from QWidget.

147.2.1. URL

Generated from http://www.troll.no/qt/qradiobutton.html.

148. QRangeControl Class Reference

`#include <qrangecontrol.h>`

Inherited by QScrollBar, QSlider and QSpinBox.

148.1. Detailed Description

The QRangeControl class provides an integer value within a range.

This class has many functions to manipulate a value inside a range. It was specifically designed for the QScrollBar widget, but it can also be practical for other purposes.

The three virtual functions valueChange(), rangeChange() and stepChange() can be reimplemented in a subclass to detect range control changes.

148.2. Member Function Documentation

QRangeControl::QRangeControl ()

Constructs a range control with min value 0, max value 99, line step 1, page step 10 and initial value 0.

QRangeControl::QRangeControl (int minValue, int maxValue, int lineStep, int pageStep, int value)

Constructs a range control with the specified parameters.

void QRangeControl::addLine ()

Equivalent to

```
setValue( value()+lineStep() )
```

See also: subtractLine().

void QRangeControl::addPage ()

Equivalent to

```
setValue( value()+pageStep() )
```

See also: subtractPage().

void QRangeControl::directSetValue (int value) `[protected]`

Sets the range control value directly without calling valueChange().

Adjusts the value if it is less than the min value or greater than the max value.

See also: setValue().

int QRangeControl::lineStep () const

Returns the current line step.

See also: setSteps() and pageStep().

int QRangeControl::maxValue () const

Returns the current maximum value in the range.

See also: setRange() and minValue().

int QRangeControl::minValue () const

Returns the current minimum value in the range.

See also: setRange() and maxValue().

int QRangeControl::pageStep () const

Returns the current page step.

See also: setSteps() and lineStep().

int QRangeControl::prevValue () const `[protected]`

Returns the previous range control value.

See also: value().

void QRangeControl::rangeChange () `[virtual protected]`

This virtual function is called whenever the range control range changes. You can reimplment it if you want to be notified when the range changes. The default implementation does nothing.

See also: setRange(), valueChange() and stepChange().

Reimplemented in QSlider and QScrollBar.

void QRangeControl::setRange (int minValue, int maxValue)

Sets the range min value to *minValue* and the max value to *maxValue.*

Calls the virtual rangeChange() function if the new min and max values are different from the previous setting. Calls the virtual valueChange() function if the current value is outside the new range and has to be adjusted.

See also: minValue() and maxValue().

Examples:
xform/xform.cpp

void QRangeControl::setSteps (int lineStep,int pageStep)

Sets the range line step to *lineStep* and page step to *pageStep.*

Calls the virtual stepChange() function if the new line step and/or page step are different from the previous setting.

See also: setRange().

void QRangeControl::setValue (int value)

Sets the range control value to *value.*

Adjusts the value if it is less than the min value or greater than the max value.

Calls the virtual valueChange() function if the value is different from the previous value.

void QRangeControl::stepChange () `[virtual protected]`

This virtual function is called whenever the range control step value changes. You can reimplment it if you want to be notified when the step changes. The default implementation does nothing.

See also: setSteps(), rangeChange() and valueChange().

Reimplemented in QScrollBar.

void QRangeControl::subtractLine ()

Equivalent to

```
setValue( value()-lineStep() )
```

See also: addLine().

void QRangeControl::subtractPage ()

Equivalent to

```
setValue( value()-pageStep() )
```

See also: addPage().

int QRangeControl::value () const

Returns the current range control value.

See also: setValue() and prevValue().

Examples:
    xform/xform.cpp

void QRangeControl::valueChange () `[virtual protected]`

This virtual function is called whenever the range control value changes. You can reimplment it if you want to be notified when the value changes. The default implementation does nothing.

See also: setValue(), addPage(), subtractPage(), addLine(), subtractLine(), rangeChange() and stepChange().

Reimplemented in QSlider and QScrollBar.

148.2.1. URL

Generated from http://www.troll.no/qt/qrangecontrol.html.

149. QRect Class Reference

```
#include <qrect.h>
```

149.1. Detailed Description

The QRect class defines a rectangle in the plane.

A rectangle is internally represented as an upper left corner and a bottom right corner, but it is normally expressed as an upper left corner and a size.

The coordinate type is QCOORD (defined in qwindowdefs.h as `short`). The minimum value of QCOORD is QCOORD_MIN (-32768) and the maximum value is QCOORD_MAX (32767).

Note that the size (width and height) of a rectange might be different from what you are used to. If the top left corner and the bottom right corner are the same, then the height and the width of the rectangle will both be 1.

Generally, *width = right - left + 1* and *height = bottom - top + 1.* We designed it this way to make it correspond to rectangular spaces used by drawing functions, where the width and height denote a number of pixels. For example, drawing a rectangle with width and height 1 draws a single pixel.

The default coordinate system has origin (0,0) in the top left corner, the positive direction of the y axis is downwards and the positive x axis is from the left to the right.

See also: QPoint and QSize.

Examples:

tictac/tictac.cpp
forever/forever.cpp
desktop/desktop.cpp
tooltip/tooltip.cpp
movies/main.cpp
xform/xform.cpp

149.2. Member Function Documentation

QRect::QRect ()

Constructs an empty rectangle.

QRect::QRect (const QPoint & topLeft, const QPoint & bottomRight)

Constructs a rectangle with *topLeft* as the top left corner and *bottomRight* as the bottom right corner.

QRect::QRect (const QPoint & topLeft, const QSize & size)

Constructs a rectangle with *topLeft* as the top left corner and *size* as the rectangle size.

QRect::QRect (int left, int top, int width, int height)

Constructs a rectangle with the *top, left* corner and *width* and *height.*

Example (creates three identical rectangles):

```
QRect r1( QPoint(100,200), QPoint(110,215) );
QRect r2( QPoint(100,200), QSize(11,16) );
QRect r3( 100, 200, 11, 16 );
```

int QRect::bottom () const

Returns the bottom coordinate of the rectangle.

See also: top(), setBottom(), bottomLeft() and bottomRight().

Examples:
desktop/desktop.cpp

QPoint QRect::bottomLeft () const

Returns the bottom left position of the rectangle.

See also: moveBottomLeft(), bottomRight(), topLeft(), topRight(), bottom() and left().

Examples:
tictac/tictac.cpp

QPoint QRect::bottomRight () const

Returns the bottom right position of the rectangle.

See also: moveBottomRight(), bottomLeft(), topLeft(), topRight(), bottom() and right().

Examples:
tictac/tictac.cpp

QPoint QRect::center () const

Returns the center point of the rectangle.

See also: moveCenter(), topLeft(), topRight(), bottomLeft() and bottomRight().

Examples:
tooltip/tooltip.cpp

bool QRect::contains (const QPoint & p, bool proper=FALSE) const

Returns TRUE if the point *p* is inside or on the edge of the rectangle.

If *proper* is TRUE, this function returns TRUE only if *p* is inside (not on the edge).

bool QRect::contains (const QRect & r, bool proper=FALSE) const

Returns TRUE if the rectangle *r* is inside this rectangle.

If *proper* is TRUE, this function returns TRUE only if *r* is entirely inside (not on the edge).

See also: unite(), intersect() and intersects().

void QRect::coords (int * xp1, int * yp1, int * xp2, int * yp2) const

Extracts the rectangle parameters as the top left point and the bottom right point.

See also: setCoords() and rect().

int QRect::height () const

Returns the height of the rectangle. The height includes both the top and bottom edges, ie. height = bottom - top + 1.

See also: width(), size() and setHeight().

Examples:
- tictac/tictac.cpp
- desktop/desktop.cpp
- movies/main.cpp
- xform/xform.cpp

QRect QRect::intersect (const QRect & r) const

Returns the intersection rectangle of this rectangle and *r*.

Returns an empty rectangle if there is no intersection.

See also: isEmpty(), intersects(), unite() and contains().

bool QRect::intersects (const QRect & r) const

Returns TRUE if this rectangle intersects with *r* (there is at least one pixel which is within both rectangles).

See also: intersect() and contains().

bool QRect::isEmpty () const

Returns TRUE if the rectangle is empty, otherwise FALSE.

An empty rectangle has a left() > right() or top() > bottom().

An empty rectangle is not valid.

```
isEmpty() == !isValid()
```

See also: isNull() and isValid().

bool QRect::isNull () const

Returns TRUE if the rectangle is a null rectangle, otherwise FALSE.

A null rectangle has both the width and the height set to 0, that is right() == left() - 1 and bottom() == top() - 1.

Remember that if right() == left() and bottom() == top(), then the rectangle has width 1 and height 1.

A null rectangle is also empty.

A null rectangle is not valid.

See also: isEmpty() and isValid().

bool QRect::isValid () const

Returns TRUE if the rectangle is valid, or FALSE if it is invalid (empty).

A valid rectangle has a left() <= right() and top() <= bottom().

```
isValid() == !isEmpty()
```

See also: isNull(), isEmpty() and normalize().

Examples:
tooltip/tooltip.cpp

int QRect::left () const

Returns the left coordinate of the rectangle. Identical to x().

See also: x(), top(), right(), setLeft(), topLeft() and bottomLeft().

Examples:
tictac/tictac.cpp
desktop/desktop.cpp
xform/xform.cpp

void QRect::moveBottomLeft (const QPoint & p)

Sets the bottom left position of the rectangle to *p,* leaving the size unchanged.

See also: bottomLeft(), moveBottomRight(), moveTopLeft(), moveTopRight(), setBottom() and setLeft().

void QRect::moveBottomRight (const QPoint & p)

Sets the bottom right position of the rectangle to *p,* leaving the size unchanged.

See also: bottomRight(), moveBottomLeft(), moveTopLeft(), moveTopRight(), setBottom() and setRight().

void QRect::moveBy (int dx, int dy)

Moves the rectangle *dx* along the X axis and *dy* along the Y axis, relative to the current position. (Positive values moves the rectangle rightwards and/or downwards.).

Examples:
xform/xform.cpp

void QRect::moveCenter (const QPoint & p)

Sets the center point of the rectangle to *p,* leaving the size unchanged.

See also: center(), moveTopLeft(), moveTopRight(), moveBottomLeft() and moveBottomRight().

Examples:
tictac/tictac.cpp

void QRect::moveTopLeft (const QPoint & p)

Sets the top left position of the rectangle to *p,* leaving the size unchanged.

See also: topLeft(), moveTopRight(), moveBottomLeft(), moveBottomRight(), setTop() and setLeft().

Examples:
xform/xform.cpp

void QRect::moveTopRight (const QPoint & p)

Sets the top right position of the rectangle to *p,* leaving the size unchanged.

See also: topRight(), moveTopLeft(), moveBottomLeft(), moveBottomRight(), setTop() and setRight().

QRect QRect::normalize () const

Returns a normalized rectangle, i.e. one that has a non-negative width and height.

It swaps left and right if left() > right(), and swaps top and bottom if top() > bottom().

See also: isValid().

void QRect::rect (int * x, int * y, int * w, int * h) const

Extracts the rectangle parameters as the position and the size.

See also: setRect() and coords().

int QRect::right () const

Returns the right coordinate of the rectangle.

See also: left(), setRight(), topRight() and bottomRight().

Examples:
desktop/desktop.cpp

void QRect::setBottom (int pos)

Sets the bottom edge of the rectangle. May change the height, but will never change the top edge of the rectangle.

See also: bottom(), setTop() and setHeight().

void QRect::setCoords (int xp1, int yp1, int xp2, int yp2)

Sets the coordinates of the rectangle's top left corner to *(xp1,yp1),* and the coordinates of its bottom right corner to *(xp2,yp2).*

See also: coords() and setRect().

void QRect::setHeight (int h)

Sets the height of the rectangle to *h.* The top edge is not moved, but the bottom edge may be moved.

See also: height(), setTop(), setBottom() and setSize().

Examples:
desktop/desktop.cpp

void QRect::setLeft (int pos)

Sets the left edge of the rectangle. May change the width, but will never change the right edge of the rectangle.

Identical to setX().

See also: left(), setTop() and setWidth().

void QRect::setRect (int x, int y, int w, int h)

Sets the coordinates of the rectangle's top left corner to *(x,y),* and its size to (w,h).

See also: rect() and setCoords().

void QRect::setRight (int pos)

Sets the right edge of the rectangle. May change the width, but will never change the left edge of the rectangle.

See also: right(), setLeft() and setWidth().

void QRect::setSize (const QSize & s)

Sets the size of the rectangle to *s.* The top left corner is not moved.

See also: size(), setWidth() and setHeight().

Examples:
xform/xform.cpp

void QRect::setTop (int pos)

Sets the top edge of the rectangle. May change the height, but will never change the bottom edge of the rectangle.

Identical to setY().

See also: top(), setBottom() and setHeight().

void QRect::setWidth (int w)

Sets the width of the rectangle to *w.* The right edge is changed, but not the left edge.

See also: width(), setLeft(), setRight() and setSize().

Examples:
desktop/desktop.cpp

void QRect::setX (int x)

Sets the x position of the rectangle (its left end). May change the width, but will never change the right edge of the rectangle.

Identical to setLeft().

See also: x() and setY().

void QRect::setY (int y)

Sets the y position of the rectangle (its top). May change the height, but will never change the bottom edge of the rectangle.

Identical to setTop().

See also: y() and setX().

QSize QRect::size () const

Returns the size of the rectangle.

See also: width() and height().

Examples:
desktop/desktop.cpp
movies/main.cpp

int QRect::top () const

Returns the top coordinate of the rectangle. Identical to y().

See also: y(), left(), bottom(), setTop(), topLeft() and topRight().

Examples:
tictac/tictac.cpp
desktop/desktop.cpp
xform/xform.cpp

QPoint QRect::topLeft () const

Returns the top left position of the rectangle.

See also: moveTopLeft(), topRight(), bottomLeft(), bottomRight(), left() and top().

Examples:
tictac/tictac.cpp

QPoint QRect::topRight () const

Returns the top right position of the rectangle.

See also: moveTopRight(), topLeft(), bottomLeft(), bottomRight(), top() and right().

Examples:
tictac/tictac.cpp

QRect QRect::unite (const QRect & r) const

Returns the union rectangle of this rectangle and *r*. The union rectangle of a nonempty rectangle and an empty or invalid rectangle is defined to be the nonempty rectangle.

See also: intersect(), intersects() and contains().

Examples:
xform/xform.cpp

int QRect::width () const

Returns the width of the rectangle. The width includes both the left and right edges, ie. width = right - left + 1.

See also: height(), size() and setHeight().

Examples:
tictac/tictac.cpp
desktop/desktop.cpp
movies/main.cpp

xform/xform.cpp

int QRect::x () const

Returns the left coordinate of the rectangle. Identical to left().

See also: left(), y() and setX().

Examples:
desktop/desktop.cpp
movies/main.cpp
xform/xform.cpp

int QRect::y () const

Returns the top coordinate of the rectangle. Identical to top().

See also: top(), x() and setY().

Examples:
desktop/desktop.cpp
movies/main.cpp
xform/xform.cpp

149.3. Related Functions

bool operator== (const QRect & r1, const QRect & r2)

Returns TRUE if *r1* and *r2* are equal, or FALSE if they are different.

bool operator!= (const QRect & r1, const QRect & r2)

Returns TRUE if *r1* and *r2* are different, or FALSE if they are equal.

QDataStream & operator<< (QDataStream & s, const QRect & r)

Writes a QRect to the stream and returns a reference to the stream.

Serialization format: [left (INT16), top (INT16), right (INT16), bottom (INT16)].

QDataStream & operator>> (QDataStream & s, QRect & r)

Reads a QRect from the stream and returns a reference to the stream.

149.3.1. URL

Generated from http://www.troll.no/qt/qrect.html.

150. QRegExp Class Reference

```
#include <qregexp.h>
```

150.1. Detailed Description

The QRegExp class provides pattern matching using regular expressions and wildcards.

QRegExp knows these regexp primitives:

- *c* matches the character 'c'
-
- ^ matches start of input (except [^x] which matches NOT [x])
- *$* matches end of input
- *[]* matches a set of characters, for example [a-z0-9_]
- *a** matches a sequence of zero or more a's
- *a+* matches a sequence of one or more a's
- *a?* matches an optional a
- *c* escape code for matching special characters like \\, [, *, +, . etc.
- *b* matches the BELL character (7)
- *t* matches the TAB character (9)
- *n* matches newline (10)
- *r* matches return (13)
- *s* matches white space (9,10,11,12,13,32)
- *x12* matches the character 0x12 (18 decimal, 12 hexadecimal).
- *022* matches the character 022 (18 decimal, 22 octal).

In wildcard mode, it only knows three primitives:

- *?* matches any character
- * matches any sequence of characters
- *[]* matches a defined set of characters, e.g. [a-zA-Z0-9\.] matches upper and lower case ASCII letters, digits, and dot, and [^z] matches everything except lower-case z.

When writing regular expressions in C++ code, remember that the C++ preprocessor processes \ characters. So in order to match a "." character, you must write "\\." in C++ source, not "\.".

Examples:
qmag/qmag.cpp

150.2. Member Function Documentation

QRegExp::QRegExp ()

Constructs an empty regular expression.

QRegExp::QRegExp (const QRegExp & r)

Constructs a regular expression which is a copy of *r*.

See also: operator=(const and QRegExp&).

QRegExp::QRegExp (const char * pattern, bool caseSensitive=TRUE, bool wildcard=FALSE)

Constructs a regular expression.

Arguments:

- *pattern* is the regular expression pattern string.
- *caseSensitive* specifies whether or not to use case sensitive matching.
- *wildcard* specifies whether the pattern string should be used for wildcard matching (also called globbing expression), normally used for matching file names.

See also: setWildcard().

QRegExp::~QRegExp ()

Destroys the regular expression and cleans up its internal data.

bool QRegExp::caseSensitive () const

Returns TRUE if case sensitivity is enabled, otherwise FALSE. The default is TRUE.

See also: setCaseSensitive().

void QRegExp::compile () `[protected]`

For internal use only.

bool QRegExp::isEmpty () const

Returns TRUE if the regexp is empty.

bool QRegExp::isValid () const

Returns TRUE if the regexp is valid, or FALSE if it is invalid.

The pattern "[a-z" is an example of an invalid pattern, since it lacks a closing bracket.

int QRegExp::match (const char * str, int index=0, int * len=0) const

Attempts to match in *str,* starting from position *index.* Returns the position of the match, or -1 if there was no match.

If *len* is not a null pointer, the length of the match is stored in **len.*

Example:

```
QRegExp r("[0-9]*\.[0-9]+");              // matches floating point
int len;
r.match("pi = 3.1416", 0, &len);          // returns 5, len == 6
```

Examples:
qmag/qmag.cpp

char * QRegExp::matchstr (ushort * rxd, char * str, char * bol) const `[protected]`

For internal use only.

bool QRegExp::operator!= (const QRegExp & r) const

Returns TRUE if this regexp is *not* equal to *r.*

See also: operator==().

QRegExp & QRegExp::operator= (const QRegExp & r)

Copies the regexp *r* and returns a reference to this regexp. The case sensitivity and wildcard options are copied, as well.

QRegExp & QRegExp::operator= (const char * pattern)

Sets the pattern string to *pattern* and returns a reference to this regexp. The case sensitivity or wildcard options do not change.

bool QRegExp::operator== (const QRegExp & r) const

Returns TRUE if this regexp is equal to *r*.

Two regexp objects are equal if they have equal pattern strings, case sensitivity options and wildcard options.

const char * QRegExp::pattern () const

Returns the pattern string of the regexp.

void QRegExp::setCaseSensitive (bool enable)

Enables or disables case sensitive matching.

In case sensitive mode, "a.e" matches "axe" but not "Axe".

See also: caseSensitive().

void QRegExp::setWildcard (bool wildcard)

Sets the wildcard option for the regular expression. The default is FALSE.

Setting *wildcard* to TRUE makes it convenient to match filenames instead of plain text.

For example, "qr*.cpp" matches the string "qregexp.cpp" in wildcard mode, but not "qicpp" (which will be matched in normal mode).

See also: wildcard().

bool QRegExp::wildcard () const

Returns TRUE if wildcard mode is on, otherwise FALSE.

See also: setWildcard().

150.2.1. URL

Generated from http://www.troll.no/qt/qregexp.html.

151. QRegion Class Reference

```
#include <qregion.h>
```

151.1. Detailed Description

The QRegion class specifies a clip region for a painter.

A region can be a rectangle, an ellipse, a polygon or a combination of these.

Regions are combined by creating a new region which is a union, intersection or difference between any two regions.

The region XOR operation is defined as:

```
a XOR b = (a UNION b) - (a INTERSECTION b)
```

Example of using complex regions:

```
void MyWidget::paintEvent( QPaintEvent * )
{
    QPainter p;                               // our painter
    QRegion r1( QRect(100,100,200,80),        // r1 = elliptic region
                QRegion::Ellipse );
    QRegion r2( QRect(100,120,90,30) );       // r2 = rectangular region
    QRegion r3 = r1.intersect( r2 );          // r3 = intersection
    p.begin( this );                          // start painting widget
    p.setClipRegion( r3 );                    // set clip region
    ...                                       // paint clipped graphics
    p.end();                                  // painting done
}
```

See also: QPainter::setClipRegion() and QPainter::setClipRect().

151.2. Member Function Documentation

QRegion::QRegion ()

Constructs an null region.

See also: isNull().

QRegion::QRegion (const QPointArray & a, bool winding=FALSE)

Constructs a polygon region from the point array *a*.

If *winding* is TRUE, the polygon region uses the winding algorithm, otherwise the alternative (even-odd) algorithm will be used.

QRegion::QRegion (const QRect & r, RegionType t = Rectangle)

Constructs a rectangular or elliptic region.

Arguments:

- *r* is the region rectangle.
- *t* is the region type: QRegion::Rectangle (default) or QRegion::Ellipse.

QRegion::QRegion (const QRegion & r)

Constructs a region which is a shallow copy of *r*.

QRegion::QRegion (int x, int y, int w, int h, RegionType t = Rectangle)

Constructs a rectangular or elliptic region.

x, y, w, and *h* specify the region rectangle. *t* is the region type: QRegion::Rectangle (default) or QRegion::Ellipse.

QRegion::˜QRegion ()

Destroys the region.

QRect QRegion::boundingRect () const

Returns the bounding rectange of this region. An empty region gives a null rectangle.

bool QRegion::contains (const QPoint & p) const

Returns TRUE if the region contains the point *p,* or FALSE if *p* is outside the region.

bool QRegion::contains (const QRect & r) const

Returns TRUE if the region contains the rectangle *r,* or FALSE if *r* is outside the region.

QRegion QRegion::eor (const QRegion & r) const

Returns a region which is this region XOR *r.*

QRegion QRegion::intersect (const QRegion & r) const

Returns a region which is the intersection of this region and *r.*

bool QRegion::isEmpty () const

Returns TRUE if the region is empty, or FALSE if it is non-empty.

Example:

```
QRegion r1( 10, 10, 20, 20 );
QRegion r2( 40, 40, 20, 20 );
QRegion r3;
r1.isNull();                // FALSE
r1.isEmpty();               // FALSE
r3.isNull();                // TRUE
r3.isEmpty();               // TRUE
r3 = r1.intersect( r2 );    // r3 = intersection of r1 and r2
r3.isNull();                // FALSE
r3.isEmpty();               // TRUE
r3 = r1.unite( r2 );        // r3 = union of r1 and r2
r3.isNull();                // FALSE
r3.isEmpty();               // FALSE
```

See also: isNull().

bool QRegion::isNull () const

Returns TRUE if the region is a null region, otherwise FALSE.

A null region is a region that has not been initialized. The documentation for isEmpty() contains an example that shows how to use isNull() and isEmpty().

See also: isEmpty().

bool QRegion::operator!= (const QRegion & r) const

Returns TRUE if the region is different from *r,* or FALSE if the regions are equal.

QRegion & QRegion::operator= (const QRegion & r)

Assigns a shallow copy of *r* to this region and returns a reference to the region.

bool QRegion::operator== (const QRegion & r) const

Returns TRUE if the region is equal to *r,* or FALSE if the regions are different.

QArray<QRect> QRegion::rects () const

Returns an array of the rectangles that make up the region. The rectangles are non-overlapping. The region is formed by the union of all these rectangles.

QRegion QRegion::subtract (const QRegion & r) const

Returns a region which is *r* subtracted from this region.

void QRegion::translate (int dx, int dy)

Translates the region *dx* along the X axis and *dy* along the Y axis.

QRegion QRegion::unite (const QRegion & r) const

Returns a region which is the union of this region and *r.*

151.3. Related Functions

QDataStream & operator<< (QDataStream & s, const QRegion & r)

Writes a region to the stream and returns a reference to the stream.

QDataStream & operator>> (QDataStream & s, QRegion & r)

Reads a region from the stream and returns a reference to the stream.

151.3.1. URL

Generated from http://www.troll.no/qt/qregion.html.

152. QResizeEvent Class Reference

`#include <qevent.h>`

Inherits QEvent.

152.1. Detailed Description

The QResizeEvent class contains event parameters for resize events.

Resize events are sent to widgets that have been resized.

The event handler QWidget::resizeEvent() receives resize events.

See also: QWidget::resize() and QWidget::setGeometry().

Examples:

mainlyMotif/editor.cpp
tictac/tictac.cpp
life/life.cpp
mainlyQt/editor.cpp
tooltip/tooltip.cpp
drawdemo/drawdemo.cpp
xform/xform.cpp
menu/menu.cpp
qmag/qmag.cpp
showimg/showimg.cpp
mainlyXt/editor.cpp

152.2. Member Function Documentation

QResizeEvent::QResizeEvent (const QSize & size, const QSize & oldSize)

Constructs a resize event with the new and old widget sizes.

const QSize & QResizeEvent::oldSize () const

Returns the old size of the widget.

const QSize & QResizeEvent::size () const

Returns the new size of the widget, which is the same as QWidget::size().

152.2.1. URL

Generated from http://www.troll.no/qt/qresizeevent.html.

153. QScrollBar Class Reference

```
#include <qscrollbar.h>
```

Inherits QRangeControl and QWidget.

153.1. Detailed Description

The QScrollBar widget provides a vertical or horizontal scroll bar.

A scroll bar is used to let the user control a value within a program-definable range, and to give the user visible indication of the current value of a range control.

Scroll bars include four separate controls, in order:

- The *line-up* control is a little triangle with which the user can move one line up. The meaning of line is configurable. In e.g. editors and list boxes means one line of text.
- The *slider* is the handle that indicates the current value of the scroll bar, and which the user can drag to change the value.
- The *page-up/down* control is the area on which the slider slides (the scroll bar's background). Clicking here moves the scroll bar up or down one page. The meaning of page too is configurable - in editors and list boxes it means as many lines as there is space for in the widget.
- Finally, the line-down control is the arrow on the other end of the scroll bar. Clicking there moves the scroll bar down/rightwards one line.

QScrollBar has not much of an API of its own; it mostly relies on QRangeControl. The most useful functions are setValue() to set the scrollbar directly to some value; addPage(), addLine(), subPage() and subLine() to simulate the effects of clicking (neat for accelerator keys; setSteps() to define the values of pageStep() and lineStep(); and last but NOT least setRange() to set the minValue() and maxValue() of the scrollbar. (QScrollBar has a convenience constructor with which you can set most of that.)

In addition to the access functions from QRangeControl, QScrollBar has a comprehensive set of signals:

- valueChanged() - emitted when the scroll bar's value has changed.
- sliderPressed() - emitted when the user starts to drag the slider
- sliderMoved() - emitted when the user drags the slider
- sliderReleased() - emitted when the user releases the slider
- nextLine() - emitted when the scroll bar has moved one line down/rightwards. Line is defined in QRangeControl.
- prevLine() - emitted when the scroll bar has moved one line up/leftwards.
- nextPage() - emitted when the scroll bar has moved one page down/rightwards.
- prevPage() - emitted when the scroll bar has moved one page up/leftwards.

QScrollBar only offers integer ranges, and the current implementation has problems when the range is greater than a million or so. (A million is more than sufficient for today's display sizes, however.)

A scroll bar can be controlled by the keyboard, but it has a default focusPolicy() of `NoFocus.` Use setFocusPolicy() to enable keyboard focus.

See also: QSlider, QSpinBox and GUI Design Handbook: Scroll Bar

153.2. Member Function Documentation

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

Constructs a vertical scroll bar.

The *parent* and *name* arguments are sent to the QWidget constructor.

QScrollBar::QScrollBar (Orientation orientation, QWidget * parent=0, const char * name=0)

Constructs a scroll bar.

The *orientation* must be QScrollBar::Vertical or QScrollBar::Horizontal.

The *parent* and *name* arguments are sent to the QWidget constructor.

QScrollBar::QScrollBar (int minValue, int maxValue, int lineStep, int pageStep, int value, Orientation orientation, QWidget * parent=0, const char * name=0)

Constructs a scroll bar.

Arguments:

- *minValue* is the minimum scroll bar value.
- *maxValue* is the maximum scroll bar value.
- *lineStep* is the line step value.
- *pageStep* is the page step value.
- *value* is the initial value.
- *orientation* must be QScrollBar::Vertical or QScrollBar::Horizontal. The *parent* and *name* arguments are sent to the QWidget constructor.

bool QScrollBar::draggingSlider () const

Returns TRUE if the user has clicked the mouse on the slider and is currenly dragging it, or FALSE if not.

void QScrollBar::keyPressEvent (QKeyEvent * e) `[virtual protected]`

Handles key press events for the scroll bar.

Reimplemented from QWidget.

void QScrollBar::mouseMoveEvent (QMouseEvent * e) `[virtual protected]`

Handles mouse move events for the scroll bar.

Reimplemented from QWidget.

void QScrollBar::mousePressEvent (QMouseEvent * e) `[virtual protected]`

Handles mouse press events for the scroll bar.

Reimplemented from QWidget.

void QScrollBar::mouseReleaseEvent (QMouseEvent * e) `[virtual protected]`

Handles mouse release events for the scroll bar.

Reimplemented from QWidget.

void QScrollBar::nextLine () `[signal]`

This signal is emitted when the scroll bar scrolls one line down/right.

void QScrollBar::nextPage () `[signal]`

This signal is emitted when the scroll bar scrolls one page down/right.

Orientation QScrollBar::orientation () const

Returns the scroll bar orientation; QScrollBar::Vertical or QScrollBar::Horizontal.

See also: setOrientation().

void QScrollBar::paintEvent (QPaintEvent * event) `[virtual protected]`

Handles paint events for the scroll bar.

Reimplemented from QWidget.

void QScrollBar::prevLine () `[signal]`

This signal is emitted when the scroll bar scrolls one line up/left.

void QScrollBar::prevPage () `[signal]`

This signal is emitted when the scroll bar scrolls one page up/left.

void QScrollBar::rangeChange () `[virtual protected]`

For internal use only.

Reimplemented from QRangeControl.

void QScrollBar::resizeEvent (QResizeEvent *) `[virtual protected]`

Handles resize events for the scroll bar.

Reimplemented from QWidget.

void QScrollBar::setOrientation (Orientation orientation)

Sets the scroll bar orientation. The *orientation* must be QScrollBar::Vertical or QScrollBar::Horizontal.

See also: orientation().

void QScrollBar::setPalette (const QPalette & p) `[virtual]`

Reimplements the virtual function QWidget::setPalette().

Sets the background color to the mid color for Motif style scroll bars.

Reimplemented from QWidget.

void QScrollBar::setTracking (bool enable)

Enables scroll bar tracking if *enable* is TRUE, or disables tracking if *enable* is FALSE.

If tracking is enabled (default), the scroll bar emits the valueChanged() signal whenever the slider is being dragged. If tracking is disabled, the scroll bar emits the valueChanged() signal when the user relases the mouse button (unless the value happens to be the same as before).

See also: tracking().

QSize QScrollBar::sizeHint () const `[virtual]`

Returns a size hint for this scroll bar.

Reimplemented from QWidget.

void QScrollBar::sliderMoved (int value) `[signal]`

This signal is emitted when the slider is moved by the user, with the new scroll bar value as an argument.

This signal is emitted even when tracking is turned off.

See also: tracking(), valueChanged(), nextLine(), prevLine(), nextPage() and prevPage().

void QScrollBar::sliderPressed () `[signal]`

This signal is emitted when the user presses the slider with the mouse.

QRect QScrollBar::sliderRect () const `[protected]`

Returns the scroll bar slider rectangle.

See also: sliderStart().

void QScrollBar::sliderReleased () `[signal]`

This signal is emitted when the user releases the slider with the mouse.

int QScrollBar::sliderStart () const `[protected]`

Returns the pixel position where the scroll bar slider starts.

It is equivalent to sliderRect().y() for vertical scroll bars or sliderRect().x() for horizontal scroll bars.

void QScrollBar::stepChange () `[virtual protected]`

For internal use only.

Reimplemented from QRangeControl.

void QScrollBar::timerEvent (QTimerEvent *) `[virtual protected]`

Handles timer events for the scroll bar.

Reimplemented from QObject.

bool QScrollBar::tracking () const

Returns TRUE if tracking is enabled, or FALSE if tracking is disabled.

Tracking is initially enabled.

See also: setTracking().

void QScrollBar::valueChange () `[virtual protected]`

For internal use only.

Reimplemented from QRangeControl.

void QScrollBar::valueChanged (int value) `[signal]`

This signal is emitted when the scroll bar value is changed, with the new scroll bar value as an argument.

153.2.1. URL

Generated from http://www.troll.no/qt/qscrollbar.html.

154. QScrollView Class Reference

#include <qscrollview.h>

Inherits QFrame.

Inherited by QListView.

154.1. Detailed Description

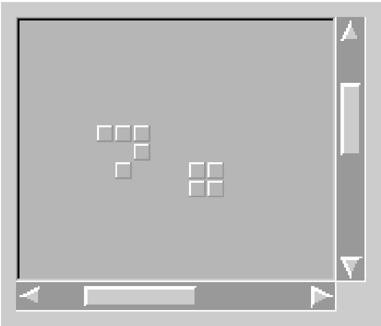
The QScrollView widget provides a scrolling area with on-demand scrollbars.

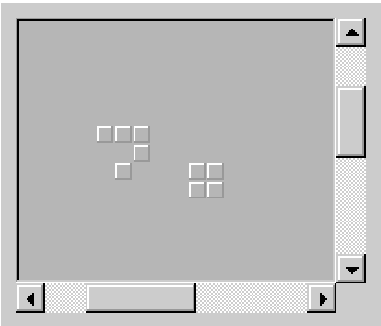
The QScrollView is a large canvas - potentially larger than the coordinate system normally supported by the underlying window system. This is important, as is is quite easy to go beyond such limitations (eg. many web pages are more than 32000 pixels high). Additionally, the QScrollView can have QWidgets positioned on it that scroll around with the drawn content. These subwidgets can also have positions outside the normal coordinate range (but they are still limited in size).

To provide content for the widget, inherit from QScrollView and override drawContentsOffset(), and use resizeContents() to set the size of the viewed area. Use addChild(), moveChild(), and showChild() to position widgets on the view.

Note also the effect of resizePolicy().

Currently the scrollbars fail beyond about 1000000 pixels due to integer overflow.





154.2. Member Function Documentation

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

Constructs a QScrollView.

If you intend to add child widgets, you may see improved refresh if you include WPaintClever in the widgets flags, *f*. WPaintClever will be propagated to the viewport() widget.

QScrollView::~QScrollView ()

Destructs the QScrollView. Any children added with addChild() will be destructed.

void QScrollView::addChild (QWidget * child)

Inserts *child* into the scrolled area. It is equivalent to addChild(child,0,0).

void QScrollView::addChild (QWidget * child, int x, int y) `[virtual]`

Inserts *child* into the scrolled area positioned at *(x, y)*. If the child is already in the view, it is just moved.

int QScrollView::bottomMargin () const `[protected]`

Returns the current bottom margin.

See also: setMargins().

void QScrollView::center (int x, int y) `[slot]`

Scrolls the content so that the point (x,y) is in the center of visible area.

void QScrollView::center (int x, int y, float xmargin, float ymargin) `[slot]`

Scrolls the content so that the point (x,y) is visible, with the given margins (as fractions of visible area).

eg.

- Margin 0.0 allows (x,y) to be on edge of visible area.
- Margin 0.5 ensures (x,y) is in middle 50% of visible area.
- Margin 1.0 ensures (x,y) is in the center of the visible area.

bool QScrollView::childIsVisible (QWidget * child)

\obsoltete

Returns TRUE if *child* is visible. This is equivalent to child->isVisible().

int QScrollView::childX (QWidget * child)

Returns the X position of the given child widget. Use this rather than QWidget::x() for widgets added to the view.

int QScrollView::childY (QWidget * child)

Returns the Y position of the given child widget. Use this rather than QWidget::y() for widgets added to the view.

int QScrollView::contentsHeight () const

Returns the height of the contents area.

void QScrollView::contentsMoving (int x, int y) `[signal]`

This signal is emitted just before the contents is moved to the given position.

See also: contentsX() and contentsY().

int QScrollView::contentsWidth () const

Returns the width of the contents area.

int QScrollView::contentsX () const

Returns the X coordinate of the contents which is at the left edge of the viewport.

int QScrollView::contentsY () const

Returns the Y coordinate of the contents which is at the top edge of the viewport.

QWidget* QScrollView::cornerWidget () const

Returns the widget in the corner between the two scrollbars.

By default, no corner widget is present.

void QScrollView::drawContentsOffset (QPainter * p, int offsetx, int offsety, int clipx, int clipy, int clipw, int cliph) `[virtual protected]`

Reimplement this method if you are viewing a drawing area rather than a widget.

Draws the rectangle (*clipx, clipy, clipw, cliph*) of the contents, offset by (*offsetx, offsety*) using painter *p*. All four are given in the scroll views's coordinates. All of *clipx, clipy, offsetx* and *offsety* are typically large positive numbers.

Note that the final coordinates you give to QPainter methods must be within the range supported by the underlying window systems - about +/- 32000 at most, often much less - +/- 4000 or so is all you should rely expect.

For example:

```
{
  // Fill a 40000 by 50000 rectangle at (100000,150000)
  // Calculate the coordinates... (don't use QPoint, QRect, etc!)
  int x1 = 100000, y1 = 150000;
  int x2 = x1+40000-1, y2 = y1+50000-1;
  // Clip the coordinates so X/Windows will not have problems...
  if (x1 < clipx) x1=clipx;
  if (y1 < clipy) y1=clipy;
  if (x2 > clipx+clipw-1) x2=clipx+clipw-1;
  if (y2 > clipy+cliph-1) y2=clipy+cliph-1;
  // Translate to scrolled coordinates...
  x1 -= ox;
  x2 -= ox;
  y1 -= oy;
  y2 -= oy;
```

```
    // Paint using the new small coordinates...
    if ( x2 >= x1 && y2 >= y1 )
        p->fillRect(x1, y1, x2-x1+1, y2-y1+1, red);
}
```

The clip rectangle of the painter *p* is already set appropriately.

Note that QPainter::translate() is not sufficient.

The default implementation does nothing.

Reimplemented in QListView.

void QScrollView::ensureVisible (int x, int y) `[slot]`

Scrolls the content so that the point (x, y) is visible with at least 50-pixel margins (if possible, otherwise centered).

void QScrollView::ensureVisible (int x, int y, int xmargin, int ymargin) `[slot]`

Scrolls the content so that the point (x, y) is visible with at least the given pixel margins (if possible, otherwise centered).

bool QScrollView::eventFilter (QObject * obj, QEvent * e) `[virtual protected]`

This event filter ensures the scrollbars are updated when a single contents widget is resized, shown, hidden, or destroyed, and passes mouse events to the QScrollView.

Reimplemented from QObject.

bool QScrollView::focusNextPrevChild (bool next) `[virtual protected]`

Override so that traversal moves among child widgets, even if they are not visible, scrolling to make them so.

Reimplemented from QWidget.

void QScrollView::frameChanged () `[virtual protected]`

An override - ensures scrollbars are correct size when frame style changes.

Reimplemented from QFrame.

QScrollView::ScrollBarMode QScrollView::hScrollBarMode() const

Returns the currently set mode for the horizontal scrollbar.

See also: setHScrollBarMode().

QScrollBar* QScrollView::horizontalScrollBar ()

Returns the component horizontal scrollbar. It is made available to allow accelerators, autoscrolling, etc., and to allow changing of arrow scrollrates: bar->setSteps(rate, bar->pageStep()).

It should not be otherwise manipulated.

int QScrollView::leftMargin () const `[protected]`

Returns the current left margin.

See also: setMargins().

void QScrollView::moveChild (QWidget * child, int x, int y) `[virtual]`

Repositions *child* to (*x, y*). This functions the same as addChild().

void QScrollView::removeChild (QWidget * child)

Removes a child from the scrolled area. Note that this happens automatically if the child is deleted.

void QScrollView::resize (const QSize & s)

An override - ensures scrollbars are correct size upon resize.

void QScrollView::resize (int w, int h)

An override - ensures scrollbars are correct size upon resize.

void QScrollView::resizeContents (int w, int h) `[virtual slot]`

Set the size of the contents area to *w* pixels wide and *h* pixels high, and updates the viewport accordingly.

void QScrollView::resizeEvent (QResizeEvent * event) `[virtual protected]`

An override - ensures scrollbars are correct size upon resize.

Reimplemented from QWidget.

QScrollView::ResizePolicy QScrollView::resizePolicy() const

Returns the currently set ResizePolicy.

int QScrollView::rightMargin () const `[protected]`

Returns the current right margin.

See also: setMargins().

void QScrollView::scrollBy (int dx, int dy) `[slot]`

Scrolls the content by *x* to the left and *y* upwards.

void QScrollView::setContentsPos (int x, int y) `[slot]`

Scrolls the content so that the point (x, y) is in the top-left corner.

void QScrollView::setCornerWidget (QWidget * corner) `[virtual]`

Sets the widget in the corner between the two scrollbars.

You will probably also want to set at least one of the scrollbar modes to AlwaysOn.

Passing 0 shows no widget in the corner.

Any previous corner widget is hidden.

You may call setCornerWidget() with the same widget at different times.

All widgets set here will be deleted by the QScrollView when it destructs unless you seperately recreate the widget after setting some other corner widget (or 0).

Any *newly* set widget should have no current parent.

By default, no corner widget is present.

See also: setVScrollBarMode() and setHScrollBarMode().

void QScrollView::setHScrollBarMode (ScrollBarMode mode) `[virtual]`

Sets the mode for the horizontal scrollbar.

- `Auto` (the default) shows a scrollbar when the content is too wide to fit.
- `AlwaysOff` never shows a scrollbar.
- `AlwaysOn` always shows a scrollbar.

See also: hScrollBarMode() and setVScrollBarMode().

void QScrollView::setMargins (int left, int top, int right, int bottom) `[protected]`

Sets the margins around the scrolling area. This is useful for applications such as spreadsheets with ‘locked’ rows and columns. The marginal space is *inside* the frameRect() and is left blank - override drawContents() or put widgets in the unused area.

By default all margins are zero.

See also: frameChanged().

void QScrollView::setResizePolicy (ResizePolicy r) `[virtual]`

Sets the resize policy to *r*.

The policies are:

- `Default` is the initial value. It converts to `Manual` if the view is resized with resizeContents(), or to `AutoOne` if a child is added.
- `Manual` means the view stays the size set by resizeContents().
- `AutoOne` means that if there is only only child widget, the view stays the size of that widget.

void QScrollView::setVScrollBarMode (ScrollBarMode mode) `[virtual]`

Sets the mode for the vertical scrollbar.

- `Auto` (the default) shows a scrollbar when the content is too tall to fit.
- `AlwaysOff` never shows a scrollbar.
- `AlwaysOn` always shows a scrollbar.

See also: vScrollBarMode() and setHScrollBarMode().

void QScrollView::show () `[virtual]`

An override - ensures scrollbars are correct size upon showing.

Reimplemented from QWidget.

void QScrollView::showChild (QWidget * child, bool y=TRUE)

\obsolete

Sets the visibility of *child.* Equivalent to QWidget::show() or QWidget::hide().

int QScrollView::topMargin () const `[protected]`

Returns the current top margin.

See also: setMargins().

void QScrollView::updateScrollBars () `[slot]`

Updates scrollbars - all possibilities considered. You should never need to call this in your code.

QScrollView::ScrollBarMode QScrollView::vScrollBarMode() const

Returns the currently set mode for the vertical scrollbar.

See also: setVScrollBarMode().

QScrollBar* QScrollView::verticalScrollBar ()

Returns the component vertical scrollbar. It is made available to allow accelerators, autoscrolling, etc., and to allow changing of arrow scrollrates: bar->setSteps(rate, bar->pageStep()).

It should not be otherwise manipulated.

QWidget* QScrollView::viewport ()

Returns the viewport widget of the scrollview. This is the widget containing the contents widget or which is the drawing area.

void QScrollView::viewportMouseDoubleClickEvent (QMouseEvent *) `[virtual protected]`

To provide simple processing of events on the contents, this method receives all mouse double click events sent to the viewport.

void QScrollView::viewportMouseMoveEvent (QMouseEvent *) `[virtual protected]`

To provide simple processing of events on the contents, this method receives all mouse move events sent to the viewport.

void QScrollView::viewportMousePressEvent (QMouseEvent *) `[virtual protected]`

To provide simple processing of events on the contents, this method receives all mouse press events sent to the viewport.

void QScrollView::viewportMouseReleaseEvent (QMouseEvent *) `[virtual protected]`

To provide simple processing of events on the contents, this method receives all mouse release events sent to the viewport.

void QScrollView::viewportPaintEvent (QPaintEvent * pe) `[virtual protected]`

This is a low-level painting routine that draws the viewport contents. Override this if drawContentsOffset() is too high-level. (for example, if you don't want to open a QPainter on the viewport).

154.2.1. URL

Generated from http://www.troll.no/qt/qscrollview.html.

155. QSemiModal Class Reference

`#include <qsemimodal.h>`

Inherits QWidget.

Inherited by QProgressDialog.

155.1. Detailed Description

The QSemiModal class is the base class of semi-modal dialog windows.

The semi-modal dialog window can disable events to other windows while it is open. To enable this, the QSemiModal must be constructed with TRUE for the *modal* argument, which is FALSE by default, for consistency with QDialog. Such a QSemiModal is modal like QDialog, but it does not have its own event loop and has no concept of a default button.

Note that the parent widget has a different meaning for semi-modal dialogs than for other types of widgets. A semi-modal dialog is placed on top of the parent widget. The dialog is centered on the screen if the parent widget is zero.

155.2. Member Function Documentation

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

Constructs a semi-modal dialog named *name,* which has a parent widget *parent.* If *modal* is FALSE (the default), the only behaviour different to a QWidget is automatic sizing and positioning.

QSemiModal::~QSemiModal ()

Destroys the QSemiModal and all its children.

void QSemiModal::move (const QPoint & p)

Reimplements QWidget::move() for internal purposes.

void QSemiModal::move (int x, int y) `[virtual]`

Reimplements QWidget::move() for internal purposes.

Reimplemented from QWidget.

void QSemiModal::resize (const QSize & s)

Reimplements QWidget::resize() for internal purposes.

void QSemiModal::resize (int w, int h) `[virtual]`

Reimplements QWidget::resize() for internal purposes.

Reimplemented from QWidget.

void QSemiModal::setGeometry (const QRect & r)

Reimplements QWidget::setGeometry() for internal purposes.

void QSemiModal::setGeometry (int x, int y, int w, int h) `[virtual]`

Reimplements QWidget::setGeometry() for internal purposes.

Reimplemented from QWidget.

void QSemiModal::show () `[virtual]`

Shows the widget. This implementation also does automatic resizing and automatic positioning. If you have not already resized or moved the dialog, it will find a size that fits the contents and a position near the middle of the screen (or centered relative to the parent widget if any).

Reimplemented from QWidget.

155.2.1. URL

Generated from http://www.troll.no/qt/qsemimodal.html.

156. QShared Class Reference

```
#include <qshared.h>
```

156.1. Detailed Description

The QShared struct is internally used for implementing shared classes.

It only contains a reference count and member functions to increment and decrement it.

Shared classes normally have internal classes that inherit QShared and add the shared data.

See also: Shared Classes

156.1.1. URL

Generated from http://www.troll.no/qt/qshared.html.

157. QSignal Class Reference

```
#include <qsignal.h>
```

157.1. Detailed Description

The QSignal class can be used to send signals without parameters.

QSignal is a simple extension of QObject that can send plain signals without parameters. If you want to send signals from a class that does not inherit QObject, you can create an internal QSignal object to emit the signal. You must also provide a function that connects the signal to an outside object slot. This is how we have implemented signals in the QMenuData class, which is not a QObject.

In general, we recommend inheriting QObject instead. QObject provides much more functionality.

Note that QObject is a *private* base class of QSignal, i.e. you cannot call any QObject member functions from a QSignal object.

Example:

```
#include <qsignal.h>
class MyClass
{
public:
    MyClass();
   ~MyClass();
    void doSomething();
    void connect( QObject *receiver, const char *member );
private:
    QSignal *sig;
};
MyClass::MyClass()
{
    sig = new QSignal;
}
MyClass::~MyClass()
{
    delete sig;
}
void MyClass::doSomething()
{
    // ... does something
    sig->activate();        // activates the signal
}
void MyClass::connect( QObject *receiver, const char *member )
{
    sig->connect( receiver, member );
}
```

157.2. Member Function Documentation

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

Constructs a signal object with the parent object *parent* and a *name*. These arguments are passed directly to QObject.

void QSignal::activate ()

Emits the signal.

See also: isBlock().

void QSignal::block (bool b)

Blocks the signal if *b* is TRUE, or unblocks the signal if *b* is FALSE.

An activated signal disappears into hyperspace if it is blocked.

See also: isBlocked(), activate() and QObject::blockSignals().

bool QSignal::connect (const QObject * receiver, const char * member)

Connects the signal to *member* in object *receiver.*

See also: disconnect() and QObject::connect().

bool QSignal::disconnect (const QObject * receiver, const char * member=0)

Disonnects the signal from *member* in object *receiver.*

See also: connect() and QObject::disconnect().

bool QSignal::isBlocked () const

Returns TRUE if the signal is blocked, or FALSE if it is not blocked.

The signal is not blocked by default.

See also: block() and QObject::signalsBlocked().

const char * QSignal::name () const

Returns the name of this signal object.

Since QObject is a private base class, we have added this function, which calls QObject::name().

void QSignal::setName (const char * name)

Sets the name of this signal object to *name.*

Since QObject is a private base class, we have added this function, which calls QObject::setName().

157.2.1. URL

Generated from http://www.troll.no/qt/qsignal.html.

158. QSignalMapper Class Reference

`#include <qsignalmapper.h>`

Inherits QObject.

158.1. Detailed Description

A QSignalMapper object bundles signals from identifiable senders.

Collects a set of parameterless signals, re-emitting them with an integer or string parameters corresponding to the object which sent the signal.

158.2. Member Function Documentation

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

Constructs a QSignalMapper. Like all QObjects, it will be deleted when the parent is deleted.

QSignalMapper::~QSignalMapper ()

Destructs the QSignalMapper.

void QSignalMapper::map () `[slot]`

This slot emits signals based on which object sends signals to it.

void QSignalMapper::mapped (const char *) `[signal]`

This signal is emitted when map() is signalled from an object which has a string mapping set.

See also: setMapping(const and char*).

void QSignalMapper::mapped (int) `[signal]`

This signal is emitted when map() is signalled from an object which has an integer mapping set.

See also: setMapping(int).

void QSignalMapper::removeMappings (const QObject * sender)

Removes all mappings for *sender*. This is done automatically when mapped objects are destroyed.

void QSignalMapper::setMapping (const QObject * sender, const char * identifier)

Adds a mapping such that when map() is signalled from the given sender, the signal mapper(identifier) is emitted.

There may be at most one string identifier for each object, and it may not be null.

void QSignalMapper::setMapping (const QObject * sender, int identifier)

Adds a mapping such that when map() is signalled from the given sender, the signal mapper(identifier) is emitted.

There may be at most one integer identifier for each object.

158.2.1. URL

Generated from http://www.troll.no/qt/qsignalmapper.html.

159. QSize Class Reference

`#include <qsize.h>`

159.1. Detailed Description

The QSize class defines the size of a two-dimensional object.

A size is specified by a width and a height.

The coordinate type is QCOORD (defined in qwindowdefs.h as `short)`. The minimum value of QCOORD is QCOORD_MIN (-32768) and the maximum value is QCOORD_MAX (32767).

See also: QPoint and QRect.

Examples:
tictac/tictac.cpp
xform/xform.cpp
pref/pref.cpp
progress/progress.cpp

159.2. Member Function Documentation

QSize::QSize ()

Constructs a size with invalid (negative) width and height.

QSize::QSize (int w, int h)

Constructs a size with width *w* and height *h.*

QSize QSize::boundedTo (const QSize & otherSize) const

Returns a size with the minimum width and height of this size and *otherSize.*

QSize QSize::expandedTo (const QSize & otherSize) const

Returns a size with the maximum width and height of this size and *otherSize.*

int QSize::height () const

Returns the height.

See also: width().

Examples:
pref/pref.cpp

bool QSize::isEmpty () const

Returns TRUE if the width is <= 0 or the height is <= 0, otherwise FALSE.

bool QSize::isNull () const

Returns TRUE if the width is 0 and the height is 0, otherwise FALSE.

bool QSize::isValid () const

Returns TRUE if the width is equal to or greater than 0 and the height is equal to or greater than 0, otherwise FALSE.

QSize & QSize::operator*= (float c)

Multiplies both the width and height with *c* and returns a reference to the size.

Note that the result is truncated.

QSize & QSize::operator*= (int c)

Multiplies both the width and height with *c* and returns a reference to the size.

QSize & QSize::operator+= (const QSize & s)

Adds *s* to the size and returns a reference to this size.

Example:

```
QSize s(  3, 7 );
QSize r( -1, 4 );
s += r;                     // s becomes (2,11)
```

QSize & QSize::operator-= (const QSize & s)

Subtracts *s* from the size and returns a reference to this size.

Example:

```
QSize s(  3, 7 );
QSize r( -1, 4 );
s -= r;                     // s becomes (4,3)
```

QSize & QSize::operator/= (float c)

Divides both the width and height by *c* and returns a reference to the size.

Note that the result is truncated.

QSize & QSize::operator/= (int c)

Divides both the width and height by *c* and returns a reference to the size.

QCOORD & QSize::rheight ()

Returns a reference to the height.

Using a reference makes it possible to directly manipulate the height.

Example:

```
QSize s( 100, 10 );
s.rheight() += 5;           // s becomes (100,15)
```

See also: rwidth().

QCOORD & QSize::rwidth ()

Returns a reference to the width.

Using a reference makes it possible to directly manipulate the width.

Example:

```
QSize s( 100, 10 );
s.rwidth() += 20;           // s becomes (120,10)
```

See also: rheight().

void QSize::setHeight (int h)

Sets the height to *h.*

See also: height() and setWidth().

Examples:
pref/pref.cpp

void QSize::setWidth (int w)

Sets the width to *w.*

See also: width() and setHeight().

Examples:
pref/pref.cpp

void QSize::transpose ()

Swaps the values of width and height.

int QSize::width () const

Returns the width.

See also: height().

Examples:
pref/pref.cpp

159.3. Related Functions

bool operator== (const QSize & s1, const QSize & s2)

Returns TRUE if *s1* and *s2* are equal, or FALSE if they are different.

bool operator!= (const QSize & s1, const QSize & s2)

Returns TRUE if *s1* and *s2* are different, or FALSE if they are equal.

QSize operator+ (const QSize & s1, const QSize & s2)

Returns the sum of *s1* and *s2;* each component is added separately.

QSize operator- (const QSize & s1, const QSize & s2)

Returns *s2* subtracted from *s1;* each component is subtracted separately.

QSize operator* (const QSize & s, int c)

Multiplies *s* by *c* and returns the result.

QSize operator* (int c, const QSize & s)

Multiplies *s* by *c* and returns the result.

QSize operator* (const QSize & s, float c)

Multiplies *s* by *c* and returns the result.

QSize operator* (float c, const QSize & s)

Multiplies *s* by *c* and returns the result.

QSize operator/ (const QSize & s, int c)

Divides *s* by *c* and returns the result.

QSize operator/ (const QSize & s, float c)

Divides *s* by *c* and returns the result.

Note that the result is truncated.

QDataStream & operator<< (QDataStream & s, const QSize & sz)

Writes the size to the stream and returns a reference to the stream.

Serialization format: [width (INT16), height (INT16)].

QDataStream & operator>> (QDataStream & s, QSize & sz)

Reads the size from the stream and returns a reference to the stream.

159.3.1. URL

Generated from http://www.troll.no/qt/qsize.html.

160. QSlider Class Reference

`#include <qslider.h>`

Inherits QRangeControl and QWidget.

160.1. Detailed Description

The QSlider widget provides a vertical or horizontal slider.

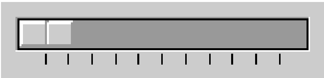
A slider is used to let the user control a value within a program-definable range. In contrast to a QScrollBar, the QSlider widget has a constant size slider and no arrow buttons.

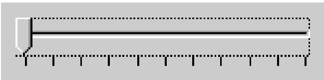
QSlider only offers integer ranges.

The recommended thickness of a slider is given by sizeHint().

Tickmarks may be added using setTickmarks().

A slider has a default focusPolicy() of *TabFocus*.





See also: QScrollBar, QSpinBox and GUI Design Handbook: Slider

Examples:
xform/xform.cpp
pref/pref.cpp
widgets/widgets.cpp

160.2. Member Function Documentation

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

Constructs a vertical slider.

The *parent* and *name* arguments are sent to the QWidget constructor.

QSlider::QSlider (Orientation orientation, QWidget * parent=0, const char * name=0)

Constructs a slider.

The *orientation* must be QSlider::Vertical or QSlider::Horizontal.

The *parent* and *name* arguments are sent to the QWidget constructor.

QSlider::QSlider (int minValue, int maxValue, int pageStep, int value, Orientation orientation, QWidget * parent=0, const char * name=0)

Constructs a slider.

Arguments:

- *minValue* is the minimum slider value.

- *maxValue* is the maximum slider value.
- *step* is the page step value.
- *value* is the initial value.
- *orientation* must be QSlider::Vertical or QSlider::Horizontal. The *parent* and *name* arguments are sent to the QWidget constructor.

void QSlider::addStep () `[slot]`

Moves the slider one pageStep() upwards.

void QSlider::drawTicks (QPainter * p, int d, int w, int i=1) const `[protected]`

Using *p,* draws tickmarks at a distance of *d* from the edge of the widget, using *w* pixels and with an interval of *i.*

void QSlider::drawWinGroove (QPainter * p, QCOORD c) `[protected]`

Draws the "groove" on which the slider moves, using the painter *p. c* gives the distance from the top (or left) edge of the widget to the center of the groove.

void QSlider::focusInEvent (QFocusEvent *) `[virtual protected]`

Handles focus in events for the slider.

Reimplemented from QWidget.

void QSlider::keyPressEvent (QKeyEvent * e) `[virtual protected]`

Handles key press events for the slider.

Reimplemented from QWidget.

void QSlider::mouseMoveEvent (QMouseEvent * e) `[virtual protected]`

Handles mouse move events for the slider.

Reimplemented from QWidget.

void QSlider::mousePressEvent (QMouseEvent * e) `[virtual protected]`

Handles mouse press events for the slider.

Reimplemented from QWidget.

void QSlider::mouseReleaseEvent (QMouseEvent *) `[virtual protected]`

Handles mouse release events for the slider.

Reimplemented from QWidget.

Orientation QSlider::orientation () const

Returns the slider orientation; QSlider::Vertical or QSlider::Horizontal.

See also: setOrientation().

void QSlider::paintEvent (QPaintEvent * e) `[virtual protected]`

Handles paint events for the slider.

Reimplemented from QWidget.

void QSlider::paintSlider (QPainter * p, const QRect & r) `[virtual protected]`

Paints the slider button using painter *p* with size and position given by *r*. Reimplement this function to change the look of the slider button.

void QSlider::rangeChange () `[virtual protected]`

Implements the virtual QRangeControl function.

Reimplemented from QRangeControl.

void QSlider::resizeEvent (QResizeEvent *) `[virtual protected]`

Handles resize events for the slider.

Reimplemented from QWidget.

void QSlider::setOrientation (Orientation orientation)

Sets the slider orientation. The *orientation* must be QSlider::Vertical or QSlider::Horizontal.

See also: orientation().

void QSlider::setPalette (const QPalette & p) `[virtual]`

Reimplements the virtual function QWidget::setPalette().

Sets the background color to the mid color for Motif style sliders.

Reimplemented from QWidget.

void QSlider::setTickInterval (int i) `[virtual]`

Sets the interval between tickmarks to *i*. This is a value interval, not a pixel interval. If *i* is 0, the slider will choose between lineStep() and pageStep(). The initial value of tickInterval() is 0.

See also: tickInterval(), QRangeControl::lineStep() and QRangeControl::pageStep().

Examples:
    widgets/widgets.cpp

void QSlider::setTickmarks (TickSetting s) `[virtual]`

Sets the way tickmarks are displayed by the slider. *s* can take the following values:

- `NoMarks`
- `Above`
- `Left`
- `Below`
- `Right`
- `Both` The initial value is `NoMarks.`

See also: tickmarks() and setTickInterval().

Examples:
widgets/widgets.cpp

void QSlider::setTracking (bool enable)

Enables slider tracking if *enable* is TRUE, or disables tracking if *enable* is FALSE.

If tracking is enabled (default), the slider emits the valueChanged() signal whenever the slider is being dragged. If tracking is disabled, the slider emits the valueChanged() signal when the user releases the mouse button (unless the value happens to be the same as before).

See also: tracking().

Examples:
xform/xform.cpp

void QSlider::setValue (int value) `[slot]`

Makes QRangeControl::setValue() available as a slot.

Examples:
xform/xform.cpp

QSize QSlider::sizeHint () const `[virtual]`

Returns the recommended size of the slider. Only the thickness is relevant.

Examples:
widgets/widgets.cpp

Reimplemented from QWidget.

void QSlider::sliderMoved (int value) `[signal]`

This signal is emitted when the slider is dragged, with the new slider value as an argument.

void QSlider::sliderPressed () `[signal]`

This signal is emitted when the user presses the slider with the mouse.

QRect QSlider::sliderRect () const

Returns the slider handle rectangle. (The actual moving-around thing.).

void QSlider::sliderReleased () `[signal]`

This signal is emitted when the user releases the slider with the mouse.

void QSlider::subtractStep () `[slot]`

Moves the slider one pageStep() downwards.

int QSlider::thickness () const `[virtual protected]`

Returns the number of pixels to use for the business part of the slider (i.e. the non-tickmark portion). The remaining space is shared equally between the tickmark regions. This function and sizeHint() are closely related; if you change one, you almost certainly have to change the other.

int QSlider::tickInterval () const

Returns the interval between tickmarks. Returns 0 if the slider chooses between pageStep() and lineStep().

See also: setTickInterval().

QSlider::TickSetting QSlider::tickmarks() const

Returns the tickmark settings for this slider.

See also: setTickmarks().

bool QSlider::tracking () const

Returns TRUE if tracking is enabled, or FALSE if tracking is disabled.

Tracking is initially enabled.

See also: setTracking().

void QSlider::valueChange () `[virtual protected]`

Implements the virtual QRangeControl function.

Reimplemented from QRangeControl.

void QSlider::valueChanged (int value) `[signal]`

This signal is emitted when the slider value is changed, with the new slider value as an argument.

160.2.1. URL

Generated from http://www.troll.no/qt/qslider.html.

161. QSocketNotifier Class Reference

```
#include <qsocketnotifier.h>
```

Inherits QObject.

161.1. Detailed Description

The QSocketNotifer class provides support for socket callbacks.

This class makes it possible to write e.g. asynchronous TCP/IP socket-based code in Qt. Using synchronous socket operations blocks the program, which is clearly not acceptable for an event-based GUI program.

Once you have opened a non-blocking socket (either for TCP, UDP, a unix-domain socket, or any other protocol family your operating system supports), you can create a socket notifier to monitor the socket. Then connect the activated() signal to the slot you want to be called when a socket event occurs.

There are three types of socket notifiers (read, write and exception) and you must specify one of these in the constructor.

The type specifies when the activated() signal is to be emitted:

1 `QSocketNotifier::Read`: There is data to be read (socket read event).

2 `QSocketNotifier::Write`: Data can be written (socket write event).

3 `QSocketNofifier::Exception`: An exception has ocurred (socket exception event). We recommend against using this.

For example, if you need to monitor both reads and writes for the same socket, you must create two socket notifiers.

Example:

```
int sockfd;                              // socket identifier
struct sockaddr_in sa;                   // should contain host address
sockfd = socket( AF_INET, SOCK_STREAM, 0 ); // create TCP socket
// make the socket non-blocking here, usually using fcntl( O_NONBLOCK )
::connect( sockfd, (struct sockaddr*)&sa,   // connect to host
           sizeof(sa) );                 //   NOT QObject::connect()!
QSocketNotifier *sn;
sn = new QSocketNotifier( sockfd, QSocketNotifier::Read, parent );
QObject::connect( sn, SIGNAL(activated(int)),
                  myObject, SLOT(dataReceived()) );
```

The optional *parent* argument can be set to make the socket notifier a child of some widget and therefore be automatically destroyed when the widget is destroyed.

For read notifiers, it makes little sense to connect the activated() signal to more than one slot, because the data can be read from the socket only once.

Make sure to disable the socket notifier for write operations when there is nothing to be written, otherwise the notifier fires on every pass of the main event loop. The socket notifier is enabled when it is created.

Also observe that if you do not read all the available data when the read notifier fires, it fires again and again.

If you disable the read notifier, your program may deadlock. Avoid it if you do not know what you are doing. (The same applies to exception notifiers if you have to use that, for instance if you *have* to use TCP urgent data.)

If you need a time-out for your sockets, you can use either timer events or the QTimer class.

Socket action is detected in the main event loop of Qt. The X11 version of Qt has has a single UNIX select() call which incorporates all socket notifiers and the X socket.

Note that on XFree86 for OS/2, select() only works in the thread in which main() is running, therefore you should use that thread for GUI operations.

161.2. Member Function Documentation

QSocketNotifier::QSocketNotifier (int socket, Type type, QObject * parent=0, const char * name=0)

Constructs a socket notifier with a *parent* and a *name.*

Arguments:

- *socket* is the socket to be monitored.
- *type* specifies the socket operation to be detected; `QSocketNotifier::Read`, `QSocketNotifier::Write` or `QSocketNotifier::Exception`. The *parent* and *name* arguments are sent to the QObject constructor.

The socket notifier is initially enabled. It is generally advisable to explicitly enable or disable it, especially for write notifiers.

See also: setEnabled() and isEnabled().

QSocketNotifier::~QSocketNotifier ()

Destroys the socket notifier.

void QSocketNotifier::activated (int socket) `[signal]`

This signal is emitted under certain conditions, specified by the notifier type:

1 `QSocketNotifier::Read`: There is data to be read (socket read event).

2 `QSocketNotifier::Write`: Data can be written (socket write event).

3 `QSocketNofifier::Exception`: An exception has ocurred (socket exception event).

The *socket* argument is the socket identifier.

See also: type() and socket().

bool QSocketNotifier::event (QEvent * e) `[virtual protected]`

Handles events for the socket notifier object.

Emits the activated() signal when a `Event_SockAct` is received.

Reimplemented from QObject.

bool QSocketNotifier::isEnabled () const

Returns TRUE if the notifier is enabled, or FALSE if it is disabled.

See also: setEnabled().

void QSocketNotifier::setEnabled (bool enable)

Enables the notifier if *enable* is TRUE, or disables it if *enable* is FALSE.

The notifier is by default enabled.

If the notifier is enabled, it emits the activated() signal whenever a socket event corresponding to its type occurs. If it is disabled, it ignores socket events (the same effect as not creating the socket notifier).

Disable the socket notifier for writes if there is nothing to be written, otherwise your program hogs the CPU.

See also: isEnabled() and activated().

int QSocketNotifier::socket () const

Returns the socket identifier specified to the constructor.

See also: type().

Type QSocketNotifier::type () const

Returns the socket event type specified to the constructor; `QSocketNotifier::Read`, `QSocketNotifier::Write` or `QSocketNotifier::Exception`.

See also: socket().

161.2.1. URL

Generated from http://www.troll.no/qt/qsocketnotifier.html.

162. QSpinBox Class Reference

#include <qspinbox.h>

Inherits QFrame and QRangeControl.

162.1. Detailed Description

The QSpinBox class provides a spin box widget, sometimes called up-down widget, little arrows widget or spin button.

QSpinBox allows the user to choose a numeric value, either by clicking the up/down buttons to increase/decrease the value currently displayed, or by typing the value directly into the spin box.

Every time the value changes, QSpinBox emits the valueChanged() signal. The current value can be fetched with value() and set with setValue().

The spin box clamps the value within a numeric range, see QRangeControl for details. Clicking the up/down down buttons (or using the keyboard accelerators: Up-arrow and Down-arrow) will increase or decrease the current value in steps of size lineStep().

Most spin boxes are directional, but QSpinBox can also operate as a circular spin box, i.e. if the range is 0-99 and the current value is 99, clicking Up will give 0. Use setWrapping() to if you want circular behavior.

The displayed value can be appended and/or prepended with an arbitray string indicating for example the unit of measurement. See setPrefix() and setSuffix().

It is often desirable to give the user a special, often default, choice in addition to the range of numeric values. See setSpecialValueText() for how to do this with QSpinBox.

The default focus policy is StrongFocus.

QSpinBox can easily be subclassed to allow the user to input other things than a numeric value, as long as the allowed input can be mapped down to a range of integers. This can be done by overriding the virtual functions mapValueToText() and mapTextToValue() and setting another, suitable validator using setValidator().

See also: QScrollBar, QSlider and GUI Design Handbook: Spin Box

Examples:
widgets/widgets.cpp

162.2. Member Function Documentation

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

Creates a spin box with the default QRangeControl range and step value.

See also: minValue(), maxValue(), setRange(), lineStep() and setSteps().

QSpinBox::QSpinBox (int minValue, int maxValue, int step = 1, QWidget * parent = 0, const char * name = 0)

Creates a spin box with range from *minValue* to *maxValue* inclusive, with step value *step.* The value is set to *minValue.*

See also: minValue(), maxValue(), setRange(), lineStep() and setSteps().

QSpinBox::˜QSpinBox ()

Deletes the spin box, freeing all memory and other resoures.

QString QSpinBox::cleanText () const `[virtual]`

Returns a copy of the current text of the spin box with any prefix and/or suffix and white space at the start and end removed.

See also: text(), setPrefix() and setSuffix().

QString QSpinBox::currentValueText () `[protected]`

Returns the full text of the current value, i.e. including any prefix, suffix or special-value text.

QPushButton* QSpinBox::downButton () const `[protected]`

Returns a pointer to the embedded 'down' button.

QLineEdit* QSpinBox::editor () const `[protected]`

Returns a pointer to the embedded QLineEdit.

void QSpinBox::enabledChange (bool) `[virtual protected]`

Reimplemented for internal purposes.

Reimplemented from QWidget.

bool QSpinBox::eventFilter (QObject * obj, QEvent * ev) `[virtual protected]`

Intercepts and handles those events coming to the embedded QLineEdit which have special meaning for the QSpinBox.

Reimplemented from QObject.

void QSpinBox::fontChange (const QFont &) `[virtual protected]`

Reimplemented for internal purposes.

Reimplemented from QWidget.

void QSpinBox::interpretText () `[virtual protected]`

Called after the user has manually edited the contents of the spin box. Interprets the text using mapTextToValue(), and calls setValue() if successful.

int QSpinBox::mapTextToValue (bool * ok) `[virtual protected]`

This virtual function is used by the spin box whenever it needs to interpret the text entered by the user as a value. The default implementation tries to interpret it as an integer in the standard way, and returns the integer value.

Override this function in in a subclass if you want a specialized spin box, handling something else than integers. It should call text() (or cleanText()) and return the value corresponding to that text. If the text does not represent a legal value (uninterpretable), the bool pointed to by *ok* should be set to FALSE.

This function need not be concerned with special-value text, the QSpinBox handles that automatically.

See also: interpretText() and mapValueToText().

QString QSpinBox::mapValueToText (int v) `[virtual protected]`

This virtual function is used by the spin box whenever it needs to display value *v*. The default implementation returns a string containing *v* printed in the standard way.

Override this function in in a subclass if you want a specialized spin box, handling something else than integers. This function need not be concerned with prefix or suffix or special-value text, the QSpinBox handles that automatically.

See also: updateDisplay() and mapTextToValue().

void QSpinBox::paletteChange (const QPalette &) `[virtual protected]`

Reimplemented for internal purposes.

Reimplemented from QWidget.

const char* QSpinBox::prefix () const `[virtual]`

Returns the currently set prefix, or 0 if no prefix is currently set.

See also: setPrefix(), setSuffix() and suffix().

void QSpinBox::rangeChange () `[virtual protected]`

This method gets called by QRangeControl whenever the range has changed. It adjusts the default validator and updates the display.

void QSpinBox::resizeEvent (QResizeEvent *) `[virtual protected]`

Handles resize events for the spin box.

Reimplemented from QWidget.

void QSpinBox::setPrefix (const char * text) `[virtual slot]`

Sets the prefix to *text*. The prefix is prepended to the start of the displayed value. Typical use is to indicate the unit of measurement to the user.

To turn off the prefix display, call this function with 0 or an empty string as parameter. The default is no prefix.

See also: prefix(), setSuffix() and suffix().

void QSpinBox::setSpecialValueText (const char * text)

Sets the special-value text to *text.* If set, the spin box will display this text instead of a numeric value whenever the current value is equal to minVal(). Typically used for indicating that this choice has a special (default) meaning.

For example, if you use a spin box for letting the user choose margin width in a print dialog, and your application is able to automatically choose a good margin width, you can set up the spin box like this:

```
QSpinBox marginBox( -1, 20, 1, parent, "marginBox" );
marginBox->setSuffix( " mm" );
marginBox->setSpecialValueText( "Auto" );
```

The user will then be able to choose a margin width from 0-20 millimeters, or select "Auto" to leave it to the application to choose. Your code must then interpret the spin box value of -1 as the user requesting automatic margin width.

Neither prefix nor suffix, if set, are added to the special-value text when displayed.

To turn off the special-value text display, call this function with 0 or an empty string as parameter. The default is no special-value text, i.e. the numeric value is shown as usual.

See also: specialValueText().

Examples:
widgets/widgets.cpp

void QSpinBox::setSuffix (const char * text) `[virtual slot]`

Sets the suffix to *text.* The suffix is appended to the end of the displayed value. Typical use is to indicate the unit of measurement to the user.

To turn off the suffix display, call this function with 0 or an empty string as parameter. The default is no suffix.

See also: suffix(), setPrefix() and prefix().

Examples:
widgets/widgets.cpp

void QSpinBox::setValidator (QValidator * v)

Sets the validator to *v.* The default is to use a suitable QIntValidator.

void QSpinBox::setValue (int value) `[virtual slot]`

Sets the current value of the spin box to *value.* This is QRangeControl::setValue() made available as a slot.

void QSpinBox::setWrapping (bool on)

Setting wrapping to TRUE will allow the value to be stepped from the highest value to the lowest, and vice versa. By default, wrapping is turned off.

See also: wrapping(), minValue(), maxValue() and setRange().

QSize QSpinBox::sizeHint () const `[virtual]`

Returns a good-looking size for the spin box.

Examples:
widgets/widgets.cpp

Reimplemented from QWidget.

const char* QSpinBox::specialValueText () const

Returns the currently special-value text, or 0 if no special-value text is currently set.

See also: setSpecialValueText().

void QSpinBox::stepDown () `[virtual slot]`

Decreases the current value one step, wrapping as necessary. This is the same as clicking on the pointing-down button, and can be used for e.g. keyboard accelerators.

See also: stepUp(), subtractLine(), lineStep(), setSteps(), setValue() and value().

void QSpinBox::stepUp () `[virtual slot]`

Increases the current value one step, wrapping as necessary. This is the same as clicking on the pointing-up button, and can be used for e.g. keyboard accelerators.

See also: stepDown(), addLine(), lineStep(), setSteps(), setValue() and value().

void QSpinBox::styleChange (GUIStyle) `[virtual protected]`

Reimplemented for internal purposes.

Reimplemented from QWidget.

const char* QSpinBox::suffix () const `[virtual]`

Returns the currently set suffix, or 0 if no suffix is currently set.

See also: setSuffix(), setPrefix() and suffix().

const char * QSpinBox::text () const

Returns the current text of the spin box.

See also: value().

void QSpinBox::textChanged () `[protected slot]`

This slot gets called whenever the user edits the text of the spin box.

QPushButton* QSpinBox::upButton () const `[protected]`

Returns a pointer to the embedded 'up' button.

void QSpinBox::updateDisplay () `[virtual protected]`

Updates the contents of the embedded QLineEdit to reflect current value, using mapValueToText(). Also enables/disables the push buttons accordingly.

See also: mapValueToText().

void QSpinBox::valueChange () `[virtual protected]`

This method gets called by QRangeControl whenever the value has changed. Updates the display and emits the valueChanged() signals.

void QSpinBox::valueChanged (const char * valueText) `[signal]`

This signal is emitted whenever the valueChanged(int) signal is emitted, i.e. every time the value of the spin box changes (whatever the cause - by setValue(), by a keyboard accelerator, by mouse clicks etc.).

The string pointed to by *valueText* is the same string that is displayed in the edit field of the spin box.

See also: value().

void QSpinBox::valueChanged (int value) `[signal]`

This signal is emitted every time the value of the spin box changes (whatever the cause - by setValue(), by a keyboard accelerator, by mouse clicks etc.).

Note that it is emitted *every* time, not just for the "final" step - if the user clicks 'up' three times, this signal is emitted three times.

See also: value().

bool QSpinBox::wrapping () const

Returns the current setWrapping() value.

162.2.1. URL

Generated from http://www.troll.no/qt/qspinbox.html.

163. QSplitter Class Reference

`#include <qsplitter.h>`

Inherits QFrame.

163.1. Detailed Description

QSplitter implements a splitter widget.

A splitter lets the user control the size of child widgets by dragging the boundary between the children.

The current implementation is limited to two children. The two widgets to be managed are the first two children added. If you need to split more than two widgets, you can nest splitters (although it may be difficult to control the relative sizing to your requirements).

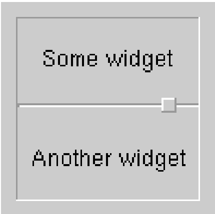
To show a QListBox and a QMultiLineEdit side by side:

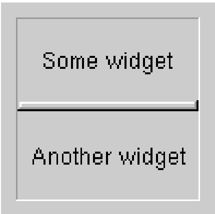
```
QSplitter *split = new QSplitter( parent );
QListBox *lb = new QListBox( split );
QMultiLineEdit *lb = new QMultiLineEdit( split );
```

In QSplitter the boundary can be either horizontal or vertical. The default is horizontal (the children are side by side) and you can use setOrientation(QSplitter::Vertical) to set it to vertical.

By default, both widgets can be as large or as small as the user wishes. You can naturally use setMinimumSize() and/or setMaximumSize() on the children. Use setResizeMode() to specify that a widget should keep its size when the splitter is resized.

QSplitter normally resizes the children only at the end of a resize operation, but if you call setOpaqueResize(TRUE), the widgets are resized as often as possible.





See also: QTabBar.

163.2. Member Function Documentation

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

Creates a horizontal splitter.

QSplitter::QSplitter (Orientation o, QWidget * parent=0, const char * name=0)

Creates splitter with orientation *o*.

int QSplitter::adjustPos (int p) `[protected]`

Returns the legal position of the splitter closest to *p*.

void QSplitter::childInsertEvent (QChildEvent * c) `[protected]`

Tells the splitter that a child widget has been inserted.

void QSplitter::childRemoveEvent (QChildEvent * c) `[protected]`

Tells the splitter that a child widget has been removed.

void QSplitter::drawSplitter (QPainter * p, QCOORD x, QCOORD y, QCOORD w, QCOORD h) `[virtual protected]`

Draws the splitter handle in the rectangle described by *x, y, w, h* using painter *p*.

bool QSplitter::event (QEvent * e) `[virtual]`

Reimplemented to provide childRemoveEvent(), childInsertEvent() and layoutHintEvent() without breaking binary compatibility.

Reimplemented from QObject.

void QSplitter::layoutHintEvent (QEvent *) `[protected]`

Tells the splitter that a child widget has changed layout parameters.

void QSplitter::moveSplitter (QCOORD p) `[protected]`

Moves the left/top edge of the splitter handle as close as possible to *p* which is the distance from the left (or top) edge of the widget.

Only has effect if both widgets are set.

void QSplitter::moveToFirst (QWidget * w)

Moves *w* to the leftmost/top position.

void QSplitter::moveToLast (QWidget * w)

Moves *w* to the rightmost/bottom position.

bool QSplitter::opaqueResize () const

Returns TRUE if opaque resize is on, FALSE otherwise.

See also: setOpaqueResize().

QSplitter::Orientation QSplitter::orientation() const

Returns the orientation (`Horizontal` or `Vertical`) of the splitter.

See also: setOrientation().

void QSplitter::refresh ()

Updates the splitter state. You should not need to call this function during normal operations.

void QSplitter::setOpaqueResize (bool on = TRUE)

Sets opaque resize to *on*. Opaque resize is initially turned off.

See also: opaqueResize().

void QSplitter::setOrientation (Orientation o)

Sets the orientation to *o*. By default the orientation is horizontal (the two widgets are side by side).

See also: orientation().

void QSplitter::setResizeMode (QWidget * w, ResizeMode mode)

Sets resize mode of *w* to *mode*. *mode* can be one of:

- `Stretch` (the default) - *w* will resize when the splitter resizes
- `KeepSize` - *w* will keep its size.

void QSplitter::setRubberband (int p) `[protected]`

Shows a rubber band at position *p*. If *p* is negative, the rubber band is removed.

163.2.1. URL

Generated from http://www.troll.no/qt/qsplitter.html.

164. QStack Class Reference

```
#include <qstack.h>
```

164.1. Detailed Description

The QStack class is a template class that provides a stack.

QStack is implemented as both a template and a macro class. Define a template instance QStack<X> to create a stack that operates on pointers to X, or X*.

A stack is a collection that can be used to push and pop items.

See also: Collection Classes

164.2. Member Function Documentation

QStack::QStack ()

Creates and empty stack.

QStack::QStack (const QStack<type> & s)

Creates a stack by making a shallow copy of another stack.

QStack::~QStack ()

Destroys the stack. All items will be deleted if autoDelete() is TRUE.

QStack::operator type* () const

Returns a reference to the top item on the stack (most recently pushed). The stack is not changed.

type* QStack::current () const

Returns a reference to the top item on the stack (most recently pushed). The stack is not changed.

bool QStack::isEmpty () const

Returns TRUE is the stack contains no elements to be popped

QStack<type>& QStack::operator= (const QStack<type> & s)

Sets the contents of this stack by making a shallow copy of another stack. Elements currently in this stack will be deleted if autoDelete() is TRUE.

type* QStack::pop ()

Removes the top item from the stack and returns it.

void QStack::push (const type * d)

Adds an element to the top of the stack. Last in, first out.

bool QStack::remove ()

Removes the top item from the stack and deletes it if if autoDelete() is TRUE. Returns TRUE if there was an item to pop.

type* QStack::top () const

Returns a reference to the top item on the stack (most recently pushed). The stack is not changed.

164.2.1. URL

Generated from http://www.troll.no/qt/qstack.html.

165. QStatusBar Class Reference

#include <qstatusbar.h>

Inherits QWidget.

165.1. Detailed Description

The QStatusBar class provides a horizontal bar suitable for presenting status messages.

Each status message falls into one of three classes:

- `Temporary` - occupies most of the status bar briefly. Used e.g. for explaining tool tip texts or menu entries)
- `Normal` - occupies part of the status bar and may be hidden by temporary messages. Used e.g. for displaying the page and line number in a word processor.
- `Permanent` - occupies the far right of the statusbar and is never hidden. Used for important mode indications. Some applications put a Caps Lock indicator here.

QStatusBar lets you display all three sorts of messages.

To display a temporary message, you can call message(), or connect a suitable signal to it. To remove a temporary message, you can call clear(), or connect a signal to it.

There are two variants of message(), one which displays the message until the next clear() or mesage(), and one which also has a time limit.

Normal and permanent messages are displayed by creating a widget (typically a QLabel) and using addWidget() to add this widget to the status bar.

Finally, in Windows style QStatusBar also provides a Windows standard resize handle. In the X11 version Qt 1.40 this resize handle generally works differently than the one provided by the system; we hope to reduce this difference in the future.

See also: QToolBar, QMainWindow, QLabel, Microsoft Style Guide, and GUI Design Handbook: Status Bar.

165.2. Member Function Documentation

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

Constructs an empty status bar.

QStatusBar::~QStatusBar ()

Destroys the object and frees any allocated resources.

void QStatusBar::addWidget (QWidget * widget, int stretch, bool permanent = FALSE)

Adds *widget* to this status bar, with a width of *stretch.*

widget is permanently visible if *permanent* is TRUE, and is obscured by temporary messages if *permanent* is FALSE. The default is FALSE.

stretch is used to compute a suitable size for *widget.*

If *permanent* is TRUE, *widget* is located at the far right of the status bar. If *permanent* is FALSE (the default) *widget* is located just to the left of the first permanent widget.

This function may cause some flicker.

See also: removeWidget().

void QStatusBar::clear () `[slot]`

Removes any temporary message being shown.

See also: message().

void QStatusBar::hideOrShow () `[protected]`

Ensures that the right widgets are visible. Used by message() and clear().

void QStatusBar::message (const char * message) `[slot]`

Hide the normal status indications and display *message,* until clear() or another message() is called.

See also: clear().

void QStatusBar::message (const char * message, int ms) `[slot]`

Hide the normal status indications and display *message* for *ms* milli-seconds, or until clear() or another message() is called, whichever is shorter.

void QStatusBar::paintEvent (QPaintEvent *) `[virtual protected]`

Shows the temporary message, if appropriate.

Reimplemented from QWidget.

void QStatusBar::reformat () `[protected]`

Changes the status bar's appearance to account for item changes.

void QStatusBar::removeWidget (QWidget * widget)

Removes *widget* from the status bar.

This function may cause some flicker.

Note that *widget* is not deleted.

See also: addWidget().

165.2.1. URL

Generated from http://www.troll.no/qt/qstatusbar.html.

166. QStoredDrag Class Reference

`#include <qdragobject.h>`

Inherits QDragObject.

Inherited by QTextDrag and QUrlDrag.

166.1. Detailed Description

Simple stored-value drag object for arbitrary MIME data.

When a block of data only has one representation, you can use a QStoredDrag to hold it.

For detailed information about drag-and-drop, see the QDragObject class.

166.2. Member Function Documentation

QStoredDrag::QStoredDrag (const char * mimeType, QWidget * dragSource = 0, const char * name = 0)

Constructs a QStoredDrag. The parameters are passed to the QDragObject constructor, and the format is set to *mimeType*.

The data will be unset. Use setEncodedData() to set it.

QStoredDrag::~QStoredDrag ()

Destroys the drag object and frees all allocated resources.

QByteArray QStoredDrag::encodedData (const char * m) const `[virtual]`

Returns the stored data.

See also: setEncodedData().

void QStoredDrag::setEncodedData (const QByteArray & encodedData)

Sets the encoded data of this drag object to *encodedData*. The encoded data is what's delivered to the drop sites, and must be in a strictly defined and portable format.

The drag object can't be dropped (by the user) until this function has been called.

166.2.1. URL

Generated from http://www.troll.no/qt/qstoreddrag.html.

167. QStrIList Class Reference

`#include <qstrlist.h>`

Inherits QStrList.

167.1. Detailed Description

The QStrIList class provides a doubly linked list of `char*` with case insensitive compare.

This class is a QList<char> instance (a list of char*).

QStrIList is similar to QStrList except that it is case insensitive. The virtual compareItems() function is reimplemented and does a case insensitive string comparision. The inSort() function will insert strings in a sorted order.

The QStrListIterator class is an iterator for QStrList.

167.2. Member Function Documentation

QStrIList::QStrIList (bool deepCopies=TRUE)

Constructs a list of strings. Will make deep copies of all inserted strings if *deepCopies* is TRUE, or uses shallow copies if *deepCopies* is FALSE.

QStrIList::~QStrIList ()

Destroys the list. All strings are removed.

167.2.1. URL

Generated from http://www.troll.no/qt/qstrilist.html.

168. QString Class Reference

```
#include <qstring.h>
```

Inherits QByteArray.

168.1. Detailed Description

The QString class provides an abstraction of the classic C zero-terminated char array (*char**).

QString inherits QByteArray, which is defined as QArray<char>.

Since QString is a QArray, it uses explicit sharing with a reference count.

Note that for the QString methods that take a *const char ** parameter the results are undefined if the QString is not zero-terminated. It is legal for the *const char ** parameter to be 0.

A QString that has not been assigned to anything is *null,* i.e. both the length and data pointer is 0. A QString that references the empty string ("", a single '\0' char) is *empty.* Both null and empty QStrings are legal parameters to the methods. Assigning *const char * 0* to QString gives a null QString.

See also: Shared classes

Examples:

table/table.cpp
dclock/dclock.cpp
grapher/grapher.cpp
forever/forever.cpp
dirview/dirview.cpp
desktop/desktop.cpp
tooltip/tooltip.cpp
drawdemo/drawdemo.cpp
movies/main.cpp
picture/picture.cpp
xform/xform.cpp
application/application.cpp
pref/pref.cpp
progress/progress.cpp
layout/layout.cpp
qmag/qmag.cpp
showimg/showimg.cpp
biff/biff.cpp
widgets/widgets.cpp

168.2. Member Function Documentation

QString::QString ()

Constructs a null string.

See also: isNull().

QString::QString (const QString & s)

Constructs a shallow copy *s.*

See also: assign().

QString::QString (const char * str)

Constructs a string that is a deep copy of *str*.

If *str* is 0 a null string is created.

See also: isNull().

QString::QString (const char * str, uint maxlen)

Constructs a string that is a deep copy of *str,* that is no more than *maxlen* bytes long including the '\0'-terminator.

Example:

```
QString str("helloworld",6); // Assigns "hello" to str.
```

If *str* contains a 0 byte within the first *maxlen* bytes, the resulting QString will be terminated by the 0. If *str* is 0 a null string is created.

See also: isNull().

QString::QString (int size)

Constructs a string with room for *size* characters, including the '\0'-terminator. Makes a null string if *size* == 0.

If *size* > 0, then the first and last characters in the string are initialized to '\0'. All other characters are uninitialized.

See also: resize() and isNull().

QString::operator const char * () const

Returns the string data.

QString& QString::append (const char * str)

Appends *str* to the string and returns a reference to the string. Equivalent to operator+=().

Examples:
dirview/dirview.cpp

int QString::contains (char c, bool cs=TRUE) const

Returns the number of times the character *c* occurs in the string.

The match is case sensitive if *cs* is TRUE, or case insensitive if *cs* if FALSE.

int QString::contains (const QRegExp & rx) const

Counts the number of overlapping occurrences of *rx* in the string.

Example:

```
QString s = "banana and panama";
QRegExp r = QRegExp("a[nm]a", TRUE, FALSE);
s.contains( r );                              // 4 matches
```

See also: find() and findRev().

int QString::contains (const char * str, bool cs=TRUE) const

Returns the number of times *str* occurs in the string.

The match is case sensitive if *cs* is TRUE, or case insensitive if *cs* if FALSE.

This function counts overlapping substrings, for example, "banana" contains two occurrences of "ana".

See also: findRev().

QString QString::copy () const

Returns a deep copy of this string.

See also: detach().

bool QString::fill (char c, int len = -1)

Fills the string with *len* bytes of value *c,* followed by a '\0'-terminator.

If *len* is negative, then the current string length is used.

Returns FALSE is *len* is nonnegative and there is no memory to resize the string, otherwise TRUE is returned.

int QString::find (char c, int index=0, bool cs=TRUE) const

Finds the first occurrence of the character *c,* starting at position *index.*

The search is case sensitive if *cs* is TRUE, or case insensitive if *cs* is FALSE.

Returns the position of *c,* or -1 if *c* could not be found.

int QString::find (const QRegExp & rx, int index=0) const

Finds the first occurrence of the regular expression *rx,* starting at position *index.*

Returns the position of the next match, or -1 if *rx* was not found.

int QString::find (const char * str, int index=0, bool cs=TRUE) const

Finds the first occurrence of the string *str,* starting at position *index.*

The search is case sensitive if *cs* is TRUE, or case insensitive if *cs* is FALSE.

Returns the position of *str,* or -1 if *str* could not be found.

int QString::findRev (char c, int index=-1, bool cs=TRUE) const

Finds the first occurrence of the character *c,* starting at position *index* and searching backwards.

The search is case sensitive if *cs* is TRUE, or case insensitive if *cs* is FALSE.

Returns the position of *c,* or -1 if *c* could not be found.

int QString::findRev (const QRegExp & rx, int index=-1) const

Finds the first occurrence of the regular expression *rx,* starting at position *index* and searching backwards.

The search will start from the end of the string if *index* is negative.

Returns the position of the next match (backwards), or -1 if *rx* was not found.

int QString::findRev (const char * str, int index=-1, bool cs=TRUE) const

Finds the first occurrence of the string *str,* starting at position *index* and searching backwards.

The search is case sensitive if *cs* is TRUE, or case insensitive if *cs* is FALSE.

Returns the position of *str,* or -1 if *str* could not be found.

QString & QString::insert (uint index, char c)

Insert *c* into the string at (before) position *index* and returns a reference to the string.

If *index* is beyond the end of the string, the string is extended with spaces (ASCII 32) to length *index* and *c* is then appended.

Example:

```
QString s = "Yes";
s.insert( 3, '!');                          // s == "Yes!"
```

See also: remove() and replace().

QString & QString::insert (uint index, const char * s)

Insert *s* into the string before position *index.*

If *index* is beyond the end of the string, the string is extended with spaces (ASCII 32) to length *index* and *s* is then appended.

```
QString s = "I like fish";
s.insert( 2, "don't ");                     // s == "I don't like fish"
s = "x";
s.insert( 3, "yz" );                        // s == "x  yz"
```

Examples:
xform/xform.cpp

bool QString::isEmpty () const

Returns TRUE if the string is empty, i.e. if length() == 0. An empty string is not always a null string.

See example in isNull().

See also: isNull(), length() and size().

Examples:
application/application.cpp
qmag/qmag.cpp
showimg/showimg.cpp

bool QString::isNull () const

Returns TRUE if the string is null, i.e. if data() == 0. A null string is also an empty string.

Example:

```
QString a;          // a.data() == 0,  a.size() == 0, a.length() == 0
QString b == "";    // b.data() == "", b.size() == 1, b.length() == 0
a.isNull();         // TRUE, because a.data() == 0
a.isEmpty();        // TRUE, because a.length() == 0
b.isNull();         // FALSE, because b.data() == ""
```

```
b.isEmpty();        // TRUE, because b.length() == 0
```

See also: isEmpty(), length() and size().

QString QString::left (uint len) const

Returns a substring that contains the *len* leftmost characters of the string.

The whole string is returned if *len* exceeds the length of the string.

Example:

```
QString s = "Pineapple";
QString t = s.left( 4 );                    // t == "Pine"
```

See also: right() and mid().

QString QString::leftJustify (uint width, char fill=' ', bool truncate=FALSE) const

Returns a string of length *width* (plus '\0') that contains this string and padded by the *fill* character.

If the length of the string exceeds *width* and *truncate* is FALSE, then the returned string is a copy of the string. If the length of the string exceeds *width* and *truncate* is TRUE, then the returned string is a left(*width*).

Example:

```
QString s("apple");
QString t = s.leftJustify(8, '.');          // t == "apple..."
```

See also: rightJustify().

uint QString::length () const

Returns the length of the string, excluding the '\0'-terminator. Equivalent to calling `strlen(data())`.

Null strings and empty strings have zero length.

See also: size(), isNull() and isEmpty().

Examples:
showimg/showimg.cpp

QString QString::lower () const

Returns a new string that is the string converted to lower case.

Presently it only handles 7-bit ASCII, or whatever tolower() handles (if $LC_CTYPE is set, most UNIX systems do the Right Thing).

Example:

```
QString s("TeX");
QString t = s.lower();                      // t == "tex"
```

See also: upper().

QString QString::mid (uint index, uint len) const

Returns a substring that contains the *len* characters of this string, starting at position *index*.

Returns a null string if the string is empty or *index* is out of range. Returns the whole string from *index* if *index+len* exceeds the length of the string.

Example:

```
QString s = "Two pineapples";
QString t = s.mid( 4, 4 );                      // t == "pine"
```

See also: left() and right().

Examples:
qmag/qmag.cpp

QString & QString::operator+= (char c)

Appends *c* to the string and returns a reference to the string.

QString& QString::operator+= (const char * str)

Appends *str* to the string and returns a reference to the string.

QString & QString::operator= (const QString & s)

Assigns a shallow copy of *s* to this string and returns a reference to this string.

QString & QString::operator= (const char * str)

Assigns a deep copy of *str* to this string and returns a reference to this string.

If *str* is 0 a null string is created.

See also: isNull().

QString & QString::prepend (const char * s)

Prepend \s to the string. Equivalent to insert(0,s).

See also: insert().

QString & QString::remove (uint index, uint len)

Removes *len* characters starting at position *index* from the string and returns a reference to the string.

If *index* is too big, nothing happens. If *index* is valid, but *len* is too large, the rest of the string is removed.

```
QString s = "Montreal";
s.remove( 1, 4 );
// s == "Meal"
```

See also: insert() and replace().

QString & QString::replace (const QRegExp & rx, const char * str)

Replaces every occurrence of *rx* in the string with *str*. Returns a reference to the string.

Example:

```
QString s = "banana";
s.replace( QRegExp("a.*a"), "" );           // becomes "b"
QString s = "banana";
s.replace( QRegExp("^[bn]a"), " " );        // becomes " nana"
QString s = "banana";
```

```
s.replace( QRegExp("^[bn]a"), "" );          // NOTE! becomes ""
```

Examples:
qmag/qmag.cpp

QString & QString::replace (uint index, uint len, const char * s)

Replaces *len* characters starting at position *index* from the string with *s,* and returns a reference to the string.

If *index* is too big, nothing is deleted and *s* is inserted at the end of the string. If *index* is valid, but *len* is too large, *str* replaces the rest of the string.

```
QString s = "Say yes!";
s.replace( 4, 3, "NO" );                     // s == "Say NO!"
```

See also: insert() and remove().

bool QString::resize (uint len)

Extends or shrinks the string to *len* bytes, including the '\0'-terminator.

A \0-terminator is set at position `len - 1` unless `len == 0`.

Example:

```
QString s = "resize this string";
s.resize( 7 );                               // s == "resize"
```

See also: truncate().

QString QString::right (uint len) const

Returns a substring that contains the *len* rightmost characters of the string.

The whole string is returned if *len* exceeds the length of the string.

Example:

```
QString s = "Pineapple";
QString t = s.right( 5 );                    // t == "apple"
```

See also: left() and mid().

QString QString::rightJustify (uint width, char fill=' ', bool truncate=FALSE) const

Returns a string of length *width* (plus '\0') that contains pad characters followed by the string.

If the length of the string exceeds *width* and *truncate* is FALSE, then the returned string is a copy of the string. If the length of the string exceeds *width* and *truncate* is TRUE, then the returned string is a right(*width).*

Example:

```
QString s("pie");
QString t = s.rightJustify(8, '.');          // t == ".....pie"
```

See also: leftJustify().

bool QString::setExpand (uint index, char c)

Sets the character at position *index* to *c* and expands the string if necessary, filling with spaces.

Returns FALSE if this *index* was out of range and the string could not be expanded, otherwise TRUE.

QString & QString::setNum (double n, char f='g', int prec=6)

Sets the string to the printed value of *n*.

Arguments:

- *f* is the format specifier: 'f', 'F', 'e', 'E', 'g', 'G' (same as sprintf()).
- *prec* is the precision. Returns a reference to the string.

QString & QString::setNum (float n, char f='g', int prec=6)

Sets the string to the printed value of *n*.

Arguments:

- *f* is the format specifier: 'f', 'F', 'e', 'E', 'g', 'G' (same as sprintf()).
- *prec* is the precision. Returns a reference to the string.

QString & QString::setNum (int n)

Sets the string to the printed value of *n* and returns a reference to the string.

QString & QString::setNum (long n)

Sets the string to the printed value of *n* and returns a reference to the string.

QString & QString::setNum (short n)

Sets the string to the printed value of *n* and returns a reference to the string.

QString & QString::setNum (uint n)

Sets the string to the printed unsigned value of *n* and returns a reference to the string.

QString & QString::setNum (ulong n)

Sets the string to the printed unsigned value of *n* and returns a reference to the string.

QString & QString::setNum (ushort n)

Sets the string to the printed unsigned value of *n* and returns a reference to the string.

QString & QString::setStr (const char * str)

Makes a deep copy of *str*. Returns a reference to the string.

QString QString::simplifyWhiteSpace () const

Returns a new string that has white space removed from the start and the end, plus any sequence of internal white space replaced with a single space (ASCII 32).

White space means any ASCII code 9, 10, 11, 12, 13 or 32.

```
QString s = "  lots\t of\nwhite    space ";
QString t = s.simplifyWhiteSpace();         // t == "lots of white space"
```

See also: stripWhiteSpace().

QString & QString::sprintf (const char * format, ...)

Implemented as a call to the native vsprintf() (see your C-library manual).

If your string is shorter than 256 characters, this sprintf() calls resize(256) to decrease the chance of memory corruption. The string is resized back to its natural length before sprintf() returns.

Example:

```
QString s;
s.sprintf( "%d - %s", 1, "first" );          // result < 256 chars
QString big( 25000 );                        // very long string
big.sprintf( "%d - %s", 2, longString );     // result < 25000 chars
```

Warning: All vsprintf() implementations will write past the end of the target string (*this) if the format specification and arguments happen to be longer than the target string, and some will also fail if the target string is longer than some arbitrary implementation limit.

Giving user-supplied arguments to sprintf() is begging for trouble. Sooner or later someone *will* paste a 3000-character line into your application.

Examples:

dclock/dclock.cpp
forever/forever.cpp
tooltip/tooltip.cpp
drawdemo/drawdemo.cpp
movies/main.cpp
picture/picture.cpp
xform/xform.cpp
application/application.cpp
pref/pref.cpp
progress/progress.cpp
layout/layout.cpp
qmag/qmag.cpp
showimg/showimg.cpp
widgets/widgets.cpp

QString QString::stripWhiteSpace () const

Returns a new string that has white space removed from the start and the end.

White space means any ASCII code 9, 10, 11, 12, 13 or 32.

Example:

```
QString s = " space ";
QString t = s.stripWhiteSpace();            // t == "space"
```

See also: simplifyWhiteSpace().

double QString::toDouble (bool * ok=0) const

Returns the string converted to a `double` value.

If *ok* is non-null, **ok* is set to TRUE if there are no conceivable errors, and FALSE if the string is not a number at all, or if it has trailing garbage.

float QString::toFloat (bool * ok=0) const

Returns the string converted to a `float` value.

If *ok* is non-null, **ok* is set to TRUE if there are no conceivable errors, and FALSE if the string is not a number at all, or if it has trailing garbage.

int QString::toInt (bool * ok=0) const

Returns the string converted to a `int` value.

If *ok* is non-null, **ok* is set to TRUE if there are no conceivable errors, and FALSE if the string is not a number at all, or if it has trailing garbage.

long QString::toLong (bool * ok=0) const

Returns the string converted to a `long` value.

If *ok* is non-null, **ok* is set to TRUE if there are no conceivable errors, and FALSE if the string is not a number at all, or if it has trailing garbage.

short QString::toShort (bool * ok=0) const

Returns the string converted to a `short` value.

If *ok* is non-null, **ok* is set to TRUE if there are no conceivable errors, and FALSE if the string is not a number at all, or if it has trailing garbage.

uint QString::toUInt (bool * ok=0) const

Returns the string converted to an `unsigned int` value.

If *ok* is non-null, **ok* is set to TRUE if there are no conceivable errors, and FALSE if the string is not a number at all, or if it has trailing garbage.

ulong QString::toULong (bool * ok=0) const

Returns the string converted to an `unsigned long` value.

If *ok* is non-null, **ok* is set to TRUE if there are no conceivable errors, and FALSE if the string is not a number at all, or if it has trailing garbage.

ushort QString::toUShort (bool * ok=0) const

Returns the string converted to an `unsigned short` value.

If *ok* is non-null, **ok* is set to TRUE if there are no conceivable errors, and FALSE if the string is not a number at all, or if it has trailing garbage.

bool QString::truncate (uint pos)

Truncates the string at position *pos*.

Equivalent to calling `resize(pos+1)`.

Example:

```
QString s = "truncate this string";
s.truncate( 5 );                              // s == "trunc"
```

See also: resize().

QString QString::upper () const

Returns a new string that is the string converted to upper case.

Presently it only handles 7-bit ASCII, or whatever toupper() handles (if $LC_CTYPE is set, most UNIX systems do the Right Thing).

Example:

```
QString s("TeX");
QString t = s.upper();                    // t == "TEX"
```

See also: lower().

168.3. Related Functions

void * memmove (void * dst, const void * src, uint len)

This function is normally part of the C library. Qt implements memmove() for platforms that do not have it.

memmove() copies *len* bytes from *src* into *dst.* The data is copied correctly even if *src* and *dst* overlap.

char * qstrdup (const char * str)

Returns a duplicate string.

Allocates space for a copy of *str* (using `new`), copies it, and returns a pointer to the copy. If *src* is null, it immediately returns 0.

uint strlen (const char * str)

A safe strlen() function that overrides the one defined by the C library. The original strlen() function has been renamed cstrlen().

Returns the number of characters in *str,* or 0 if *str* is null.

char * strcpy (char * dst, const char * str)

A safe strcpy() function that overrides the one defined by the C library. The original strcpy() function has been renamed cstrcpy().

Copies all characters (including \0) from *str* into *dst* and returns a pointer to *dst.* If *src* is null, it immediately returns 0.

See also: qstrncpy().

char * qstrncpy (char * dst, const char * src, uint len)

A safe strncpy() function.

Copies all characters up to *len* bytes from *str* into *dst* and returns a pointer to *dst.* Guarantees that *dst* is \0-terminated. If *src* is null, it immediately returns 0.

See also: strcpy().

int strcmp (const char * str1, const char * str2)

A safe strcmp() function that overrides the one defined by the C library. The original strcmp() function has been renamed cstrcmp().

Compares *str1* and *str2*. Returns a negative value if *str1* is less than *str2,* 0 if *str1* is equal to *str2* or a positive value if *str1* is greater than *str2.*

Special case I: Returns 0 if *str1* and *str2* are both null.

Special case II: Returns a random nonzero value if *str1* is null or *str2* is null (but not both).

See also: strncmp(), stricmp() and strnicmp().

Examples:
desktop/desktop.cpp

int strncmp (const char * str1, const char * str2, uint len)

A safe strncmp() function that overrides the one defined by the C library. The original strncmp() function has been renamed cstrncmp().

Compares *str1* and *str2* up to *len* bytes.

Returns a negative value if *str1* is less than *str2,* 0 if *str1* is equal to *str2* or a positive value if *str1* is greater than *str2.*

Special case I: Returns 0 if *str1* and *str2* are both null.

Special case II: Returns a random nonzero value if *str1* is null or *str2* is null (but not both).

See also: strcmp(), stricmp() and strnicmp().

int stricmp (const char * str1, const char * str2)

A safe stricmp() function that overrides the one defined by the C library, if the C library has one.

Compares *str1* and *str2* ignoring the case.

Returns a negative value if *str1* is less than *str2,* 0 if *str1* is equal to *str2* or a positive value if *str1* is greater than *str2.*

Special case I: Returns 0 if *str1* and *str2* are both null.

Special case II: Returns a random nonzero value if *str1* is null or *str2* is null (but not both).

See also: strcmp(), strncmp() and strnicmp().

int strnicmp (const char * str1, const char * str2, uint len)

A safe strnicmp() function that overrides the one defined by the C library, if the C library has one.

Compares *str1* and *str2* up to *len* bytes ignoring the case.

Returns a negative value if *str1* is less than *str2,* 0 if *str1* is equal to *str2* or a positive value if *str1* is greater than *str2.*

Special case I: Returns 0 if *str1* and *str2* are both null.

Special case II: Returns a random nonzero value if *str1* is null or *str2* is null (but not both).

See also: strcmp(), strncmp() and stricmp().

QDataStream & operator<< (QDataStream & s, const QString & str)

Writes a string to the stream.

Output format: [length (Q_UINT32) data...].

QDataStream & operator>> (QDataStream & s, QString & str)

Reads a string from the stream.

bool operator== (const QString & s1, const QString & s2)

Returns TRUE if the two strings are equal, or FALSE if they are different.

Equivalent to `strcmp(s1,s2) == 0`

bool operator== (const QString & s1, const char * s2)

Returns TRUE if the two strings are equal, or FALSE if they are different.

Equivalent to `strcmp(s1,s2) == 0`

bool operator== (const char * s1, const QString & s2)

Returns TRUE if the two strings are equal, or FALSE if they are different.

Equivalent to `strcmp(s1,s2) == 0`

bool operator!= (const QString & s1, const QString & s2)

Returns TRUE if the two strings are different, or FALSE if they are equal.

Equivalent to `strcmp(s1,s2) != 0`

bool operator!= (const QString & s1, const char * s2)

Returns TRUE if the two strings are different, or FALSE if they are equal.

Equivalent to `strcmp(s1,s2) != 0`

bool operator!= (const char * s1, const QString & s2)

Returns TRUE if the two strings are different, or FALSE if they are equal.

Equivalent to `strcmp(s1,s2) != 0`

bool operator< (const QString & s1, const char * s2)

Returns TRUE if *s1* is alphabetically less than *s2,* otherwise FALSE.

Equivalent to `strcmp(s1,s2) < 0`

bool operator< (const char * s1, const QString & s2)

Returns TRUE if *s1* is alphabetically less than *s2,* otherwise FALSE.

Equivalent to `strcmp(s1,s2) < 0`

bool operator<= (const QString & s1, const char * s2)

Returns TRUE if *s1* is alphabetically less than or equal to *s2,* otherwise FALSE.

Equivalent to `strcmp(s1,s2) <= 0`

bool operator<= (const char * s1, const QString & s2)

Returns TRUE if *s1* is alphabetically less than or equal to *s2,* otherwise FALSE.

Equivalent to `strcmp(s1,s2) <= 0`

bool operator> (const QString & s1, const char * s2)

Returns TRUE if *s1* is alphabetically greater than *s2,* otherwise FALSE.

Equivalent to `strcmp(s1,s2) > 0`

bool operator> (const char * s1, const QString & s2)

Returns TRUE if *s1* is alphabetically greater than *s2,* otherwise FALSE.

Equivalent to `strcmp(s1,s2) > 0`

bool operator>= (const QString & s1, const char * s2)

Returns TRUE if *s1* is alphabetically greater than or equal to *s2,* otherwise FALSE.

Equivalent to `strcmp(s1,s2) >= 0`

bool operator>= (const char * s1, const QString & s2)

Returns TRUE if *s1* is alphabetically greater than or equal to *s2,* otherwise FALSE.

Equivalent to `strcmp(s1,s2) >= 0`

QString operator+ (const QString & s1, const QString & s2)

Returns the concatenated string of s1 and s2.

QString operator+ (const QString & s1, const char * s2)

Returns the concatenated string of s1 and s2.

QString operator+ (const char * s1, const QString & s2)

Returns the concatenated string of s1 and s2.

QString operator+ (const QString & s, char c)

Returns the concatenated string of s and c.

QString operator+ (char c, const QString & s)

Returns the concatenated string of c and s.

168.3.1. URL

Generated from http://www.troll.no/qt/qstring.html.

169. QStrList Class Reference

#include <qstrlist.h>

Inherits QList.

Inherited by QStrIList.

169.1. Detailed Description

The QStrList class provides a doubly linked list of `char*`.

This class is a QList<char> instance (a list of char*).

QStrList can make deep or shallow copies of the strings that are inserted.

A deep copy means to allocate space for the string and then copy the string data into it. A shallow copy is just a copy of the pointer value and not the string data.

The disadvantage with shallow copies is that since a pointer can only be deleted once, the program must put all strings in a central place and know when it is safe to delete them (i.e. when the strings are no longer referenced by other parts of the program). This can make the program more complex. The advantage of shallow copies is that shallow copies consume far less memory than deep copies. It is also much faster to copy a pointer (typically 4 or 8 bytes) than to copy string data.

The virtual compareItems() function is reimplemented and does a case sensitive string comparision. The inSort() function will insert strings in a sorted order.

The QStrListIterator class is an iterator for QStrList.

Examples:
    showimg/showimg.cpp

169.2. Member Function Documentation

QStrList::QStrList (bool deepCopies=TRUE)

Constructs an empty list of strings. Will make deep copies of all inserted strings if *deepCopies* is TRUE, or uses shallow copies if *deepCopies* is FALSE.

QStrList::QStrList (const QStrList & list)

Constructs a copy of *list*.

If *list* has deep copies, this list will also get deep copies. Only the pointers are copied (shallow copy) if the other list does not use deep copies.

QStrList::~QStrList ()

Destroys the list. All strings are removed.

QStrList& QStrList::operator= (const QStrList & list)

Assigns *list* to this list and returns a reference to this list.

If *list* has deep copies, this list will also get deep copies. Only the pointers are copied (shallow copy) if the other list does not use deep copies.

169.2.1. URL

Generated from http://www.troll.no/qt/qstrlist.html.

170. QTabBar Class Reference

```
#include <qtabbar.h>
```

Inherits QWidget.

170.1. Detailed Description

The QTabBar class provides a tab bar, for use in e.g. tabbed dialogs.

The class is quite simple; it draws the tabs in one of four shapes and emits a signal when one is selected. It can be subclassed to tailor the look and feel.

The four possible shapes are

- `RoundedAbove` - (the default) rounded tabs to be located above the pages.
- `RoundedBelow` - rounded tabs to be located below the pages.
- `TriangularAbove` - triangular tabs to be located above the pages (very unusual, included for completeness).
- `TriangularBelow` - triangular tabs to be located below the pages.

The choice of tab shape is still a matter of taste, to a large degree. Tab dialogs (preferences and the like) invariable use `RoundedAbove` and nobody uses `TriangularAbove.` Tab controls in windows other than dialogs almost always either `RoundedBelow` or `TriangularBelow.` Many spreadsheets and other tab controls where all the pages are essentially similar to use `TriangularBelow,` while `RoundedBelow` is used mostly when the pages are different (e.g. a multi-page tool palette). There is no strong tradition yet, however, so use your taste and create the tradition.

The most important part of QTabBar's API is the signal selected(). It's emitted whenever the selected page changes (even at startup, when the selected page changes from 'none'). There are also a slot, setCurrentTab(), which can be used to select a page programmatically.

QTabBar creates automatic accelerator keys in the manner of QButton; e.g. if a tab's label is "&Graphics" Alt-G becomes an accelerator key for switching to that tab.

The following virtual functions may need to be reimplemented:

- paint() paints a single tab. paintEvent() calls paint() for each tab in such a way that any overlap will look right.
- addTab() creates a new tab and adds it to the bar.
- selectTab() decides which, if any, tab the user selects with the mouse.

170.2. Member Function Documentation

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

Create a new, empty tab bar.

QTabBar::˜QTabBar ()

Delete the tab control and free the memory it used.

int QTabBar::addTab (QTab * newTab) `[virtual]`

Add *newTab* to the tab control.

Allocate a new id, set t's id, locate it just to the right of the existing tabs, insert an accelerator if the tab's label contains the string "&p" for some value of p, add it to the bar, and return the newly allocated id.

int QTabBar::currentTab () const

If a page is currently visible, returns its ID. If no page is currently visible, returns either -1 or the ID of one of the pages.

Even if the return value is not -1, you cannot assume either that the user can see the relevant page, or that the tab is enabled.

When you need to display something, the return value from this function represents the best page to display. That's all.

See also: selected().

bool QTabBar::isTabEnabled (int id) const

Return TRUE if the tab with id *id* is enabled, or FALSE if it is disabled or there is no such tab.

See also: setTabEnabled().

void QTabBar::keyPressEvent (QKeyEvent * e) `[virtual protected]`

Handles the tab bar's keyboard interface (if enabled).

The right and left arrow keys move a selector, the space bar makes the tab with the selector active. All other keys are ignored.

Reimplemented from QWidget.

int QTabBar::keyboardFocusTab () const

If this tab control has keyboard focus, returns the ID of the tab Space will select. Otherwise, returns -1.

void QTabBar::mousePressEvent (QMouseEvent * e) `[virtual protected]`

Handles mouse press events; records what tab the mouse points to.

Reimplemented from QWidget.

void QTabBar::mouseReleaseEvent (QMouseEvent * e) `[virtual protected]`

Handles mouse release events for the tab control. Checks that the mouse is released over the tab where it was pressed, and if it was, selects that tab.

Reimplemented from QWidget.

void QTabBar::paint (QPainter * p, QTab * t, bool selected) const `[virtual protected]`

Paint the single tab *t* using *p*. If and only if *selected* is TRUE, *t* is currently selected.

This virtual function may be reimplemented to change the look of QTabBar. If you decide to reimplement it, you may also need to reimplement sizeHint().

void QTabBar::paintEvent (QPaintEvent * e) `[virtual protected]`

Repaints the tab row. All the painting is done by paint(); paintEvent() only decides which tabs need painting and in what order.

See also: paint().

Reimplemented from QWidget.

QTab * QTabBar::selectTab (const QPoint & p) const `[virtual protected]`

This virtual functions is called by the mouse event handlers to determine which tab is pressed. The default implementation returns a pointer to the tab whose bounding rectangle contains *p,* if exactly one tab's bounding rectangle contains *p.* It returns 0 else.

See also: mousePressEvent() and mouseReleaseEvent().

void QTabBar::selected (int id) `[signal]`

QTabBar emits this signal whenever any tab is selected, whether by the program or the user. The argument *id* is the ID if the tab as returned by addTab().

show() is guaranteed to emit this signal, so that you can display your page in a slot connected to this signal.

void QTabBar::setCurrentTab (QTab * tab) `[slot]`

"Raises" *tab* and emits the selected() signal unless the tab was already current.

See also: currentTab() and selected().

void QTabBar::setCurrentTab (int id) `[slot]`

"Raises" the tab with ID *id* and emits the selected() signal.

See also: currentTab(), selected() and tab().

void QTabBar::setShape (Shape s)

Sets the shape of this tab bar to *s* and refreshes the bar.

void QTabBar::setTabEnabled (int id, bool enabled)

Enable tab *id* if *enable* is TRUE, or disable it if *enable* is FALSE. If *id* is currently selected, setTabEnabled() makes another tab selected.

setTabEnabled() calls repaint() if this causes a change in *id's* status.

See also: update() and isTabEnabled().

QTabBar::Shape QTabBar::shape() const

Returns the shape of this tab bar.

See also: setShape().

void QTabBar::show () `[virtual]`

Shows the widget, and ensures that one tab is selected.

Reimplemented from QWidget.

QSize QTabBar::sizeHint () const `[virtual]`

Returns a suitable size for the tab control.

Reimplemented from QWidget.

QTab * QTabBar::tab (int id)

Returns a pointer to the tab with ID *id,* or 0 if there is no such tab.

QListT<QTab> * QTabBar::tabList () `[protected]`

The list of QTab objects added.

170.2.1. URL

Generated from http://www.troll.no/qt/qtabbar.html.

171. QTabDialog Class Reference

`#include <qtabdialog.h>`

Inherits QDialog.

171.1. Detailed Description

The QTabDialog class provides a stack of tabbed widgets.

A tabbed dialog is one in which several "pages" are available, and the user selects which page to see and use by clicking on its tab, or by pressing the indicated Alt-(letter) key combination.

QTabDialog does not provide more than one row of tabs, and does not provide tabs along the sides or bottom of the pages. It also does not offer any way to find out which page is currently visible or to set the visible page.

QTabDialog provides an OK button and optionally Apply, Cancel and Defaults buttons.

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() to set up a tab and keyboard accelerator for it,

3 set up the buttons for the tab dialog (Apply, Cancel and so on), and finally

4 connect to the signals and slots.

The pref.cpp example does all this.

If you don't call addTab(), the page you have created will not be visible. Please don't confuse the object name you supply to the QWidget constructor and the tab label you supply to addTab(): addTab() takes a name which indicates an accelerator and is meaningful and descriptive to the user, while 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.
- 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 it show() is called, you must connect this signal to a slot which resets the state of the dialog.
- selected() is emitted when the user selects some page.

Each tab is either enabled or disabled at any given time. 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 can not select that tab. Note that even though a tab is disabled, the page can still be visible, for example if all of the tabs happen to be disabled.

While tab dialogs can be a very good way to split up a complex dialog, it's also very easy to make a royal mess out of a tab dialog. Here is some advice. For more, see e.g. the UIE web page on tab dialogs.

1 Make sure that each page forms a logical whole which is adequately described by the label on the tab.

If two related functions are on different pages, users will often not find one of the functions, or will spend far too long searching for it.

2 Do not join several independent dialogs into one tab dialog. Several aspects of one complex dialog is acceptable (such as the various aspects of "preferences") but a tab dialog is no substitute for a pop-up menu leading to several smaller dialogs.

The OK button (and the other buttons) apply to the *entire* dialog. If the tab dialog is really several independent smaller dialogs, users often press Cancel to cancel just the changes he/she has made on the current page: Many users will treat that page as independent of the other pages.

3 Do not use tab dialogs for frequent operations. The tab dialog is probably the most complex widget in common use at the moment, and subjecting the user to this complexity during his/her normal use of your application is most often a bad idea.

The tab dialog is good for complex operations which have to be performed seldom, like Preferences dialogs. Not for common operations, like setting left/right alignment in a word processor. (Often, these common operations are actually independent dialogs and should be treated as such.)

The tab dialog is not a navigational aid, it is an organizational aid. It is a good way to organize aspects of a complex operation (such as setting up caching and proxies in a web browser), but a bad way to navigate towards a simple operation (such as emptying the cache in a web browser - emptying the cache is *not* part of setting up the cache, it is a separate and independent operation).

4 The changes should take effect when the user presses Apply or OK. Not before.

Providing Apply, Cancel or OK buttons on the individual pages is likely to weaken the users' mental model of how tab dialogs work. If you think a page needs its own buttons, consider making it a separate dialog.

5 There should be no implicit ordering of the pages. If there is, it is probably better to use a wizard dialog.

If some of the pages seem to be ordered and others not, perhaps they ought not to be joined in a tab dialog.

Most of the functionality in QTabDialog is provided by a QTabBar (at the top, providing the tabs) and a QWidgetStack (most of the area, organizing the individual pages).

See also: QDialog.

Examples:
pref/pref.cpp

171.2. 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.

QTabDialog::˜QTabDialog ()

Destroys the tab dialog.

void QTabDialog::aboutToShow () `[signal]`

This signal is emitted by show() when it's time to set the state of the dialog's contents. The dialog should reflect the current state of the application when if appears; if there is any chance that the state of the application can 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, show()n and deleted afterwards.

See also: applyButtonPressed(), show() and cancelButtonPressed().

void QTabDialog::addTab (QWidget * child, QTab * tab)

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.

void QTabDialog::addTab (QWidget * child, const char * label)

Add another tab and page to the tab view.

The tab will be labelled *label* and *child* constitutes the new page. Note the difference between the widget name (which you supply to widget constructors and to e.g. setTabEnabled()) and the tab label: The name is internal to the program and invariant, while the label is shown on screen and may vary according to e.g. language.

label is written in the QButton style, where &P makes Qt create an accelerator key on Alt-P for this page. For example:

```
td->addTab( graphicsPane, "&Graphics" );
td->addTab( soundPane, "&Sound" );
```

If the user presses Alt-S the sound page of the tab dialog is shown, if the user presses Alt-P the graphics page is shown.

If you call addTab() after show(), the screen will flicker and the user will be confused.

Examples:
pref/pref.cpp

void QTabDialog::applyButtonPressed () `[signal]`

This signal is emitted when the Apply or OK buttons are clicked.

It should be connected to a slot (or several slots) which change the application's state according to the state of the dialog.

See also: cancelButtonPressed(), defaultButtonPressed() and setApplyButton().

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 in any way, so generally you should not need to connect it to any slot.

See also: applyButtonPressed(), defaultButtonPressed() and setCancelButton().

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(), cancelButtonPressed() and setDefaultButton().

bool QTabDialog::hasApplyButton () const

Returns TRUE if the tab dialog has an Apply button, FALSE if not.

See also: setDefaultsButton(), defaultButtonPressed(), hasApplyButton() and hasCancelButton().

bool QTabDialog::hasCancelButton () const

Returns TRUE if the tab dialog has a Cancel button, FALSE if not.

See also: setCancelButton(), cancelButtonPressed(), hasDefaultButton() and hasApplyButton().

bool QTabDialog::hasDefaultButton () const

Returns TRUE if the tab dialog has a Defaults button, FALSE if not.

See also: setApplyButton(), applyButtonPressed(), hasCancelButton() and hasDefaultButton().

bool QTabDialog::hasOkButton () const

Returns TRUE if the tab dialog has an OK button, FALSE if not.

See also: setDefaultsButton(), defaultButtonPressed(), hasOkButton() and hasCancelButton().

bool QTabDialog::isTabEnabled (const char * name) const

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() and QWidget::isEnabled().

void QTabDialog::paintEvent (QPaintEvent *) `[virtual protected]`

Handles paint events for the tabbed dialog.

Reimplemented from QWidget.

void QTabDialog::resizeEvent (QResizeEvent * e) `[virtual protected]`

Handles resize events for the tab dialog.

Reimplemented from QWidget.

void QTabDialog::selected (const char * tabLabel) `[signal]`

This signal is emitted whenever a tab is selected (raised), including during the first show().

See also: raise().

void QTabDialog::setApplyButton (const char * text = "Apply")

Add an Apply button to the dialog. The button's text is set to *text* (and defaults to "Apply").

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.

See also: setCancelButton(), setDefaultButton() and applyButtonPressed().

Examples:
pref/pref.cpp

void QTabDialog::setCancelButton (const char * text = "Cancel")

Add a Cancel button to the dialog. The button's text is set to *text* (and defaults to "Cancel").

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 the the state immediately after the last Apply.

When Cancel is clicked, the cancelButtonPressed() signal is emitted. The dialog is closed at the same time.

See also: setApplyButton, setDefaultButton() and cancelButtonPressed().

Examples:
pref/pref.cpp

void QTabDialog::setDefaultButton (const char * text = "Defaults")

Add a Defaults button to the dialog. The button's text is set to *text* (and defaults to "Defaults").

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.

See also: setApplyButton(), setCancelButton() and defaultButtonPressed().

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.

void QTabDialog::setOkButton (const char * text = "OK")

Set the OK button's text to *text* (which defaults to "OK").

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. Then the dialog closes.

See also: setCancelButton(), setDefaultButton() and applyButtonPressed().

void QTabDialog::setTabBar (QTabBar * tb) `[protected]`

Replaces the QTabBar heading the dialog by the given tab bar. Note that this must be called *before* any tabs have been added, or the behavior is undefined.

See also: tabBar().

void QTabDialog::setTabEnabled (const char * name, bool enable)

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 keep a separate flag.

Note that even a disabled tab/page may be visible. If the page is visible already, QTabDialog will not hide it, and 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() and QWidget::setEnabled().

void QTabDialog::show () `[virtual]`

Shows the tab view and its children. Reimplemented in order to delay show()'ing of every page except the initially visible one, and in order to emit the aboutToShow() signal.

See also: hide() and aboutToShow().

Examples:
pref/pref.cpp

Reimplemented from QWidget.

void QTabDialog::showPage (QWidget * w)

Ensures that *w* is shown. This is useful mainly for accelerators.

Warning: Used carelessly, this function can easily surprise or confuse the user.

See also: QTabBar::setCurrentTab().

void QTabDialog::styleChange (GUIStyle s) `[virtual protected]`

Reimplemented to hndle a change of GUI style while on-screen.

Reimplemented from QWidget.

QTabBar* QTabDialog::tabBar () const `[protected]`

Returns the currently set QTabBar.

See also: setTabBar().

const char * QTabDialog::tabLabel (QWidget * w)

Returns the text in the tab for page *w.*

171.2.1. URL

Generated from http://www.troll.no/qt/qtabdialog.html.

172. QTableView Class Reference

`#include <qtableview.h>`

Inherits QFrame.

Inherited by QHeader, QListBox, QMultiLineEdit and QPopupMenu.

172.1. Detailed Description

This is the abstract base class of all the table views.

A table view consists of a number of abstract cells organized in rows and columns and a visible part called a view. The cells are identified with a row index and a column index. The top left cell is in row 0, column 0.

The behavior of the widget can be finely tuned using setTableFlags(); subclasses like QListBox are little more than a call to setTableFlags(), some table content manipulation, and an implementation of paintCell(). Subclasses that need cells with variable width or height must reimplement cellHeight() and/or cellWidth(). Use updateTableSize() to tell QTableView when the width or height has changed.

When you read this documentation, it is important to understand the distinctions between the four pixel coordinate systems involved.

1 The *cell* coordinates. (0,0) is the top left corner of a cell. This is used by functions such as paintCell().

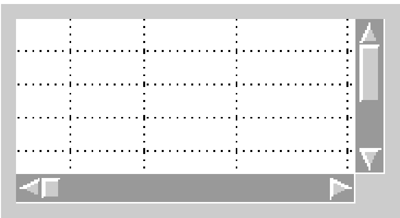
2 The *table* coordinates. (0,0) is the top left corner of the cell at row 0 and column 0. These coordinates are absolute; that is, they are independent of what part of the table is visible at the moment. This is used by functions such as setXOffset() or maxYOffset().

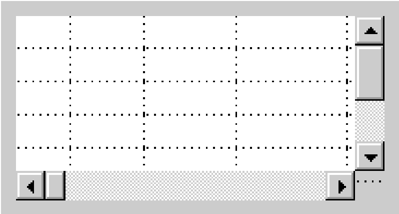
3 The *widget* coordinates. (0,0) is the top left corner of the widget, *including* the frame. This is used by functions such as repaint().

4 The *view* coordinates. (0,0) is the top left corner of the view, *excluding* the frame. This is the least-used coordinate system, used by functions such as viewWidth().

It is rather unfortunate that we have to use four different coordinate systems, but if we were to provide a flexible and powerful base class, there wasn't any way around it.

Note: The row,column indices are always given in that order, i.e. first the vertical (row), then the horizontal (column). This is the opposite order of all pixel operations, which take first the horizontal (x), then the vertical (y).





Warning: in the current (and previous) version of the library, the functions setNumRows(), setNumCols(), setCellHeight(), setCellWidth(), setTableFlags() and clearTableFlags() may cause virtual functions like cellWidth() and cellHeight() to be called, even if autoUpdate() is FALSE. This may cause errors if relevant state variables are not initialized.

See also: QScrollView and GUI Design Handbook: Table

Examples:
table/table.cpp

172.2. Member Function Documentation

QTableView::QTableView (QWidget * parent=0, const char * name=0, WFlags f=0) `[protected]`

Constructs a table view. All the arguments are passed to the QFrame constructor.

The table flags are all cleared (set to zero). Set `Tbl_autoVScrollBar` or `Tbl_autoHScrollBar` to get automatic scroll bars and `Tbl_clipCellPainting` to get safe clipping.

The cell height and cell width are set to 0.

Frame line shapes (QFrame::HLink and QFrame::VLine) are disallowed, see QFrame::setStyle().

Note that the *f* argument is *not* table flags but rather widget flags.

QTableView::~QTableView () `[protected]`

Destroys the table view.

bool QTableView::autoUpdate () const `[protected]`

Returns TRUE if the view updates itself automatically whenever it is changed in some way.

See also: setAutoUpdate().

int QTableView::cellHeight () const `[protected]`

Returns the row height, in pixels. Returns 0 if the rows have variable heights.

See also: setCellHeight() and cellWidth().

int QTableView::cellHeight (int) `[virtual protected]`

Returns the height of row *row,* in pixels.

This function is virtual and must be reimplemented by subclasses that have variable cell heights. Note that if the total table height changes, updateTableSize() must be called.

See also: setCellHeight(), cellWidth() and totalHeight().

Reimplemented in QListBox, QHeader and QPopupMenu.

QRect QTableView::cellUpdateRect () const `[protected]`

This function should only be called from the paintCell() function in subclasses. It returns the portion of a cell that actually needs to be updated, in *cell* coordinates. This is only useful for non-trivial paintCell().

int QTableView::cellWidth () const `[protected]`

Returns the column width, in pixels. Returns 0 if the columns have variable widths.

See also: setCellWidth() and cellHeight().

int QTableView::cellWidth (int) `[virtual protected]`

Returns the width of column *col,* in pixels.

This function is virtual and must be reimplemented by subclasses that have variable cell widths. Note that if the total table width changes, updateTableSize() must be called.

See also: setCellWidth(), cellHeight(), totalWidth() and updateTableSize().

Reimplemented in QHeader and QPopupMenu.

void QTableView::clearTableFlags (uint f = ˜0) `[protected]`

Clears the table flags that are set in *f.*

Example (clears a single flag):

```
clearTableFlags( Tbl_snapToGrid );
```

The default argument clears all flags.

See also: setTableFlags(), testTableFlags() and tableFlags().

bool QTableView::colIsVisible (int col) const `[protected]`

Returns TRUE if *col* is at least partially visible.

See also: rowIsVisible().

bool QTableView::colXPos (int col, int * xPos) const `[protected]`

Computes the position in the widget of column *column.*

Returns TRUE and stores the result in **xPos* (in *widget* coordinates) if the column is visible. Returns FALSE and does not modify **xPos* if *col* is invisible or invalid.

See also: rowYPos() and findCol().

int QTableView::findCol (int xPos) const `[protected]`

Returns the index of the column at position *xPos,* where *xPos* is in *widget* coordinates. Returns -1 if *xPos* is outside the valid range.

See also: findRow() and colXPos().

int QTableView::findRow (int yPos) const `[protected]`

Returns the index of the row at position *yPos,* where *yPos* is in *widget* coordinates. Returns -1 if *yPos* is outside the valid range.

See also: findCol() and rowYPos().

const QScrollBar * QTableView::horizontalScrollBar () const `[protected]`

Returns a pointer to the horizontal scroll bar, mainly so you can connect() to its signals. Note that the scroll bar works in pixel values, use findCol() to translate to cell numbers.

int QTableView::lastColVisible () const `[protected]`

Returns the index of the last (right) column in the view. The index of the first column is 0.

If no columns are visible it returns -1. This can happen if the view is too narrow for the first column and Tbl_cutCellsH is set.

See also: lastRowVisible().

int QTableView::lastRowVisible () const `[protected]`

Returns the index of the last (bottom) row in the view. The index of the first row is 0.

If no rows are visible it returns -1. This can happen if the view is too small for the first row and Tbl_cutCellsV is set.

See also: lastColVisible().

int QTableView::leftCell () const `[protected]`

Returns the index of the first column in the table that is visible in the view. The index of the very leftmost column is 0.

See also: topCell() and setLeftCell().

int QTableView::maxColOffset () `[protected]`

Returns the index of the last column which may be at the left edge of the view.

Depending on the Tbl_scrollLastHCell flag, this may or may not be the last column.

See also: maxXOffset() and maxRowOffset().

int QTableView::maxRowOffset () `[protected]`

Returns the index of the last row which may be at the top edge of the view.

Depending on the Tbl_scrollLastVCell flag, this may or may not be the last row.

See also: maxYOffset() and maxColOffset().

int QTableView::maxViewX () const `[protected]`

Returns the rightmost pixel of the table view in *view* coordinates. This excludes the frame and any scroll bar, but includes blank pixels to the right of the visible table data.

See also: maxViewY(), viewWidth() and contentsRect().

int QTableView::maxViewY () const `[protected]`

Returns the bottom pixel of the table view in *view* coordinates. This excludes the frame and any scroll bar, but includes blank pixels below the visible table data.

See also: maxViewX(), viewHeight() and contentsRect().

int QTableView::maxXOffset () `[protected]`

Returns the maximum horizontal offset within the table of the view's left edge, in *table* coordinates.

This is used mainly to set the horizontal scroll bar's range.

See also: maxColOffset(), maxYOffset() and totalWidth().

int QTableView::maxYOffset () `[protected]`

Returns the maximum vertical offset within the table of the view's top edge, in *table* coordinates.

This is used mainly to set the vertical scroll bar's range.

See also: maxRowOffset(), maxXOffset() and totalHeight().

int QTableView::minViewX () const `[protected]`

Returns the leftmost pixel of the table view in *view* coordinates. This excludes the frame and any header.

See also: maxViewY(), viewWidth() and contentsRect().

int QTableView::minViewY () const `[protected]`

Returns the top pixel of the table view in *view* coordinates. This excludes the frame and any header.

See also: maxViewX(), viewHeight() and contentsRect().

int QTableView::numCols () const `[protected]`

Returns the number of columns in the table

See also: numRows() and setNumCols().

int QTableView::numRows () const `[protected]`

Returns the number of rows in the table.

See also: numCols() and setNumRows().

void QTableView::paintCell (QPainter * p, int row, int col) `[virtual protected]`

This pure virtual function is called to paint the single cell at *(row,col)* using *p,* which is open when paintCell() is called and must remain open.

The coordinate system is translated such that the origin is at the top left corner of the cell to be painted; i.e. *cell* coordinates. Do not scale or shear the coordinate system (or if you do, restore the transformation matrix before you return).

By default, the painter is not clipped, for maximum efficiency. For safety, call setTableFlags(Tbl_clipCellPainting) to enable clipping.

See also: paintEvent(), QPainter() and setTableFlags().

Reimplemented in QListBox, QMultiLineEdit, QHeader and QPopupMenu.

void QTableView::paintEvent (QPaintEvent * e) `[virtual protected]`

Handles paint events for the table view, by calling paintCell() and drawFrame().

It is optimized to repaint only the cells that need to be repainted.

Reimplemented from QWidget.

void QTableView::repaint (int x, int y, int w, int h, bool erase=TRUE)

Repaints the table view directly by calling paintEvent() directly, unless updates are disabled.

Erases the view area *(x,y,w,h)* if *erase* is TRUE. Parameters *(x,y)* are in *widget* coordinates.

If *w* is negative, it is replaced with `width() - x`. If *h* is negative, it is replaced width `height() - y`.

Doing a repaint() usually is faster than doing an update(), but calling update() many times in a row will generate a single paint event.

At present, QTableView is the only widget that reimplements repaint(). It does this because by clearing and then repainting one cell at at time, it can make the screen flicker less than it would otherwise.

void QTableView::repaint (bool erase=TRUE)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Repaints the entire view.

void QTableView::repaint (const QRect & r, bool erase=TRUE)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

void QTableView::resizeEvent (QResizeEvent *) `[virtual protected]`

Handles resize events for the table view.

The scroll bars are moved and cells repainted as necessary.

Reimplemented from QWidget.

bool QTableView::rowIsVisible (int row) const `[protected]`

Returns TRUE if *row* is at least partially visible.

See also: colIsVisible().

bool QTableView::rowYPos (int row, int * yPos) const `[protected]`

Computes the position in the widget of row *row.*

Returns TRUE and stores the result in **yPos* (in *widget* coordinates) if the row is visible. Returns FALSE and does not modify **yPos* if *row* is invisible or invalid.

See also: colXPos() and findRow().

void QTableView::scroll (int xPixels, int yPixels) `[protected]`

Moves the visible area of the table rightwards by *xPixels* and downwards by *yPixels* pixels. Both may be negative.

This function is *not* the same as QWidget::scroll().

See also: setXOffset(), setYOffset(), setOffset(), setTopCell(), setLeftCell() and setTopLeftOffset().

void QTableView::setAutoUpdate (bool enable) `[protected]`

Sets the auto-update option of the table view to *enable.*

If *enable* is TRUE (this is the default) then the view updates itself automatically whenever it has changed in some way (for example when a flag is changed).

If *enable* is FALSE, the view does NOT repaint itself, or update its internal state variables itself when it is changed. This can be useful to avoid flicker during large changes, and is singularly useless otherwise: Disable auto-update, do the changes, re-enable auto-update, and call repaint().

Warning: Do not leave the view in this state for a long time (i.e. between events). If, for example, the user interacts with the view when auto-update is off, strange things can happen.

Setting auto-update to TRUE does not repaint the view, you must call repaint() to do this.

See also: autoUpdate() and repaint().

void QTableView::setBackgroundColor (const QColor & c) `[virtual]`

For internal use only.

Reimplemented from QWidget.

void QTableView::setCellHeight (int cellHeight) `[protected]`

Sets the height in pixels of the table cells to *cellHeight.*

Setting it to zero means that the row height is variable. When set to 0 (this is the default) QTableView calls the virtual function cellHeight() to get the height.

See also: cellHeight(), setCellWidth(), totalHeight() and numRows().

Examples:
    table/table.cpp

void QTableView::setCellWidth (int cellWidth) `[protected]`

Sets the width in pixels of the table cells to *cellWidth.*

Setting it to zero means that the column width is variable. When set to 0 (this is the default) QTableView calls the virtual function cellWidth() to get the width.

See also: cellWidth(), setCellHeight(), totalWidth() and numCols().

Examples:
table/table.cpp

void QTableView::setLeftCell (int col) `[protected]`

Scrolls the table such that *col* becomes the leftmost column. The index of the very leftmost column is 0.

See also: setXOffset(), setTopLeftCell() and setTopCell().

Examples:
table/table.cpp

void QTableView::setNumCols (int cols) `[protected]`

Sets the number of columns of the table to *cols* (must be non-negative). Does not change leftCell().

The table repaints itself automatically if autoUpdate() is set.

See also: numCols(), numRows() and setNumRows().

Examples:
table/table.cpp

void QTableView::setNumRows (int rows) `[protected]`

Sets the number of rows of the table to *rows* (must be non-negative). Does not change topCell().

The table repaints itself automatically if autoUpdate() is set.

See also: numCols(), setNumCols() and numRows().

Examples:
table/table.cpp

void QTableView::setOffset (int x, int y, bool updateScrBars = TRUE) `[protected]`

Scrolls the table such that *(x,y)* becomes the top left pixel in the view. Parameters *(x,y)* are in *table* coordinates.

The interaction with Tbl_snapTo*Grid is tricky. If *updateScrBars* is TRUE, the scroll bars are updated.

See also: xOffset(), yOffset(), setXOffset(), setYOffset() and setTopLeftCell().

void QTableView::setPalette (const QPalette & p) `[virtual]`

Reimplements QWidget::setPalette(). Sets the palette for the scroll bars, too.

See also: setBackgroundColor().

Reimplemented from QWidget.

void QTableView::setTableFlags (uint f) `[protected]`

Sets the table flags to *f*.

If a flag setting changes the appearance of the table the table is repainted if and only if autoUpdate() is TRUE.

The table flags are mostly single bits, though there are some multibit flags for convenience. Here is a complete list:

Tbl_vScrollBar
: The table has a vertical scroll bar.

Tbl_hScrollBar
: The table has a horizontal scroll bar.

Tbl_autoVScrollBar
: The table has a vertical scroll bar if and only if the table is taller than the view.

Tbl_autoHScrollBar
: The table has a horizontal scroll bar if and only if the table is wider than the view.

Tbl_autoScrollBars
: The union of the previous two flags.

Tbl_clipCellPainting
: The table uses QPainter::setClipRect() to make sure that paintCell() will not draw outside the cell boundaries.

Tbl_cutCellsV
: The table will never show part of a cell at the bottom of the table; if there is not space for all of a cell the space is left blank.

Tbl_cutCellsH
: The table will never show part of a cell at the right side of the table; if there is not space for all of a cell the space is left blank.

Tbl_cutCells
: The union of the previous two flags.

Tbl_scrollLastHCell
: When the user scrolls horizontally, let him/her scroll the last cell leftward until it is at the left edge of the view. If this flag is not set, the user can only scroll to the point where last cell is completely visible.

Tbl_scrollLastVCell
: When the user scrolls vertically, let him/her scroll the last cell upward until it is at the top edge of the view. If this flag is not set, the user can only scroll to the point where last cell is completely visible.

Tbl_scrollLastCell
: The union of the previous two flags.

Tbl_smoothHScrolling
: The table scrolls as smoothly as possible when the user scrolls horizontally. When this flag is not set scrolling is done one cell at a time.

Tbl_smoothVScrolling
: The table scrolls as smoothly as possible when scrolling vertically. When this flag is not set scrolling is done one cell at a time.

Tbl_smoothScrolling
: The union of of previous two flags.

Tbl_snapToHGrid
: Except when the user is actually scrolling, the leftmost column shown snaps to the leftmost edge of the view.

Tbl_snapToVGrid
Except when the user is actually scrolling, the top row snaps to the top edge of the view.

Tbl_snapToGrid
The union of the previous two flags.

You can specify more than one flag at a time using bitwise OR.

Example:

```
setTableFlags( Tbl_smoothScrolling | Tbl_autoScrollBars );
```

See also: clearTableFlags(), testTableFlags() and tableFlags().

Examples:
table/table.cpp

void QTableView::setTopCell (int row) `[protected]`

Scrolls the table such that *row* becomes the top row. The index of the very first row is 0.

See also: setYOffset(), setTopLeftCell() and setLeftCell().

Examples:
table/table.cpp

void QTableView::setTopLeftCell (int row, int col) `[protected]`

Scrolls the table such that the cell at row *row* and colum *col* becomes the top left cell in the view. The cell at the extreme top left of the table is at position (0,0).

See also: setLeftCell(), setTopCell() and setOffset().

void QTableView::setXOffset (int x) `[protected]`

Scrolls the table such that *x* becomes the leftmost pixel in the view. The *x* parameter is in *table* coordinates.

The interaction with Tbl_snapToHGrid is tricky.

See also: xOffset(), setYOffset(), setOffset() and setLeftCell().

void QTableView::setYOffset (int y) `[protected]`

Scrolls the table such that *y* becomes the top pixel in the view. The *y* parameter is in *table* coordinates.

The interaction with Tbl_snapToVGrid is tricky.

See also: yOffset(), setXOffset(), setOffset() and setTopCell().

void QTableView::setupPainter (QPainter *) `[virtual protected]`

This virtual function is called before painting of table cells is started. It can be reimplemented by subclasses that want to to set up the painter in a special way and that do not want to do so for each cell.

The default implementation does nothing.

Reimplemented in QHeader.

void QTableView::show () `[virtual]`

Reimplements QWidget::show() in order to do scroll bar tricks.

Reimplemented from QWidget.

uint QTableView::tableFlags () const `[protected]`

Returns the union of the table flags that are currently set.

See also: setTableFlags(), clearTableFlags() and testTableFlags().

bool QTableView::testTableFlags (uint f) const `[protected]`

Returns TRUE if any of the table flags in *f* are currently set, otherwise FALSE.

See also: setTableFlags(), clearTableFlags() and tableFlags().

int QTableView::topCell () const `[protected]`

Returns the index of the first row in the table that is visible in the view. The index of the very first row is 0.

See also: leftCell() and setTopCell().

int QTableView::totalHeight () `[virtual protected]`

Returns the total height of the table in pixels.

This function is virtual and should be reimplmented by subclasses that have variable cell heights and a non-trivial cellHeight() function, or a large number of rows in the table.

The default implementation may be slow for very tall tables.

See also: cellHeight() and totalWidth().

int QTableView::totalWidth () `[virtual protected]`

Returns the total width of the table in pixels.

This function is virtual and should be reimplmented by subclasses that have variable cell widths and a non-trivial cellWidth() function, or a large number of columns in the table.

The default implementation may be slow for very wide tables.

See also: cellWidth() and totalHeight().

void QTableView::updateCell (int row, int col, bool erase=TRUE) `[protected]`

Repaints the cell at row *row,* column *col* if it is inside the view.

If *erase* is TRUE, the relevant part of the view is cleared to the background color/pixmap before the contents are repainted.

See also: isVisible().

Examples:
table/table.cpp

void QTableView::updateScrollBars () `[protected]`

Updates the scroll bars' contents and presence to match the table's state. Generally you should not need to call this.

See also: setTableFlags().

void QTableView::updateTableSize () `[protected]`

Updates the scroll bars and internal state.

Call this function when the table view's total size is changed; typically because the result of cellHeight() or cellWidth() have changed.

This function does not repaint the widget.

const QScrollBar * QTableView::verticalScrollBar () const `[protected]`

Returns a pointer to the vertical scroll bar, mainly so you can connect() to its signals. Note that the scroll bar works in pixel values, use findRow() to translate to cell numbers.

int QTableView::viewHeight () const `[protected]`

Returns the height of the table view, as such, in *view* coordinates. This does not include any header, scroll bar or frame, but does include background pixels below the table data.

See also: minViewY(), maxViewY(), viewWidth(), contentsRect() and viewRect().

QRect QTableView::viewRect () const `[protected]`

Returns the rectangle which is the actual table, excluding any frame, in *widget* coordinates.

Somewhat similar to clientRect(), but does not include any frames.

int QTableView::viewWidth () const `[protected]`

Returns the width of the table view, as such, in *view* coordinates. This does not include any header, scroll bar or frame, but does include background pixels to the right of the table data.

See also: minViewX(), maxViewX(), viewHeight(), contentsRect() and viewRect().

int QTableView::xOffset () const `[protected]`

Returns the x coordinate in *table* coordinates of the pixel which is currently on the left edge of the view.

See also: setXOffset(), yOffset() and leftCell().

int QTableView::yOffset () const `[protected]`

Returns the y coordinate in *table* coordinates of the pixel which is currently on the top edge of the view.

See also: setYOffset(), xOffset() and topCell().

172.2.1. URL

Generated from http://www.troll.no/qt/qtableview.html.

173. QTextDrag Class Reference

`#include <qdragobject.h>`

Inherits QStoredDrag.

173.1. Detailed Description

The QTextDrag provides a drag-and-drop object for transferring plain text.

Plain text is defined as single- or multi-line US-ASCII or an unspecified 8-bit character set.

Qt provides no built-in mechanism for delivering only single-line or only US-ASCII text.

Drag&Drop text does *not* have a NUL terminator when it is dropped onto the target.

For detailed information about drag-and-drop, see the QDragObject class.

173.2. Member Function Documentation

QTextDrag::QTextDrag (QWidget * dragSource = 0, const char * name = 0)

Creates a default text drag object. *dragSource* must be the drag source, *name* is the object name.

QTextDrag::QTextDrag (const char * text, QWidget * dragSource = 0, const char * name = 0)

Creates a text drag object and sets it to *text. dragSource* must be the drag source, *name* is the object name.

QTextDrag::~QTextDrag ()

Destroys the text drag object and frees all allocated resources.

bool QTextDrag::canDecode (QDragMoveEvent * e) `[static]`

Returns TRUE if the information in *e* can be decoded into a QString.

See also: decode().

bool QTextDrag::decode (QDropEvent * e, QString & str) `[static]`

Attempts to decode the dropped information in *e* into *str,* returning TRUE if successful.

See also: decode().

void QTextDrag::setText (const char * text)

Sets the text to be dragged. You will need to call this if you did not pass the text during construction.

173.2.1. URL

Generated from http://www.troll.no/qt/qtextdrag.html.

174. QTextStream Class Reference

```
#include <qtextstream.h>
```

174.1. Detailed Description

The QTextStream class provides basic functions for reading and writing text using a QIODevice.

The text stream class has a functional interface that is very similar to that of the standard C++ iostream class. The difference between iostream and QTextStream is that our stream operates on a QIODevice, which is easily subclassed, while iostream operates on FILE * pointers, which can not be subclassed.

Qt provides several global functions similar to the ones in iostream:

- `bin` sets the QTextStream to output binary numbers
- `oct` sets the QTextStream to output octal numbers
- `dec` sets the QTextStream to output decimal numbers
- `hex` sets the QTextStream to output hexadecimal numbers
- `endl` forces a line break
- `flush` forces the QIODevice to flush any buffered data
- `ws` eats any available white space (on input)
- `reset` resets the QTextStream to its default mode (see reset()).

The QTextStream class reads and writes ASCII text and it is not appropriate for dealing with binary data (but QDataStream is).

See also: QDataStream.

Examples:
grapher/grapher.cpp
application/application.cpp

174.2. Member Function Documentation

QTextStream::QTextStream ()

Constructs a data stream that has no IO device.

QTextStream::QTextStream (QByteArray a, int mode)

Constructs a text stream that operates on a byte array throught an internal QBuffer device.

Example:

```
QString str;
QTextStream ts( str, IO_WriteOnly );
ts << "pi = " << 3.14;                    // str == "pi = 3.14"
```

Writing data to the text stream will modify the contents of the string. The string will be expanded when data is written beyond the end of the string.

Same example, using a QBuffer:

```
QString str;
QBuffer buf( str );
buf.open( IO_WriteOnly );
QTextStream ts( &buf );
ts << "pi = " << 3.14;                    // str == "pi = 3.14"
buf.close();
```

Note that QStrings created in this way will not have NUL terminators. So unless you are using the string purely as an array of bytes, you should terminate them:

```
QString str;
...
ts << "pi = " << 3.14;                      // str == "pi = 3.14"
ts << '\0';
```

QTextStream::QTextStream (QIODevice * d)

Constructs a text stream that uses the IO device *d*.

QTextStream::QTextStream (FILE * fh, int mode)

Constructs a text stream that operates on an existing file handle *fh* throught an internal QFile device.

Example:

```
QTextStream cout( stdout, IO_WriteOnly );
QTextStream cin ( stdin,  IO_ReadOnly );
QTextStream cerr( stderr, IO_WriteOnly );
```

QTextStream::~QTextStream () `[virtual]`

Destroys the text stream.

The destructor will not affect the current IO device.

QIODevice * QTextStream::device () const

Returns the IO device currently set.

See also: setDevice() and unsetDevice().

bool QTextStream::eof () const

Returns TRUE if the IO device has reached the end position (end of stream or file) or if there is no IO device set.

Returns FALSE if the current position of the read/write head of the IO device is somewhere before the end position.

See also: QIODevice::atEnd().

Examples:
grapher/grapher.cpp
application/application.cpp

int QTextStream::fill () const

Returns the fill character. The default value is ' ' (space).

int QTextStream::fill (int f)

Sets the fill character to *f*. Returns the previous fill character.

int QTextStream::flags () const

Returns the current stream flags. The default value is 0.

The meaning of the flags are:

- *skipws* - Not currently used - whitespace always skipped
- *left* - Numeric fields are left-aligned
- *right* - Not currently used (by default numerics are right aligned)
- *internal* - Put any padding spaces between +/- and value
- *bin* - Output *and* input only in binary
- *oct* - Output *and* input only in octal
- *dec* - Output *and* input only in decimal
- *hex* - Output *and* input only in hexadecimal
- *showbase* - Annotate numeric outputs with 0b, 0, or 0x if in *bin, oct,* or *hex* format
- *showpoint* - Not currently used
- *uppercase* - Use 0B and 0X rather than 0b and 0x
- *showpos* - Show + for positive numeric values
- *scientific* - Use scientific notation for floating point values
- *fixed* - Use fixed-point notation for floating point values

Note that unless *bin, oct, dec,* or *hex* is set, the input base is octal if the value starts with 0, hexadecimal if it starts with 0x, binary if the value starts with 0b, and decimal otherwise.

See also: setf() and unsetf().

int QTextStream::flags (int f)

Sets the stream flags to *f.* Returns the previous stream flags.

See also: setf(), unsetf() and flags().

QTextStream & QTextStream::operator<< (char c)

Writes a `char` to the stream and returns a reference to the stream.

QTextStream & QTextStream::operator<< (const char * s)

Writes a string to the stream and returns a reference to the stream.

QTextStream & QTextStream::operator<< (double f)

Writes a `double` to the stream and returns a reference to the stream.

QTextStream & QTextStream::operator<< (float f)

Writes a `float` to the stream and returns a reference to the stream.

QTextStream & QTextStream::operator<< (signed int i)

Writes an `int` to the stream and returns a reference to the stream.

QTextStream & QTextStream::operator<< (signed long i)

Writes a `long int` to the stream and returns a reference to the stream.

QTextStream & QTextStream::operator<< (signed short i)

Writes a `short` integer to the stream and returns a reference to the stream.

QTextStream & QTextStream::operator<< (unsigned int i)

Writes an `unsigned int` to the stream and returns a reference to the stream.

QTextStream & QTextStream::operator<< (unsigned long i)

Writes an `unsigned long int` to the stream and returns a reference to the stream.

QTextStream & QTextStream::operator<< (unsigned short i)

Writes an `unsigned short` integer to the stream and returns a reference to the stream.

QTextStream & QTextStream::operator<< (void * ptr)

Writes a pointer to the stream and returns a reference to the stream.

The *ptr* is output as an unsigned long hexadecimal integer.

QTextStream & QTextStream::operator>> (QString & str)

Reads a word from the stream and returns a reference to the stream.

QTextStream & QTextStream::operator>> (char & c)

Reads a `char` from the stream and returns a reference to the stream.

QTextStream & QTextStream::operator>> (char * s)

Reads a word from the stream and returns a reference to the stream.

QTextStream & QTextStream::operator>> (double & f)

Reads a `double` from the stream and returns a reference to the stream. See flags() for an explanation of expected input format.

QTextStream & QTextStream::operator>> (float & f)

Reads a `float` from the stream and returns a reference to the stream. See flags() for an explanation of expected input format.

QTextStream & QTextStream::operator>> (signed int & i)

Reads a signed `int` from the stream and returns a reference to the stream. See flags() for an explanation of expected input format.

QTextStream & QTextStream::operator>> (signed long & i)

Reads a signed `long` int from the stream and returns a reference to the stream. See flags() for an explanation of expected input format.

QTextStream & QTextStream::operator>> (signed short & i)

Reads a signed `short` integer from the stream and returns a reference to the stream. See flags() for an explanation of expected input format.

QTextStream & QTextStream::operator>> (unsigned int & i)

Reads an unsigned `int` from the stream and returns a reference to the stream. See flags() for an explanation of expected input format.

QTextStream & QTextStream::operator>> (unsigned long & i)

Reads an unsigned `long` int from the stream and returns a reference to the stream. See flags() for an explanation of expected input format.

QTextStream & QTextStream::operator>> (unsigned short & i)

Reads an unsigned `short` integer from the stream and returns a reference to the stream. See flags() for an explanation of expected input format.

int QTextStream::precision () const

Returns the precision. The default value is 6.

int QTextStream::precision (int p)

Sets the precision to *p*. Returns the previous precision setting.

QString QTextStream::readLine ()

Reads a line from the stream and returns a string containing the text.

The returned string does not contain any trailing newline or carriage return. Note that this is different from QIODevice::readLine(), which does not strip the newline at the end of the line.

See also: QIODevice::readLine().

Examples:
application/application.cpp

QTextStream & QTextStream::readRawBytes (char * s, uint len)

Reads *len* bytes from the stream into *e* s and returns a reference to the stream.

The buffer *s* must be preallocated.

See also: QIODevice::readBlock().

void QTextStream::reset ()

Resets the text stream.

- All flags are set to 0.
- The field width is set to 0.
- The fill character is set to ' ' (space).
- The precision is set to 6.

See also: setf(), width(), fill() and precision().

void QTextStream::setDevice (QIODevice * d)

Sets the IO device to *d*.

See also: device() and unsetDevice().

int QTextStream::setf (int bits)

Sets the stream flag bits *bits*. Returns the previous stream flags.

Equivalent to `flags( flags() | bits )`.

See also: setf() and unsetf().

int QTextStream::setf (int bits, int mask)

Sets the stream flag bits *bits* with a bit mask *mask*. Returns the previous stream flags.

Equivalent to `flags( (flags() & ~mask) | (bits & mask) )`.

See also: setf() and unsetf().

void QTextStream::unsetDevice ()

Unsets the IO device. Equivalent to setDevice(0).

See also: device() and setDevice().

int QTextStream::unsetf (int bits)

Clears the stream flag bits *bits*. Returns the previous stream flags.

Equivalent to `flags( flags() & ~mask )`.

See also: setf().

int QTextStream::width () const

Returns the field width. The default value is 0.

int QTextStream::width (int w)

Sets the field width to *w*. Returns the previous field width.

QTextStream & QTextStream::writeRawBytes (const char * s, uint len)

Writes the *len* bytes from *s* to the stream and returns a reference to the stream.

See also: QIODevice::writeBlock().

174.2.1. URL

Generated from http://www.troll.no/qt/qtextstream.html.

175. QTime Class Reference

```
#include <qdatetime.h>
```

175.1. Detailed Description

The QTime class provides time functions 24 hours a day.

The time resolution of QTime is a millisecond, although the accuracy depends on the underlying operating system. Some operating systems (e.g. Linux and Window NT) support a one-millisecond resolution, while others (MS-DOS and Windows 3.1) support only a 55 millisecond resolution.

See also: QDate and QDateTime.

Examples:

tictac/tictac.cpp
aclock/aclock.cpp

175.2. Member Function Documentation

QTime::QTime ()

Constructs a time 00:00:00.000, which is valid.

QTime::QTime (int h, int m, int s=0, int ms=0)

Constructs a time with hour *h,* minute *m,* seconds *s* and milliseconds *ms.*

QTime QTime::addMSecs (int ms) const

Returns the time plus *ms* milliseconds.

QTime QTime::addSecs (int nsecs) const

Returns the time plus *nsecs* seconds.

See also: secsTo().

QTime QTime::currentTime () `[static]`

Returns the current time.

int QTime::elapsed ()

Returns the number of milliseconds that have elapsed since start() or restart() were called.

See also: start() and restart().

int QTime::hour () const

Returns the hour part (0..23) of the time.

Examples:

tictac/tictac.cpp

bool QTime::isNull () const

Returns TRUE if the time is equal to 00:00:00.000. A null time is valid.

See also: isValid().

bool QTime::isValid () const

Returns TRUE if the time is valid, or FALSE if the time is invalid. The time 23:30:55.746 is valid, while 24:12:30 is invalid.

See also: isNull().

bool QTime::isValid (int h, int m, int s, int ms=0) `[static]`

Returns TRUE if the specified time is valid, otherwise FALSE.

Example:

```
QTime::isValid(21, 10, 30);          // returns TRUE
QTime::isValid(22, 5,  62);          // returns FALSE
```

int QTime::minute () const

Returns the minute part (0..59) of the time.

Examples:
tictac/tictac.cpp
aclock/aclock.cpp

int QTime::msec () const

Returns the millisecond part (0..999) of the time.

int QTime::msecsTo (const QTime & t) const

Returns the number of milliseconds between this time and *t*.

bool QTime::operator!= (const QTime & t) const

Returns TRUE if this time is different from *t*, or FALSE if they are equal.

bool QTime::operator< (const QTime & t) const

Returns TRUE if this time is before *t*, otherwise FALSE.

bool QTime::operator<= (const QTime & t) const

Returns TRUE if this time is before or equal to *t*, otherwise FALSE.

bool QTime::operator== (const QTime & t) const

Returns TRUE if this time is equal to *t*, or FALSE if they are different.

bool QTime::operator> (const QTime & t) const

Returns TRUE if this time is after *t,* otherwise FALSE.

bool QTime::operator>= (const QTime & t) const

Returns TRUE if this time is equal to or after *t,* otherwise FALSE.

int QTime::restart ()

Sets *this to the current time, and returns the number of milliseconds that have elapsed since the last start() or restart().

restart is guaranteed to be atomic, and so is very handy for repeated measurements; call start() to start the first measurement, then restart() for each later measurement.

See also: start() and elapsed().

int QTime::second () const

Returns the second part (0..59) of the time.

Examples:
tictac/tictac.cpp

int QTime::secsTo (const QTime & t) const

Returns the number of seconds from this time to *t* (which is negative if *t* is in the past).

Since QTime measures time within a day and there are 86400 seconds in a day, the result is between -86400 and 86400.

See also: addSecs() and QDateTime::secsTo().

bool QTime::setHMS (int h, int m, int s, int ms=0)

Sets the hour *h,* minute *m,* seconds *s* and milliseconds *ms.* Returns TRUE if the time is valid, otherwise FALSE.

void QTime::start ()

Sets the time to the current time, e.g. for timing:

```
QTime t;
t.start();                          // start clock
... // some lengthy task
debug( "%d\n", t.elapsed() );       // prints # msecs elapsed
```

See also: restart() and elapsed().

QString QTime::toString () const

Converts the date to a string, which is returned. Milliseconds are not included. The string format is "03:40:13".

175.3. Related Functions

QDataStream & operator<< (QDataStream & s, const QTime & t)

Writes a time to the stream.

Serialization format: [Q_UINT32], milliseconds since midnight.

QDataStream & operator>> (QDataStream & s, QTime & t)

Reads a time from the stream.

175.3.1. URL

Generated from http://www.troll.no/qt/qtime.html.

176. QTimer Class Reference

```
#include <qtimer.h>
```

Inherits QObject.

176.1. Detailed Description

The QTimer class provides timer signals and single-shot timers.

It uses timer events internally to provide a more versatile timer. QTimer is very easy to use, create a QTimer, call start() to start it and connect its timeout() to the appropriate slots, then when the time is up it will emit timeout().

Note that a QTimer object is destroyed automatically when its parent object is destroyed.

Example:

```
QTimer *timer = new QTimer( myObject );
connect( timer, SIGNAL(timeout()),
         myObject, SLOT(timerDone()) );
timer->start( 2000, TRUE );                 // 2 seconds single-shot
```

As a special case, a QTimer with timeout 0 times out as soon as all the events in the window system's event queue have been processed.

This can be used to do heavy work while providing a snappy user interface:

```
QTimer *t = new QTimer( myObject );
connect( t, SIGNAL(timeout()), SLOT(processOneThing()) );
t->start( 0, FALSE );
```

myObject->processOneThing() will be called repeatedly and should return quickly (typically after processing one data item) so that Qt can deliver events to widgets, and stop the timer as soon as it has done all its work. This is the traditional way of implementing heavy work in GUI applications; multi-threading is now becoming available on more and more platforms and we expect that null events will eventually be replaced by threading.

Note that QTimer's accuracy depends on the underlying operating system and hardware. Most platforms support an accuracy of 20ms; some provide more. If Qt is unable to deliver the requested number of timer clicks, it will silently discard some.

An alternative to using QTimer is to call QObject::startTimer() for your object and reimplement the QObject::timerEvent() event handler in your class (which must of course inherit QObject). The disadvantage is that timerEvent() does not support such high-level features as single-shot timers or signals.

Examples:
hello/hello.cpp
aclock/aclock.cpp

176.2. Member Function Documentation

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

Constructs a timer with a *parent* and a *name.*

Notice that the destructor of the parent object will destroy this timer object.

QTimer::~QTimer ()

Destroys the timer.

void QTimer::changeInterval (int msec)

Changes the timeout interval to *msec* milliseconds.

If the timer signal is pending, it will be stopped and restarted, otherwise it will be started.

See also: start() and isActive().

bool QTimer::event (QEvent * e) `[virtual protected]`

Handles timer events. Emits timeout() when a timer event is received.

Reimplemented from QObject.

bool QTimer::isActive () const

Returns TRUE if the timer is running (pending), or FALSE is the timer is idle.

void QTimer::singleShot (int msec, QObject * receiver, const char * member) `[static]`

This static function calls a slot after a given time interval.

It is very convenient to use this function because you do not need to bother with a timerEvent or to create a local QTimer object.

Example:

```
#include <qapplication.h>
#include <qtimer.h>
int main( int argc, char **argv )
{
    QApplication a( argc, argv );
    QTimer::singleShot( 10*60*1000, &a, SLOT(quit()) );
        ... // create and show your widgets
    return a.exec();
}
```

This sample program automatically terminates after 10 minutes (i.e. 600000 milliseconds).

int QTimer::start (int msec, bool sshot = FALSE)

Starts the timer with a *msecs* milliseconds timeout.

If *sshot* is TRUE, the timer will be activated only once, otherwise it will continue until it is stopped.

Any pending timer will be stopped.

See also: stop(), changeInterval() and isActive().

Examples:
hello/hello.cpp
aclock/aclock.cpp

void QTimer::stop ()

Stops the timer.

See also: start().

void QTimer::timeout () `[signal]`

This signal is emitted when the timer is activated.

176.2.1. URL

Generated from http://www.troll.no/qt/qtimer.html.

177. QTimerEvent Class Reference

`#include <qevent.h>`

Inherits QEvent.

177.1. Detailed Description

The QTimerEvent class contains parameters that describe a timer event.

Timer events are sent at regular intervals to objects that have started one or more timers. Each timer has a unique identifier.

If the interval is zero, the event will be sent on every iteration of the event loop.

The QTimer class provides a high-level programming interface with one-shot timers and timer signals instead of events.

The event handler QObject::timerEvent() receives timer events.

See also: QTimer, QObject::timerEvent(), QObject::startTimer(), QObject::killTimer() and QObject::killTimers().

Examples:

dclock/dclock.cpp
grapher/grapher.cpp
forever/forever.cpp
progress/progress.cpp
qmag/qmag.cpp
biff/biff.cpp

177.2. Member Function Documentation

QTimerEvent::QTimerEvent (int timerId)

Constructs a timer event object with the timer identifier set to *timerId.*

int QTimerEvent::timerId () const

Returns the unique timer identifier, which is the same identifier as returned from QObject::startTimer().

177.2.1. URL

Generated from http://www.troll.no/qt/qtimerevent.html.

178. QToolBar Class Reference

`#include <qtoolbar.h>`

Inherits QWidget.

178.1. Detailed Description

The QToolBar class provides a simple tool bar.

To use QToolBar, you simply create a QToolBar as child of a QMainWindow, create a number of QToolButton widgets (or other widgets) in left to right (or top to bottom) order, call addSeparator() when you want a separator, and that's all.

The application/application.cpp example does precisely this.

Each QToolBar lives in a dock in a QMainWindow, and can optionally start a new line in its dock. Tool bars that start a new line are always positioned at the left end or top of the tool bar dock; others are placed next to the previous tool bar and word-wrapped as necessary.

The tool bar is able to arrange its buttons horizontally or vertically (see setOrientation() for details) and draws the appropriate frames around the tool button in Windows and Motif style. Generally, QMainWindow will set the orientation correctly for you.

See also: QToolButton, QMainWindow, Parts of Isys on Visual Design,, Microsoft Style Guide,, some common buttons (NB: read the license), and GUI Design Handbook: Tool Bar.

Examples:
    application/application.cpp

178.2. Member Function Documentation

QToolBar::QToolBar (QMainWindow * parent = 0, const char * name = 0)

Constructs an empty tool bar in the top dock of its parent, without any label and without requiring a newline. This is mostly useless.

QToolBar::QToolBar (const char * label, QMainWindow * mainWindow, QWidget * parent, bool newLine = FALSE, const char * name = 0, WFlags f = 0)

Constructs an empty horizontal tool bar which is a parent of *parent* and managed by *mainWindow*. The *label* and *newLine* are passed straight to QMainWindow::addToolBar(). *name* is the object name and *f* is the widget flags.

This is the constructor to use if you want to create torn-off toolbars, or toolbars in the status bar.

QToolBar::QToolBar (const char * label, QMainWindow * parent, QMainWindow::ToolBarDock dock = QMainWindow::Top, bool newLine = FALSE, const char * name = 0)

Constructs an empty tool bar which is a chilf od *parent* and managed by *parent,* initially in *dock,* labelled *and* starting a new line in the dock if *newLine* is TRUE. *name* is the object name, as usual.

QToolBar::~QToolBar ()

Destroys the object and frees any allocated resources.

void QToolBar::addSeparator ()

Adds a separator in here. Cool, man.

bool QToolBar::event (QEvent * e) `[virtual]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QObject.

QMainWindow * QToolBar::mainWindow ()

Returns a pointer to the QMainWindow which controls this tool bar.

QToolBar::Orientation QToolBar::orientation() const

Returns the current orientation of the toolbar.

void QToolBar::paintEvent (QPaintEvent *) `[virtual protected]`

Paint the handle. The Motif style is rather close to Netscape and even closer to KDE.

Reimplemented from QWidget.

void QToolBar::setOrientation (Orientation newOrientation) `[virtual]`

Sets this toolbar to organize its content vertically if *newOrientation* is `Vertical` and horizontally if *newOrientation* is `Horizontal`.

void QToolBar::setStretchableWidget (QWidget * w)

Sets *w* to be expanded if this toolbar is requested to stretch (because QMainWindow right-justifies the dock it's in).

void QToolBar::show () `[virtual]`

Reimplemented to set up geometry management.

Reimplemented from QWidget.

178.2.1. URL

Generated from http://www.troll.no/qt/qtoolbar.html.

179. QToolButton Class Reference

`#include <qtoolbutton.h>`

Inherits QButton.

179.1. Detailed Description

The QToolButton class provides a push button whose appearance has been tailored for use in a QToolBar.

This means that it implements the ridiculous Microsoft auto-raise feature using QIconSet. Apart from that, it's pretty much like a QPushButton. The two classes may at some point be merged.

See also: QPushButton, QToolButton and GUI Design Handbook: Push Button

Examples:
    application/application.cpp

179.2. Member Function Documentation

QToolButton::QToolButton (QIconSet iconSet, const char * textLabel, const char * grouptext, QObject * receiver, const char * slot, QToolBar * parent, const char * name = 0)

Creates a tool button that is a child of *parent* (which must be a QToolBar) and named *name.*

The tool button will display *iconSet,* with text label or tool tip *textLabel* and status-bar message *grouptext,* connected to *slot* in object *receiver,* and returns the button.

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

Constructs an empty tool button.

QToolButton::QToolButton (const QPixmap & pm, const char * textLabel, const char * grouptext, QObject * receiver, const char * slot, QToolBar * parent, const char * name = 0)

Creates a tool button that is a child of *parent* (which must be a QToolBar) and named *name.*

The tool button will display *pm,* with text label or tool tip *textLabel* and status-bar message *grouptext,* connected to *slot* in object *receiver,* and returns the button.

QToolButton::~QToolButton ()

Destroys the object and frees any allocated resources.

void QToolButton::drawButton (QPainter * p) `[virtual protected]`

Draws the edges and decoration of the button (pretty much nothing) and calls drawButtonLabel().

See also: drawButtonLabel() and QButton::paintEvent().

Reimplemented from QButton.

void QToolButton::drawButtonLabel (QPainter * p) `[virtual protected]`

Draws the contents of the button (pixmap and optionally text).

See also: drawButton() and QButton::paintEvent().

Reimplemented from QButton.

void QToolButton::enterEvent (QEvent * e) `[virtual protected]`

Reimplemented to handle the automatic 3D effects in Windows style.

Reimplemented from QWidget.

QIconSet QToolButton::iconSet () const

Returns a copy of the icon set in use. If no icon set has been set, iconSeT() creates one from the pixmap().

If the button doesn't have a pixmap either, iconSet()'s return value is meaningless.

See also: setIconSet() and QIconSet.

void QToolButton::leaveEvent (QEvent * e) `[virtual protected]`

Reimplemented to handle the automatic 3D effects in Windows style.

Reimplemented from QWidget.

void QToolButton::setIconSet (const QIconSet & set)

Sets this tool button to display the icons in *set.* (setPixmap() is effectively a wrapper for this function.)

QToolButton makes a copy of *set,* so you must delete *set* yourself.

See also: iconSet() and QIconSet.

void QToolButton::setOn (bool enable) `[slot]`

Sets this tool button to be on if *enable* is TRUE, and off it *enable* is FALSE.

This function has no effect on non-toggling buttons.

See also: isToggleButton() and toggle().

void QToolButton::setTextLabel (const char * newLabel, bool tipToo = TRUE) `[virtual slot]`

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) `[slot]`

Makes the tool button a toggle button if *enable* is TRUE, or a normal tool button if *enable* is FALSE.

Toggle buttons have an on/off state similar to check boxes. A tool button is initially not a toggle button.

See also: setOn(), toggle(), toggleButton() and toggled().

void QToolButton::setUsesBigPixmap (bool enable) `[virtual slot]`

Sets this button to use the big pixmaps provided by its QIconSet if *enable* is TRUE, and to use the small ones else.

QToolButton automatically connects this slot to the relevant signal in the QMainWindow in which is resides.

void QToolButton::setUsesTextLabel (bool enable) `[virtual slot]`

Sets this button to draw a text label below the icon if *enable* is TRUE, and to not draw it if *enable* is FALSE.

QToolButton automatically connects this slot to the relevant signal in the QMainWindow in which is resides.

QSize QToolButton::sizeHint () const `[virtual]`

Returns a size suitable for this tool button. This depends on GUI style, usesBigPixmap(), textLabel() and usesTextLabel().

Reimplemented from QWidget.

const char * QToolButton::textLabel () const

Returns the text label in use by this tool button, or 0.

See also: setTextLabel(), usesTextLabel(), setUsesTextLabel() and setText().

void QToolButton::toggle () `[slot]`

Toggles the state of this tool button.

This function has no effect on non-toggling buttons.

See also: isToggleButton() and toggle().

bool QToolButton::uses3D () const `[protected]`

Returns TRUE if this button should be drawn using raised edges.

See also: drawButton().

bool QToolButton::usesBigPixmap () const

Returns TRUE if this tool button uses the big (32-pixel) pixmaps, and FALSE if it does not.

See also: setUsesBigPixmap(), setPixmap() and usesTextLabel.

bool QToolButton::usesTextLabel () const

Returns TRUE if this tool button puts a text label below the button pixmap, and FALSE if it does not.

See also: setUsesTextLabel(), setTextLabel() and usesBigPixmap().

179.2.1. URL

Generated from http://www.troll.no/qt/qtoolbutton.html.

180. QToolTip Class Reference

```
#include <qtooltip.h>
```

180.1. Detailed Description

The QToolTip class provides tool tips (sometimes called balloon help) for any widget or rectangular part of a widget.

The tip is a short, one-line 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. In Motif style, Qt's tool tips look much like Motif's but feel more like Windows 95 tool tips.

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 lets the mouse rest on a tip-equipped region for a second or so, and remains in active mode until the user either clicks a mouse button, presses a key, lets the mouse rest for five seconds, or moves the mouse outside *all* tip-equupped 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 already connected to the appropriate status bar:

```
QToolTip::add( quitButton, "Leave the application", g,
               "Leave the application, without asking for confirmation" );
QToolTip::add( closeButton, "Close this window", g,
               "Close this window, without asking for confirmation" );
```

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 the above 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 e.g. tips for text that can move as the user scrolls.

To use QToolTip like this, you need to subclass QToolTip and reimplement maybeTip(). maybeTip() will be called when there's a chance that a tip should pop up. It must decide whether to show a tip, and possibly call add() with the rectangle the tip applies to, the tip's text and optionally the QToolTipGroup details. The tip will disappear once the mouse moves outside the rectangle you supply, and *not reappear* - maybeTip() will be called again if the user lets the mouse rest within the same rectangle again. You can forcibly remove the tip by calling remove() with no arguments. This is handy if the widget scrolls.

See also: QStatusBar, QWhatsThis, QToolTipGroup and GUI Design Handbook: Tool Tip

Examples:
tooltip/tooltip.cpp

180.2. Member Function Documentation

QToolTip::QToolTip (QWidget * parent, QToolTipGroup * group = 0)

Creates 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).

parent is widget you want to add dynamic tool tips to and *group* (optional) is the tool tip group they should belong to.

See also: maybeTip().

void QToolTip::add (QWidget * widget, const QRect & rect, const char * text) `[static]`

Adds a tool tip to a fixed rectangle within *widget. text* is the text shown in the tool tip. QToolTip makes a deep copy of this string.

void QToolTip::add (QWidget * widget, const QRect & rect, const char * text, QToolTipGroup * group, const char * groupText) `[static]`

Adds a tool tip to an entire *widget,* and to tool tip group *group.*

text is the text shown in the tool tip and *longText* is the text emitted from *group.* QToolTip makes deep copies of both strings.

Normally, *longText* is shown in a status bar or similar.

void QToolTip::add (QWidget * widget, const char * text) `[static]`

Adds a tool tip to *widget. text* is the text to be shown in the tool tip. QToolTip makes a deep copy of this string.

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:
tooltip/tooltip.cpp
widgets/widgets.cpp

void QToolTip::add (QWidget * widget, const char * text, QToolTipGroup * group, const char * longText) `[static]`

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.* QToolTip makes deep copies of both strings.

Normally, *longText* is shown in a status bar or similar.

void QToolTip::clear () `[protected]`

Remove all tool tips for this widget immediately.

QFont QToolTip::font () `[static]`

Returns the font common to all tool tips.

See also: setFont().

QToolTipGroup * QToolTip::group () const

Returns the tool tip group this QToolTip is a member of, of 0 if it isn't a member of any group.

The tool tip group is the object responsible for relaying contact betweeen tool tips and a status bar or something else which can show a longer help text.

See also: parentWidget() and QToolTipGroup.

virtual 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 chance that a tool tip should be shown, and must decide whether there is a tool tip for the point *p* in the widget this QToolTip object relates to.

\p is given in that widget's local coordinates. Most maybeTip() implementation will be of the form:

```
if ( &lt;something&gt; ) {
    tip( &lt;something&gt;, &lt;something&gt; );
}
```

The first argument to tip() (a rectangle) should include the *p,* or QToolTip, the user or both can be confused.

See also: tip().

QPalette QToolTip::palette () `[static]`

Returns the palette common to all tool tips.

See also: setPalette().

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().

void QToolTip::remove (QWidget * widget) `[static]`

Remove the tool tip from *widget.*

If there are 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]`

Remove the tool tip for *rect* from *widget.*

If there are more than one tool tip on *widget,* only the one covering rectangle *rect* is removed.

void QToolTip::setFont (const QFont & font) `[static]`

Sets the font for all tool tips to *font*.

See also: font().

void QToolTip::setPalette (const QPalette & palette) `[static]`

Sets the palette for all tool tips to *palette*.

See also: palette().

void QToolTip::tip (const QRect & rect, const char * text) `[protected]`

Pop up a tip saying *text* right now, and remove that 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 come back if the cursor moves back; your maybeTip() has to reinstate it each time.

Examples:
tooltip/tooltip.cpp

void QToolTip::tip (const QRect & rect, const char * text, const char * groupText) `[protected]`

Pop up a tip saying *text* right now, and remove that tip once the cursor moves out of rectangle *rect*.

The tip will not come back if the cursor moves back; your maybeTip() has to reinstate it each time.

180.2.1. URL

Generated from http://www.troll.no/qt/qtooltip.html.

181. QToolTipGroup Class Reference

`#include <qtooltip.h>`

Inherits QObject.

181.1. Detailed Description

The QToolTipGroup class collects tool tips into natural groups.

Tool tips can display *two* texts, the one in the tip and optionally another one, typically in a status bar. QToolTipGroup provides a way to link tool tips to this status bar.

QToolTipGroup has practically no API, it is only used 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 char *)),
         myLabel, SLOT(setText(const char *)) );
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.

181.2. Member Function Documentation

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

Constructs a tool tip group.

QToolTipGroup::~QToolTipGroup ()

Destroy this tool tip groups and all tool tips in it.

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().

void QToolTipGroup::showTip (const char * longText) `[signal]`

This signal is emitted when one of the tool tips in the group is displayed. *longText* is the supplementary text for the displayed tool tip.

See also: removeTip().

181.2.1. URL

Generated from http://www.troll.no/qt/qtooltipgroup.html.

182. QUrlDrag Class Reference

`#include <qdragobject.h>`

Inherits QStoredDrag.

182.1. Detailed Description

Provides for drag-and-drop of a list of file references.

URLs are a useful way to refer to files that may be distributed across multiple machines. Much of the time a URL will refer to a file on a machine local to both the drag source and the drop target, and so the URL will be equivalent to passing a filename, but more extensible.

182.2. Member Function Documentation

QUrlDrag::QUrlDrag (QStrList urls, QWidget * dragSource = 0, const char * name = 0)

Creates an object to drag the list of urls in *urls.* The *dragSource* and *name* arguments are passed on to QStoredDrag.

QUrlDrag::QUrlDrag (QWidget * dragSource = 0, const char * name = 0)

Creates a object to drag. You will need to call setUrls() before you start the drag().

QUrlDrag::~QUrlDrag ()

Destroys the object.

bool QUrlDrag::canDecode (QDragMoveEvent * e) `[static]`

Returns TRUE if decode() would be able to decode *e.*

bool QUrlDrag::decode (QDropEvent * e, QStrList & l) `[static]`

Decodes URLs from *e,* placing the result in *l* (which is first cleared).

Returns TRUE if the event contained a valid list of URLs.

bool QUrlDrag::decodeLocalFiles (QDropEvent * e, QStrList & l) `[static]`

Decodes URLs from *e,* converts them to local files if they refer to local files, and places them in *l* (which is first cleared).

Returns TRUE if the event contained a valid list of URLs. The list will be empty if no URLs were local files.

void QUrlDrag::setUrls (QStrList urls)

Changes the list of *urls* to be dragged.

QString QUrlDrag::urlToLocalFile (const char * url) `[static]`

Returns the name of a local file equivalent to *url,* or a null string if *url* is not a local file.

182.2.1. URL

Generated from http://www.troll.no/qt/qurldrag.html.

183. QValidator Class Reference

`#include <qvalidator.h>`

Inherits QObject.

Inherited by QDoubleValidator and QIntValidator.

183.1. Detailed Description

The QValidator class provides validation of input text.

The class itself is abstract; two subclasses provide rudimentary numeric range checking.

The class includes two virtual functions, validate() and fixup().

validate() is pure virtual, so it must be implemented by every subclass. It returns Invalid, Valid or Acceptable depending on whether its argument is valid (for the class' definition of valid).

fixup() is provided for validators that can repair some or all user errors. The default does nothing. QLineEdit, for example, will call fixup() if the user presses Return and the content is not currently valid, in case fixup() can do magic.

QValidator is generally used with QLineEdit, QSpinBox and QComboBox.

183.2. Member Function Documentation

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

Sets up the internal data structures used by the validator. At the moment there aren't any.

QValidator::~QValidator ()

Deletes the validator and frees any storage and other resources used.

void QValidator::fixup (QString & input) `[virtual]`

Attempts to change *to* be valid according to this validator's rules. Need not result in a valid string - callers of this function must re-test afterwards. The default does nothing.

Reimplementation notes:

Note that *input* may not be the only QString object referencing this string, so it's almost always necessary to detach() the string at the start of the code:

```
input.detach();
```

You can change *input* even if you aren't able to produce a valid string. For example an ISBN validator might want to delete every character except digits and "-", even if the result is not a valid ISBN, and a last-name validator might want to remove white space from the start and end of the string, even if the resulting string is not in the list of known last names.

QValidator::State QValidator::validate(QString & input, int & pos) `[virtual]`

This pure virtual function returns `Invalid` if *input* is invalid according to this validator's rules, `Valid` if it is likely that a little more editing will make the input acceptable (e.g. the user types '4' into a widget which accepts 10-99) and `Acceptable` if the input is completely acceptable.

The function can change *input* and *pos* (the cursor position) if it wants to.

Reimplemented in QIntValidator and QDoubleValidator.

183.2.1. URL

Generated from http://www.troll.no/qt/qvalidator.html.

184. QVBoxLayout Class Reference

`#include <qlayout.h>`

Inherits QBoxLayout.

184.1. Detailed Description

The QVBoxLayout class lines up child widgets vertically.

This class provides an easier way to construct vertical box layout objects. See QBoxLayout for more details.

The simplest way to use this class is:

```
QBoxLayout * l = new QVBoxLayout( widget );
l->addWidget( aWidget );
l->addWidget( anotherWidget );
l->activate()
```

See also: QHBoxLayout and QGridLayout.

Examples:
layout/layout.cpp
widgets/widgets.cpp

184.2. Member Function Documentation

QVBoxLayout::QVBoxLayout (QWidget * parent, int border=0, int autoBorder = -1, const char * name=0)

Creates a new top-level vertical box.

QVBoxLayout::QVBoxLayout (int autoBorder = -1, const char * name=0)

Creates a new vertical box. You have to add it to another layout before using it.

QVBoxLayout::~QVBoxLayout ()

Destroys this box.

184.2.1. URL

Generated from http://www.troll.no/qt/qvboxlayout.html.

185. QWhatsThis Class Reference

```
#include <qwhatsthis.h>
```

185.1. Detailed Description

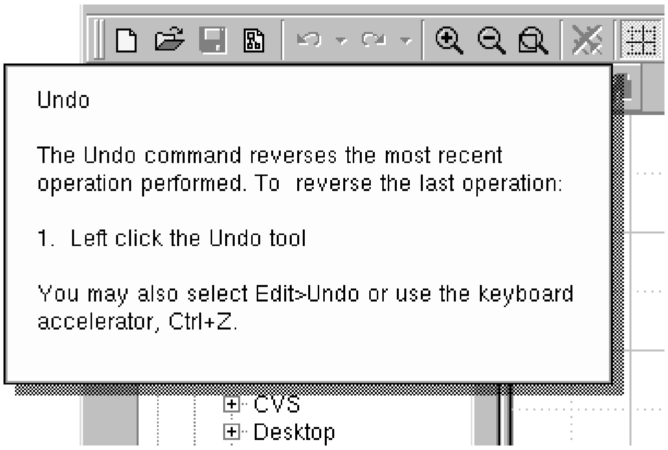
The QWhatsThis class provides a simple description of any widget, e.g. answering the question "what's this?"

What's This help lies between tool tips and fully-blown online help systems:

- Tool Tips - flyweight help, extremely brief, entirely integrated in the user interface.
- What's This? - also lightweight, but can encompass a three-paragraph explanation.
- Online Help - can encompass any amount of information, but is typically a little slower to call up, a little separated from the user's work, and often users feel that using online help is a digression from their real task.

QWhatsThis, then, offers a single window with a single explanatory text, which pops up quickly when the user asks "what's this?", and goes away as soon as the user does something else. There are two ways to make QWhatsThis pop up: Click a "What's This?" button and then click on some other widget to get help for that other widget, or press Shift-F1 to get help for the widget that has keyboard focus.

QWhatsThis provides functions to add() and remove() What's This help for a widget, and it provides a function to create a What's This button suitable for typical tool bars.



More functionality will be provided in the coming releases of Qt.

See also: QToolTip.

185.2. Member Function Documentation

void QWhatsThis::add (QWidget * widget, const QPixmap & icon, const char * title, const char * text, bool deepCopy = TRUE) `[static]`

Adds *text* as What's This help for *widget,* with title line *title* and icon *icon.* If *deepCopy* is TRUE, QWhatsThis makes a deep copy of *title* and *text;* if it is FALSE QWhatsThis just copies the pointers.

void QWhatsThis::add (QWidget * widget, const char * text, bool deepCopy = TRUE) `[static]`

Adds *text* as What's This help for *widget.* If *deepCopy* is TRUE, QWhatsThis makes a deep copy of the string; if it is FALSE QWhatsThis just copies the pointer *text.*

Examples:
application/application.cpp

void QWhatsThis::remove (QWidget * widget) `[static]`

Removes the What's This help for *widget.*

See also: add().

const char * QWhatsThis::textFor (QWidget * widget) `[static]`

Returns the text (not the title) for *widget,* or 0 if there isn't any What's This help for *widget.*

See also: add().

QToolButton * QWhatsThis::whatsThisButton (QWidget * parent) `[static]`

Returns a pointer to a specially configured QToolButton, suitable for use to enter What's This mode.

See also: QToolButton.

185.2.1. URL

Generated from http://www.troll.no/qt/qwhatsthis.html.

186. QWidget Class Reference

```
#include <qwidget.h>
```

Inherits QObject and QPaintDevice.

Inherited by QButton, QComboBox, QDialog, QFrame, QGLWidget, QLineEdit, QMainWindow, QNPWidget, QScrollBar, QSemiModal, QSlider, QStatusBar, QTabBar, QToolBar, QWindow and QXtWidget.

186.1. Detailed Description

The QWidget class is the base class of all user interface objects.

The widget is the atom of the user interface: It receives mouse, keyboard and other events from the window system, and paints a representation of itself on the screen. Every widget is rectangular, and they are sorted in a Z-order. A widget is clipped by its parent and by the widgets in front of it.

A widget without a parent, called a top-level widget, is a window with a frame and a title bar (though it is also possible to create top level widgets without such decoration). A widget with a parent is a child window in its parent. You usually cannot distinguish a child widget from its parent visually.

QWidget has many member functions, but some of them have little direct functionality - for example it has a font but never uses it itself. There are many subclasses which provide real functionality, as diverse as QPushButton, QListBox and QTabDialog.

Groups of functions:

- Window functions: show(), hide(), raise(), lower(), close().
- Top level windows: caption(), setCaption(), icon(), setIcon(), iconText(), setIconText(), isActiveWindow(), setActiveWindow(), iconify().
- Window contents: update(), repaint(), erase(), drawRect(), scroll().
- Geometry: move(), resize(), setGeometry(), pos(), size(), rect(), x(), y(), width(), height(), frameGeometry(), geometry(), childrenRect(), sizeHint(), adjustSize(), mapFromGlobal(), mapFromParent() mapToGlobal(), mapToParent(), maximumSize(), minimumSize(), sizeIncrement(), setMaximumSize(), setMinimumSize(), setSizeIncrement(), setFixedSize()
- Mode: isVisible(), isVisibleTo(), isVisibleToTLW(), isDesktop(), isEnabled(), isEnabledTo(), isEnabledToTLW(), isModal(), isPopup(), isTopLevel(), setEnabled(), hasMouseTracking(), setMouseTracking(), isUpdatesEnabled(), setUpdatesEnabled(), setFontPropagation(), fontPropagation(), setPalettePropagation(), palettePropagation().
- Look and feel: style(), setStyle(), cursor(), setCursor() font(), setFont(), palette(), setPalette(), backgroundMode(), setBackgroundMode(), backgroundPixmap(), setBackgroundPixmap(), backgroundBrush(), colorGroup(), fontMetrics(), fontInfo().
- Keyboard focus functions: isFocusEnabled(), setFocusPolicy(), focusPolicy(), hasFocus(), setFocus(), clearFocus(), setTabOrder(), setFocusProxy().
- Mouse and keyboard grabbing: grabMouse(), releaseMouse(), grabKeyboard(), releaseKeyboard(), mouseGrabber(), keyboardGrabber().
- Event handlers: event(), mousePressEvent(), mouseReleaseEvent(), mouseDoubleClickEvent(), mouseMoveEvent(), keyPressEvent(), keyReleaseEvent(), focusInEvent(), focusOutEvent(), enterEvent(), leaveEvent(), paintEvent(), moveEvent(), resizeEvent(), closeEvent().
- Change handlers: backgroundColorChange(), backgroundPixmapChange(), enabledChange(), fontChange(), paletteChange(), styleChange().

- System functions: parentWidget(), topLevelWidget(), recreate(), winId(), find(), metric().
- Internal kernel functions: setFRect(), setCRect(), focusNextPrevChild(), wmapper(), clearWFlags(), getWFlags(), setWFlags(), testWFlags().

Every widget's constructor accepts two or three standard arguments:

- `QWidget *parent = 0` is the parent of the new widget. If it is 0 (the default), the new widget will be a top-level window. If not, it will be a child of *parent,* and be constrained by *parent's* geometry.
- `const char *name = 0` is the widget name of the new widget. The widget name is little used at the moment - the dumpObjectTree() debugging function uses it, and you can access it using name(). It will become more important when our visual GUI builder is ready (you can name a a widget in the builder, and connect() to it by name in your code).
- `WFlags f = 0` (where available) sets the widget flags; the default is good for almost all widgets, but to get e.g. top-level widgets without a window system frame you must use special flags.

The tictac/tictac.cpp example program is good example of a simple widget. It contains a few event handlers (as all widgets must), a few custom routines that are peculiar to it (as all useful widgets must), and has a few children and connections. Everything it does is done in response to an event: This is by far the most common way to design GUI applications.

You will need to supply the content for your widgets yourself, but here is a brief run-down of the events, starting with the most common ones:

- paintEvent() - called whenever the widget needs to be repainted. Every widget which displays output must implement it, and it is sensible to *never* paint on the screen outside paintEvent(). You are guaranteed to receive a paint event immediately after every resize events, and at once when the widget is first shown.
- resizeEvent() - called when the widget has been resized.
- mousePressEvent() - called when a mouse button is pressed. There are six mouse-related events, mouse press and mouse release events are by far the most important. A widget receives mouse press events when the widget is inside it, or when it has grabbed the mouse using grabMouse().
- mouseReleaseEvent() - called when a mouse button is released. A widget receives mouse release events when it has received the corresponding mouse press event. This means that if the user presses the mouse inside *your* widget, then drags the mouse to somewhere else, then releases, *your* widget receives the release event. There is one exception, however: If a popup menu appears while the mouse button is held down, that popup steals the mouse events at once.
- mouseDoubleClickEvent() - not quite as obvious as it might seem. If the user double-clicks, the widget receives a mouse press event (perhaps a mouse move event or two if he/she does not hold the mouse quite steady), a mouse release event and finally this event. It is *not possible* to distinguish a click from a double click until you've seen whether the second click arrives. (This is one reason why most GUI books recommend that double clicks be an extension of single clicks, rather than a different action.)

If your widget only contains child widgets, you probably do not need to implement any event handlers (except resizeEvent() for custom layout management).

Widgets that accept keyboard input need to reimplement a few more event handlers:

- keyPressEvent() - called whenever a key is pressed, and again when a key has been held down long enough for it to auto-repeat. Note that the Tab and shift-Tab keys are only passed to the widget if they are not used by the focus-change mechanisms. To force those keys to be processed by your widget, you must override QWidget::event().

- focusInEvent() - called when the widget gains keyboard focus (assuming you have called setFocusPolicy(), of course). Well written widgets indicate that they own the keyboard focus in a clear but discreet way.
- focusOutEvent() - called when the widget loses keyboard focus.

Some widgets will need to reimplement some more obscure event handlers, too:

- mouseMoveEvent() - called whenever the mouse moves while a button is held down. This is useful for e.g. dragging. If you call setMouseTracking(TRUE), you get mouse move events even when no buttons are held down. (Note that applications which make use of mouse tracking are often not very useful on low-bandwidth X connections.)
- keyReleaseEvent() - called whenever a key is released, and also while it is held down if the key is auto-repeating. In that case the widget receives a key release event and immediately a key press event for every repeat. Note that the Tab and shift-Tab keys are only passed to the widget if they are not used by the focus-change mechanisms. To force those keys to be processed by your widget, you must override QWidget::event().
- enterEvent() - called when the mouse enters the widget's screen space. (This excludes screen space owned by any children of the widget.)
- leaveEvent() - called when the mouse leaves the widget's screen space.
- moveEvent() - called when the widget has been moved relative to its parent.
- closeEvent() - called when the user closes the widget (or when close() is called).

There are also some *really* obscure events. They are listed in qevent.h and you need to reimplement event() to handle them. The default implementation of event() handles Tab and shift-Tab (to move the keyboard focus), and passes on every other event to one of the more specialized handlers above.

When writing a widget, there are a few more things to look out for. In the constructor, be sure to set up your member variables early on, before there's any chance that you might receive an event.

It is often a good idea to reimplement sizeHint(), so users of your class can set up layout management more easily. If you do, consider offering size management using autoMinimumSize() too.

If your widget is a top-level window, setCaption() and setIcon() set the title bar and icon respectively.

See also: QEvent, QPainter, QGridLayout and QBoxLayout.

Examples:

tictac/tictac.cpp
life/life.cpp
dclock/dclock.cpp
forever/forever.cpp
desktop/desktop.cpp
connect/connect.cpp
tooltip/tooltip.cpp
hello/hello.cpp
xform/xform.cpp
aclock/aclock.cpp
menu/menu.cpp
pref/pref.cpp
progress/progress.cpp
layout/layout.cpp
qmag/qmag.cpp
showimg/showimg.cpp
biff/biff.cpp

widgets/widgets.cpp

186.2. Member Function Documentation

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

Constructs a widget which is a child of *parent,* with the name *name* and widget flags set to *f.*

If *parent* is 0, the new widget becomes a top-level window. If *parent* is another widget, this widget becomes a child window inside *parent.*

The *name* is sent to the QObject constructor.

The widget flags argument *f* is normally 0, but it can be set to customize the window frame of a top-level widget (i.e. *parent* must be zero). To customize the frame, set the `WStyle_Customize` flag OR'ed with any of these flags:

- `WStyle_NormalBorder` gives the window a normal border. Cannot be combined with `WStyle_DialogBorder` or `WStyle_NoBorder`.
- `WStyle_DialogBorder` gives the window a thin dialog border. Cannot be combined with `WStyle_NormalBorder` or `WStyle_NoBorder`.
- `WStyle_NoBorder` gives a borderless window. Note that the user cannot move or resize a borderless window via the window system. Cannot be combined with `WStyle_NormalBorder` or `WStyle_DialogBorder`.
- `WStyle_Title` gives the window a title bar.
- `WStyle_SysMenu` adds a window system menu.
- `WStyle_Minimize` adds a minimize button.
- `WStyle_Maximize` adds a maximize button.
- `WStyle_MinMax` is equal to `(WStyle_Minimize | WStyle_Maximize)`
- `WStyle_Tool` makes the window a tool window, usually combined with `WStyle_NoBorder`. A tool window is a small window that lives for a short time and it is typically used for creating popup windows.

Note that X11 does not necessarily support all style flag combinations. X11 window managers live their own lives and can only take hints. Win32 supports all style flags.

Example:

```
QLabel *toolTip = new QLabel( 0, "myToolTip",
                              WStyle_Customize | WStyle_NoBorder |
                              WStyle_Tool );
```

The widget flags are defined in qwindowdefs.h (which is included by qwidget.h).

QWidget::~QWidget ()

Destroys the widget.

All children of this widget are deleted first. The application exits if this widget is (was) the main widget.

bool QWidget::acceptDrops () const

Returns TRUE if drop events are enabled for this widget.

See also: setAcceptDrops().

void QWidget::adjustSize () `[virtual]`

Adjusts the size of the widget to fit the contents.

Uses sizeHint() if valid (i.e if the size hint's width and height are equal to or greater than 0), otherwise sets the size to the children rectangle (the union of all child widget geometries).

See also: sizeHint() and childrenRect().

Examples:
movies/main.cpp

Reimplemented in QMessageBox.

const QColor & QWidget::backgroundColor () const

Returns the background color of this widget.

The background color is independent of the color group.

Setting a new palette overwrites the background color.

See also: setBackgroundColor(), foregroundColor(), colorGroup() and palette().

Examples:
grapher/grapher.cpp
xform/xform.cpp
widgets/widgets.cpp

void QWidget::backgroundColorChange (const QColor & oldBackgroundColor) `[virtual protected]`

This virtual function is called from setBackgroundColor(). *oldBackgroundColor* is the previous background color; you can get the new background color from backgroundColor().

Reimplement this function if your widget needs to know when its background color changes. You will almost certainly need to update the widget using either repaint(TRUE) or update().

The default implementation calls update().

See also: setBackgroundColor(), backgroundColor(), setPalette(), repaint() and update().

QWidget::BackgroundMode QWidget::backgroundMode() const

Returns the mode most recently set by setBackgroundMode(). The default is PaletteBackground

const QPixmap * QWidget::backgroundPixmap () const

Returns the background pixmap, or null if no background pixmap has not been set. If the widget has been made empty, this function will return a pixmap which isNull() rather than a null pointer.

See also: setBackgroundPixmap() and setBackgroundMode().

void QWidget::backgroundPixmapChange (const QPixmap & oldBackgroundPixmap) `[virtual protected]`

This virtual function is called from setBackgroundPixmap(). *oldBackgroundPixmap* is the previous background pixmap; you can get the new background pixmap from backgroundPixmap().

Reimplement this function if your widget needs to know when its background pixmap changes. You will almost certainly need to update the widget using either repaint(TRUE) or update().

The default implementation calls update().

See also: setBackgroundPixmap(), backgroundPixmap(), repaint() and update().

const char * QWidget::caption () const

Returns the widget caption, or null if no caption has been set.

See also: setCaption(), icon() and iconText().

QRect QWidget::childrenRect () const

Returns the bounding rectangle of the widget's children.

Examples:
widgets/widgets.cpp

void QWidget::clearFocus ()

Takes keyboard input focus from the widget.

If the widget has active focus, a focus out event is sent to this widget to tell it that it is about to loose the focus.

This widget must enable focus setting in order to get the keyboard input focus, i.e. it must call setFocusPolicy().

See also: hasFocus(), setFocus(), focusInEvent(), focusOutEvent(), setFocusPolicy() and QApplication::focusWidget().

void QWidget::clearMask ()

Removes any mask set by setMask().

See also: setMask().

void QWidget::clearWFlags (WFlags f) `[protected]`

For internal use only.

bool QWidget::close (bool forceKill=FALSE) `[virtual]`

Closes this widget. Returns TRUE if the widget was closed, otherwise FALSE.

First it sends the widget a QCloseEvent. The widget is hidden if it accepts the close event. The default implementation of QWidget::closeEvent() accepts the close event.

If *forceKill* is TRUE, the widget is deleted whether it accepts the close event or not.

The application is terminated when the main widget is closed.

The QApplication::lastWindowClosed() signal is emitted when the last visible top level widget is closed.

See also: closeEvent(), QCloseEvent, hide(), QApplication::quit() and QApplication::setMainWidget().

Examples:
application/application.cpp

void QWidget::closeEvent (QCloseEvent * e) `[virtual protected]`

This event handler can be reimplemented in a subclass to receive widget close events.

The default implementation calls e->accept(), which hides this widget. See the QCloseEvent documentation for more details.

See also: event(), hide(), close() and QCloseEvent.

Reimplemented in QDialog.

const QColorGroup & QWidget::colorGroup () const

Returns the current color group of the widget palette.

The color group is determined by the state of the widget.

A disabled widget returns the QPalette::disabled() color group, a widget in focus returns the QPalette::active() color group and a normal widget returns the QPalette::normal() color group.

See also: palette() and setPalette().

Examples:
life/life.cpp

void QWidget::create (WId window) `[protected]`

For internal use only.

void QWidget::create (WId window, bool initializeWindow, bool destroyOldWindow) `[protected]`

Creates a new widget window if *window* is null, otherwise sets the widget's window to *window.*

Initializes the window (sets the geometry etc.) if *initializeWindow* is TRUE. If *initializeWindow* is FALSE, no initialization is performed. This parameter makes only sense if *window* is a valid window.

Destroys the old window if *destroyOldWindow* is TRUE. If *destroyOldWindow* is FALSE, you are responsible for destroying the window yourself (using platform native code).

The QWidget constructor calls create(0,TRUE,TRUE) to create a window for this widget.

const QCursor & QWidget::cursor () const

Returns the widget cursor.

See also: setCursor().

void QWidget::destroy (bool destroyWindow, bool destroySubWindows) `[protected]`

Frees up window system resources. Destroys the widget window if *destroyWindow* is TRUE.

destroy() calls itself recursively for all the child widgets, passing *destroySubWindows* for the *destroyWindow* parameter. To have more control over destruction of subwidgets, destroy subwidgets selectively first.

This function is usually called from the QWidget destructor.

void QWidget::drawText (int x, int y, const char * str)

Writes *str* at position *x,y.*

The *y* position is the base line position of the text. The text is drawn using the default font and the default foreground color.

This function is provided for convenience. You will generally get more flexible results and often higher speed by using a a painter instead.

See also: setFont(), foregroundColor() and QPainter::drawText().

void QWidget::drawText (const QPoint & pos, const char * str)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

void QWidget::enabledChange (bool oldEnabled) `[virtual protected]`

This virtual function is called from setEnabled(). *oldEnabled* is the previous setting; you can get the new setting from enabled().

Reimplement this function if your widget needs to know when it becomes enabled or disabled. You will almost certainly need to update the widget using either repaint(TRUE) or update().

The default implementation calls repaint(TRUE).

See also: setEnabled(), isEnabled(), repaint() and update().

Reimplemented in QSpinBox and QButton.

void QWidget::enterEvent (QEvent *) `[virtual protected]`

This event handler can be reimplemented in a subclass to receive widget enter events.

An event is sent to the widget when the mouse cursor enters the widget.

The default implementation does nothing.

See also: leaveEvent(), mouseMoveEvent() and event().

Reimplemented in QToolButton.

void QWidget::erase (int x, int y, int w, int h)

Erases the specified area *(x,y,w,h)* in the widget without generating a paint event.

If *w* is negative, it is replaced with `width() - x`. If *h* is negative, it is replaced width `height() - y`.

Child widgets are not affected.

See also: repaint().

Examples:
connect/connect.cpp

void QWidget::erase ()

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

This version erases the entire widget.

void QWidget::erase (const QRect & r)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

bool QWidget::event (QEvent * e) `[virtual protected]`

This is the main event handler. You may reimplement this function in a subclass, but we recommend using one of the specialized event handlers instead.

The main event handler first passes an event through all event filters that have been installed. If none of the filters intercept the event, it calls one of the specialized event handlers.

Key press/release events are treated differently from other events. event() checks for Tab and shift-Tab and tries to move the focus appropriately. If there is no widget to move the focus to (or the key press is not Tab or shift-Tab), event() calls keyPressEvent().

This function returns TRUE if it is able to pass the event over to someone, or FALSE if nobody wanted the event.

See also: closeEvent(), focusInEvent(), focusOutEvent(), enterEvent(), keyPressEvent(), keyReleaseEvent(), leaveEvent(), mouseDoubleClickEvent(), mouseMoveEvent(), mousePressEvent(), mouseReleaseEvent(), moveEvent(), paintEvent(), resizeEvent(), QObject::event() and QObject::timerEvent().

Reimplemented from QObject.

QWExtra * QWidget::extraData () `[protected]`

For internal use only.

QWidget * QWidget::find (WId id) `[static]`

Returns a pointer to the widget with window identifer/handle *id.*

The window identifier type depends by the underlying window system, see qwindowdefs.h for the actual definition. If there is no widget with this identifier, a null pointer is returned.

See also: wmapper() and id().

QFocusData * QWidget::focusData () `[protected]`

Returns a pointer to the focus data for this widget's top-level widget.

Focus data always belongs to the top-level widget. The focus data list contains all the widgets in this top-level widget that can accept focus, in tab order. An iterator points to the current focus widget (focusWidget() returns a pointer to this widget).

This information is useful for implementing advanced versions of focusNextPrevChild().

void QWidget::focusInEvent (QFocusEvent *) `[virtual protected]`

This event handler can be reimplemented in a subclass to receive keyboard focus events (focus received) for the widget.

A widget must accept focus initially in order to receive focus events.

The default implementation calls repaint() since the widget's color group changes from normal to active. You may want to call repaint(FALSE) to reduce flicker in any reimplementation.

As a special case to support applications not utilizing focus, Top-level widgets that have NoFocus policy will receive focus events and gain keyboard events, but the repaint is not done by default.

See also: focusOutEvent(), setFocusPolicy(), keyPressEvent(), keyReleaseEvent(), event() and QFocusEvent.

Reimplemented in QSlider, QPushButton, QListBox, QButton, QMultiLineEdit, QListView, QLineEdit and QComboBox.

bool QWidget::focusNextPrevChild (bool next) `[virtual protected]`

Finds a new widget to give the keyboard focus to, as appropriate for Tab/shift-Tab, and returns TRUE if is can find a new widget and FALSE if it can't,

If *next* is true, this function searches "forwards", if *next* is FALSE, "backwards".

Sometimes, you will want to reimplement this function. For example, a web browser might reimplement it to move its "current active link" forwards or backwards, and call QWidget::focusNextPrevChild() only when it reaches the last/first.

Child widgets call focusNextPrevChild() on their parent widgets, and only the top-level widget will thus make the choice of where to redirect focus. By overriding this method for an object, you thus gain control of focus traversal for all child widgets.

See also: focusData().

Reimplemented in QScrollView.

void QWidget::focusOutEvent (QFocusEvent *) `[virtual protected]`

This event handler can be reimplemented in a subclass to receive keyboard focus events (focus lost) for the widget.

A widget must accept focus initially in order to receive focus events.

The default implementation calls repaint() since the widget's color group changes from active to normal. You may want to call repaint(FALSE) to reduce flicker in any reimplementation.

See also: focusInEvent(), setFocusPolicy(), keyPressEvent(), keyReleaseEvent(), event() and QFocusEvent.

Reimplemented in QListBox, QButton, QMultiLineEdit, QListView and QLineEdit.

QWidget::FocusPolicy QWidget::focusPolicy() const

Returns `QWidget::TabFocus` if the widget accepts focus by tabbing, `QWidget::ClickFocus` if the widget accepts focus by clicking, `QWidget::StrongFocus` if it accepts both and `QWidget::NoFocus` if it does not accept focus at all.

See also: isFocusEnabled(), setFocusPolicy(), focusInEvent(), focusOutEvent(), keyPressEvent(), keyReleaseEvent() and isEnabled().

QWidget * QWidget::focusProxy () const

Returns a pointer to the focus proxy, or 0 if there is no focus proxy.

See also: setFocusProxy().

QWidget * QWidget::focusWidget () const

Returns the focus widget in this widget's window. This is not the same as QApplication::focusWidget(), which returns the focus widget in the currently active window.

const QFont & QWidget::font () const

Returns the font currently set for the widget.

fontInfo() tells you what font is actually being used.

See also: setFont(), fontInfo() and fontMetrics().

Examples:
grapher/grapher.cpp
hello/hello.cpp
xform/xform.cpp

void QWidget::fontChange (const QFont & oldFont) `[virtual protected]`

This virtual function is called from setFont(). *oldFont* is the previous font; you can get the new font from font().

Reimplement this function if your widget needs to know when its font changes. You will almost certainly need to update the widget using either repaint(TRUE) or update().

The default implementation calls update().

See also: setFont(), font(), repaint() and update().

Reimplemented in QSpinBox and QMenuBar.

QFontInfo QWidget::fontInfo () const

Returns the font info for the widget.

The font info object is automatically updated if somebody sets a new widget font. We have decided to change this policy in Qt 2.0: setting a new font for a widget should not affect existing QFontInfo objects.

See also: font(), fontMetrics() and setFont().

Examples:
xform/xform.cpp

QFontMetrics QWidget::fontMetrics () const

Returns the font metrics for the widget.

The font metrics object is automatically updated if somebody sets a new widget font. We have decided to change this policy in Qt 2.0: setting a new font for a widget should not affect existing QFontMetrics objects.

See also: font(), fontInfo() and setFont().

Examples:

drawdemo/drawdemo.cpp
xform/xform.cpp
qmag/qmag.cpp
showimg/showimg.cpp
widgets/widgets.cpp

QWidget::PropagationMode QWidget::fontPropagation() const

Returns the font propagation mode of this widget. The default font propagation mode is `NoChildren`, but you can set it to *SameFont* or *AllChildren.*

See also: setFontPropagation().

const QColor & QWidget::foregroundColor () const

Returns the foreground color of this widget.

The foreground color equals `colorGroup().foreground()`.

See also: backgroundColor() and colorGroup().

Examples:

aclock/aclock.cpp

const QRect & QWidget::frameGeometry () const

Returns the geometry of the widget, relative to its parent and including the window frame.

See also: geometry(), x(), y() and pos().

const QRect & QWidget::geometry () const

Returns the geometry of the widget, relative to its parent widget and excluding the window frame.

See also: frameGeometry(), size() and rect().

Examples:

qmag/qmag.cpp

WFlags QWidget::getWFlags () const `[protected]`

For internal use only.

void QWidget::grabKeyboard ()

Grabs all keyboard input.

This widget will receive all keyboard events, independent of the active window.

Warning: Grabbing the keyboard might lock the terminal.

See also: releaseKeyboard(), grabMouse() and releaseMouse().

void QWidget::grabMouse ()

Grabs the mouse input.

This widget will be the only one to receive mouse events until releaseMouse() is called.

Warning: Grabbing the mouse might lock the terminal.

It is almost never necessary to grab the mouse when using Qt since Qt grabs and releases it sensibly. In particular, Qt grabs the mouse when a button is pressed and keeps it until the last button is released.

See also: releaseMouse(), grabKeyboard() and releaseKeyboard().

Examples:
qmag/qmag.cpp

void QWidget::grabMouse (const QCursor & cursor)

Grabs the mouse intput and changes the cursor shape.

The cursor will assume shape *cursor* (for as long as the mouse focus is grabbed) and this widget will be the only one to receive mouse events until releaseMouse() is called().

Warning: Grabbing the mouse might lock the terminal.

See also: releaseMouse(), grabKeyboard(), releaseKeyboard() and setCursor().

bool QWidget::hasFocus () const

Returns TRUE if this widget (or its focus proxy) has the keyboard input focus, otherwise FALSE.

Equivalent to `qApp->focusWidget() == this`.

See also: setFocus(), clearFocus(), setFocusPolicy() and QApplication::focusWidget().

bool QWidget::hasMouseTracking () const

Returns TRUE if mouse tracking is enabled for this widget, or FALSE if mouse tracking is disabled.

See also: setMouseTracking().

int QWidget::height () const

Returns the height of the widget, excluding the window frame.

See also: geometry(), width() and size().

Examples:
grapher/grapher.cpp
tooltip/tooltip.cpp
drawdemo/drawdemo.cpp
xform/xform.cpp
aclock/aclock.cpp
pref/pref.cpp
qmag/qmag.cpp
showimg/showimg.cpp

void QWidget::hide () `[virtual slot]`

Hides the widget.

The QApplication::lastWindowClosed() signal is emitted when the last visible top level widget is hidden,

See also: show(), iconify() and isVisible().

Examples:
xform/xform.cpp
progress/progress.cpp

Reimplemented in QPopupMenu and QMenuBar.

const QPixmap * QWidget::icon () const

Returns the widget icon pixmap, or null if no icon has been set.

See also: setIcon(), iconText() and caption().

const char * QWidget::iconText () const

Returns the widget icon text, or null if no icon text has been set.

See also: setIconText(), icon() and caption().

void QWidget::iconify () `[slot]`

Iconifies the widget.

Calling this function has no effect for other than top-level widgets.

See also: show(), hide() and isVisible().

bool QWidget::isActiveWindow () const

Returns TRUE if the top-level widget containing this widget is the active window.

See also: setActiveWindow() and topLevelWidget().

bool QWidget::isDesktop () const

Returns TRUE if the widget is a desktop widget, otherwise FALSE.

A desktop widget is also a top-level widget.

See also: isTopLevel() and QApplication::desktop().

bool QWidget::isEnabled () const

Returns TRUE if the widget is enabled, or FALSE if it is disabled.

See also: setEnabled().

bool QWidget::isEnabledTo (QWidget * ancestor) const

Returns TRUE if this widget and every parent up to but excluding *ancestor* is enabled, otherwise returns FALSE.

See also: setEnabled() and isEnabled().

bool QWidget::isEnabledToTLW () const

Returns TRUE if this widget and every parent up to the top level widget is enabled, otherwise returns FALSE.

This is equivalent to isEnabledTo(0).

See also: setEnabled() and isEnabled().

bool QWidget::isFocusEnabled () const

Returns TRUE if the widget accepts keyboard focus, or FALSE if it does not.

Keyboard focus is initially disabled (i.e. focusPolicy() == `QWidget::NoFocus`).

You must enable keyboard focus for a widget if it processes keyboard events. This is normally done from the widget's constructor. For instance, the QLineEdit constructor calls setFocusPolicy(`QWidget::StrongFocus`).

See also: setFocusPolicy(), focusInEvent(), focusOutEvent(), keyPressEvent(), keyReleaseEvent() and isEnabled().

bool QWidget::isModal () const

Returns TRUE if the widget is a modal widget, otherwise FALSE.

A modal widget is also a top-level widget.

See also: isTopLevel() and QDialog.

bool QWidget::isPopup () const

Returns TRUE if the widget is a popup widget, otherwise FALSE.

A popup widget is created by specifying the widget flag `WType_Popup` to the widget constructor.

A popup widget is also a top-level widget.

See also: isTopLevel().

bool QWidget::isTopLevel () const

Returns TRUE if the widget is a top-level widget, otherwise FALSE.

A top-level widget is a widget which usually has a frame and a caption (title bar). Popup and desktop widgets are also top-level widgets. Modal dialog widgets are the only top-level widgets that can have parent widgets; all other top-level widgets have null parents. Child widgets are the opposite of top-level widgets.

See also: topLevelWidget(), isModal(), isPopup(), isDesktop() and parentWidget().

bool QWidget::isUpdatesEnabled () const

Returns TRUE if updates are enabled, otherwise FALSE.

See also: setUpdatesEnabled().

bool QWidget::isVisible () const

Returns TRUE if the widget itself is set to visible status, or else FALSE. Calling show() sets the widget to visible status; calling hide() sets it to hidden status. Iconified top-level widgets also have hidden status.

If a widget is set to visible status, but its parent widget is set to hidden status, this function returns TRUE. isVisibleToTLW() looks at the visibility status of the parent widgets up to the top level widget.

This function returns TRUE if the widget currently is obscured by other windows on the screen, but would be visible if moved.

See also: show(), hide() and isVisibleToTLW().

bool QWidget::isVisibleTo (QWidget * ancestor) const

Returns TRUE if this widget and every parent up to but excluding *ancestor* is visible, otherwise returns FALSE.

This function returns TRUE if the widget it is obscured by other windows on the screen, but would be visible if moved.

See also: show(), hide() and isVisible().

bool QWidget::isVisibleToTLW () const

Returns TRUE if this widget and every parent up to the top level widget is visible, otherwise returns FALSE.

This function returns TRUE if the widget it is obscured by other windows on the screen, but would be visible if moved.

This is equivalent to isVisibleTo(0).

See also: show(), hide() and isVisible().

void QWidget::keyPressEvent (QKeyEvent * e) `[virtual protected]`

This event handler can be reimplemented in a subclass to receive key press events for the widget.

A widget must accept focus initially and have focus in order to receive a key press event.

If you reimplement this handler, it is very important that you ignore() the press if you do not understand it, so that the widget's parent can interpret it.

The default implementation ignores the event.

As a special case to support applications not utilizing focus, Top-level widgets that have NoFocus policy will receive keyboard events.

See also: keyReleaseEvent(), QKeyEvent::ignore(), setFocusPolicy(), focusInEvent(), focusOutEvent(), event() and QKeyEvent.

Reimplemented in QSlider, QListBox, QButton, QTabBar, QMultiLineEdit, QListView, QPopupMenu, QMessageBox, QScrollBar, QLineEdit, QDialog, QRadioButton, QFileDialog, QMenuBar and QComboBox.

void QWidget::keyReleaseEvent (QKeyEvent * e) `[virtual protected]`

This event handler can be reimplemented in a subclass to receive key release events for the widget.

A widget must accept focus initially and have focus in order to receive a key release event.

If you reimplement this handler, it is very important that you ignore() the release if you do not understand it, so that the widget's parent can interpret it.

The default implementation ignores the event.

See also: keyPressEvent(), QKeyEvent::ignore(), setFocusPolicy(), focusInEvent(), focusOutEvent(), event() and QKeyEvent.

QWidget * QWidget::keyboardGrabber () `[static]`

Returns a pointer to the widget that is currently grabbing the keyboard input.

If no widget in this application is currently grabbing the keyboard, 0 is returned.

See also: grabMouse() and mouseGrabber().

void QWidget::leaveEvent (QEvent *) `[virtual protected]`

This event handler can be reimplemented in a subclass to receive widget leave events.

A leave event is sent to the widget when the mouse cursor leaves the widget.

The default implementation does nothing.

See also: enterEvent(), mouseMoveEvent() and event().

Reimplemented in QToolButton, QMultiLineEdit, QLineEdit, QXtWidget and QMenuBar.

void QWidget::lower () `[slot]`

Lowers the widget to the bottom of the parent widget's stack.

If there are siblings of this widget that overlap it on the screen, this widget will be obscured by its siblings afterwards.

See also: raise().

QPoint QWidget::mapFromGlobal (const QPoint & pos) const

Translates the global screen coordinate *pos* to widget coordinates.

See also: mapToGlobal().

QPoint QWidget::mapFromParent (const QPoint & p) const

Translates the parent widget coordinate *pos* to widget coordinates.

Same as mapFromGlobal() if the widget has no parent.

See also: mapToParent().

QPoint QWidget::mapToGlobal (const QPoint & pos) const

Translates the widget coordinate *pos* to global screen coordinates. For example,

```
mapToGlobal(QPoint(0,0))
```

would give the global coordinates of the top-left pixel of the widget.

See also: mapFromGlobal().

QPoint QWidget::mapToParent (const QPoint & p) const

Translates the widget coordinate *pos* to a coordinate in the parent widget.

Same as mapToGlobal() if the widget has no parent.

See also: mapFromParent().

QSize QWidget::maximumSize () const

Returns the maximum widget size.

The widget cannot be resized to a larger size than the maximum widget size.

See also: setMaximumSize(), minimumSize() and sizeIncrement().

int QWidget::metric (int m) const `[virtual protected]`

Internal implementation of the virtual QPaintDevice::metric() function.

Use the QPaintDeviceMetrics class instead.

Reimplemented from QPaintDevice.

QSize QWidget::minimumSize () const

Returns the minimum widget size.

The widget cannot be resized to a smaller size than the minimum widget size.

See also: setMinimumSize(), maximumSize() and sizeIncrement().

void QWidget::mouseDoubleClickEvent (QMouseEvent * e) `[virtual protected]`

This event handler can be reimplemented in a subclass to receive mouse double click events for the widget.

The default implementation generates a normal mouse press event.

Note that the widgets gets a mousePressEvent() and a mouseReleaseEvent() before the mouseDoubleClickEvent().

See also: mousePressEvent(), mouseReleaseEvent(), mouseMoveEvent(), event() and QMouseEvent.

Reimplemented in QListBox, QMultiLineEdit, QListView, QLineEdit and QComboBox.

QWidget * QWidget::mouseGrabber () `[static]`

Returns a pointer to the widget that is currently grabbing the mouse input.

If no widget in this application is currently grabbing the mouse, 0 is returned.

See also: grabMouse() and keyboardGrabber().

void QWidget::mouseMoveEvent (QMouseEvent *) `[virtual protected]`

This event handler can be reimplemented in a subclass to receive mouse move events for the widget.

If mouse tracking is switched off, mouse move events only occur if a mouse button is down while the mouse is being moved. If mouse tracking is switched on, mouse move events occur even if no mouse button is down.

QMouseEvent::pos() reports the position of the mouse cursor, relative to this widget. For press and release events, the position is usually the same as the position of the last mouse move event, but it might be different if the user moves and clicks the mouse fast. This is a feature of the underlying window system, not Qt.

The default implementation does nothing.

See also: setMouseTracking(), mousePressEvent(), mouseReleaseEvent(), mouseDoubleClickEvent(), event() and QMouseEvent.

Reimplemented in QSlider, QListBox, QMultiLineEdit, QButton, QListView, QPopupMenu, QScrollBar, QLineEdit, QMenuBar and QComboBox.

void QWidget::mousePressEvent (QMouseEvent *) `[virtual protected]`

This event handler can be reimplemented in a subclass to receive mouse press events for the widget.

The default implementation does nothing.

If you create new widgets in the mousePressEvent() the mouseReleaseEvent() may not end up where you expect, depending on the underlying window system (or X11 window manager), the widgets' location and maybe more.

See also: mouseReleaseEvent(), mouseDoubleClickEvent(), mouseMoveEvent(), event() and QMouseEvent.

Reimplemented in QSlider, QListBox, QTabBar, QButton, QMultiLineEdit, QListView, QPopupMenu, QScrollBar, QLineEdit, QMenuBar and QComboBox.

void QWidget::mouseReleaseEvent (QMouseEvent *) `[virtual protected]`

This event handler can be reimplemented in a subclass to receive mouse release events for the widget.

The default implementation does nothing.

See also: mouseReleaseEvent(), mouseDoubleClickEvent(), mouseMoveEvent(), event() and QMouseEvent.

Reimplemented in QSlider, QListBox, QTabBar, QButton, QMultiLineEdit, QListView, QPopupMenu, QScrollBar, QLineEdit, QRadioButton, QMenuBar and QComboBox.

void QWidget::move (int x, int y) `[virtual slot]`

Moves the widget to the position *(x,y)* relative to the parent widget.

A move event is generated immediately if the widget is visible. If the widget is invisible, the move event is generated when show() is called.

This function is virtual, and all other overloaded move() implementations call it.

Warning: If you call move() or setGeometry() from moveEvent(), you may see infinite recursion.

See also: pos(), resize(), setGeometry() and moveEvent().

Examples:
xform/xform.cpp

Reimplemented in QSemiModal and QDialog.

void QWidget::move (const QPoint &) `[slot]`

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Examples:
drawdemo/drawdemo.cpp
xform/xform.cpp

void QWidget::moveEvent (QMoveEvent *) `[virtual protected]`

This event handler can be reimplemented in a subclass to receive widget move events. When the widget receives this event, it is already at the new position.

The old position is accessible through QMoveEvent::oldPos().

The default implementation does nothing.

See also: resizeEvent(), event(), move() and QMoveEvent.

void QWidget::paintEvent (QPaintEvent *) `[virtual protected]`

This event handler can be reimplemented in a subclass to receive widget paint events. Actually, it more or less *must* be reimplemented.

The default implementation does nothing.

When the paint event occurs, the update rectangle QPaintEvent::rect() normally has been cleared to the background color or pixmap. An exception is repaint() with erase=FALSE.

For many widgets it is sufficient to redraw the entire widget each time, but some need to consider the update rectangle to avoid flicker or slow update.

Pixmaps can also be used to implement flicker-free update.

update() and repaint() can be used to force a paint event.

See also: event(), repaint(), update(), QPainter, QPixmap and QPaintEvent.

Reimplemented in QGLWidget, QSlider, QTabBar, QButton, QToolBar, QTableView, QGroupBox, QPopupMenu, QScrollBar, QLineEdit, QFrame, QMainWindow, QStatusBar, QTabDialog and QComboBox.

const QPalette & QWidget::palette () const

Returns the widget palette.

See also: setPalette(), colorGroup() and QApplication::palette().

void QWidget::paletteChange (const QPalette & oldPalette) `[virtual protected]`

This virtual function is called from setPalette(). *oldPalette* is the previous palette; you can get the new palette from palette().

Reimplement this function if your widget needs to know when its palette changes. You will almost certainly need to update the widget using either repaint(TRUE) or update().

The default implementation calls update().

See also: setPalette(), palette(), repaint() and update().

Reimplemented in QSpinBox.

QWidget::PropagationMode QWidget::palettePropagation() const

Returns the palette propagation mode of this widget. The default palette propagation mode is `NoChildren,` but you can set it to *SamePalette* or *AllChildren.*

See also: setPalettePropagation().

QWidget * QWidget::parentWidget () const

Returns a pointer to the parent of this widget, or a null pointer if it does not have any parent widget.

QPoint QWidget::pos () const

Returns the postion of the widget in its parent widget, including the window frame.

See also: frameGeometry(), x() and y().

Examples:
qmag/qmag.cpp

void QWidget::raise () `[slot]`

Raises this widget to the top of the parent widget's stack.

If there are any siblings of this widget that overlap it on the screen, this widget will be visually in front of its siblings afterwards.

See also: lower().

void QWidget::recreate (QWidget * parent, WFlags f, const QPoint & p, bool showIt=FALSE)

Reparents the widget. The widget gets a new *parent,* new widget flags (*f,* but as usual, use 0) at a new position in its new parent (*p).*

If *showIt* is TRUE, show() is called once the widget has been recreated.

If the new parent widget is in a different top-level widget, the reparented widget and its children are appended to the end of the TAB chain of the new parent widget, in the same internal order as before. If one of the moved widgets had keyboard focus, recreate() calls clearFocus() for that widget.

If the new parent widget is in the same top-level widget as the old parent, recreate doesn't change the TAB order or keyboard focus.

See also: getWFlags().

QRect QWidget::rect () const

Returns the the internal geometry of the widget, excluding the window frame. rect() equals QRect(0,0,width(),height()).

See also: size().

Examples:
grapher/grapher.cpp
desktop/desktop.cpp
picture/picture.cpp
menu/menu.cpp

void QWidget::releaseKeyboard ()

Releases the keyboard grab.

See also: grabKeyboard(), grabMouse() and releaseMouse().

void QWidget::releaseMouse ()

Releases the mouse grab.

See also: grabMouse(), grabKeyboard() and releaseKeyboard().

Examples:
qmag/qmag.cpp

void QWidget::repaint (int x, int y, int w, int h, bool erase=TRUE) `[slot]`

Repaints the widget directly by calling paintEvent() directly, unless updates are disabled or the widget is hidden.

Erases the widget area *(x,y,w,h)* if *erase* is TRUE.

If *w* is negative, it is replaced with `width() - x`. If *h* is negative, it is replaced width `height() - y`.

Doing a repaint() usually is faster than doing an update(), but calling update() many times in a row will generate a single paint event.

Warning: If you call repaint() in a function which may itself be called from paintEvent(), you may see infinite recursion. The update() function never generates recursion.

See also: update(), paintEvent(), setUpdatesEnabled() and erase().

Examples:
forever/forever.cpp
tooltip/tooltip.cpp
hello/hello.cpp
qmag/qmag.cpp
showimg/showimg.cpp
biff/biff.cpp

void QWidget::repaint (bool erase=TRUE) `[slot]`

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

This version repaints the entire widget.

void QWidget::repaint (const QRect & r, bool erase=TRUE) `[slot]`

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Examples:
- life/life.cpp
- forever/forever.cpp
- pref/pref.cpp

void QWidget::resize (int w, int h) `[virtual slot]`

Resizes the widget to size *w* by *h* pixels.

A resize event is generated immediately if the widget is visible. If the widget is invisible, the resize event is generated when show() is called.

The size is adjusted if it is outside the minimum or maximum widget size.

This function is virtual, and all other overloaded resize() implementations call it.

Warning: If you call resize() or setGeometry() from resizeEvent(), you may see infinite recursion.

See also: size(), move(), setGeometry(), resizeEvent(), minimumSize() and maximumSize().

Examples:
- tictac/tictac.cpp
- hello/hello.cpp
- xform/xform.cpp
- layout/layout.cpp
- qmag/qmag.cpp
- showimg/showimg.cpp
- biff/biff.cpp

Reimplemented in QSemiModal and QDialog.

void QWidget::resize (const QSize &) `[slot]`

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Examples:
- mainlyMotif/editor.cpp
- table/table.cpp
- life/life.cpp
- drawdemo/drawdemo.cpp
- xform/xform.cpp
- application/application.cpp
- pref/pref.cpp
- qmag/qmag.cpp
- mainlyXt/editor.cpp

void QWidget::resizeEvent (QResizeEvent *) `[virtual protected]`

This event handler can be reimplemented in a subclass to receive widget resize events. When resizeEvent() is called, the widget already has its new geometry.

The old size is accessible through QResizeEvent::oldSize().

The default implementation does nothing.

See also: moveEvent(), event(), resize() and QResizeEvent.

Reimplemented in QGLWidget, QSlider, QSpinBox, QListBox, QHeader, QLCDNumber, QTableView, QListView, QMessageBox, QScrollBar, QLineEdit, QFrame, QProgressDialog, QScrollView, QFileDialog, QMenuBar, QTabDialog and QComboBox.

void QWidget::scroll (int dx, int dy)

Scrolls the contents of the widget *dx* pixels rightwards and *dy* pixels downwards. If *dx/dy* is negative, the scroll direction is leftwards/upwards. Child widgets are moved accordingly.

The areas of the widget that are exposed will be erased and paint events may be generated immediately, or after some further event processing.

Warning: If you call scroll() in a function which may itself be called from the moveEvent() or paintEvent() of a direct child of the widget being scrolled, you may see infinite recursion.

See also: erase() and bitBlt().

void QWidget::setAcceptDrops (bool on)

Announces to the system that this widget *may* be able to accept drop events.

In Qt 1.x, drop event handlers are in QDropSite.

See also: acceptDrops().

void QWidget::setActiveWindow ()

Sets the top-level widget containing this widget to be the active window.

An active window is a top-level window that has the keyboard input focus.

This function performs the same operation as clicking the mouse on the title bar of a top-level window.

See also: isActiveWindow() and topLevelWidget().

void QWidget::setBackgroundColor (const QColor & color) `[virtual]`

This function is deprecated. Use setBackgroundMode() or setPalette(), as they ensure the appropriate clearing color is used when the widget is in the Active, Normal, or Disabled state.

If you want to change the color scheme of a widget, the setPalette() function is better suited. Here is how to set *thatWidget* to use a light green (RGB value 80, 255, 80) as background color, with shades of green used for all the 3D effects:

```
thatWidget->setPalette( QPalette( QColor(80, 255, 80) ) );
```

See also: setPalette(), QApplication::setPalette(), backgroundColor(), setBackgroundPixmap() and setBackgroundMode().

Examples:
    tictac/tictac.cpp

desktop/desktop.cpp
connect/connect.cpp
drawdemo/drawdemo.cpp
xform/xform.cpp
layout/layout.cpp
widgets/widgets.cpp

Reimplemented in QTableView and QComboBox.

void QWidget::setBackgroundMode (BackgroundMode m)

Tells the window system which color to clear this widget to when sending a paint event.

In other words, this color is the color of the widget when paintEvent() is called. To minimize flicker, this should be the most common color in the widget.

The following values are valid:

- `PaletteForeground` - use palette() . foreground()
- `PaletteBackground` - use palette() . background()
- `PaletteLight` - use palette() . light()
- `PaletteMidlight` - use palette() . midlight()
- `PaletteDark` - use palette() . dark()
- `PaletteMid` - use palette() . mid()
- `PaletteText` - use palette() . text()
- `PaletteBase` - use palette() . base()
- `NoBackground` - no color or pixmap is used - the paintEvent() must completely cover the drawing area. This can help avoid flicker.

If setBackgroundPixmap() or setBackgroundColor() is called, the mode will be one of:

- `FixedPixmap` - the pixmap set by setBackgroundPixmap()
- `FixedColor` - the color set by setBackgroundColor()

These values may not be used as parameters to setBackgroundMode().

For most widgets the default (PaletteBackground, normally gray) suffices, but some need to use PaletteBase (the background color for text output, normally white) and a few need other colors.

QListBox, which is "sunken" and uses the base color to contrast with its environment, does this:

```
setBackgroundMode( PaletteBase );
```

If you want to change the color scheme of a widget, the setPalette() function is better suited. Here is how to set *thatWidget* to use a light green (RGB value 80, 255, 80) as background color, with shades of green used for all the 3D effects:

```
thatWidget->setPalette( QPalette( QColor(80, 255, 80) ) );
```

You can also use QApplication::setPalette() if you want to change the color scheme of your entire application, or of all new widgets.

Examples:
table/table.cpp
widgets/widgets.cpp

void QWidget::setBackgroundPixmap (const QPixmap & pixmap) `[virtual]`

Sets the background pixmap of the widget to *pixmap.*

The background pixmap is tiled. Some widgets (e.g. QLineEdit) do not work well with a background pixmap.

See also: backgroundPixmap(), backgroundPixmapChange() and setBackgroundColor().

Examples:
desktop/desktop.cpp

void QWidget::setCRect (const QRect & r) `[protected]`

For internal use only.

void QWidget::setCaption (const char * caption) `[slot]`

Sets the window caption (title).

See also: caption(), setIcon() and setIconText().

Examples:
drawdemo/drawdemo.cpp
movies/main.cpp
application/application.cpp
pref/pref.cpp
progress/progress.cpp
showimg/showimg.cpp
widgets/widgets.cpp

void QWidget::setCursor (const QCursor & cursor) `[virtual]`

Sets the widget cursor shape to *cursor*.

The mouse cursor will assume this shape when it's over this widget. See a list of predefined cursor objects with a range of useful shapes in the QCursor documentation.

An editor widget would for example use an I-beam cursor:

```
setCursor( ibeamCursor );
```

See also: cursor() and QApplication::setOverrideCursor().

Examples:
cursor/cursor.cpp

void QWidget::setEnabled (bool enable) `[virtual slot]`

Enables widget input events if *enable* is TRUE, otherwise disables input events.

An enabled widget receives keyboard and mouse events; a disabled widget does not. Note that an enabled widget receives keyboard events only when it is in focus.

Some widgets display themselves differently when they are disabled. For example a button might draw its label grayed out.

See also: isEnabled(), QKeyEvent and QMouseEvent.

Reimplemented in QLineEdit and QComboBox.

void QWidget::setFRect (const QRect & r) `[protected]`

For internal use only.

void QWidget::setFixedHeight (int h)

Sets both the minimum and maximum heights of the widget to *h* without changing the widths. Provided for convenience.

See also: sizeHint(), minimumSize(), maximumSize(), setFixedSize() and more.

Examples:
widgets/widgets.cpp

void QWidget::setFixedSize (const QSize & s)

Sets both the minimum and maximum sizes of the widget to *s,* thereby preventing it from ever growing or shrinking.

See also: setMaximumSize() and setMinimumSize().

void QWidget::setFixedSize (int w, int h)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Examples:
pref/pref.cpp
widgets/widgets.cpp

void QWidget::setFixedWidth (int w)

Sets both the minimum and maximum width of the widget to *w* without changing the heights. Provided for convenience.

See also: sizeHint(), minimumSize(), maximumSize(), setFixedSize() and more.

void QWidget::setFocus ()

Gives the keyboard input focus to the widget (or its focus proxy).

First, a focus out event is sent to the focus widget (if any) to tell it that it is about to loose the focus. Then a focus in event is sent to this widget to tell it that it just received the focus.

setFocus() gives focus to a widget regardless of its focus policy. However, QWidget::focusWidget() (which determines where Tab/shift-Tab) moves from) is changed only if the widget accepts focus. This can be used to implement "hidden focus"; see focusNextPrevChild() for details.

Warning: If you call setFocus() in a function which may itself be called from focusOutEvent() or focusInEvent(), you may see infinite recursion.

See also: hasFocus(), clearFocus(), focusInEvent(), focusOutEvent(), setFocusPolicy() and QApplication::focusWidget().

void QWidget::setFocusPolicy (FocusPolicy policy)

Enables or disables the keyboard focus for the widget.

The keyboard focus is initially disabled (i.e. *policy* == `QWidget::NoFocus).`

You must enable keyboard focus for a widget if it processes keyboard events. This is normally done from the widget's constructor. For instance, the QLineEdit constructor calls setFocusPolicy(`QWidget::StrongFocus).`

The *policy* can be:

- `QWidget::TabFocus,` the widget accepts focus by tabbing.
- `QWidget::ClickFocus,` the widget accepts focus by clicking.
- `QWidget::StrongFocus,` the widget accepts focus by both tabbing and clicking.
- `QWidget::NoFocus,` the widget does not accept focus

As a special case to support applications not utilizing focus, Top-level widgets that have NoFocus policy will receive focus events and gain keyboard events.

See also: isFocusEnabled(), focusInEvent(), focusOutEvent(), keyPressEvent(), keyReleaseEvent() and isEnabled().

Examples:
table/table.cpp
widgets/widgets.cpp

void QWidget::setFocusProxy (QWidget * w)

Sets this widget's focus proxy to *w.*

Some widgets, such as QComboBox, can "have focus," but create a child widget to actually handle the focus. QComboBox, for example, creates a QLineEdit.

setFocusProxy() sets the widget which will actually get focus when "this widget" gets it. If there is a focus proxy, focusPolicy(), setFocusPolicy(), setFocus() and hasFocus() all operate on the focus proxy.

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

Sets the font for the widget.

The fontInfo() function reports the actual font that is being used by the widget.

This code fragment sets a 12 point helvetica bold font:

```
QFont f("Helvetica", 12, QFont::Bold);
setFont( f );
```

If *fontPropagation()* is `AllChildren` or `SameFont,` setFont() calls setFont() for children of the object, or those with whom the object shares the font, respectively. The default for QWidget is *NoChildren,* so setFont() will not change the children's fonts.

See also: font(), fontChange(), fontInfo(), fontMetrics() and setFontPropagation().

Examples:
grapher/grapher.cpp
xform/xform.cpp
widgets/widgets.cpp

Reimplemented in QMultiLineEdit, QPopupMenu, QListView, QLineEdit, QTabDialog, QComboBox and QListBox.

void QWidget::setFontPropagation (PropagationMode m)

Sets the font propagation mode to *m.*

if *m* is `NoChildren` (the default), setFont() does not change any children's fonts. If it is `SameFont,` setFont() changes the font of the children that have the exact same font as this widget (see QFont::isCopyOf() for details). If it is `AllChildren,` setFont() changes the font of all children.

See also: fontPropagation(), setFont() and setPalettePropagation().

void QWidget::setGeometry (int x, int y, int w, int h) `[virtual slot]`

Sets the widget geometry to *w* by *h,* positioned at *x,y* in its parent widget.

A resize event and a move event are generated immediately if the widget is visible. If the widget is invisible, the events are generated when show() is called.

The size is adjusted if it is outside the minimum or maximum widget size.

This function is virtual, and all other overloaded setGeometry() implementations call it.

Warning: If you call setGeometry() from resizeEvent() or moveEvent(), you may see infinite recursion.

See also: geometry(), move(), resize(), moveEvent(), resizeEvent(), minimumSize() and maximumSize().

Examples:
tictac/tictac.cpp
mainlyQt/editor.cpp
drawdemo/drawdemo.cpp
xform/xform.cpp
cursor/cursor.cpp
qmag/qmag.cpp

Reimplemented in QPushButton, QSemiModal, QXtWidget and QDialog.

void QWidget::setGeometry (const QRect &) `[slot]`

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

void QWidget::setIcon (const QPixmap & pixmap) `[slot]`

Sets the window icon pixmap.

See also: icon(), setIconText() and setCaption().

void QWidget::setIconText (const char * iconText) `[slot]`

Sets the text of the window's icon to *iconText.*

See also: iconText(), setIcon() and setCaption().

void QWidget::setMask (QBitmap bitmap)

Causes only the pixels of the widget for which *bitmap* has a corresponding 1 bit to be visible. If the region includes pixels outside the rect() of the widget, window system controls in that area may or may not be visible, depending on the platform.

Note that this effect can be slow if the region is particularly complex.

See also: setMask(const, QRegion&) and clearMask().

void QWidget::setMask (const QRegion & region)

Causes only the parts of the widget which overlap *region* to be visible. If the region includes pixels outside the rect() of the widget, window system controls in that area may or may not be visible, depending on the platform.

Note that this effect can be slow if the region is particularly complex.

See also: setMask(QBitmap) and clearMask().

void QWidget::setMaximumHeight (int h)

Sets the maximum height of the widget to *h* without changing the width. Provided for convenience.

See also: sizeHint(), minimumSize(), maximumSize(), setFixedSize() and more.

void QWidget::setMaximumSize (int maxw, int maxh)

Sets the maximum size of the widget to *w* by *h* pixels.

The widget cannot be resized to a larger size than the maximum widget size. The widget's size is forced to the maximum size if the current size is greater.

See also: maximumSize(), setMinimumSize(), setSizeIncrement(), resize() and size().

Examples:
life/life.cpp
biff/biff.cpp

void QWidget::setMaximumSize (const QSize & size)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

void QWidget::setMaximumWidth (int w)

Sets the maximum width of the widget to *w* without changing the height. Provided for convenience.

See also: sizeHint(), minimumSize(), maximumSize(), setFixedSize() and more.

void QWidget::setMinimumHeight (int h)

Sets the minimum height of the widget to *h* without changing the width. Provided for convenience.

See also: sizeHint(), minimumSize(), maximumSize(), setFixedSize() and more.

Examples:
layout/layout.cpp
widgets/widgets.cpp

void QWidget::setMinimumSize (int minw, int minh)

Sets the minimum size of the widget to *w* by *h* pixels.

The widget cannot be resized to a smaller size than the minimum widget size. The widget's size is forced to the minimum size if the current size is smaller.

See also: minimumSize(), setMaximumSize(), setSizeIncrement(), resize() and size().

Examples:
life/life.cpp
tooltip/tooltip.cpp
menu/menu.cpp

pref/pref.cpp
layout/layout.cpp
qmag/qmag.cpp
biff/biff.cpp
widgets/widgets.cpp

void QWidget::setMinimumSize (const QSize & size)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

void QWidget::setMinimumWidth (int w)

Sets the minimum width of the widget to *w* without changing the height. Provided for convenience.

See also: sizeHint(), minimumSize(), maximumSize(), setFixedSize() and more.

void QWidget::setMouseTracking (bool enable) `[slot]`

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

If mouse tracking is disabled (default), this widget only receives mouse move events when at least one mouse button is pressed down while the mouse is being moved.

If mouse tracking is enabled, this widget receives mouse move events even if no buttons are pressed down.

See also: hasMouseTracking(), mouseMoveEvent() and QApplication::setGlobalMouseTracking().

Examples:
qmag/qmag.cpp
showimg/showimg.cpp

void QWidget::setPalette (const QPalette & p) `[virtual]`

Sets the widget palette to *p*. The widget background color is set to `colorGroup().background()`.

If *palettePropagation()* is `AllChildren` or `SamePalette`, setPalette() calls setPalette() for children of the object, or those with whom the object shares the palette, respectively. The default for QWidget is *NoChildren,* so setPalette() will not change the children's palettes.

See also: QApplication::setPalette(), palette(), paletteChange(), colorGroup(), setBackgroundColor() and setPalettePropagation().

Reimplemented in QTableView, QListView, QScrollBar, QLineEdit, QComboBox and QSlider.

void QWidget::setPalettePropagation (PropagationMode m)

Sets the palette propagation mode to *m.*

if *m* is `NoChildren` (the default), setPalette() does not change any children's palettes. If it is `SamePalette`, setPalette() changes the palette of the children that have the exact same palette as this widget (see QPalette::isCopyOf() for details). If it is `AllChildren`, setPalette() changes the palette of all children.

See also: palettePropagation(), setPalette() and setFontPropagation().

void QWidget::setSizeGrip (bool sizegrip) `[protected]`

Informs the underlying window system that this widget is a size grip (if sizegrip is TRUE). An example is the nifty decoration in the bottom right corner of a QStatusBar.

This function does yet nothing under Windows. Under X11, the window manager has to support the QT_SIZEGRIP protocol.

void QWidget::setSizeIncrement (int w, int h)

Sets the size increment of the widget. When the user resizes the window, the size will move in steps of *w* pixels horizontally and *h* pixels vertically.

Note that while you can set the size increment for all widgets, it has no effect except for top-level widgets.

Warning: The size increment has no effect under Windows, and may be disregarded by the window manager on X.

See also: sizeIncrement(), setMinimumSize(), setMaximumSize(), resize() and size().

void QWidget::setSizeIncrement (const QSize &)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Examples:
life/life.cpp

void QWidget::setStyle (GUIStyle style) `[virtual]`

Sets the GUI style for this widget. The valid values are listed in qwindowdefs.h.

See also: style(), styleChange() and QApplication::setStyle().

Examples:
grapher/grapher.cpp

Reimplemented in QListView, QMessageBox and QComboBox.

void QWidget::setTabOrder (QWidget * first, QWidget * second) `[static]`

Moves the *second* widget around the ring of focus widgets so that keyboard focus moves from *first* widget to *second* widget when Tab is pressed.

Note that since the tab order of the *second* widget is changed, you should order a chain like this:

```
setTabOrder(a, b ); // a to b
setTabOrder(b, c ); // a to b to c
setTabOrder(c, d ); // a to b to c to d
```

not like this:

```
setTabOrder(c, d); // c to d
setTabOrder(a, b); // a to b AND c to d
setTabOrder(b, c); // a to b to c, but not c to d
```

If either *first* or *second* has a focus proxy, setTabOrder() substitutes its/their proxies.

See also: setFocusPolicy() and setFocusProxy().

void QWidget::setUpdatesEnabled (bool enable) `[slot]`

Enables widget updates if *enable* is TRUE, or disables widget updates if *enable* is FALSE.

Calling update() and repaint() has no effect if updates are disabled. Paint events from the window system are processed as normally even if updates are disabled.

This function is normally used to disable updates for a short period of time, for instance to avoid screen flicker during large changes.

Example:

```
setUpdatesEnabled( FALSE );
bigVisualChanges();
setUpdatesEnabled( TRUE );
repaint();
```

See also: isUpdatesEnabled(), update(), repaint() and paintEvent().

void QWidget::setWFlags (WFlags f) `[protected]`

For internal use only.

void QWidget::show () `[virtual slot]`

Shows the widget and its child widgets.

If its size or position has changed, Qt guarantees that a widget gets move and resize events just before the widget is shown.

See also: hide(), iconify() and isVisible().

Examples:

forever/forever.cpp
connect/connect.cpp
mainlyQt/editor.cpp
drawdemo/drawdemo.cpp
movies/main.cpp
picture/picture.cpp
xform/xform.cpp
menu/menu.cpp
pref/pref.cpp
progress/progress.cpp
cursor/cursor.cpp
layout/layout.cpp
qmag/qmag.cpp
showimg/showimg.cpp
widgets/widgets.cpp

Reimplemented in QTableView, QTabBar, QToolBar, QSemiModal, QListView, QPopupMenu, QDialog, QMainWindow, QScrollView, QMenuBar, QTabDialog and QWidgetStack.

QSize QWidget::size () const

Returns the size of the widget, excluding the window frame.

See also: geometry(), width() and height().

QSize QWidget::sizeHint () const `[virtual]`

Returns a recommended size for the widget, or an invalid size if no size is recommended.

The default implementation returns an invalid size.

See also: QSize::isValid(), resize() and setMinimumSize().

Examples:
widgets/widgets.cpp

Reimplemented in QToolButton, QPushButton, QSpinBox, QListBox, QTabBar, QHeader, QLCDNumber, QListView, QScrollBar, QLineEdit, QCheckBox, QProgressDialog, QRadioButton, QLabel, QComboBox, QSlider and QProgressBar.

QSize QWidget::sizeIncrement () const

Returns the widget size increment.

See also: setSizeIncrement(), minimumSize() and maximumSize().

GUIStyle QWidget::style () const

Returns the GUI style for this widget.

See also: setStyle() and QApplication::style().

void QWidget::styleChange (GUIStyle oldStyle) `[virtual protected]`

This virtual function is called from setStyle(). *oldStyle* is the previous style; you can get the new style from style().

Reimplement this function if your widget needs to know when its GUI style changes. You will almost certainly need to update the widget using either repaint(TRUE) or update().

The default implementation calls update().

See also: setStyle(), style(), repaint() and update().

Reimplemented in QSpinBox, QProgressDialog and QTabDialog.

bool QWidget::testWFlags (WFlags n) const

Returns non-zero if any of the widget flags in *n* are set. The widget flags are listed in qwindowdefs.h, and are strictly for internal use.

QWidget * QWidget::topLevelWidget () const

Returns the top-level widget for this widget.

A top-level widget is an overlapping widget. It usually has no parent. Modal dialog widgets are the only top-level widgets that can have parent widgets.

See also: isTopLevel().

void QWidget::update () `[slot]`

Updates the widget unless updates are disabled or the widget is hidden.

Updating the widget will erase the widget contents and generate a paint event from the window system. The paint event is processed after the program has returned to the main event loop.

See also: repaint(), paintEvent(), setUpdatesEnabled() and erase().

Examples:
desktop/desktop.cpp
connect/connect.cpp
picture/picture.cpp
aclock/aclock.cpp
showimg/showimg.cpp

void QWidget::update (int x, int y, int w, int h) `[slot]`

Updates a rectangle *(x, y, w, h)* inside the widget unless updates are disabled or the widget is hidden.

Updating the widget erases the widget area *(x,y,w,h),* which in turn generates a paint event from the window system. The paint event is processed after the program has returned to the main event loop.

If *w* is negative, it is replaced with `width() - x`. If *h* is negative, it is replaced width `height() - y`.

See also: repaint(), paintEvent(), setUpdatesEnabled() and erase().

Examples:
grapher/grapher.cpp
drawdemo/drawdemo.cpp

void QWidget::update (const QRect & r) `[slot]`

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

int QWidget::width () const

Returns the width of the widget, excluding the window frame.

See also: geometry(), height() and size().

Examples:
grapher/grapher.cpp
tooltip/tooltip.cpp
drawdemo/drawdemo.cpp
xform/xform.cpp
aclock/aclock.cpp
menu/menu.cpp
pref/pref.cpp
qmag/qmag.cpp
showimg/showimg.cpp

WId QWidget::winId () const

Returns the window system identifier of the widget.

Portable in principle, but if you use it you are probably about to do something non-portable. Be careful.

See also: find().

QWidgetMapper * QWidget::wmapper () `[static]`

For internal use only.

int QWidget::x () const

Returns the x coordinate of the widget, relative to its parent widget and including the window frame.

See also: frameGeometry(), y() and pos().

int QWidget::y () const

Returns the y coordinate of the widget, relative to its parent widget and including the window frame.

See also: frameGeometry(), x() and pos().

Examples:
drawdemo/drawdemo.cpp

186.2.1. URL

Generated from http://www.troll.no/qt/qwidget.html.

187. QWidgetStack Class Reference

`#include <qwidgetstack.h>`

Inherits QFrame.

187.1. Detailed Description

The QWidgetStack class provides a stack of widgets, where the user can see only the top widget.

This is often used e.g. in tab and wizard dialogs.

The application programmer can move any widget to the top of the stack at any time using the slot raiseWidget(), and add or remove widgets using addWidget() and removeWidget().

visibleWidget() is the *get* equivalent of raiseWidget(); it returns a pointer ot the widget that is currently on the top of the stack.

QWidgetStack also provides the ability to manipulate widgets through application-specfied integer IDs, and to translate from widget pointers to IDs using id() and from IDs to widget pointers using widget(). These numeric IDs have and unique (per QWidgetStack, not globally) and cannot be -1, but apart from that QWidgetStack does not attach any meaning to them.

The default widget stack is frame-less and propagates its font and palette to all its children, but you can use the usual QFrame functions (like setFrameStyle()) to add a frame, and use setFontPropagation() and setPalettePropagation() to change the propagation style.

Finally, QWidgetStack provides a signal, aboutToShow(), which is emitted just before a managed widget is shown.

See also: QTabDialog, QTabBar and QFrame.

187.2. Member Function Documentation

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

Constructs an empty widget stack.

QWidgetStack::˜QWidgetStack ()

Destroys the object and frees any allocated resources.

void QWidgetStack::aboutToShow (QWidget *) `[signal]`

This signal is emitted just before a managed widget is shown. The argument is a pointer to the widget.

void QWidgetStack::aboutToShow (int) `[signal]`

This signal is emitted just before a managed widget is shown, if that managed widget has a non-zero ID. The argument is the numeric ID of the widget.

void QWidgetStack::addWidget (QWidget * w, int id)

Adds *w* to this stack of widgets, with id *id.*

If *w* is not a child of `this,` QWidgetStack moves it using recreate().

bool QWidgetStack::event (QEvent * e) `[virtual]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QObject.

void QWidgetStack::frameChanged () `[virtual protected]`

Reimpelemented in order to set the children's geometries appropriately.

Reimplemented from QFrame.

int QWidgetStack::id (QWidget * widget) const

Returns the ID of the *widget.* If *widget* is 0 or is not being managed by this widget stack, this function returns -1.

See also: widget() and addWidget().

void QWidgetStack::raiseWidget (QWidget * w) `[slot]`

Raises *w* to the top of the widget stack.

void QWidgetStack::raiseWidget (int id) `[slot]`

Raises *id* to the top of the widget stack.

See also: visibleWidget().

void QWidgetStack::removeWidget (QWidget * w)

Removes *w* from this stack of widgets. Does not delete *w.* If *w* is the currently visible widget, no other widget is substituted.

See also: visibleWidget() and raiseWidget().

void QWidgetStack::setChildGeometries () `[protected]`

Fix up the children's geometries.

void QWidgetStack::show () `[virtual]`

Reimplemented in order to set the children's geometries appropriately.

Reimplemented from QWidget.

QWidget * QWidgetStack::visibleWidget () const

Returns a pointer to the currently visible widget (the one on the top of the stack), of 0 if nothing is currently being shown.

See also: aboutToShow(), id() and raiseWidget().

QWidget * QWidgetStack::widget (int id) const

Returns a pointer to the widget with ID *id.* If this widget stack does not manage a widget with ID *id,* this function return 0.

See also: id() and addWidget().

187.2.1. URL

Generated from http://www.troll.no/qt/qwidgetstack.html.

188. QWindow Class Reference

`#include <qwindow.h>`

Inherits QWidget.

188.1. Detailed Description

The QWindow class is reserved for future extensions.

This class is not yet finished. It will contain intelligent handling of menus etc. in a future Qt release.

188.2. Member Function Documentation

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

Constructs a window named *name,* which will be a child widget of *parent.*

The widget flags *f* should normally be set to zero unless you know what you are doing.

These arguments are sent to the QWidget constructor.

QWindow::~QWindow ()

Destroys the window.

188.2.1. URL

Generated from http://www.troll.no/qt/qwindow.html.

189. QWMatrix Class Reference

```
#include <qwmatrix.h>
```

189.1. Detailed Description

The QWMatrix class specifies 2D transformations of a coordinate system.

The standard coordinate system of a paint device has the origin located at the top left position. X values increase to the right, and Y values increase downwards.

This coordinate system is default for the QPainter, which renders graphics in a paint device. A user-defined coordinate system can be specified by setting a QWMatrix for the painter.

Example:

```
MyWidget::paintEvent( QPaintEvent * )
{
  QPainter p;                        // our painter
  QWMatrix m;                        // our transformation matrix
  m.rotate( 22.5 );                  // rotated coordinate system
  p.begin( this );                   // start painting
  p.setWorldMatrix( m );             // use rotated coordinate system
  p.drawText( 30,20, "detator" );    // draw rotated text at 30,20
  p.end();                           // painting done
}
```

A matrix specifies how to translate, scale, shear or rotate the graphics, and the actual transformation is performed by the drawing routines in QPainter and by QPixmap::xForm().

The QWMatrix class contains a 3*3 matrix of the form:

```
m11  m12  0
m21  m22  0
dx   dy   1
```

A matrix transforms a point in the plane to another point:

```
x' = m11*x + m21*y + dx
y' = m22*y + m12*x + dy
```

The point *(x,y)* is the original point, and *(x',y')* is the transformed point. *(x',y')* can be transformed back to *(x,y)* by performing the same operation on the inverted matrix.

The elements *dx* and *dy* specify horisontal and vertical translation. The elements *m11* and *m22* specify horisontal and vertical scaling. The elements *m12* and *m21* specify horisontal and vertical shearing.

The identity matrix has *m11* and *m22* set to 1, all others set to 0. This matrix maps a point to itself.

Translation is the simplest transformation. Setting *dx* and *dy* will move the coordinate system *dx* units along the X axis and *dy* units along the Y axis.

Scaling can be done by setting *m11* and *m22.* For example, setting *m11* to 2 and *m22* to 1.5 will double the height and increase the width by 50%.

Shearing is controlled by *m12* and *m21.* Setting these elements to values different from zero will twist the coordinate system.

Rotation is achieved by carefully setting both the shearing factors and the scaling factors. The QWMatrix has a function that sets rotation directly.

QWMatrix lets you combine transformations like this:

```
QWMatrix m;                                // identity matrix
m.translate(10, -20);                      // first translate (10,-20)
m.rotate(25);                              // then rotate 25 degrees
m.scale(1.2, 0.7);                         // finally scale it
```

The same example, but using basic matrix operations:

```
float a    = pi/180 * 25;                // convert 25 to radians
float sina = sin(a);
float cosa = cos(a);
QWMatrix m1(0, 0, 0, 0, 10, -20);        // translation matrix
QWMatrix m2( cosa, sina,                 // rotation matrix
            -sina, cosa, 0, 0 );
QWMatrix m3(1.2, 0, 0, 0.7, 0, 0);       // scaling matrix
QWMatrix m;
m = m3 * m2 * m1;                        // combine all transformations
```

QPainter has functions that translate, scale, shear and rotate the coordinate system without using a QWMatrix. These functions are very convenient, however, if you want to perform more than a single transform operation, it is more efficient to build a QWMatrix and call QPainter::setWorldMatrix().

See also: QPainter::setWorldMatrix() and QPixmap::xForm().

Examples:

qtimage/qtimage.cpp
desktop/desktop.cpp
drawdemo/drawdemo.cpp
movies/main.cpp
xform/xform.cpp
aclock/aclock.cpp
qmag/qmag.cpp
showimg/showimg.cpp

189.2. Member Function Documentation

QWMatrix::QWMatrix ()

Constructs an identity matrix. All elements are set to zero, except *m11* and *m22* (scaling) which are set to 1.

QWMatrix::QWMatrix (float m11, float m12, float m21, float m22, float dx, float dy)

Constructs a matrix with the specified elements.

float QWMatrix::dx () const

Returns the horizontal translation.

float QWMatrix::dy () const

Returns the vertical translation.

QWMatrix QWMatrix::invert (bool * invertible = 0) const

Returns the inverted matrix.

If the matrix is singular (not invertible), then the identity matrix is returned.

If **invertible* is not null, then the value of **invertible* will be set to TRUE or FALSE to tell if the matrix is invertible or not.

float QWMatrix::m11 () const

Returns the X scaling factor.

float QWMatrix::m12 () const

Returns the vertical shearing factor.

float QWMatrix::m21 () const

Returns the horizontal shearing factor.

float QWMatrix::m22 () const

Returns the Y scaling factor.

QPoint QWMatrix::map (const QPoint & p) const

Returns the transformed *p.*

QPointArray QWMatrix::map (const QPointArray & a) const

Returns the point array *a* transformed by calling map for each point.

QRect QWMatrix::map (const QRect & r) const

Returns the tranformed rectangle *r.*

If rotation or shearing has been specified, then the bounding rectangle will be returned.

void QWMatrix::map (float x, float y, float * tx, float * ty) const

Transforms *(x,y)* to *(*tx,*ty),* using the formulae:

```
*tx = m11*x + m21*y + dx
*ty = m22*y + m12*x + dy
```

void QWMatrix::map (int x, int y, int * tx, int * ty) const

Transforms *(x,y)* to *(*tx,*ty),* using the formulae:

```
*tx = m11*x + m21*y + dx  --  (rounded to the nearest integer)
*ty = m22*y + m12*x + dy  --  (rounded to the nearest integer)
```

Examples:
xform/xform.cpp

bool QWMatrix::operator!= (const QWMatrix & m) const

Returns TRUE if this matrix is not equal to *m.*

QWMatrix & QWMatrix::operator*= (const QWMatrix & m)

Returns the result of multiplying this matrix with *m*.

bool QWMatrix::operator== (const QWMatrix & m) const

Returns TRUE if this matrix is equal to *m*.

void QWMatrix::reset ()

Resets the matrix to an identity matrix.

All elements are set to zero, except *m11* and *m22* (scaling) that are set to 1.

QWMatrix & QWMatrix::rotate (float a)

Rotates the coordinate system *a* degrees counterclockwise.

Returns a reference to the matrix.

See also: translate(), scale() and shear().

Examples:
- desktop/desktop.cpp
- drawdemo/drawdemo.cpp
- xform/xform.cpp
- aclock/aclock.cpp

QWMatrix & QWMatrix::scale (float sx, float sy)

Scales the coordinate system unit by *sx* horizontally and *sy* vertically.

Returns a reference to the matrix.

See also: translate(), shear() and rotate().

Examples:
- qtimage/qtimage.cpp
- movies/main.cpp
- xform/xform.cpp
- aclock/aclock.cpp
- showimg/showimg.cpp

void QWMatrix::setMatrix (float m11, float m12, float m21, float m22, float dx, float dy)

Sets the matrix elements to the specified values.

QWMatrix & QWMatrix::shear (float sh, float sv)

Shears the coordinate system by *sh* horizontally and *sv* vertically.

Returns a reference to the matrix.

See also: translate(), scale() and rotate().

Examples:
- drawdemo/drawdemo.cpp
- xform/xform.cpp

QWMatrix & QWMatrix::translate (float dx, float dy)

Moves the coordinate system *dx* along the X-axis and *dy* along the Y-axis.

Returns a reference to the matrix.

See also: scale(), shear() and rotate().

Examples:
drawdemo/drawdemo.cpp
xform/xform.cpp
aclock/aclock.cpp

189.3. Related Functions

QWMatrix operator* (const QWMatrix & m1, const QWMatrix & m2)

Returns the product *m1 * m2*.

Remember that matrix multiplication is not commutative, thus a*b != b*a.

QDataStream & operator<< (QDataStream & s, const QWMatrix & m)

Writes a matrix to the stream and returns a reference to the stream.

Serialization format: m11, m12, m21, m22, dx and dy are serialized as `float` in the listed order.

QDataStream & operator>> (QDataStream & s, QWMatrix & m)

Reads a matrix from the stream and returns a reference to the stream.

189.3.1. URL

Generated from http://www.troll.no/qt/qwmatrix.html.

190. QXtApplication Class Reference

`#include <qxt.h>`

Inherits QApplication.

190.1. Detailed Description

This class is defined in the **Qt Xt/Motif Extension**, which can be found in the `qt/extensions` directory. It is not included in the main Qt API.

Allows mixing of Xt/Motif and Qt widgets.

The QXtApplication and QXtWidget classes allow old Xt or Motif widgets to be used in new Qt applications. They also allow Qt widgets to be used in primarily Xt/Motif applications. The facility is intended to aid migration from Xt/Motif to the more comfortable Qt system.

Examples:
mainlyMotif/editor.cpp
mainlyQt/editor.cpp
mainlyXt/editor.cpp

190.2. Member Function Documentation

QXtApplication::QXtApplication (Display * display)

Constructs a QApplication from the *display* of an already-initialized Xt application.

Use this constructor when introducing Qt widgets into an existing Xt/Motif application.

QXtApplication::QXtApplication (int & argc, char ** argv, const char * appclass=0, XrmOptionDescRec * options=0, int num_options=0, const char ** resources=0)

Constructs a QApplication and initializes the Xt toolkit. The *appclass, options, num_options,* and *resources* arguments are passed on to XtAppSetFallbackResources and XtDisplayInitialize.

Use this constructor when writing a new Qt application which needs to use some existing Xt/Motif widgets.

QXtApplication::~QXtApplication ()

Destructs the application. Does not close the Xt toolkit.

bool QXtApplication::x11EventFilter (XEvent * ev)

Reimplemented to pass client messages to Xt.

190.2.1. URL

Generated from http://www.troll.no/qt/qxtapplication.html.

191. QXtWidget Class Reference

`#include <qxt.h>`

Inherits QWidget.

191.1. Detailed Description

This class is defined in the **Qt Xt/Motif Extension**, which can be found in the `qt/extensions` directory. It is not included in the main Qt API.

Allows mixing of Xt/Motif and Qt widgets.

QXtWidget acts as a bridge between Xt and Qt. For utilizing old Xt widgets, it can be a QWidget based on a Xt widget class. For including Qt widgets in an existing Xt/Motif application, it can be a special Xt widget class that is a QWidget. See the constructors for the different behaviors.

Examples:
mainlyMotif/editor.cpp
mainlyQt/editor.cpp
mainlyXt/editor.cpp

191.2. Member Function Documentation

QXtWidget::QXtWidget (const char * name, Widget parent, bool managed=FALSE)

Constructs a QXtWidget of the special Xt widget class known as "QWidget" to the resource manager.

Use this constructor to utilize Qt widgets in an Xt/Motif application. The QXtWidget is a QWidget, so you can create subwidgets, layouts, etc. using Qt functionality.

QXtWidget::QXtWidget (const char * name, WidgetClass widget_class, QXtWidget * parent, ArgList args=0, Cardinal num_args=0, bool managed=FALSE)

Constructs a QXtWidget of the given *widget_class.*

Use this constructor to utilize Xt or Motif widgets in a Qt application. The QXtWidget looks and behaves like the Xt class, but can be used like any QWidget.

Note that the parent must be a QXtWidget (possibly NULL). This is necessary since all Xt widgets must have Xt widgets as ancestors up to the top-level widget. *WWA: This restriction may be avoidable by reimplementing the low-level Qt window creation/destruction functions.*

QXtWidget::~QXtWidget ()

Destructs the QXtWidget.

void QXtWidget::leaveEvent (QEvent * ev) `[virtual protected]`

Reimplemented to produce the Xt effect of losing focus when the mouse goes out of the widget. *This may be changed.*

Reimplemented from QWidget.

void QXtWidget::setGeometry (const QRect & r)

Reimplemented to pass the new geometry to Xt via XtSetValues().

void QXtWidget::setGeometry (int x, int y, int w, int h) `[virtual]`

Reimplemented to pass the new geometry to Xt via XtSetValues().

Examples:
mainlyQt/editor.cpp

Reimplemented from QWidget.

bool QXtWidget::x11Event (XEvent * ev) `[protected]`

Reimplemented to pass events to Xt.

Widget QXtWidget::xtWidget () const

Returns the Xt widget equivalent for the Qt widget.

Examples:
mainlyMotif/editor.cpp
mainlyXt/editor.cpp

191.2.1. URL

Generated from http://www.troll.no/qt/qxtwidget.html.