Troll Tech AS

# Qt 2.0.1

## Tutorial and Reference

August 4, 1999

http://www.troll.no/

sales@troll.no / qt-bugs@troll.no

fax: +47 22806380

# 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!", 0 );
    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 QWidget. 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 QPixmap.

```
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!", 0 );
```

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 specifies 0 for the parent window which the button should be inside).

```
        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", 0 );
    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", 0 );
```

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. Still we pass 0 as the parent, since the button is a top-level window.

```
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 (using QApplication::setFont()) 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>
#include <qvbox.h>
int main( int argc, char **argv )
{
    QApplication a( argc, argv );
    QVBox box;
    box.resize( 200, 120 );
    QPushButton quit( "Quit", &box );
    quit.setFont( QFont( "Times", 18, QFont::Bold ) );
    QObject::connect( &quit, SIGNAL(clicked()), &a, SLOT(quit()) );
    a.setMainWidget( &box );
    box.show();
    return a.exec();
}
```

### 3.1. Line by Line Walk-Through

```
#include <qvbox.h>
```

We add an include of qvbox.h, to get the layout class we'll use.

```
QVBox box;
```

Here we simply create a vertical box container. The QVBox arranges its child widgets in a vertical row, one above the other, handing out space according to each child's QWidget::sizePolicy().

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

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

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

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.

The mother widget, the QVBox, automatically adds the child, centered in its box. Since nothing else is added, the button gets all the space the mother has.

```
box.show();
```

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

### 3.2. Behavior

The button no longer fills the entire widget. Instead, it gets a "natural" size. This is because there is now a new top-level widget, which uses layout management to set a good size and position for the button.

### 3.3. Excercises

Try resizing the window. How does the button change? What is the button's size change policy? What happens if you try to make the window *really* small?

### 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, this 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 window's title bar or in the button. It is a name associated with a widget to make it possible to look up this widget later, and there is also a handy debugging function that will list a complete widget hierarchy.

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

The implementation of the constructor starts here. Like most 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 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.

#### 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 <qslider.h>
#include <qlcdnumber.h>
#include <qfont.h>
#include <qvbox.h>
class MyWidget : public QVBox
{
public:
    MyWidget( QWidget *parent=0, const char *name=0 );
};
MyWidget::MyWidget( QWidget *parent, const char *name )
        : QVBox( parent, name )
{
    QPushButton *quit = new QPushButton( "Quit", this, "quit" );
```

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

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

    QLCDNumber *lcd  = new QLCDNumber( 2, this, "lcd" );

    QSlider * slider = new QSlider( Horizontal, this, "slider" );

    slider->setRange( 0, 99 );

    slider->setValue( 0 );

    connect( slider, SIGNAL(valueChanged(int)), lcd, SLOT(display(int)) );

}

int main( int argc, char **argv )

{

    QApplication a( argc, argv );

    MyWidget w;

    a.setMainWidget( &w );

    w.show();

    return a.exec();

}
```

### 5.1. Line by Line Walk-Through

```
#include <qapplication.h>
#include <qpushbutton.h>
#include <qslider.h>
#include <qlcdnumber.h>
#include <qfont.h>
#include <qvbox.h>
```

Three new include files here. qslider.h and qlcdnumber.h are there because we use two new widgets; QSlider and QLCDNumber. qvbox.h is there because we use Qt's automatic layout support.

```
class MyWidget : public QVBox

{

public:

    MyWidget( QWidget *parent=0, const char *name=0 );

};

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

        : QVBox( parent, name )

{
```

Nothing new compared to the previous chapter.

```
QLCDNumber *lcd  = new QLCDNumber( 2, this, "lcd" );
```

`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".

```
QSlider * slider = new QSlider( Horizontal, this, "slider" );
slider->setRange( 0, 99 );
slider->setValue( 0 );
```

QSlider is a classical slider; a widget where the user can drag something to adjust an integer value in a range. Here, we create a horizontal one, set its range to 0-99 (inclusive, see the QSlider::setRange() documentation) and its initial value to 0.

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

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

Whenever the slider'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.

### 5.2. Behavior

The LCD number reflects everything you do to the slider, and the widget handles resizing well. Notice how the LCD number widget changes in size when the window is resized (because it can), but the others stay about the same (because they would look stupid otherwise).

### 5.3. Excercises

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

You can also change the slider's range.

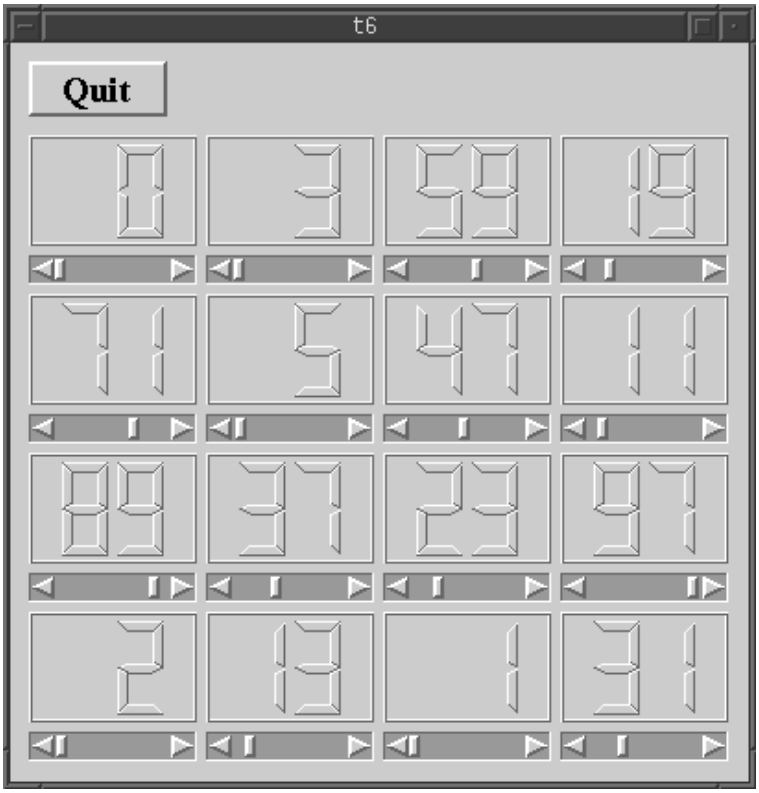
Perhaps it would have been better to use QSpinBox than a slider?

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 <qslider.h>
#include <qlcdnumber.h>
#include <qfont.h>
#include <qvbox.h>
#include <qgrid.h>
```

```
class LCDRange : public QVBox
{
public:
    LCDRange( QWidget *parent=0, const char *name=0 );
};
LCDRange::LCDRange( QWidget *parent, const char *name )
        : QVBox( parent, name )
{
    QLCDNumber *lcd  = new QLCDNumber( 2, this, "lcd"  );
    QSlider * slider = new QSlider( Horizontal, this, "slider" );
    slider->setRange( 0, 99 );
    slider->setValue( 0 );
    connect( slider, SIGNAL(valueChanged(int)), lcd, SLOT(display(int)) );
}
class MyWidget : public QVBox
{
public:
    MyWidget( QWidget *parent=0, const char *name=0 );
};
MyWidget::MyWidget( QWidget *parent, const char *name )
        : QVBox( parent, name )
{
    QPushButton *quit = new QPushButton( "Quit", this, "quit" );
    quit->setFont( QFont( "Times", 18, QFont::Bold ) );
    connect( quit, SIGNAL(clicked()), qApp, SLOT(quit()) );
    QGrid *grid = new QGrid( 4, this );
    for( int c = 0 ; c < 4 ; c++ )
        for( int r = 0 ; r < 4 ; r++ )
            (void)new LCDRange( grid );
}
int main( int argc, char **argv )
{
    QApplication a( argc, argv );
```

```
    MyWidget w;

    a.setMainWidget( &w );

    w.show();

    return a.exec();

}
```

### 6.1. Line by Line Walk-Through

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

The LCDRange widget is a widget without any API. It just has a constructor. This sort of widget is not very useful, so later on we'll add some API.

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

        : QVBox( parent, name )

{

    QLCDNumber *lcd  = new QLCDNumber( 2, this, "lcd"  );

    QSlider * slider = new QSlider( Horizontal, this, "slider" );

    slider->setRange( 0, 99 );

    slider->setValue( 0 );

    connect( slider, SIGNAL(valueChanged(int)), lcd, SLOT(display(int)) );

}
```

This is lifted straight from the MyWidget constructor in chapter five. The only differences are that the button is left out and the class renamed.

```
class MyWidget : public QVBox

{

public:

    MyWidget( QWidget *parent=0, const char *name=0 );

};
```

MyWidget, too, contains no API except a constructor.

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

        : QVBox( parent, name )
```

```
{
    QPushButton *quit = new QPushButton( "Quit", this, "quit" );
    quit->setFont( QFont( "Times", 18, QFont::Bold ) );
    connect( quit, SIGNAL(clicked()), qApp, SLOT(quit()) );
```

The push button that used to be in what's now LCDRange has been separated, so that we can have one "Quit" button and many LCDRange objects.

```
    QGrid *grid = new QGrid( 4, this );
```

We create a QGrid object with four columns. The QGRid widget automatically arranges its children in rows and columns; you can specify the number of rows or of columns, and QGrid will discover its new children and fit them into the grid.

```
    for( int c = 0 ; c < 4 ; c++ )
        for( int r = 0 ; r < 4 ; r++ )
            (void)new LCDRange( grid );
```

Four columns, for rows.

We create 4*4 LCDRanges, all of which children of the grid object. The QGrid widget will arrange them.

```
}
```

That's all.

### 6.2. Behavior

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

### 6.3. Excercises

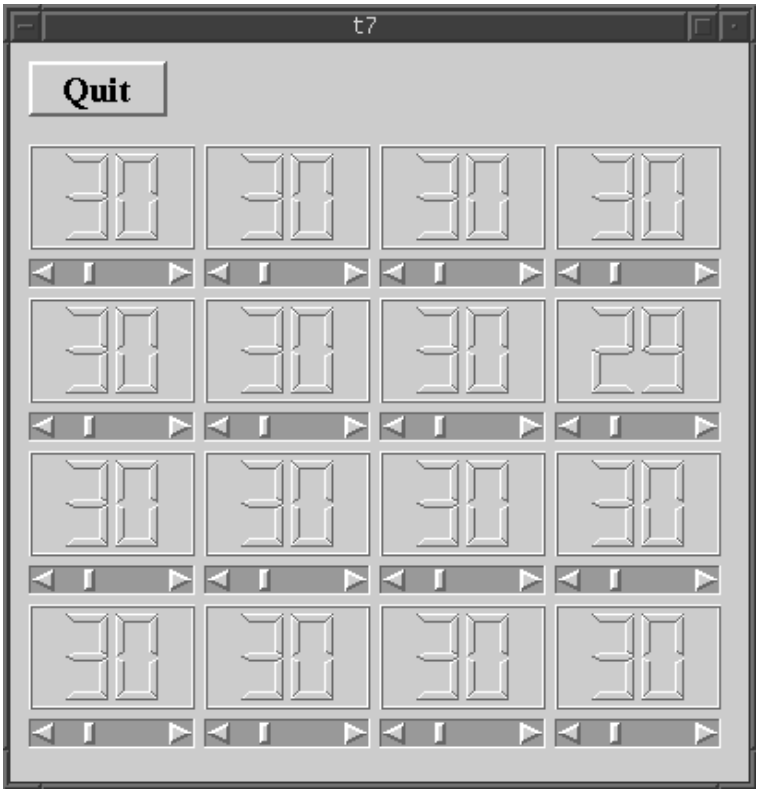
Initialize each slider with a different/random value on startup.

The source contains three occurences of "4". What happens if you change the one in the QGrid constructor call? What about the other two? Why is this?

### 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 classic 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 <qvbox.h>
```

`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`.

```
class QSlider;
```

This is another classic trick, but one that's much less used. Since we don't need QSlider in the *interface* of the class, only in the implementation, we use a forward declaration of the class in the header file, and #include the header file for QSlider in the .cpp file.

This makes compilation of big projects much faster, since when a header file has changed, fewer files need to be recompiled. It can often speed up big compiled by a factor of two or more.

```
class LCDRange : public QVBox
{
    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. If you are 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.

The signal valueChanged() is used when the LCDRange's value has changed - just as you guessed from the name. This is not the last signal you'll see called *something*Changed().

**lcdrange.cpp**

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

```
        connect( slider, SIGNAL(valueChanged(int)), lcd, SLOT(display(int)) );
        connect( slider, SIGNAL(valueChanged(int)), SIGNAL(valueChanged(int)) );
    }
    int LCDRange::value() const
```

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 slider'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 slider: The slider 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 - LCDRange::valueChanged() may be emitted before or after QLCDNumber::display(), entirely arbitrarily.

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

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

**main.cpp**

```
LCDRange *previous = 0;
for( int r = 0 ; r < 4 ; r++ ) {
    for( int c = 0 ; c < 4 ; c++ ) {
        LCDRange* lr = new LCDRange( grid );
        if ( previous )
            connect( lr, SIGNAL(valueChanged(int)),
                     previous, SLOT(setValue(int)) );
        previous = lr;
    }
}
```

All of main.cpp is copied from the previous chapter, except that in the constructor for MyWidget, when we create the 16 LCDRange object 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 slider to the bottom right...

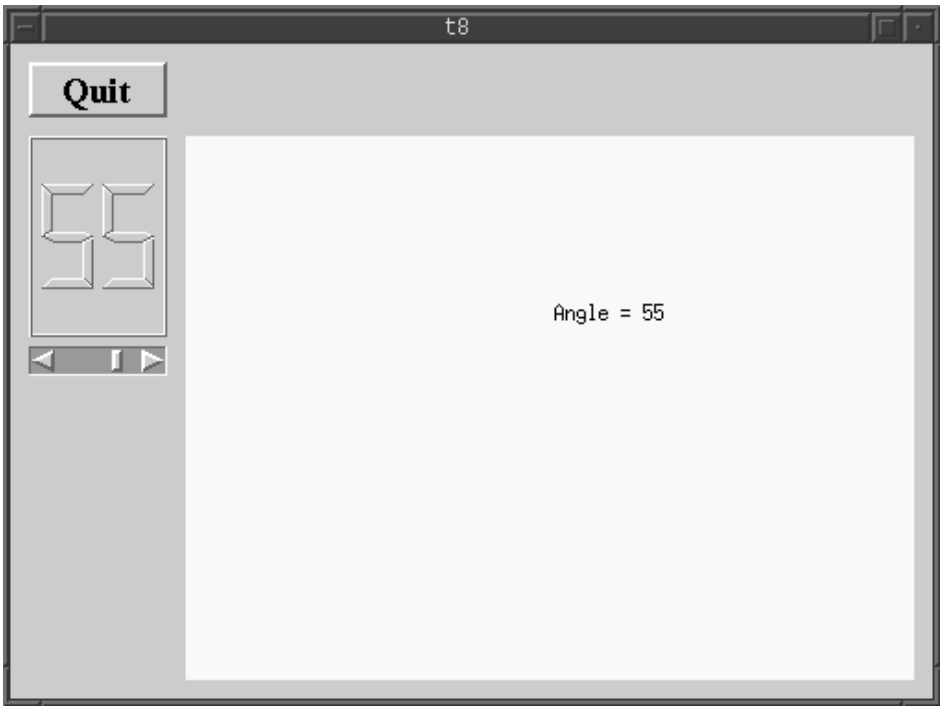
### 7.3. Excercises

Use the bottom right slider to set all LCDs to 30. Then set the top half to 29 by using the rightmost slider 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 slider. 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. We also add a useful keyboard interface (with two lines of code).

- 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**

There is a change to the constructor, we'll discuss that later.

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

setRange() sets the range of the slider 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 down to -9 but chose not to.) If the arguments are illegal, we use Qt's qWarning() function to issue a warning to the user and return immediately. qWarning() is a printf-like function that by default sends its output to `stderr`. You can install your own handler function using ::qInstallMsgHandler() 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; }

    QSizePolicy sizePolicy() const;

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;

    setPalette( QPalette( QColor( 250, 250, 200) ) );

}
```

The constructor initializes the angle value to 45 degrees, and sets a custom palette for this widget. This palette uses the indicated color as background, and picks other colors suitably. (For this widget, only the background and text colors wil be used.)

```
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 of 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 important to only emit the signal angleChanged() when the angle *really* has changed.

Then we set the new angle value and repaint our widget. The QWidget::repaint() function clears the widget (usually filling it with its background color) and send a paint event to the widget. This results in a call the paint event function of the widget.

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 = "Angle = " + QString::number( ang );
    QPainter p( this );
    p.drawText( 200, 200, 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 region in the widget that must be updated. For the time being, we will be lazy and just paint everything.

Our code displays the angle value in the widget at a fixed position. First we create a QString with some text and the angle, then we create a QPainter operating on this widget, and use it to paint the

string. We'll come back to QPainter later; it can do a great many things.

**main.cpp**

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

We include our new class.

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

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

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

In the constructor, we create and set up our LCDRange.

```
    angle->setFixedHeight( 70 );
```

We set the LCDRange to accept ranges from 5 to 70 degrees.

```
    angle->setRange( 5, 70 );
    CannonField *cannonField = new CannonField( this, "cannonField" );
```

We create our CannonField.

```
    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 omitted this check, the program would have entered an infinite loop upon the first change of one of the values.

```
    QGridLayout *grid = new QGridLayout( this, 2, 2, 10 ); //2x2, 10 pixel border
    grid->addWidget( quit, 0, 0 );
```

Until now we've used the no-assembly-required QVBox and QGrid widgets for geometry management. However, now we want to have a little more control over the layout, and switch to the more powerful QGridLayout class. QGridLayout isn't a widget, it is a different class that can manage the children of *any* widget.

As the comment indicates, we create a two-by-two array, with ten pixel borders. (The constructor for QGridLayout can be a little cryptic, so it's good to put in such comments.)

```
    grid->addWidget( angle, 1, 0, Qt::AlignTop );
```

We add the Quit button in the top-left cell of the grid: 0, 0.

```
    grid->addWidget( cannonField, 1, 1 );
```

We put the angle LCDRange in the bottom-left cell, aligned to the top of its cell. (This alignment is one of the things QGridLayout allows but QGrid does not.)

```
    grid->setColStretch( 1, 10 );
```

We put the CannonField object in the bottom right cell. (The top right cell is empty, yes.)

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

We tell QGridLayout that the right column (column 1) is stretchable. Since the left column isn't (it has stretch factor 0 - the default value), QGridLayout will try to let the left-hand widgets' sizes be unchanged and resize just the CannonField when the MyWidget is resized.

```
}
```

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

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

And our last action is to set `angle` to have keyboard focus, so that by default, keyboard input will go to the LCDRange widget.

LCDRange does not contain any keyPressEvent(), so that would seem not to be terribly useful. However, its constructor just got a new line:

The LCDRange sets the slider to be its focus proxy. That means that when someone (the program or the user) wants to give the LCDRange keyboard focus, the slider should take care of it. QSlider has a decent keyboard interface, so with just one line of code we've give LCDRange one.

### 8.2. Behavior

The keyboard now does something - the arrow keys, Home, End, PageUp and PageDown now all do something vaguely sensible.

When the slider 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

Try to resize the window. What happens if you make it really narrow or really squat?

If you remove the AlignTop, what happens to the LCDRange's position and size? Why?

If you give the left-hand column a non-zero stretch factor, what happens when you resize the window?

Leave out the setFocus() call. Which behavior do you prefer?

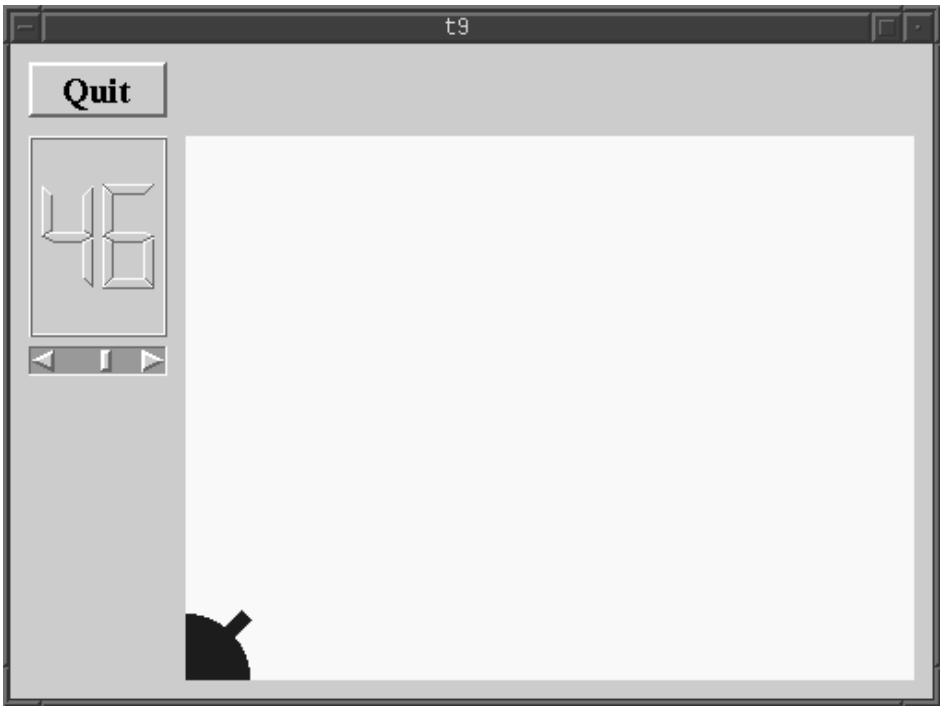
Try to change "Quit" to "&Quit" in the QButton::setText() call. How does the button's look change? What happens if you press Alt-Q while the program's running? (Meta-Q on a few keyboards.)

Center the text in 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 become graphic, by drawing 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( this );
```

We'll now start to use QPainter in ernest. We create a painter that operates on this widget.

```
p.setBrush( blue );
```

When QPainter fills a rectangle, a circle or something, it fills the shape using its brush. Here, we set it to use a blue brush. (We could also use a pattern.)

```
p.setPen( NoPen );
```

And the edges of what QPainter draws are drawn using the pen. Here we set it to NoPen, meaning that there will be no special edge when we draw something. The blue brush will go all the way to the edges of the things we draw.

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

The QPainter::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 for more information about Qt's 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 QPainter::rotate() function rotates the coordinate system of the QPainter around the origin. 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 QPainter::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:

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

We tell Qt that we want a different color allocation strategy for this program. There is no single correct color allocation strategy. Since this program uses an unusual yellow but not many colors, `CustomColor` is best. There are several other allocation strategies. You can read about them in the QApplication::setColorSpec() documentation.

Mostly, you can ignore this - the default is good. Occasionally some applications with unusual color use look bad, and often changing the allocation strategy helps.

### 9.2. Behavior

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

The Q on the Quit button is now underlined, and Alt-Q does what you think it does. If you do not know why, you didn't do the exercises in chapter eight.

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

### 9.3. Excercises

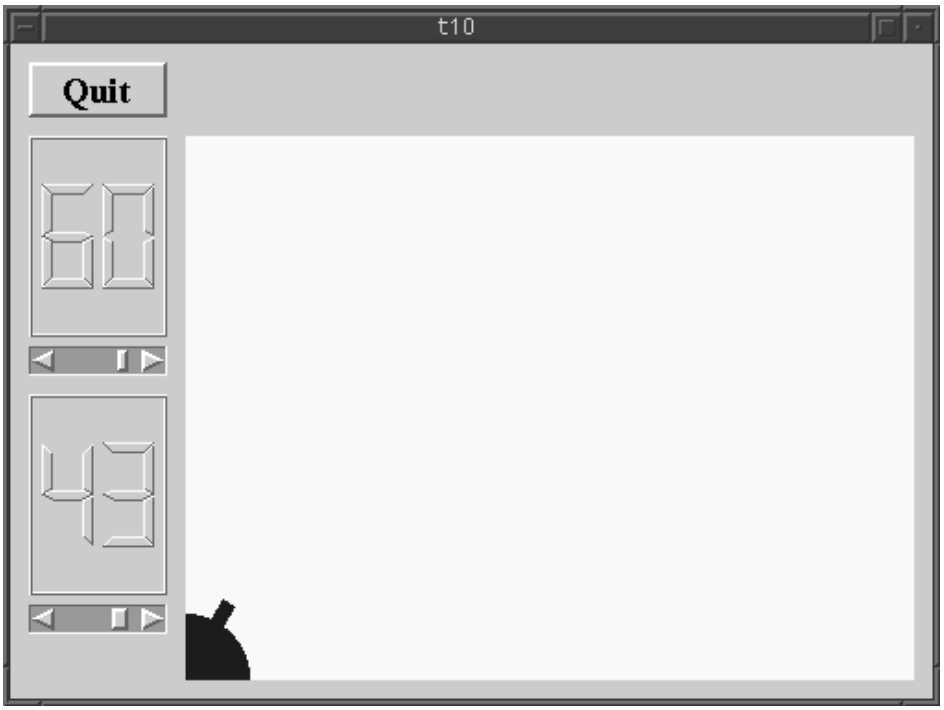
Set a different pen instead of NoPen. Set a patterned brush.

Try "Q&uit" or "Qu&it" as button text instead of "&Quit" - what happens?

#### 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; }
    QSizePolicy sizePolicy() const;
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 put the definition 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. This speeds up and smooths drawing a little bit.

```
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 since we don't show the force value, we don't need to repaint the widget.

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

```
    }
}
void CannonField::paintCannon( QPainter *p )
{
    QRect cr = cannonRect();
    QPixmap pix( cr.size() );
    pix.fill( this, cr.topLeft() );
    QPainter tmp( &pix );
    tmp.setBrush( blue );
    tmp.setPen( NoPen );
    tmp.translate( 0, pix.height() - 1 );
    tmp.drawPie( QRect( -35,-35, 70, 70 ), 0, 90*16 );
    tmp.rotate( -ang );
    tmp.drawRect( QRect(33, -4, 15, 8) );
    tmp.end();
    p->drawPixmap( cr.topLeft(), pix );
}
QRect CannonField::cannonRect() const
{
    QRect r( 0, 0, 50, 50 );
    r.moveBottomLeft( rect().bottomLeft() );
    return r;
```

We have now optimized the paint event to repaint only the parts of the widget that need updating. First we check whether we have to paint anything at all, and return if we don't.

```
}
QSizePolicy CannonField::sizePolicy() const
```

Then we create a temporary pixmap, which we use for flicker-free painting. All the painting operations are done into this pixmap, and then the pixmap is drawn on the screen in a single operation.

This is the essence of flicker-free drawing: Draw on each pixel precisely once. Less, and you get drawing errors. More, and you get flicker. It doesn't matter much in this example - when the code

was written there were still machines slow enough for it to flicker, but not any more. We've kept the code for educational purposes.

```
{
```

We fill the pixmap with the background from this widget.

```
    return QSizePolicy( QSizePolicy::Expanding, QSizePolicy::Expanding );
}
```

We paint, as in chapter 9, but now we paint in the pixmap.

At this point, we have a painter variable and a pixmap that looks precisely right, but we still haven't painted on the screen.

So we open the painter on the CannonField itself and then draw the pixmap.

That's all. A couple of extra lines at the top and a couple at the bottom, and the code is 100% flickerfree.

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

The QWidget::rect() function returns the widget's enclosing rectangle in the widget's own coordinates (where the top left corner is 0,0).

**main.cpp**

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

The constructor is mostly the same, but some new bits have been added.

```
    LCDRange *force  = new LCDRange( this, "force" );
    force->setRange( 10, 50 );
```

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

```
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.

```
QVBoxLayout *leftBox = new QVBoxLayout;
grid->addLayout( leftBox, 1, 0 );
leftBox->addWidget( angle );
leftBox->addWidget( force );
```

In chapter 9, we put `angle` in the lower-left cell of the layout. Now, we want to have two widgets in that cell, so we make a vertical box, put the vertical box in the grid cell, and put each of `angle` and `range` in the vertical box.

```
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.

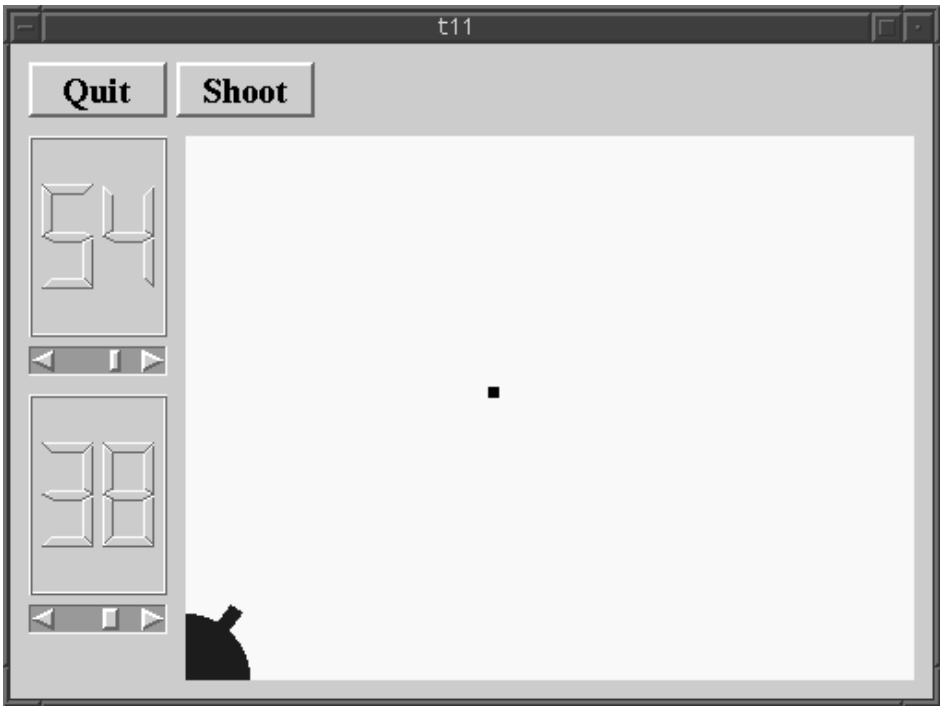
Put the cannon in the bottom right corner.

Try adding a better keyboard interface. For example, make + and - increase and decrease the force and enter shoot. Hint: QAccel and new addStep() and subtractStep() slots in LCDRange, like QSlider::addStep(). If you're bothered by the way the left and right keys work (I am!), change that too.

### 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 a timer to implement animated 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; }
```

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

```
    QSizePolicy sizePolicy() const;

public slots:
```

This private slot is used to move the shot while it is in the air, using a QTimer.

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

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.

```
    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 and connect the QTimer::timeout() signal to our moveShot() slot. We'll move the shot every time the timer times out.

```
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. Finally we start the timer.

moveShot() is the slot that moves the shot, called every 50 milliseconds when the QTimer fires.

Its tasks are to compute the new position, repaint the screen with the shot in the new position, and if necessary stop the timer.

First we make a QRegion that holds the old shotRect(). A QRegion is capable of holding any sort of region, and we'll use it here to simplify the painting. shotRect() returns the rectangle where the shot is now - it is explained in detail later.

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 the timer. Else, we add the new shotRect() to the QRegion.

Finally, we repaint the QRegion. This will send a single paint event, for just the one or two rectangles that need updating.

The paint event function has been split in two since the previous chapter. Now, we fetch the bounding rectangle of the region that needs painting, check whether it intersects either the cannon and/or the shot, and if necessary call paintCannon() and/or paintShot().

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

We leave out the implementation of paintCannon(); it is the same as the paintEvent() from the previous chapter.

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 coordinate 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 );
};
```

The only addition is the shoot button.

```
QPushButton *shoot = new QPushButton( "Shoot", this, "shoot" );

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

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

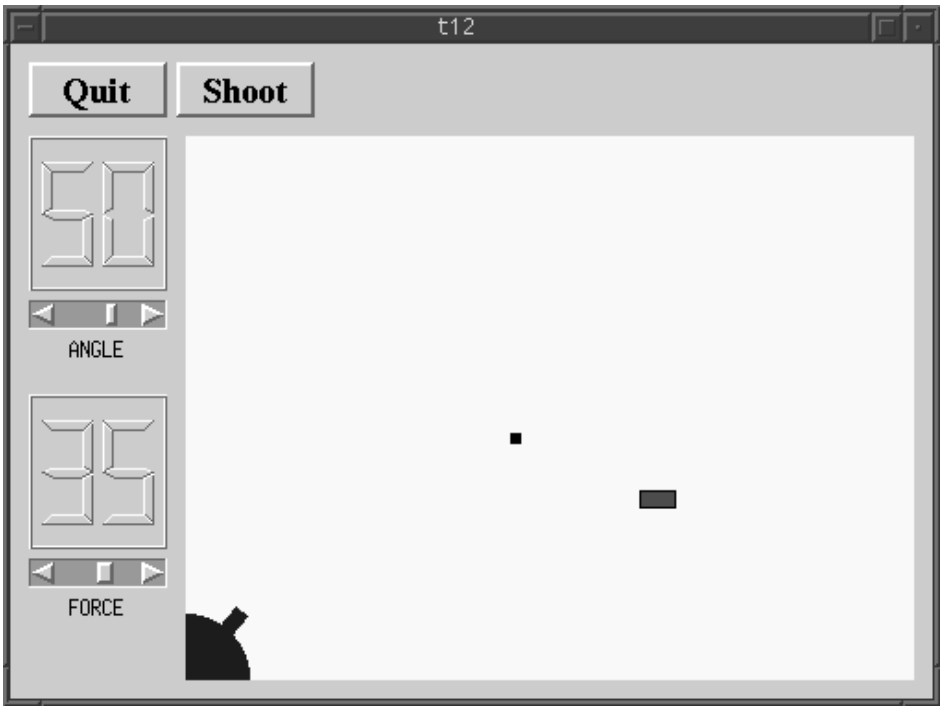
Make the shot a filled circle (hint: QPainter::drawEllipse() may help).

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

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 )
        : QVBox( 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 )
        : QVBox( parent, name )
{
    init();
    setText( s );
}
```

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

```
void LCDRange::init()
{
    QLCDNumber *lcd  = new QLCDNumber( 2, this, "lcd"  );
    slider = new QSlider( 0, 99,        // range
                          10,           // page steps
                          0,            // inital value
                          Horizontal,   // orientation
                          this, "slider" );
    label  = new QLabel( " ", this, "label"  );
    label->setAlignment( AlignHCenter );
    label->setFixedHeight( label->sizeHint().height() );
    connect( slider, SIGNAL(valueChanged(int)), lcd, SLOT(display(int)) );
    connect( slider, SIGNAL(valueChanged(int)), SIGNAL(valueChanged(int)) );
}
```

The set up of `lcd` and `slider` 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.

**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 is certain that it has not and will 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 calling repaint() on 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 for QDate, QTime and QDateTime for more information.

Finally, we calculate the target's center point. We keep it within 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.

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

This `if` statement checks if the shot rectangle intersects the target rectangle. If it does, the shot has hit the target (ouchh!). We stop the shoot timer 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.

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.

And the rest of the function is as before.

CannonField::paintEvent() is as before, except that this has been added:

These two lines make sure the target too is painted, when necessary.

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

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 QRect::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.

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

We set the angle text label to "ANGLE".

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

We set the force text label to "FORCE".

## 12.2. Behavior

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

The LCDRange widgets look a bit strange - the built-in layout management in QVBox gives the labels too much space and the rest not enough. We'll fix that in the next chapter.

## 12.3. Excercises

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

If you did the "round shot" exercise from the previous chapter, try changing the shotRect() to a shotRegion() that returns a QRegion so you can have really accurate collision detection.

Make a moving target.

Make sure the target is always created entirely on-screen.

Make sure that the widget cannot be resized so that the target isn't visible. Hint: QWidget::setMinimumSize() is your friend.

(Not easy:) 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, with a score. 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.

The layout problems in LCDRange are fixed.

- 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

#### lcdrange.h

```
#include <qvbox.h>
class QSlider;
class QLabel;
class LCDRange : public QVBox
{
    Q_OBJECT
public:
    LCDRange( QWidget *parent=0, const char *name=0 );
```

We inherit QWidget rather than QVBox: QVBox is very easy to use, but again it showed its limitations, so we switch to the more powerful and slightly harder to use QVBoxLayout. (As you remember, QVBoxLayout is not a widget, it manages one.)

#### lcdrange.cpp

We need to include qlayout.h now, to get the other layout management API.

We inherit QWidget in the usual way.

The other constructor has the same change. init() is unchanged, except that we've added some lines at the end:

We create a QVBoxLayout with all the default values, managing this widget's children.

At the top, we add the QLCDNumber, with a non-zero stretch.

Then we add the other two, both with the default zero stretch.

This stretch control is something QVBoxLayout (and QHBoxLayout, and QGridLayout) offers, but classes like QVBox do not. In this case, we're saying that the QLCDNumber should stretch and the others should not.

### 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().

This new signal indicates that the CannonField is in a state where the shoot() slot makes sense.

We'll use it below to enable/disable the Shoot button.

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

```
gameEnded     = FALSE;
```

This line has been added to the constructor. Initially, the game is not over (luckily for the player :-).

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

We added a new isShooting() function, so shoot() uses it instead of testing directly. Also, shoot tells the world that the CannonField cannot shot now.

```
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 with different rules. For example, a multi-player version of this, where the first one to hit ten times could use the CannonField unchanged.

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.

moveShot() too emits the new canShoot(TRUE) signal at the same time as either hit() or miss().

Modifications in CannonField::paintEvent():

```
void CannonField::paintEvent( QPaintEvent *e )
{
```

```
    QRect updateR = e->rect();
    QPainter p;
    p.begin( this );
    if ( updateR.intersects( cannonRect() ) )
        paintCannon( &p );
    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 X servers with Unicode fonts) 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 when shooting and the target only when playing (that is, when the game is not ended).

**gamebrd.h**

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

```
class QPushButton;
class LCDRange;
class QLCDNumber;
class CannonField;
#include "lcdrange.h"
#include "cannon.h"
class GameBoard : public QWidget
{
    Q_OBJECT
public:
    GameBoard( QWidget *parent=0, const char *name=0 );
protected slots:
    void  fire();
    void  hit();
    void  missed();
    void  newGame();
private:
    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.

### gamebrd.cpp

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

We have made some changes in the GameBoard constructor.

```
    cannonField = new CannonField( this, "cannonField" );
```

`cannonField` is now a member variable, so we carefully change the constructor to use it. (The *good* programmers at Troll Tech never forget this, but I do. Caveat programmor. If "programmor" is latin, at least. Anyway, back to the code.)

```
connect( cannonField, SIGNAL(hit()),SLOT(hit()) );
connect( cannonField, SIGNAL(missed()),SLOT(missed()) );
angle->setValue( 60 );
force->setValue( 25 );
```

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.

```
QPushButton *restart = new QPushButton( "New Game", this, "newgame" );
restart->setFont( QFont( "Times", 18, QFont::Bold ) );
```

We also use the cannonField's canShoot() signal to enable or disable the Shoot button appropriately.

```
connect( restart, SIGNAL(clicked()), SLOT(newGame()) );
```

We create, set up and connect the New Game 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" );
QGridLayout *grid = new QGridLayout( this, 2, 2, 10 );
```

We create four new widgets. Note that we don't bother to keep the pointers to the QLabel widgets in the GameBoard class since there's nothing much we want to do with them. Qt will delete them when the GameBoard widget is destroyed, and the layout classes will resize them appropriately.

```
    QHBoxLayout *topBox = new QHBoxLayout;
    grid->addLayout( topBox, 0, 1 );
    topBox->addWidget( shoot );
    topBox->addWidget( hits );
    topBox->addWidget( hitsL );
    topBox->addWidget( shotsLeft );
    topBox->addWidget( shotsLeftL );
    topBox->addStretch( 1 );
    topBox->addWidget( restart );
```

The number of widgets in the top-right cell is getting large. Once it was empty, now it's full enough that we group together the layout setting for better overview.

Notice how we let all the widgets have their preferred sizes, instead putting the stretch just to the left of the New Game button.

```
    newGame();
}
```

We're all done constructing the GameBoard, so we start it all using newGame(). (newGame() is a slot, but as we said, slots can be used as ordinary functions too.)

```
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, unchanged except for the name change.

## 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.

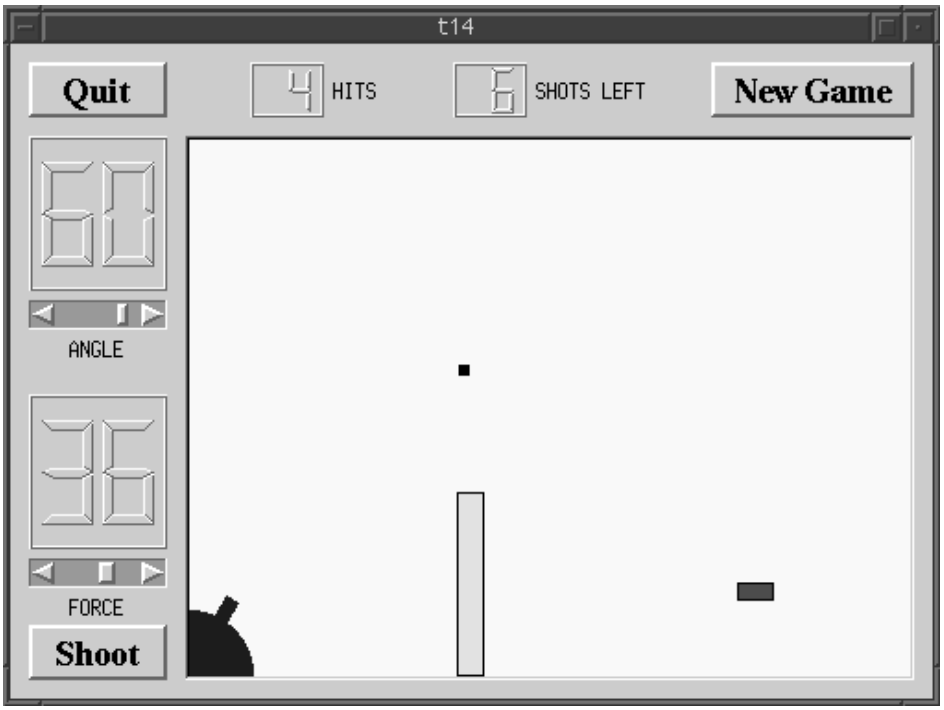
Make some splatter effects when the shot hits the target.

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 mouse is not pressed on the barrel.

Now that we have a barrier there are three ways to miss. We test for the third, too.

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.

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 too, 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.

This Qt event handler is called whenever the user releases a mouse button and it was pressed inside this widget.

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

The paint event has two extra lines:

paintBarrier() does the same sort of thing as paintShot(), paintTarget() and paintCannon().

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

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

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.cpp**

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

We include the class definition of QAccel.

```
QVBox *box = new QVBox( this, "cannonFrame" );
box->setFrameStyle( QFrame::WinPanel | QFrame::Sunken );
cannonField = new CannonField( box, "cannonField" );
```

We create and set up a QVBox, set its frame style, and then create `cannonField` as a child of that box. Since nothing else is in the box, the effect is that the QVBox will put a frame around the CannonField.

```
QAccel *accel = new QAccel( this );
accel->connectItem( accel->insertItem( Key_Space), this, SLOT(fire()) );
accel->connectItem( accel->insertItem( Key_Q), qApp, SLOT(quit()) );
QGridLayout *grid = new QGridLayout( this, 2, 2, 10 );
grid->addWidget( quit, 0, 0 );
```

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 is a child of a widget and will be destroyed when that widget 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 Enter, and we want the application to quit when key Control-Q is pressed. Since Enter is sometimes Return and there are even keyboard with *both* keys, we make both Enter and Return invoke fire().

```
grid->addWidget( box, 1, 1 );

grid->setColStretch( 1, 10 );
```

And then we set up Control-Q to do the same thing as Alt-Q. Some people are more used to Control-Q, and anyway it shows how do do it.

CTRL, Key_Enter, Key_Return and Key_Q are all constants provided by Qt. They're actually Qt::Key_Enter etc. but practically all classes inherit the Qt namespace class.

```
QVBoxLayout *leftBox = new QVBoxLayout;

grid->addLayout( leftBox, 1, 0 );

leftBox->addWidget( angle );

leftBox->addWidget( force );
```

We put `box` (the QVBox), not the CannonField, in the lower right cell.

## 14.2. Behavior

The cannon now shoots when you press Enter. 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.

(This excercise was first done by Igor Rafienko. You can download his game.)

The new excercise is:

Write a Breakout 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 and shortcut 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 `UNICODE_ACCEL`).

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

## 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().

**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( UNICODE_ACCEL + 'q', 202 );  // Unicode 'q' to quit
a->insertItem( Key_D );                     // gets id 3
a->insertItem( Key_P + CTRL + SHIFT );      // gets id 4
```

**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.

**QString QAccel::keyToString ( int k )** `[static]`

Creates an accelerator string for the key *k.* For instance CTRL+Key_O gives "Ctrl+O". The "Ctrl" etc. are translated in the "QAccel" scope.

See also: stringToKey().

**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::reparent().

**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().

**void QAccel::setWhatsThis ( int id, const QString & text )**

Sets a Whats This help for a certain accelerator.

Arguments:

- *id* is the accelerator item id.
- *text* is the Whats This help text. The text will be shown when the application is in What's This mode and the user either hits the respective accelerator key or selects a menu item that has been attached to this accelerator.

See also: whatsThis() and QWhatsThis::inWhatsThisMode().

**int QAccel::shortcutKey ( const QString & str )** `[static]`

Returns the shortcut key for *string,* or 0 if *string* has no shortcut sequence.

For example, acceleratorKey("E&xit") returns ALT+Key_X, shortcutChar("&Exit") returns ALT+Key_E and shortcutChar("Exit") returns 0. (In code that does not inherit the Qt namespace class, you need to write e.g. Qt::ALT+Qt::Key_X.)

We provide a list of common accelerators in English. (At the time of writing the Microsoft and The Open Group appear to not have issued such recommendations for other languages.).

**int QAccel::stringToKey ( const QString & s )** `[static]`

Creates an accelerator code for the string *s.* For example "Ctrl+O" gives CTRL+UNICODE_ACCEL+'O'. The strings "Ctrl", "Shift", "Alt" and their translated equivalents in the "QAccel" scope are recognized.

A common usage of this function is to provide translatable accelerator values for menus:

```
QPopupMenu* file = new QPopupMenu(this);
file->insertItem( p1, tr("&Open..."), this, SLOT(open()),
    QAccel::stringToKey(tr("Ctrl+O")) );
```

Note that this function currently only supports character accelerators (unlike keyToString() which can produce Ctrl+Backspace, etc. from the appropriate key codes).

**QString QAccel::whatsThis ( int id ) const**

Returns the Whats This help text for the specified item *id* or QString::null if no text has been defined yet.

See also: setWhatsThis().

### 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 GUI application's control flow and main settings.

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

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

This object (you can access is using the global variable *qApp)* does a great many things, most importantly:

- It initializes the application to the user's desktop settings like palette(), font() or the doubleClickInterval(). It keeps track of these properties in case the user changes the desktop globally in some kind of control panel.
- It performs event handling, meaning that it receives events from the underlying window system and sends them to the destination widgets. By using sendEvent() and postEvent() you can send your own events to widgets.
- It parses common command line arguments and sets its internal state accordintly. See the constructor documentation below for more details about this.
- It defines the application's look and feel, which is encapsulate in a QStyle object. This can be changed during runtime with setStyle().

- It specifies how the application is to allocate colors. See setColorSpec() for details.
- It specifies the default text encoding (see setDefaultCodec() ) and provides localization of strings that are visible to the user via translate().
- It provides some magic objects like the desktop() and the clipboard().
- It knows about the application's windows, and lets you ask which widget is at a certain position using widgetAt(), lets you closeAllWindows(), gives you a list of topLevelWidgets(), etc.
- It manages the application's mouse cursor handling, see setOverrideCursor() and setGlobalMouseTracking().
- On the X window system, it provides functions to flush and sync the communication stream, see flushX() and syncX().
- It provides support to implement sophisticated session management. This makes it possible for applications to terminate gracefully when the user logs out, to cancel a shutdown process if termination isn't possible and even to preserve the entire application state for a future session. See isSessionRestored(), sessionId() and commitData() and saveState() for details.

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

Since the QApplication object does so much initialization, it is absolutely necessary to create it before any other objects related to the user interface are created.

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

**Groups of functions:**

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

**Non-GUI programs**

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

Examples:

qdir/qdir.cpp
showimg/main.cpp
rangecontrols/main.cpp
validator/main.cpp
themes/main.cpp

listviews/main.cpp
aclock/main.cpp
checklists/main.cpp
buttons_groups/main.cpp
drawlines/connect.cpp
dclock/main.cpp
wizard/main.cpp
xform/xform.cpp
application/main.cpp
qiconview/main.cpp
cursor/cursor.cpp
layout/layout.cpp
life/main.cpp
i18n/main.cpp
drawdemo/drawdemo.cpp
lineedits/main.cpp
popup/popup.cpp
listbox/main.cpp
menu/menu.cpp
progress/progress.cpp
scribble/main.cpp
qmag/qmag.cpp
table/main.cpp
splitter/splitter.cpp
progressbar/main.cpp
richtext/main.cpp
tooltip/main.cpp
qwerty/main.cpp
forever/forever.cpp
rot13/rot13.cpp
desktop/desktop.cpp
scrollview/scrollview.cpp
qfd/qfd.cpp
addressbook/main.cpp
movies/main.cpp
picture/picture.cpp
listbox_combo/main.cpp
hello/main.cpp
qfileiconview/main.cpp
qbrowser/main.cpp
biff/main.cpp
tictac/main.cpp
customlayout/main.cpp
dirview/main.cpp

### 16.2. Member Function Documentation

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

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

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

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

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 (not available if Qt was compiled with the NO_DEBUG flag defined):

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

See Debugging Techniques for a more detailed explanation.

All Qt programs automatically support the following command line options:

- `-style=` *style,* sets the application GUI style. Possible values are `motif, windows,` and `platinum.`
- `-session=` *session,* restores the application from an earlier session.
- `-qdevel` activates the Application Builder window, which allows run-time inspection of the program.
- `-qtranslate` activates the Application Translator window, which allows translation of the texts shown in the program.

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

- `-display` *display,* sets the X display (default is $DISPLAY).
- `-geometry` *geometry,* sets the client geometry of the main widget.
- `-fn` or `-font` *font,* defines the application font.
- `-bg` or `-background` *color,* sets the default background color and an application palette (light and dark shades are calculated).
- `-fg` or `-foreground` *color,* sets the default foreground color.
- `-btn` or `-button` *color,* sets the default button color.
- `-name` *name,* sets the application name.
- `-title` *title,* sets the application title (caption).
- `-visual TrueColor,` forces the application to use a TrueColor visual on an 8-bit display.
- `-ncols` *count,* limits the number of colors allocated in the color cube on 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 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]`

Cleans up any window system resources that were allocated by this application. Sets the global variable `qApp` to null. Unlike former versions of Qt the destructor does *not* delete all remaining widgets.

**void QApplication::aboutToQuit ()** `[signal]`

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

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

See also: quit().

**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().

**QWidget * QApplication::activeWindow () const**

Returns the application toplevel window that has the keyboard input focus, or null if no application window has the focus. Note that there might be an activeWindow even if there is no focusWidget, if no widget in that window accepts key events.

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

**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.

**void QApplication::closeAllWindows ()** `[slot]`

A convenience function that closes all toplevel windows.

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

```
// the "Quit" menu entry should try to close all windows
QPopupMenu* file = new QPopupMenu( this );
file->insertItem( tr("&Quit"), qApp, SLOT(closeAllWindows()), CTRL+Key_Q );
// when the last window was closed, the application should quit
connect( qApp, SIGNAL( lastWindowClosed() ), qApp, SLOT( quit() ) );
```

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

See also: QWidget::close(), QWidget::closeEvent(), lastWindowClosed(), quit(), topLevelWidgets() and QWidget::isTopLevel().

**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().

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

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

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

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

**Important**

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

Details about session management in general can be found here.

See also: isSessionRestored(), sessionId() and saveState().

**int QApplication::cursorFlashTime () `[static]`**

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

The default value is 1000 milliseconds. Under Windows, the control panel value is used.

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

See also: setCursorFlashTime().

**QTextCodec* QApplication::defaultCodec () const**

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

**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

**bool QApplication::desktopSettingsAware ()** `[static]`

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

See also: setDesktopSettingsAware().

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

Returns the maximum duration for a double click.

The default value is 400 milliseconds. Under Windows, the control panel value is used.

See also: setDoubleClickInterval().

**int QApplication::enter_loop ()**

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

See also: exit_loop().

**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() and setMainWidget().

Examples:

showimg/main.cpp
rangecontrols/main.cpp
validator/main.cpp

themes/main.cpp
listviews/main.cpp
aclock/main.cpp
checklists/main.cpp
buttons_groups/main.cpp
drawlines/connect.cpp
dclock/main.cpp
mainlyQt/editor.cpp
xform/xform.cpp
application/main.cpp
qiconview/main.cpp
cursor/cursor.cpp
layout/layout.cpp
life/main.cpp
i18n/main.cpp
drawdemo/drawdemo.cpp
lineedits/main.cpp
popup/popup.cpp
listbox/main.cpp
menu/menu.cpp
progress/progress.cpp
scribble/main.cpp
qmag/qmag.cpp
table/main.cpp
splitter/splitter.cpp
progressbar/main.cpp
richtext/main.cpp
tooltip/main.cpp
qwerty/main.cpp
forever/forever.cpp
rot13/rot13.cpp
scrollview/scrollview.cpp
qfd/qfd.cpp
addressbook/main.cpp
movies/main.cpp
picture/picture.cpp
listbox_combo/main.cpp
hello/main.cpp
qfileiconview/main.cpp
qbrowser/main.cpp
biff/main.cpp
tictac/main.cpp
customlayout/main.cpp
dirview/main.cpp

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

Tells the application to exit with a return code.

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

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

Note that unlike the C library 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(), QWidget::hasFocus() and activeWindow().

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

Returns the default font for a widget. Basically this function uses the class name of the widget to get a font for it.

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

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::installTranslator ( QTranslator * mf )**

Adds *mf* to the list of message files to be used for localization. Message files are searched starting with the most recently added file.

See also: removeTranslator(), translate() and QObject::tr().

Examples:

i18n/main.cpp

**bool QApplication::isSessionRestored () const**

Returns whether the application has been restored from an earlier session.

See also: sessionId(), commitData() and saveState().

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

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

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

For convenience, transient toplevel widgets such as popup menus and dialogs are omitted.

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

**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, easiest 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 ( const QWidget * w = 0 )** `[static]`

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

If a widget is passed as argument, the default palette for the widget's class is returned. This may or may not be the application palette, but in most cases there won't be a special palette for certain types of widgets. An exception is the popup menu under Windows, when the user defined a special background color for menus in the display settings.

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

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

Polishing of widgets.

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

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

The default implementation sets a class specific font or palette if available and no font or palette has been set yet. Then it calls QStyle::polish().

See also: QStyle::polish(), QWidget::polish(), setPalette() and setFont().

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

Stores the event in a queue and returns immediately.

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: processOneEvent(), 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() and aboutToQuit().

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

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

The events are *not* dispatched, simply removed from the queue. You should never need to call this function.

**void QApplication::removeTranslator ( QTranslator * mf )**

Removes *mf* from the list of message files used by this application. Does not, of course, delete mf.

See also: installTranslator(), translate() and QObject::tr().

**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().

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

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

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

Note that you should never exit the application within this function. Instead, the session manager may or may not do this afterwards, depending on the context. Futhermore, most session managers

will very likely request a saved state immediately after the application has been started. This permits the session manager to learn about the application's restart policy.

**Important**

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

Details about session management in general can be found here

See also: isSessionRestored(), sessionId() and commitData().

**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().

Examples:

popup/popup.cpp

**void QApplication::sendPostedEvents ()** `[static]`

Dispatches all posted events.

**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*.

If *receiver* is 0, all objects get their events. If *event_type* is 0, all types of events are dispatched.

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

**QString QApplication::sessionId () const**

Returns the identifier of the current session.

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

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

See also: isSessionRestored(), commitData() and saveState().

**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 on Windows machines it can cause dithering of non-standard colours in 256-color mode.
- `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 currently the same as `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, for eample. They provide an 8 bit visual by 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 certain colors.

See QColor::enterAllocContext() for more information.

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

The color cube used by Qt 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:

showimg/main.cpp
themes/main.cpp
tetrix/tetrix.cpp
qbrowser/main.cpp

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

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

Under windows, calling this function sets the double click interval for all windows.

See also: cursorFlashTime().

**void QApplication::setDefaultCodec ( QTextCodec * codec )**

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

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

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

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

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

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

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

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

See also: desktopSettingsAware().

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

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

Under windows, calling this function sets the double click interval for all windows.

See also: doubleClickInterval().

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

Changes the default application font to *font.*

The default font depends on the underlying window system, under X Windows on the X server in use.

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

If a className is passed, then the font is only set for widgets that inherit this class in the sense of QObject::inherits()

Widgets created after this call get *font* as their font when they access it.

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

Examples:

showimg/main.cpp
desktop/desktop.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 )** `[virtual]`

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:

rangecontrols/main.cpp
validator/main.cpp
themes/main.cpp
listviews/main.cpp
aclock/main.cpp
checklists/main.cpp
buttons_groups/main.cpp
drawlines/connect.cpp
dclock/main.cpp
mainlyQt/editor.cpp
xform/xform.cpp
qiconview/main.cpp
cursor/cursor.cpp
layout/layout.cpp
life/main.cpp
i18n/main.cpp
drawdemo/drawdemo.cpp
lineedits/main.cpp
popup/popup.cpp
listbox/main.cpp
menu/menu.cpp
progress/progress.cpp

scribble/main.cpp
qmag/qmag.cpp
table/main.cpp
splitter/splitter.cpp
progressbar/main.cpp
richtext/main.cpp
tooltip/main.cpp
forever/forever.cpp
rot13/rot13.cpp
qfd/qfd.cpp
addressbook/main.cpp
picture/picture.cpp
listbox_combo/main.cpp
hello/main.cpp
qfileiconview/main.cpp
biff/main.cpp
tictac/main.cpp
customlayout/main.cpp
dirview/main.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().

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

Changes the default application palette to *palette.*

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

If a className is passed, then the palette is only set for widgets that inherit this class in the sense of

QObject::inherits()

Widgets created after this call get *palette* as their palette when they access it.

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

See also: QWidget::setPalette(), palette() and QStyle::polish().

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

Sets the application GUI style to *style.*

See also: style() and QStyle.

Examples:

themes/main.cpp

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

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

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

repaint all widgets if the color is changed.

The default color is `darkBlue`.

See also: winStyleHighlightColor().

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

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

See also: closingDown().

**QStyle& QApplication::style ()** `[static]`

Returns the style object of the application.

See also: setStyle() and QStyle.

**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, for example all the popup menus of a

menubar.

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().

### QString QApplication::translate ( const char * scope, const char * key ) const

Returns the best available translation for *key* in *scope,* by querying the installed messages files. The message file that was installed last is asked first.

QObject::tr() offers a more convenient way to use this functionality.

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

If none of the message files contain a translation for *key* in *scope,* this function returns *key.*

This function is not virtual, but you can add alternative translation techniques by installing subclasses of QTranslator.

See also: QObject::tr(), installTranslator(), removeTranslator() and QTranslator.

### 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]**

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

Returns the color used to mark selections in windows style.

See also: setWinStyleHighlightColor().

### 16.3. Related Functions

**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.

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 void qWarning (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 list of arguments, similar to the C printf() function.

Example:

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

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

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

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

Examples:

progress/progress.cpp
tictac/main.cpp

**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

**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 truly weird (i.e. different endianness for 16 bit and 32 bit integers).

### 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

### Q_EXPORT void qFatal (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 list of arguments, similar to the C printf() function.

Example:

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

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

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

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

**Q_EXPORT const char * qVersion ()**

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

**Q_EXPORT void qDebug (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 list of arguments, similar to the C printf() function.

Example:

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

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

Note: If DEBUG was not defined when the Qt library was built (i.e. NO_DEBUG was defined), this function does nothing.

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

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

Examples:

qdir/qdir.cpp
mainlyXt/editor.cpp

**void qAddPostRoutine (Q_CleanUpFunction p)**

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

The function given by *p* should take no arguments and return 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
}
```

### 16.3.1. URL

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

# 17. QArray Class Reference

`#include <qarray.h>`

Inherits QGArray.

Inherited by QPointArray.

### 17.1. Detailed Description

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

QArray is implemented as a template 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
```

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[]().

**ConstIterator QArray::begin () const**

Returns a const iterator pointing at the beginning of this array. This iterator can be used as the iterators of QValueList and QMap for example. In fact it does not only behave like a usual pointer: It is a pointer.

**Iterator QArray::begin ()**

Returns an iterator pointing at the beginning of this array. This iterator can be used as the iterators of QValueList and QMap for example. In fact it does not only behave like a usual pointer: It is a pointer.

**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().

**ConstIterator QArray::end () const**

Returns a const iterator pointing behind the last element of this array. This iterator can be used as the iterators of QValueList and QMap for example. In fact it does not only behave like a usual pointer: It is a pointer.

**Iterator QArray::end ()**

Returns an iterator pointing behind the last element of this array. This iterator can be used as the iterators of QValueList and QMap for example. In fact it does not only behave like a usual pointer: It is a pointer.

**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. QAsciiCache Class Reference

`#include <qasciicache.h>`

Inherits QGCache.

## 18.1. Detailed Description

The QAsciiCache class is a template class that provides a cache based on `char*` keys.

QAsciiCache is implemented as a template class. Define a template instance QAsciiCache<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. The cache items are accessed via `char*` keys. QAsciiCache cannot handle Unicode keys, instead use the QCache template, which uses `QString` keys. A QCache has the same performace as a QAsciiCache.

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 removed.

Apart from insert(), by far the most important function is find() (which also exists as operator[]). This function looks up an item, returns it, and by default marks it as being the most recently used item.

There are also methods to remove() or take() an object from the cache. Calling setAutoDelete(TRUE) for a cache tells it to delete items that are removed. The default is to not delete items when then are removed (i.e. remove() and take() are equivalent).

When inserting an item into the cache, 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 cache item, the virtual function QCollection::deleteItem() is called. It's default implementation in QAsciiCache is to delete the item if auto-deletion is enabled.

There is a QAsciiCacheIterator which may be used to traverse the items in the cache in arbitrary order.

See also: QAsciiCacheIterator, QCache, QIntCache and Collection Classes

## 18.2. Member Function Documentation

### QAsciiCache::QAsciiCache ( const QAsciiCache<type> & c )

For internal use only.

### QAsciiCache::QAsciiCache ( int maxCost=100, int size=17, bool caseSensitive=TRUE, bool copyKeys=TRUE )

Constructs a cache with the following properties:

Arguments:

- *maxCost* is the maximum allowed total cost.
- *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. Each inserted item is associated with a cost. When inserting a new item, if the total cost of all items in the cache will exceeds *maxCost,* the cache will start throwing out the older (recently least used) items until there is room enough for the new item to be inserted.

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 comparison includes the whole Unicode alphabeth.

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.

**QAsciiCache::~QAsciiCache ()**

Removes all items from the cache and destroys it. All iterators that access this cache will be reset.

**void QAsciiCache::clear ()** `[virtual]`

Removes all items from the cache, and deletes them if auto-deletion has been enabled.

All cache iterators that operate this on cache are reset.

See also: remove() and take().

Reimplemented from QCollection.

**uint QAsciiCache::count () const** `[virtual]`

Returns the number of items in the cache.

See also: totalCost().

Reimplemented from QCollection.

**type * QAsciiCache::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 QAsciiCache::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, QAsciiCache will remove old, least-used items until there is room for this new item.

**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 QAsciiCache::isEmpty () const**

Returns TRUE if the cache is empty, or TRUE if there is at least one object in it.

**int QAsciiCache::maxCost () const**

Returns the maximum allowed total cost of the cache.

See also: setMaxCost() and totalCost().

**QAsciiCache<type>& QAsciiCache::operator= ( const QAsciiCache<type> & c )**

For internal use only.

**type * QAsciiCache::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 QAsciiCache::remove ( const char * k )**

Removes the item associated with *k,* and returns TRUE if the item was present in the cache or FALSE if it was not.

The item is deleted if auto-deletion has been enabled, i.e. you have called setAutoDelete(TRUE).

If there are two or more items with equal keys, then the one that was inserted last is is removed.

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 QAsciiCache::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 removed immediately.

See also: maxCost() and totalCost().

**uint QAsciiCache::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 * QAsciiCache::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 QAsciiCache::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().

### 18.2.1. URL

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

# 19. QAsciiCacheIterator Class Reference

`#include <qasciicache.h>`

Inherits QGCacheIterator.

## 19.1. Detailed Description

The QAsciiCacheIterator class provides an iterator for QAsciiCache 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 QAsciiCache. QAsciiCache updates all iterators that refer an item when that item is removed.

QAsciiCacheIterator 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: QAsciiCache.

## 19.2. Member Function Documentation

### QAsciiCacheIterator::QAsciiCacheIterator ( const QAsciiCache<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).

### QAsciiCacheIterator::QAsciiCacheIterator ( const QAsciiCacheIterator<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.

**QAsciiCacheIterator::operator type * () const**

Cast operator. Returns a pointer to the current iterator item. Same as current().

**uint QAsciiCacheIterator::count () const**

Returns the number of items in the cache on which this iterator operates.

See also: isEmpty().

**type * QAsciiCacheIterator::current () const**

Returns a pointer to the current iterator item.

**const char * QAsciiCacheIterator::currentKey () const**

Returns the key for the current iterator item.

**bool QAsciiCacheIterator::isEmpty () const**

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

See also: count().

**type * QAsciiCacheIterator::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 * QAsciiCacheIterator::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 * QAsciiCacheIterator::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 * QAsciiCacheIterator::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 * QAsciiCacheIterator::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.

**QAsciiCacheIterator<type>& QAsciiCacheIterator::operator= ( const QAsciiCacheIterator<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 * QAsciiCacheIterator::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 * QAsciiCacheIterator::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().

### 19.2.1. URL

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

# 20. QAsciiDict Class Reference

`#include <qasciidict.h>`

Inherits QGDict.

## 20.1. Detailed Description

The QAsciiDict class is a template class that provides a dictionary based on `char*` keys.

QAsciiDict is implemented as a template class. Define a template instance QAsciiDict<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. QAsciiDict has `char*` keys. QAsciiDict cannot handle Unicode keys, instead use the QDict template which uses QString keys. A QDict has the same performace as a QAsciiDict.

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)
    QAsciiDict<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 (specified in the constructor). QAsciiDict implements the [] operator to lookup an item.

QAsciiDict 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 QAsciiDict 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)
    QAsciiDict<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 QAsciiDictIterator 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. QAsciiDict's default implementation is to delete the item if auto-deletion is enabled.

See also: QAsciiDictIterator, QDict, QIntDict, QPtrDict and Collection Classes

## 20.2. Member Function Documentation

### QAsciiDict::QAsciiDict ( const QAsciiDict<type> & dict )

Constructs a copy of *dict.*

Each item in *dict* are inserted into this dictionary. Only the pointers are copied (shallow copy).

### QAsciiDict::QAsciiDict ( 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 comparison 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.

**QAsciiDict::~QAsciiDict ()**

Removes all items from the dictionary and destroys it.

All iterators that access this dictionary will be reset.

See also: setAutoDelete().

**void QAsciiDict::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 QAsciiDict::count () const** `[virtual]`

Returns the number of items in the dictionary.

See also: isEmpty().

Reimplemented from QCollection.

**type * QAsciiDict::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 QAsciiDict::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 QAsciiDict::isEmpty () const**

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

See also: count().

**QAsciiDict<type> & QAsciiDict::operator= ( const QAsciiDict<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 * QAsciiDict::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 QAsciiDict::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 QAsciiDict::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:

```
QAsciiDict<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 QAsciiDict::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 QAsciiDict::size () const**

Returns the size of the internal hash array (as specified in the constructor).

See also: count().

**void QAsciiDict::statistics () const**

Debugging-only function that prints out the dictionary distribution using qDebug().

**type * QAsciiDict::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().

### 20.2.1. URL

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

# 21. QAsciiDictIterator Class Reference

```
#include <qasciidict.h>
```

Inherits QGDictIterator.

## 21.1. Detailed Description

The QAsciiDictIterator class provides an iterator for QAsciiDict collections.

QAsciiDictIterator is implemented as a template class. Define a template instance QAsciiDictIterator<X> to create a dictionary iterator that operates on QAsciiDict<X> (dictionary of X*).

Example:

```
#include <qasciidict.h>
#include <stdio.h>
void main()
{
  // Creates a dictionary that maps char* ==> char* (case insensitive)
    QAsciiDict<char> dict( 17, FALSE );
    dict.insert( "France", "Paris" );
    dict.insert( "Russia", "Moscow" );
    dict.insert( "Norway", "Oslo" );
    QAsciiDictIterator<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.

Multiple iterators may independently traverse the same dictionary. A QAsciiDict knows about all iterators that are operating on the dictionary. When an item is removed from the dictionary, QAsciiDict update all iterators that are referring the removed item to point to the next item in the traversing order.

See also: QAsciiDict and Collection Classes

## 21.2. Member Function Documentation

**QAsciiDictIterator::QAsciiDictIterator ( const QAsciiDict<type> & dict )**

Constructs an iterator for *dict*. The current iterator item is set to point on the first item in the *dict*.

**QAsciiDictIterator::~QAsciiDictIterator ()**

Destroys the iterator.

**QAsciiDictIterator::operator type * () const**

Cast operator. Returns a pointer to the current iterator item. Same as current().

**uint QAsciiDictIterator::count () const**

Returns the number of items in the dictionary this iterator operates on.

See also: isEmpty().

**type * QAsciiDictIterator::current () const**

Returns a pointer to the current iterator item.

**const char * QAsciiDictIterator::currentKey () const**

Returns a pointer to the key for the current iterator item.

**bool QAsciiDictIterator::isEmpty () const**

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

See also: count().

**type * QAsciiDictIterator::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 * QAsciiDictIterator::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 * QAsciiDictIterator::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 * QAsciiDictIterator::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.

### 21.2.1. URL

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

# 22. QAsyncIO Class Reference

`#include <qasyncio.h>`

Inherited by QDataSink and QDataSource.

### 22.1. Detailed Description

Encapsulates I/O asynchronicity.

The Qt classes for asynchronous input/output provide a simple mechanism to allow large files or slow data sources 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.

### 22.2. 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.

#### 22.2.1. URL

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

# 23. QBitArray Class Reference

```
#include <qbitarray.h>
```

Inherits QByteArray.

## 23.1. 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]
```

## 23.2. 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 ()**

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().

**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().

### 23.3. Related Functions

**QBitArray operator^ (const QBitArray & a1, const QBitArray & a2)**

Returns the XOR result between the bit arrays *a1* and *a2*.

See also: QBitArray::operator^().

**QBitArray operator& (const QBitArray & a1, const QBitArray & a2)**

Returns the AND 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

**QBitArray operator| (const QBitArray & a1, const QBitArray & a2)**

Returns the OR result between the bit arrays *a1* and *a2*.

See also: QBitArray::operator|=().

**QDataStream & operator>> (QDataStream & s, QBitArray & a)**

Reads a bit array from a stream.

### 23.3.1. URL

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

# 24. QBitmap Class Reference

`#include <qbitmap.h>`

Inherits QPixmap.

## 24.1. 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

## 24.2. Member Function Documentation

**QBitmap::QBitmap ()**

Constructs a null bitmap.

See also: QPixmap::isNull().

**QBitmap::QBitmap ( const QSize & size, const uchar * bits, bool isXbitmap=FALSE )**

Overloaded constructor; takes a QSize parameter instead of *(w,h).*

**QBitmap::QBitmap ( const QString & 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 ( const QBitmap & bitmap )**

Constructs a bitmap which is a copy of *bitmap.*

**QBitmap::QBitmap ( int w, int h, bool clear = FALSE, QPixmap::Optimization optimization = QPixmap::DefaultOptim )**

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 *Qt::color0).*

The optional *optimization* argument specifies the optimization setting for the bitmap. The default optimization should be used in most cases. Games and other pixmap-intensive applications may benefit from setting this argument.

See also: QPixmap::setOptimization() and QPixmap::setDefaultOptimization().

**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 ( const QSize & size, bool clear = FALSE, QPixmap::Optimization optimization = QPixmap::DefaultOptim )**

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

**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().

### 24.2.1. URL

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

# 25. QBitVal Class Reference

```
#include <qbitarray.h>
```

### 25.1. 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.

### 25.2. 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.

#### 25.2.1. URL

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

# 26. QBoxLayout Class Reference

`#include <qlayout.h>`

Inherits QLayout.

Inherited by QHBoxLayout and QVBoxLayout.

### 26.1. 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 mainWidget()), 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 constructor 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 of 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.

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.

See also the Layout Overview documentation.

Examples:

layout/layout.cpp
popup/popup.cpp
customlayout/main.cpp

## 26.2. Member Function Documentation

**QBoxLayout::QBoxLayout ( QLayout * parentLayout, Direction d, int space = -1, const char * name=0 )**

Creates a new QBoxLayout with direction *d* and inserts it into *parentLayout.*

**QBoxLayout::QBoxLayout ( QWidget * parent, Direction d, int border=0, int space = -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. *space* is the default number of pixels between neighbouring children. If *space* is -1 the value of *border* is used.

*name* is the internal object name

See also: direction().

**QBoxLayout::QBoxLayout ( Direction d, int space = -1, const char * name=0 )**

If *space* is -1, this QBoxLayout will inherit its parent's spacing(), otherwise *space* is used.

You have to insert this box into another layout.

Examples:

layout/layout.cpp

**QBoxLayout::~QBoxLayout ()**

Destroys this box.

**void QBoxLayout::addItem ( QLayoutItem * item )** `[virtual]`

Adds *item* to this box.

Reimplemented from QLayout.

**void QBoxLayout::addLayout ( QLayout * layout, int stretch = 0 )**

Adds *layout* to the box, with serial stretch factor *stretch.*

See also: setAutoAdd(), addWidget() and addSpacing().

**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().

**void QBoxLayout::addStretch ( int stretch = 0 )**

Adds a stretchable space with zero minimum size and stretch factor *stretch.*

See also: addSpacing().

**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 alignmnt = 0 )**

Adds *widget* to the box, with stretch factor *stretch* and alignment *alignmnt.*

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 has a stretch factor greater than zero, the space is distributed according to the QWidget:sizePolicy() of each widget that's involved.

Alignment is specified by *alignmnt* which takes the same arguments as QLabel::setAlignment(). The default alignment is 0, which means that the widget fills the entire cell.

Note: The alignment parameter is interpreted more aggressively than in previous versions of Qt. A non-default alignment now indicates that the widget should not grow to fill the available space, but should be sized according to sizeHint().

See also: setAutoAdd(), addLayout() and addSpacing().

Examples:

layout/layout.cpp
popup/popup.cpp
scrollview/scrollview.cpp
customlayout/main.cpp

**bool QBoxLayout::hasHeightForWidth () const** `[virtual]`

Returns whether this layout's preferred height depends on its width.

Reimplemented from QLayoutItem.

**int QBoxLayout::heightForWidth ( int w ) const** `[virtual]`

Returns the layout's preferred height when it is *w* pixels wide.

Reimplemented from QLayoutItem.

**void QBoxLayout::invalidate ()** `[virtual]`

Resets cached information.

Reimplemented from QLayoutItem.

**QLayoutIterator QBoxLayout::iterator ()** `[virtual]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QLayoutItem.

**QSize QBoxLayout::maximumSize () const** `[virtual]`

Returns the maximum size needed by this box.

Reimplemented from QLayoutItem.

**QSize QBoxLayout::minimumSize () const** `[virtual]`

Returns the minimum size needed by this box.

Reimplemented from QLayoutItem.

**void QBoxLayout::setGeometry ( const QRect & s )** `[virtual]`

Resizes managed widgets within the rectangle *s*.

Reimplemented from QLayoutItem.

**bool QBoxLayout::setStretchFactor ( QWidget * w, int stretch )**

Sets the stretch factor for widget *w* to *stretch* and returns TRUE, if *w* is found in this layout (not including child layouts).

Returns FALSE if *w* is not found.

**QSize QBoxLayout::sizeHint () const**

Returns the preferred size of this grid.

### 26.2.1. URL

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

# 27. QBrush Class Reference

`#include <qbrush.h>`

Inherits Qt.

### 27.1. 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:

drawdemo/drawdemo.cpp
picture/picture.cpp

## 27.2. 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 current 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().

## 27.3. Related Functions

**QDataStream & operator>> (QDataStream & s, QBrush & b)**

Reads a brush from the stream and returns a reference to the stream.

**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)

### 27.3.1. URL

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

# 28. QBuffer Class Reference

```
#include <qbuffer.h>
```

Inherits QIODevice.

## 28.1. 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

## 28.2. 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.

### 28.2.1. URL

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

# 29. QButton Class Reference

`#include <qbutton.h>`

Inherits QWidget.

Inherited by QCheckBox, QPushButton, QRadioButton and QToolButton.

## 29.1. 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(bool) is emitted when the state of a toggle button changes.
- stateChanged(int) is emitted when the state of a tristate 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.png") );
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:

scrollview/scrollview.cpp

## 29.2. Member Function Documentation

**QButton::QButton ( QWidget * parent=0, const char * name=0, WFlags f=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(), toggled(), and stateChanged() 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 QRadioButton, QCheckBox, QToolButton and QPushButton.

**void QButton::drawButtonLabel ( QPainter * )** `[virtual protected]`

Draws the button text or pixmap.

This virtual function is reimplemented by subclasses to draw real buttons.

Reimplemented in QToolButton, QCheckBox, QRadioButton and QPushButton.

**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.

**bool QButton::focusNextPrevChild ( bool next )** `[virtual]`

Reimplemented for internal reasons; the API is not affected.

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().

**bool QButton::isExclusiveToggle () const**

Returns TRUE if this button behaves exclusively inside a QButtonGroup. In that case, this button can only be toggled off by another buton beinhg toggled on.

**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 calls animateClick(), the arrow keys cause focus changes.

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 )** `[virtual]`

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 )** `[virtual]`

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 )** `[virtual]`

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 )** `[virtual]`

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 )** `[virtual]`

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::setState ( ToggleState )** `[virtual protected]`

Protected function to set the button state into one of the three states:

- `QButton::Off` - isOn() is FALSE
- `QButton::On` - isOn() is TRUE
- `QButton::NoChange` - the button is in the NoChanged state.

See also: setToggleType().

**void QButton::setText ( const QString & text )** `[virtual]`

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

**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().

**void QButton::setToggleType ( ToggleType type )** `[virtual protected]`

Sets the type of toggling behavior. *type* is one of:

- `SingleShot` - pressing the button causes an action, then the button returns to the unpressed state.
- `Toggle` - pressing the button toggles between an On and and Off state.
- `Tristate` - pressing the button cycles between three states - On, Off, and NoChanged.

Subclasses use this, and present it with a more comfortable interface.

**void QButton::stateChanged ( int state )** `[signal]`

This signal is emitted whenever a toggle button changes status. *state* is 2 if the button is on, 1 if it is in the "no change" state or 0 if the button is off.

This may be the result of a user action, toggle() slot activation, setState(), or because setOn() was

called.

See also: clicked().

**QString QButton::text () const**

Returns the button text, or null string 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().

### 29.2.1. URL

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

# 30. QButtonGroup Class Reference

```
#include <qbuttongroup.h>
```

Inherits QGroupBox.

Inherited by QHButtonGroup and QVButtonGroup.

### 30.1. 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.

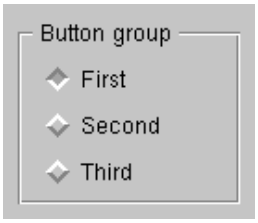
An exclusive button group switches off all toggle buttons except the one that was clicked. A button group is by default non-exclusive. All radio buttons that are inserted, will be mutually exclusive even if the button group is non-exclusive.

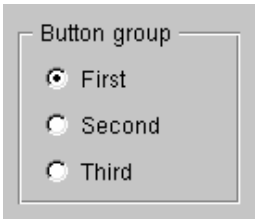
There are two ways of using a button group:

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 constructor. 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:

tabdialog/tabdialog.cpp
xform/xform.cpp
drawdemo/drawdemo.cpp

## 30.2. 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 QString & 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 ( int strips, Orientation orientation, QWidget * parent=0, const char * name=0 )**

Constructs a button group with no title. Child widgets will be arranged in *strips* rows or columns (depending on *orientation).*

The *parent* and *name* arguments are passed to the QWidget constructor.

**QButtonGroup::QButtonGroup ( int strips, Orientation orientation, const QString & title, QWidget * parent=0, const char * name=0 )**

Constructs a button group with a *title.* Child widgets will be arranged in *strips* rows or columns (depending on *orientation).*

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.

**int QButtonGroup::count () const**

Returns the number of buttons in the group.

**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::id ( QButton * button ) const**

Returns the id of *button,* or -1 if *button* is not a member of this group.

**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 identifiers 0, 1, 2, etc.

See also: find(), remove() and setExclusive().

Examples:

xform/xform.cpp

**bool QButtonGroup::isExclusive () const**

Returns TRUE if the button group is exclusive, otherwise FALSE.

See also: setExclusive().

**bool QButtonGroup::isRadioButtonExclusive () const**

Returns whether this button group will treat radio buttons as mutually exclusive.

See also: setRadioButtonExclusive().

**void QButtonGroup::moveFocus ( int key )** `[virtual]`

Moves the keyboard focus according to *key,* and if appropriate checks the new focus item.

This function does nothing unless the keyboard focus points to one of the button group members and

*key* is one of `Key_Up, Key_Down, Key_Left` and `Key_Right.`

**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().

**QButton * QButtonGroup::selected ()**

Returns a pointer to the selected radio button in this group, if one exists, or 0 if there is no selected radio button in this group.

**void QButtonGroup::setButton ( int id )** `[virtual]`

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 )** `[virtual]`

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 switched on at the

same time.

The default setting is FALSE.

See also: isExclusive().

**void QButtonGroup::setRadioButtonExclusive ( bool on )** `[virtual]`

If *on* is TRUE, this button group will treat radio buttons as mutually exclusive, and other buttons according to isExclusive(). This function is called automatically whenever a QRadioButton is inserted, so you should normally never have to call it.

### 30.2.1. URL

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

# 31. QCache Class Reference

`#include <qcache.h>`

Inherits QGCache.

## 31.1. Detailed Description

The QCache class is a template class that provides a cache based on `QString` keys.

QCache is implemented as a template 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. The cache items are accessed via QString keys, which are Unicode strings. If you want to use non-Unicode, plain 8-bit `char*` keys, use the QAsciiCache template. A QCache has the same performace as a QAsciiCache.

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 removed.

Apart from insert(), by far the most important function is find() (which also exists as operator[]). This function looks up an item, returns it, and by default marks it as being the most recently used item.

There are also methods to remove() or take() an object from the cache. Calling setAutoDelete(TRUE) for a cache tells it to delete items that are removed. The default is to not delete items when then are removed (i.e. remove() and take() are equivalent).

When inserting an item into the cache, 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 cache item, the virtual function QCollection::deleteItem() is called. It's default implementation in QCache is to delete the item if auto-deletion is enabled.

There is a QCacheIterator which may be used to traverse the items in the cache in arbitrary order.

See also: QCacheIterator, QAsciiCache, QIntCache and Collection Classes

## 31.2. Member Function Documentation

### QCache::QCache ( const QCache<type> & c )

For internal use only.

### QCache::QCache ( int maxCost=100, int size=17, bool caseSensitive=TRUE )

Constructs a cache with the following properties:

Arguments:

- *maxCost* is the maximum allowed total cost.
- *size* is the size of the internal hash array.
- *caseSensitive* specifies whether to use case sensitive lookup or not. Each inserted item is associated with a cost. When inserting a new item, if the total cost of all items in the cache will exceeds *maxCost,* the cache will start throwing out the older (recently least used) items until there is room enough for the new item to be inserted.

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 comparison includes the whole Unicode alphabeth.

### QCache::~QCache ()

Removes all items from the cache and destroys it. All iterators that access this cache will be reset.

### void QCache::clear () [virtual]

Removes all items from the cache, and deletes them if auto-deletion has been enabled.

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 QString & 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 QString & 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 remove old, least-used items until there is room for this new item.

**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 QString & 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 QString & k )**

Removes the item associated with *k,* and returns TRUE if the item was present in the cache or FALSE if it was not.

The item is deleted if auto-deletion has been enabled, i.e. you have called setAutoDelete(TRUE).

If there are two or more items with equal keys, then the one that was inserted last is is removed.

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 QString & 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().

### 31.2.1. URL

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

# 32. QCacheIterator Class Reference

`#include <qcache.h>`

Inherits QGCacheIterator.

## 32.1. 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 item 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.

## 32.2. 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 from there on.

**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 which this iterator operates.

See also: isEmpty().

**type * QCacheIterator::current () const**

Returns a pointer to the current iterator item.

**QString QCacheIterator::currentKey () const**

Returns 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().

### 32.2.1. URL

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

# 33. QCDEStyle Class Reference

`#include <qcdestyle.h>`

Inherits QMotifStyle.

## 33.1. Detailed Description

CDE Look and Feel

This style provides a slightly improved Motif look similar to some versions of the Common Desktop Environment (CDE). The main difference are thinner frames and more modern radio buttons and check boxes. Together with a dark background and a bright text/foreground color, the style looks quite attractive (at least for Motif fans).

## 33.2. Member Function Documentation

**QCDEStyle::QCDEStyle ( bool useHighlightCols = FALSE )**

Constructs a QCDEStyle

If useHighlightCols is FALSE (the default value), then the style will polish the application's color palette to emulate the Motif way of highlighting, which is a simple inversion between the base and the text color.

**int QCDEStyle::defaultFrameWidth () const** `[virtual]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QStyle.

**void QCDEStyle::drawArrow ( QPainter * p, ArrowType type, bool down, int x, int y, int w, int h, const QColorGroup & g, bool, const QBrush * = 0 )**

Reimplemented for internal reasons; the API is not affected.

**void QCDEStyle::drawExclusiveIndicator ( QPainter * p, int x, int y, int w, int h, const QColorGroup & g, bool on, bool down = FALSE, bool = TRUE )** `[virtual]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QStyle.

**void QCDEStyle::drawIndicator ( QPainter * p, int x, int y, int w, int h, const QColorGroup & g, int s, bool down = FALSE, bool = TRUE )** `[virtual]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QStyle.

### 33.2.1. URL

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

# 34. QChar Class Reference

```
#include <qstring.h>
```

### 34.1. Detailed Description

A Unicode character.

A QChar is a simple 16-bit value representing a Unicode character. Most C++ compilers will process them much as they would a "short int".

QChar values are normally used in combination with QString.

They can be trivially constructed from:

`char` the char is assumed to be a Latin-1 character, for example QChar('Q') is the Unicode character U0051 and QChar('t') is the Unicode character U0074.

`short`

the MSB is the high-order byte (row) of the Unicode character, for example QChar(ushort(0x6817)) is the Unicode character U6817 meaning *kuri* - chestnut tree - in Japanese.

`char, char`

the characters are LSB and MSB respectively, for example QChar(0x25,0x04) is the Unicode character U0425 which is the Cyrillic letter YA which looks like a mirrored R.

### 34.2. Member Function Documentation

**uchar QChar::cell () const**

Returns the cell (least significant byte) of the Unicode character.

**uchar& QChar::cell ()**

Returns a reference to the cell (least significant byte) of the Unicode character.

**QString QChar::decomposition () const**

Decomposes a character into its parts. Returns QString::null if no decomposition exists.

**int QChar::digitValue () const**

Returns the numeric value of the digit, or -1 if the character is not a digit.

**bool QChar::isDigit () const**

Returns whether the character is a decimal digit (Number_DecimalDigit).

**bool QChar::isLetter () const**

Returns whether the character is a letter (Letter_* categories).

**bool QChar::isMark () const**

Returns whether the character is a mark (Mark_* categories).

**bool QChar::isNull () const**

Returns TRUE if the charaters is the unicode character 0x0000, ie. ASCII NUL.

**bool QChar::isNumber () const**

Returns whether the character is a number (of any sort - Number_* categories).

See also: isDigit().

**bool QChar::isPrint () const**

Returns whether the character is a printable character. This is and character of category Cc or Cn. Note that this gives no indication of whether the character is available in some font.

**bool QChar::isPunct () const**

Returns whether the character is puntuation (Punctuation_* categories).

**bool QChar::isSpace () const**

Returns whether the character is a separator character (Separator_* categories).

**QChar QChar::lower () const**

Returns the lowercase equivalent if the character is uppercase, or the character itself otherwise.

**bool QChar::mirrored () const**

Returns whether the character is a mirrored character (one that should be reversed if the text direction is reversed).

**uchar QChar::row () const**

Returns the row (most significant byte) of the Unicode character.

**uchar& QChar::row ()**

Returns a reference to the row (most significant byte) of the Unicode character.

**ushort QChar::unicode () const**

Returns the numeric Unicode value equal to the QChar. Normally, you should use QChar objects as they are equivalent, but for some low-level tasks (eg. indexing into an array of Unicode information), this function is useful.

**QChar QChar::upper () const**

Returns the uppercase equivalent if the character is lowercase, or the character itself otherwise.

### 34.2.1. URL

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

# 35. QCheckBox Class Reference

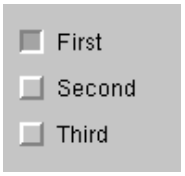
```
#include <qcheckbox.h>
```

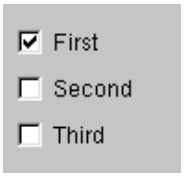
Inherits QButton.

## 35.1. 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:

tabdialog/tabdialog.cpp
xform/xform.cpp

## 35.2. Member Function Documentation

### QCheckBox::QCheckBox ( QWidget * parent, 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 QString & 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:

tabdialog/tabdialog.cpp

**void QCheckBox::setNoChange ()**

Sets the checkbox into the "no change" state.

See also: setTristate().

**void QCheckBox::setTristate ( bool y=TRUE )**

Makes the checkbox a tristate checkbox if *y* is TRUE. A tristate checkbox is useful when you need to give the use the option of neither setting nor unsetting an option, for example choosing Italic or non-Italic when the selected text is partially Italic and partially not.

See also: setNoChange() and stateChanged().

**QSize QCheckBox::sizeHint () const** `[virtual]`

Returns a size which fits the contents of the check box.

Reimplemented from QWidget.

**QSizePolicy QCheckBox::sizePolicy () const** `[virtual]`

Specifies that this widget may stretch horizontally, but is fixed vertically.

Reimplemented from QWidget.

### 35.2.1. URL

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

# 36. QCheckListItem Class Reference

```
#include <qlistview.h>
```

Inherits QListViewItem.

## 36.1. 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.

## 36.2. Member Function Documentation

**QCheckListItem::QCheckListItem ( QListView * parent, const QString & 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 QString & text, const QPixmap & p )**

Constructs a Controller item with parent *parent,* text *text* and pixmap *p.*

**QCheckListItem::QCheckListItem ( QListViewItem * parent, const QString & text, Type tt = Controller )**

Constructs a checkable item with parent *parent,* text *text* and type *tt.* Note that this item must not be a a RadioButton. Radio buttons must be children on a Controller.

**QCheckListItem::QCheckListItem ( QListViewItem * parent, const QString & text, const QPixmap & p )**

Constructs a Controller item with parent *parent,* text *text* and pixmap *p.*

**QCheckListItem::QCheckListItem ( QCheckListItem * parent, const QString & 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 radio button 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::paintFocus ( QPainter * p, const QColorGroup & cg, const QRect & r )** `[virtual]`

Draws the focus rectangle.

**void QCheckListItem::setOn ( bool b )** `[virtual]`

Sets this button on if *b* is TRUE, off otherwise. Maintains radio button 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.

**QString QCheckListItem::text () const**

Returns the text of this item.

**QString 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.

### 36.2.1. URL

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

# 37. QChildEvent Class Reference

```
#include <qevent.h>
```

Inherits QEvent.

### 37.1. Detailed Description

The QChildEvent class contains event parameters for child object events.

Child events are sent to objects when children are inserted or removed.

The handler for these events is QObject::childEvent().

### 37.2. Member Function Documentation

**QChildEvent::QChildEvent ( Type type, QObject * child )**

Constructs a child event object.

The *type* parameter must be either *QEvent::ChildInserted* or *QEvent::ChildRemoved.*

**QObject * 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 object lost a child.

#### 37.2.1. URL

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

# 38. QClipboard Class Reference

`#include <qclipboard.h>`

Inherits QObject.

## 38.1. 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 the same formats that drag and drop supports, and uses much of the same mechanisms.

Only a single QClipboard object may exist in an application. This is because QClipboard is a shared window system resource. Call QApplication::clipboard() to access the clipboard.

Example:

```
QClipboard *cb = QApplication::clipboard();
QString text;
// Copy text from the clipboard (paste)
text = cb->text();
if ( text )
    qDebug( "The clipboard contains: %s", text );
// Copy text into the clipboard
cb->setText( "This text can be pasted by other programs" );
```

## 38.2. Member Function Documentation

**void QClipboard::clear ()**

Clears the clipboard contents.

**void QClipboard::connectNotify ( const char * )** `[virtual protected]`

For internal use only.

Reimplemented from QObject.

**QMimeSource* QClipboard::data () const**

Returns a reference to a QMimeSource representation of the current clipboard data.

**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.

**QImage QClipboard::image () const**

Returns the clipboard image, or null if the clipboard does not contain an image.

See also: setText().

**QPixmap QClipboard::pixmap () const**

Returns the clipboard pixmap, or null if the clipboard does not contains any pixmap. Note that this usually looses more information than image().

See also: setText() and image().

**void QClipboard::setData ( QMimeSource * src )**

Sets the clipboard data. Ownership of the data is transferred to the clipboard - the only way to remove this data is to set something else, or to call clear(). The QDragObject subclasses are reasonable things to put on the clipboard (but do not try to drag the same object). Do not put QDragMoveEvent or QDropEvent subclasses on the clipboard, as they do not belong to the event handler which receives them.

The setText() and setPixmap() functions are shorthand ways of setting the data.

**void QClipboard::setImage ( const QImage & image )**

Copies *image* into the clipboard.

This is just a shorthand for:

```
setData(new QImageDrag(image))
```

See also: image() and setData().

**void QClipboard::setPixmap ( const QPixmap & pixmap )**

Copies *pixmap* into the clipboard. Note that this usually looses more information than setImage(), as the data may be converted to an image for transfer.

See also: pixmap().

**void QClipboard::setText ( const QString & text )**

Copies *text* into the clipboard.

See also: text() and setData().

**QString QClipboard::text () const**

Returns the clipboard text, or a null string if the clipboard does not contain any text.

See also: setText().

### 38.2.1. URL

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

# 39. QCloseEvent Class Reference

```
#include <qevent.h>
```

Inherits QEvent.

### 39.1. 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
}
```

A typical reimplementation of a close event handler is shown in the qwerty/qwerty.cpp example: If the document wasn't changed, the close event is accepted with *e->except()*. If there are unsaved changes, it asks the user whether she wants to save the data. This is done with QMessageBox::warning(). The close event is only accepted if the data was either saved successfully or if the user explicitely stated that the modifications shall be discarded.

If you want your widget also to be deleted when it is closed, simply create it with the `WDestructiveClose` widget flag. This is very useful for the independent toplevel windows of a multi window application. The qwerty/qwerty.cpp example also makes use of this.

**Warning:** Be careful. destructive close implies 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 operation. Be sure that you cannot have any other pointers to the widget hanging around.

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.

If the last toplevel window is closed, the QApplication::lastWindowClosed() signal is emitted.

See also: QWidget::close(), QWidget::hide(), QObject::destroyed(), QApplication::setMainWidget(), QApplication::lastWindowClosed() and QApplication::quit().

Examples:

popup/popup.cpp

## 39.2. Member Function Documentation

**QCloseEvent::QCloseEvent ()**

Constructs a close event object with the accept parameter flag set to FALSE.

**void QCloseEvent::accept ()**

Sets the accept 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. If the widget was created with the WDestructiveClose widget flag, it is also destroyed.

See also: ignore() and QWidget::hide().

Examples:

popup/popup.cpp

**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 closed.

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().

### 39.2.1. URL

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

# 40. QCollection Class Reference

`#include <qcollection.h>`

Inherited by QGCache, QGDict and QGList.

### 40.1. 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 (see setAutoDelete()).

A collection is implemented using the `Item` (generic collection item) type, which is a `void*`. The template classes that create the real collections cast the `Item` to the required type.

See also: Collection Classes

### 40.2. Member Type Documentation

#### QCollection::Item

This type is the generic "item" in a QCollection.

### 40.3. 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 QAsciiCache, QGCache, QIntDict, QDict, QPtrDict, QList, QGDict, QGList, QIntCache, QAsciiDict and QCache.

**virtual uint QCollection::count () const** `[virtual]`

Returns the number of objects in the collection.

Reimplemented in QPtrDict, QIntCache, QDict, QAsciiDict, QCache, QGDict, QIntDict, QGList, QList, QAsciiCache and QGCache.

**void QCollection::deleteItem ( Item 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.

**Warning:** If you reimplement this function you must also reimplement the destructor and call the virtual function clear() from your destructor. This is due to the way virtual functions and destructors work in C++: virtual functions in derived classes cannot be called from a destructor. If you do not do this your deleteItem() function will not be called when the container is destructed.

See also: newItem() and setAutoDelete().

**QCollection::Item QCollection::newItem( Item d )** `[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

### 40.3.1. URL

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

# 41. QColor Class Reference

```
#include <qcolor.h>
```

### 41.1. 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 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.

Since many people don't know the HSV color model very well, we'll cover it briefly here.

The RGB model is hardware-oriented. Its representation is close to what most monitors show. In contrast, HSV represents color in a way more suited to traditional human perception of color. For example, the relationships "stronger than", "darker than", "the opposite of" are easily expressed in HSV, but are much harder to express in RGB.

HSV, like RGB, has three components. They are:

- H, for hue, is either 0-360 if the color is chromatic (not gray), or meaningless if it is gray. It represents degrees on the color wheel familiar to most people. Red is 0 (degrees), green is 120 and blue is 240.
- S, for saturation, is 0-255 and the bigger it is, the stronger the color is. Grayish colors have saturation near 0, very strong colors have saturation near 255.
- V, for value, is 0-255 and represents lightness or brightness of the color. 0 is black, 255 is far from black as possible.

Here are some examples: Pure red is H=0, S=255, V=255. A dark red, moving slightly towards the magenta, could be H=350 (equvalent to -10), S=255, V=180. A grayish light red could have H about 0 (say 350-359 or 0-10), S about 50-100, and S=255.

See also: QPalette, QColorGroup, QApplication::setColorSpec() and Color FAQ.

Examples:

grapher/grapher.cpp
drawlines/connect.cpp
drawdemo/drawdemo.cpp
progress/progress.cpp
desktop/desktop.cpp
picture/picture.cpp

### 41.2. 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 QString & name )**

Constructs a named color in the same way as setNamedColor().

See also: setNamedColor().

**QColor::QColor ( const QColor & c )**

Constructs a color that is a copy of *c*.

**QColor::QColor ( const char * name )**

Constructs a named color in the same way as setNamedColor().

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, but should be considered an arbitrary platform-dependent value.

The pixel() function calls alloc() if necessary, so in general you don't need to call this function.

See also: setLazyAlloc() and enterAllocContext().

**int QColor::blue () const**

Returns the B (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

**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().

**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( QString 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().

**void QColor::getHsv ( int & h, int & s, int & v ) const**

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

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().

**int QColor::green () const**

Returns the G (green) component of the RGB value.

**void QColor::hsv ( int * h, int * s, int * v ) const**

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

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 FALSE 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().

**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

**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, but the value is an arbitrary 32-bit value.

See also: setLazyAlloc() and alloc().

**int QColor::red () const**

Returns the R (red) component of the RGB value.

**QRgb QColor::rgb () const**

Returns the RGB value.

The return type *QRgb* is equivalent to `unsigned int`.

See also: setRgb(), hsv(), qRed(), qBlue() and qGreen().

**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: getHsv() 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 QString & name )**

Sets the RGB value to that of the named color.

The color name may be in one of five formats:

- #RGB (each of R, G and B is a single hex digit)
- #RRGGBB
- #RRRGGGBBB
- #RRRRGGGGBBBB
- A named from the X color database (rgb.txt), e.g. "steelblue" or "gainsboro". These color names also work under Qt for Windows.

**void QColor::setRgb ( QRgb rgb )**

Sets the RGB value to *rgb.*

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().

### 41.3. Related Functions

**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)*.

The return type *QRgb* is equivalent to `unsigned int`.

See also: qRed(), qGreen() and qBlue().

Examples:

desktop/desktop.cpp

**QDataStream & operator<< (QDataStream & s, const QColor & c)**

Writes a color object to the stream.

Serialization format: RGB value serialized as an Q_UINT32.

**QDataStream & operator>> (QDataStream & s, QColor & c)**

Reads a color object from the stream.

**int qGreen (QRgb rgb)**

Returns the green component of the RGB triplet *rgb*.

See also: qRgb() and QColor::green().

Examples:

qmag/qmag.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 qRed (QRgb rgb)**

Returns the red component of the RGB triplet *rgb*.

See also: qRgb() and QColor::red().

Examples:

qmag/qmag.cpp

**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.

### 41.3.1. URL

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

# 42. QColorDialog Class Reference

`#include <qcolordialog.h>`

Inherits QDialog.

## 42.1. Detailed Description

The QColorDialog provides a dialog widget for specifying colors.

This version of Qt provides the static getColor() function that pops up a modal color dialog.

## 42.2. Member Function Documentation

**QColorDialog::~QColorDialog ()**

Destroys the dialog and frees allocated memory.

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

Pops up a modal color dialog letting the user choose a color and returns that color. The color is initially set to *initial*. Returns an invalid color if the user cancels the dialog. All colors allocated by the dialog will be deallocated before this function returns.

### 42.2.1. URL

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

# 43. QColorGroup Class Reference

```
#include <qpalette.h>
```

### 43.1. 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 fourty-two distinct color roles:

1 Foreground (graphics foreground color)

2 Button (general button color)

3 Light (lighter than button color, for shadow effects)

4 Midlight (between Button and Light, for shadow effects)

5 Dark (darker than the button color, for shadow effects)

6 Medium (between button color 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 BrightText (a text color that contrasts to the Dark color)

9 ButtonText (a text color that contrasts to the Button color)

10 Base (used as background color for some widgets). Usually white or another light color.

11 Background (general background color)

12 Shadow (a very dark color used for shadow effects, usually black)

13 Highlight (a color to indicate a selected or highlighted item)

14 HighlightedText (a text color that contrasts to Highlight)

A QPalette contains three color groups.

The current widget color group is returned by QWidget::colorGroup().

See also: QColor and QPalette.

## 43.2. Member Function Documentation

**QColorGroup::QColorGroup ()**

Constructs a color group with all colors set to black.

**QColorGroup::QColorGroup ( const QBrush & foreground, const QBrush & button, const QBrush & light, const QBrush & dark, const QBrush & mid, const QBrush & text, const QBrush & bright_text, const QBrush & base, const QBrush & background )**

Constructs a color group. You can pass either brushes, pixmaps or plain colors for each parameter.

See also: QBrush.

**QColorGroup::QColorGroup ( const QColor & foreground, const QColor & button, const QColor & light, const QColor & dark, const QColor & mid, const QColor & text, const QColor & base )**

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

Constructs a color group with the specified colors. The background color will be set to the button

color.

**QColorGroup::QColorGroup ( const QColorGroup & other )**

Constructs a color group that is an independent copy of another color group.

**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::brightText () const**

Returns the bright text foreground color of the color group.

**const QBrush & QColorGroup::brush ( ColorRole r ) const**

Returns the brush that has been set for color role *r*.

**const QColor & QColorGroup::button () const**

Returns the button color of the color group.

**const QColor & QColorGroup::buttonText () const**

Returns the button text foreground color of the color group.

**const QColor & QColorGroup::color ( ColorRole r ) const**

Returns the color that has been set for color role *r*.

See also: brush().

**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::highlight () const**

Returns the highlight color of the color group.

**const QColor & QColorGroup::highlightedText () const**

Returns the highlighted text 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.

**const QColor& QColorGroup::midlight () const**

Returns the midlight color of the color group. Currently, this is a lightened version of the button color, 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==().

**void QColorGroup::setBrush ( ColorRole r, const QBrush & b )**

Sets the brush used for color role *r* to *b.*

**void QColorGroup::setColor ( ColorRole r, const QColor & c )**

Sets the brush used for color role *r* to a solid color *c.*

**const QColor & QColorGroup::shadow () const**

Returns the shadow color of the color group.

**const QColor & QColorGroup::text () const**

Returns the text foreground color of the color group.

### 43.3. Related Functions

**QDataStream & operator>> (QDataStream & s, QColorGroup & g)**

Reads a color group from the stream.

**QDataStream & operator<< (QDataStream & s, const QColorGroup & g)**

Writes a color group to the stream.

Serialization format:

1 QBrush foreground

2 QBrush button

3 QBrush light

4 QBrush midLight

5 QBrush dark

6 QBrush mid

7 QBrush text

8 QBrush brightText

9 QBrush ButtonText

10 QBrush base

11 QBrush background

12 QBrush shadow

13 QBrush highlight

14 QBrush highlightedText The colors are serialized in the listed order.

### 43.3.1. URL

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

# 44. QComboBox Class Reference

```
#include <qcombobox.h>
```

Inherits QWidget.

### 44.1. 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 QComboBox( 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 QComboBox( FALSE, this, "read-only combo" );
QComboBox * c2 = new QComboBox( 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.

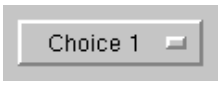
Read-only combo boxes can contain pixmaps as well as texts; the insert() and changeItem() functions are suitably overloaded. If you try to insert a pixmap in a read-write combo box, QComboBox simply ignores you. For read-write combo boxes, the function clearEdit() is provided, to clear the displayed 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.

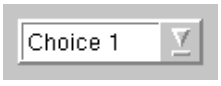
When the user enters a new string in a read-write combo, the widget may or may not insert it, and it can insert it in several locations. The default policy is is `AtBottom,` you can change it using setInsertionPolicy().

It is possible to constrain the input to an editable combo box using QValidator; see setValidator(). By default, all input is accepted.

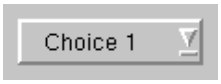
A combo box has a default focusPolicy() of `TabFocus,` i.e. it will not grab focus if clicked. This differs from both Windows and Motif.



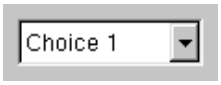

(Motif 1, read-only)




(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:

qmag/qmag.cpp

## 44.2. Member Type Documentation

### QComboBox::Policy

This enum type specifies what QComboBox should do with a new string entered by the user. The following policies are defined:

- `NoInsertion` means not to insert the string in the combo.
- `AtTop` means to insert the string at the top of the combo box.
- `AtCurrent` means to replace the previously selected item with the typed string.
- `AtBottom` means to insert the string at the bottom of the combo box.
- `AfterCurrent` means to to insert the string just after the previously selected item.
- `BeforeCurrent` means to to insert the string just before the previously selected item.

activated() is always emitted, of course.

If inserting the new string would cause the combo box to breach its content size limit, the item at the other end of the list is deleted. The definition of "other end" is implementation-dependent.

## 44.3. 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 QString & 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 is 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 QString & 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().

**QString 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]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QObject.

**void QComboBox::focusInEvent ( QFocusEvent * )** `[virtual protected]`

Handles focus in events.

Reimplemented from QWidget.

**void QComboBox::highlighted ( const QString & 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 QString & t, int index=-1 )**

Inserts a text item at position *index*. The item will be appended if *index* is negative.

**void QComboBox::insertStrList ( const QStrList * list, int index=-1 )**

Inserts the list of strings at the index *index* in the combo box.

This is only for compatibility, as it does not support Unicode strings. See insertStringList().

Examples:

qmag/qmag.cpp

**void QComboBox::insertStrList ( const char ** strings, int numStrings=-1, int index=-1 )**

Inserts the array of ASCII 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 );
```

**void QComboBox::insertStrList ( const QStrList & list, int index=-1 )**

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

**void QComboBox::insertStringList ( const QStringList & list, int index=-1 )**

Inserts the list of strings at the index *index* in the combo box.

**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 * ) [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, no selections appear.

**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 )** `[virtual]`

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 default).

The combo box uses the list of items as candidates for completion.

See also: autoCompletion() and setEditText().

**void QComboBox::setAutoResize ( bool enable )** `[virtual]`

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.

**void QComboBox::setCurrentItem ( int index )** `[virtual]`

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 QString & newText )** `[virtual 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.

**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.

**void QComboBox::setInsertionPolicy ( Policy policy )** `[virtual]`

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 )** `[virtual]`

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 because of the line edit in QComboBox.

**void QComboBox::setMaxCount ( int count )** `[virtual]`

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.

**void QComboBox::setSizeLimit ( int lines )** `[virtual]`

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::setValidator ( const QValidator * v )** `[virtual]`

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.

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().

**QSizePolicy QComboBox::sizePolicy () const** `[virtual]`

Specifies that this widget may stretch horizontally, but is fixed vertically.

**void QComboBox::styleChange ( QStyle & s )** `[virtual protected]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QWidget.

**QString QComboBox::text ( int index ) const**

Returns the text item at a given index, or null string if the item is not a string.

See also: currentText().

**void QComboBox::textChanged ( const QString & string )** `[signal]`

This signal is useful for editable comboboxes. It is emitted whenever the contents of the text entry field changes.

**const 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.

### 44.3.1. URL

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

# 45. QCommonStyle Class Reference

`#include <qcommonstyle.h>`

Inherits QStyle.

Inherited by QMotifStyle and QWindowsStyle.

## 45.1. Detailed Description

Encapsulates common Look and Feel of a GUI.

This abstract class implements some of the widget's look and feel that is common to all GUI styles

provided and shipped as part of Qt.

## 45.2. Member Function Documentation

**QRect QCommonStyle::comboButtonFocusRect ( int x, int y, int w, int h )** `[virtual]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QStyle.

**QRect QCommonStyle::comboButtonRect ( int x, int y, int w, int h )** `[virtual]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QStyle.

**int QCommonStyle::defaultFrameWidth () const** `[virtual]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QStyle.

**void QCommonStyle::drawComboButton ( QPainter * p, int x, int y, int w, int h, const QColorGroup & g, bool sunken = FALSE, bool = FALSE, bool = TRUE, const QBrush * fill = 0 )** `[virtual]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QStyle.

**void QCommonStyle::drawComboButtonMask ( QPainter * p, int x, int y, int w, int h ) [virtual]**

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QStyle.

**void QCommonStyle::drawPushButtonLabel ( QPushButton * btn, QPainter * p ) [virtual]**

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QStyle.

**void QCommonStyle::drawSliderGrooveMask ( QPainter * p, int x, int y, int w, int h, QCOORD, Orientation orient ) [virtual]**

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QStyle.

**void QCommonStyle::drawSliderMask ( QPainter * p, int x, int y, int w, int h, Orientation, bool, bool ) [virtual]**

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QStyle.

**void QCommonStyle::drawTab ( QPainter * p, const QTabBar * tb, QTab * t, bool selected ) [virtual]**

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QStyle.

**void QCommonStyle::drawTabMask ( QPainter * p, const QTabBar *, QTab * t, bool ) [virtual]**

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QStyle.

**void QCommonStyle::getButtonShift ( int & x, int & y ) [virtual]**

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QStyle.

**int QCommonStyle::maximumSliderDragDistance () const** `[virtual]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QStyle.

**int QCommonStyle::popupSubmenuIndicatorWidth ( const QFontMetrics & fm )**

Reimplemented for internal reasons; the API is not affected.

**void QCommonStyle::tabbarMetrics ( const QTabBar * t, int & hframe, int & vframe, int & overlap )** `[virtual]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QStyle.

### 45.2.1. URL

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

# 46. QConnection Class Reference

```
#include <qconnection.h>
```

### 46.1. Detailed Description

The QConnection class is an internal class, used in the signal/slot mechanism.

Do not use this class directly in application programs.

### 46.2. 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.

### 46.2.1. URL

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

# 47. QConstString Class Reference

```
#include <qstring.h>
```

### 47.1. Detailed Description

A QString which uses constant Unicode data.

In order to minimize copying, highly optimized applications can use QConstString to provide a QString-compatible object from existing Unicode data. The Unicode data must exist for the entire lifetime of the QConstString object.

### 47.2. Member Function Documentation

**QConstString::QConstString ( QChar * unicode, uint length )**

Creates a QConstString that uses the first *length* Unicode characters in the array *unicode*. Any attempt to modify copies of the string will cause it to create a copy of the data, thus it remains forever unmodified.

**QConstString::~QConstString ()**

Destroys the QConstString, creating a copy of the data if other strings are still using it.

**const QString& QConstString::string () const**

Returns a constant string referencing the data passed during construction.

#### 47.2.1. URL

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

# 48. QCString Class Reference

```
#include <qcstring.h>
```

Inherits QByteArray.

## 48.1. Detailed Description

The QCString class provides an abstraction of the classic C zero-terminated char array (*char**).

QCString inherits QByteArray, which is defined as QArray<char>.

Since QCString is a QArray, it uses *explicit* sharing with a reference count.

You might use QCString for text that is never exposed to the user, but for text the user sees, you should use QString (which provides implicit sharing, Unicode and other internationalization support).

Note that for the QCString methods that take a *const char* * parameter the results are undefined if the QCString is not zero-terminated. It is legal for the *const char* * parameter to be 0.

A QCString that has not been assigned to anything is *null,* i.e. both the length and data pointer is 0.

A QCString that references the empty string ("", a single '\0' char) is *empty.* Both null and empty QCStrings are legal parameters to the methods. Assigning *const char * 0* to QCString gives a null QCString.

See also: Shared classes

## 48.2. Member Function Documentation

### QCString::QCString ()

Constructs a null string.

See also: isNull().

### QCString::QCString ( const QCString & s )

Constructs a shallow copy *s.*

See also: assign().

### QCString::QCString ( 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().

### QCString::QCString ( 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:

```
QCString str("helloworld",6); // Assigns "hello" to str.
```

If *str* contains a 0 byte within the first *maxlen* bytes, the resulting QCString will be terminated by the

0. If *str* is 0 a null string is created.

See also: isNull().

### QCString::QCString ( 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().

### QCString::operator const char * () const

Returns the string data.

### QCString& QCString::append ( const char * str )

Appends *str* to the string and returns a reference to the string. Equivalent to operator+=().

**int QCString::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 QCString::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 QCString::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().

**QCString QCString::copy () const**

Returns a deep copy of this string.

See also: detach().

**bool QCString::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 QCString::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 QCString::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 QCString::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 QCString::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 QCString::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 QCString::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.

### QCString & QCString::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:

```
QCString s = "Yes";
s.insert( 3, '!');                          // s == "Yes!"
```

See also: remove() and replace().

### QCString & QCString::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.

```
QCString s = "I like fish";
s.insert( 2, "don't ");                     // s == "I don't like fish"
s = "x";
s.insert( 3, "yz" );                        // s == "x  yz"
```

### bool QCString::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().

### bool QCString::isNull () const

Returns TRUE if the string is null, i.e. if data() == 0. A null string is also an empty string.

Example:

```
QCString a;          // a.data() == 0,  a.size() == 0, a.length() == 0
QCString 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().

**QCString QCString::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:

```
QCString s = "Pineapple";
QCString t = s.left( 4 );                    // t == "Pine"
```

See also: right() and mid().

**QCString QCString::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:

```
QCString s("apple");
QCString t = s.leftJustify(8, '.');          // t == "apple..."
```

See also: rightJustify().

**uint QCString::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().

**QCString QCString::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:

```
QCString s("TeX");
QCString t = s.lower();                     // t == "tex"
```

See also: upper().

**QCString QCString::mid ( uint index, uint len=0xffffffff ) 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:

```
QCString s = "Two pineapples";
QCString t = s.mid( 4, 4 );                 // t == "pine"
```

See also: left() and right().

**QCString & QCString::operator+= ( char c )**

Appends *c* to the string and returns a reference to the string.

**QCString& QCString::operator+= ( const char * str )**

Appends *str* to the string and returns a reference to the string.

**QCString & QCString::operator= ( const QCString & s )**

Assigns a shallow copy of *s* to this string and returns a reference to this string.

**QCString & QCString::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().

**QCString & QCString::prepend ( const char * s )**

Prepend *s* to the string. Equivalent to insert(0,s).

See also: insert().

**QCString & QCString::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.

```
QCString s = "Montreal";
s.remove( 1, 4 );
// s == "Meal"
```

See also: insert() and replace().

**QCString & QCString::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 ""
```

**QCString & QCString::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.

```
QCString s = "Say yes!";
s.replace( 4, 3, "NO" );                    // s == "Say NO!"
```

See also: insert() and remove().

**bool QCString::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:

```
QCString s = "resize this string";
s.resize( 7 );                              // s == "resize"
```

See also: truncate().

**QCString QCString::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:

```
QCString s = "Pineapple";
QCString t = s.right( 5 );                  // t == "apple"
```

See also: left() and mid().

**QCString QCString::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:

```
QCString s("pie");
QCString t = s.rightJustify(8, '.');                // t == "....pie"
```

See also: leftJustify().

**bool QCString::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.

**QCString & QCString::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.

**QCString & QCString::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.

**QCString & QCString::setNum ( int n )**

Sets the string to the printed value of *n* and returns a reference to the string.

**QCString & QCString::setNum ( long n )**

Sets the string to the printed value of *n* and returns a reference to the string.

**QCString & QCString::setNum ( short n )**

Sets the string to the printed value of *n* and returns a reference to the string.

**QCString & QCString::setNum ( uint n )**

Sets the string to the printed unsigned value of *n* and returns a reference to the string.

**QCString & QCString::setNum ( ulong n )**

Sets the string to the printed unsigned value of *n* and returns a reference to the string.

**QCString & QCString::setNum ( ushort n )**

Sets the string to the printed unsigned value of *n* and returns a reference to the string.

**QCString & QCString::setStr ( const char * str )**

Makes a deep copy of *str*. Returns a reference to the string.

**QCString QCString::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.

```
QCString s = "  lots\t of\nwhite    space ";

QCString t = s.simplifyWhiteSpace();               // t == "lots of white space"
```

See also: stripWhiteSpace().

### QCString & QCString::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:

```
QCString s;
s.sprintf( "%d - %s", 1, "first" );         // result < 256 chars
QCString 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.

### QCString QCString::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:

```
QCString s = " space ";
QCString t = s.stripWhiteSpace();           // t == "space"
```

See also: simplifyWhiteSpace().

### double QCString::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 QCString::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 QCString::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 QCString::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 QCString::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 QCString::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 QCString::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 QCString::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 QCString::truncate ( uint pos )**

Truncates the string at position *pos*.

Equivalent to calling `resize(pos+1)`.

Example:

```
QCString s = "truncate this string";
s.truncate( 5 );                            // s == "trunc"
```

See also: resize().

**QCString QCString::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:

```
QCString s("TeX");
QCString t = s.upper();                     // t == "TEX"
```

See also: lower().

### 48.3. Related Functions

**bool operator!= (const QCString & 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 QCString & 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 QCString & s2)**

Returns TRUE if the two strings are equal, or FALSE if they are different.

Equivalent to `strcmp(s1,s2) == 0`

**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().

**QDataStream & operator<< (QDataStream & s, const QCString & str)**

Writes a string to the stream.

Output format: [length (Q_UINT32) data...].

**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().

**bool operator< (const char * s1, const QCString & s2)**

Returns TRUE if *s1* is alphabetically less than *s2,* otherwise FALSE.

Equivalent to `strcmp(s1,s2) < 0`

**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.

**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

**bool operator> (const char * s1, const QCString & s2)**

Returns TRUE if *s1* is alphabetically greater than *s2,* otherwise FALSE.

Equivalent to `strcmp(s1,s2) > 0`

**QDataStream & operator>> (QDataStream & s, QCString & str)**

Reads a string from the stream.

**bool operator<= (const QCString & 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 QCString & s1, const QCString & 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 QCString & 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 QCString & s2)**

Returns TRUE if *s1* is alphabetically greater than or equal to *s2,* otherwise FALSE.

Equivalent to `strcmp(s1,s2) >= 0`

**QCString operator+ (const QCString & s, char c)**

Returns the concatenated string of s and c.

**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().

**QCString operator+ (const QCString & s1, const QCString & s2)**

Returns the concatenated string of s1 and s2.

**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().

**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.

**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().

**QCString operator+ (char c, const QCString & s)**

Returns the concatenated string of c and s.

**bool operator>= (const QCString & 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 QCString & 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 QCString & s1, const char * s2)**

Returns TRUE if *s1* is alphabetically greater than *s2,* otherwise FALSE.

Equivalent to `strcmp(s1,s2) > 0`

**QCString operator+ (const QCString & s1, const char * s2)**

Returns the concatenated string of s1 and s2.

**bool operator< (const QCString & s1, const char * s2)**

Returns TRUE if *s1* is alphabetically less than *s2,* otherwise FALSE.

Equivalent to `strcmp(s1,s2) < 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.

**QCString operator+ (const char * s1, const QCString & s2)**

Returns the concatenated string of s1 and s2.

**bool operator== (const QCString & s1, const QCString & s2)**

Returns TRUE if the two strings are equal, or FALSE if they are different.

Equivalent to `strcmp(s1,s2) == 0`

### 48.3.1. URL

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

# 49. QCursor Class Reference

```
#include <qcursor.h>
```

### 49.1. 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
- splitVCursor :vertical splitting
- splitHCursor :horziontal splitting

- pointingHandCursor : a pointing hand

See also: QWidget and GUI Design Handbook: Cursors.

Examples:

cursor/cursor.cpp

## 49.2. 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 QPixmap & pixmap, int hotX=-1, int hotY=-1 )**

Constructs a custom pixmap cursor.

Arguments:

- *pixmap* is the image (usually it should have a mask)
- *hotX* and
- *hotY* define the hot spot of this cursor. If *hotX* is negative, it is set to the pixmap().width()/2. If *hotY* is negative, it is set to the pixmap().height()/2.

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.

Currently, only black-and-white pixmaps can be used.

**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 deinitializes the predefined cursors. This function is called from the QApplication destructor.

See also: initialize().

**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().

### 49.3. 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.

#### 49.3.1. URL

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

# 50. QCustomEvent Class Reference

`#include <qevent.h>`

Inherits QEvent.

### 50.1. 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.**

### 50.2. Member Function Documentation

**QCustomEvent::QCustomEvent ( Type 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 cast to something useful based on the event type. Again, it is not recommended to use custom events unless you are implementing Qt kernel enhancements.

#### 50.2.1. URL

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

# 51. QDataPump Class Reference

`#include <qasyncio.h>`

Inherits QObject.

## 51.1. 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).

## 51.2. 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*.

#### 51.2.1. URL

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

# 52. QDataSink Class Reference

`#include <qasyncio.h>`

Inherits QAsyncIO.

## 52.1. 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.

## 52.2. 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.

### 52.2.1. URL

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

# 53. QDataSource Class Reference

`#include <qasyncio.h>`

Inherits QAsyncIO.

Inherited by QIODeviceSource.

## 53.1. 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.

## 53.2. 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.

### 53.2.1. URL

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

# 54. QDataStream Class Reference

```
#include <qdatastream.h>
```

### 54.1. 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.

### 54.2. Member Function Documentation

#### QDataStream::QDataStream ()

Constructs a data stream that has no IO device.

#### QDataStream::QDataStream ( QIODevice * d )

Constructs a data stream that uses the IO device *d*.

#### QDataStream::QDataStream ( QByteArray a, int mode )

Constructs a data stream that operates on a byte array through 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 () [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.

**bool QDataStream::atEnd () 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().

**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**

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

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::setVersion ( int v )**

Sets the version number of the data serialization format.

In Qt 2.0, the datastream serialization format of many Qt classes was changed. In order to read data that was created with the QDatastream of Qt 1.x, call this function with the version number \v set to 1.

See also: version().

**void QDataStream::unsetDevice ()**

Unsets the IO device. This is the same as calling setDevice( 0 ).

See also: device() and setDevice().

**int QDataStream::version () const**

Returns the version number of the data serialization format. In Qt 2.0, this number is 2.

See also: setVersion().

**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().

### 54.2.1. URL

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

# 55. QDate Class Reference

`#include <qdatetime.h>`

## 55.1. Detailed Description

The QDate class provides date functions.

The QDate is based on the Gregorian (modern western) calendar. This calendar was first adopted by England in 1752, and subsequently by most other western countries. QDate computes the wrong day of week for all dates earlier than 14th September 1752, and the day it computes is correct only in some countries for years between 1752 and 1923.

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

See also: QTime and QDateTime.

## 55.2. 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().

**QString 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().

**QString 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().

### 55.3. 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.

### 55.3.1. URL

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

# 56. QDateTime Class Reference

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

### 56.1. 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.

### 56.2. 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 negative 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) );

qDebug( "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.

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

**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 03:40:13 1998".

### 56.3. 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.

#### 56.3.1. URL

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

# 57. QDialog Class Reference

`#include <qdialog.h>`

Inherits QWidget.

Inherited by QColorDialog, QFileDialog, QFontDialog, QMessageBox, QTabDialog and QWizard.

## 57.1. 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, QString name );
};
Modal::Modal( QWidget *parent, QString 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 an additional meaning for modal dialogs. A modal 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 toplevel widgets. The only difference is that they have the default button mechanism.

See also: QTabDialog, QWidget, QSemiModal and GUI Design Handbook: Dialogs, Standard.

## 57.2. 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 construct a modal dialog.

The *f* argument is the widget flags, which can be used to customize the window frame style.

A dialog is always a top level widget. The optional parent, however, will know about this child and also delete it on destruction. Furthermore, the window system will be able to tell that both the dialog and the parent belong together. This works for Windows and also some X11 window managers, that will for instance provide a common taskbar entry in that case.

It is recommended to pass a parent.

**QDialog::~QDialog ()**

Destroys the QDialog and all its children.

**void QDialog::accept ()** `[virtual 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 also 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 ()**

For modal dialogs: Starts the dialog and returns the result code.

Equivalent to calling show(), then result().

This function is very useful for modal dialogs, but makes no sense for modeless dialog. It enters a new local event loop. The event loop is terminated when the dialog is hidden, usually by calling done().

A warning message is printed if you call this function for a modeless dialog.

See also: show() and result().

Examples:

qdir/qdir.cpp
wizard/main.cpp

**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 ()** `[virtual 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.

**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 )** `[virtual]`

Reimplements QWidget::setGeometry() for internal purposes.

Reimplemented from QWidget.

**void QDialog::setGeometry ( int x, int y, int w, int h )** `[virtual]`

Reimplements QWidget::setGeometry() for internal purposes.

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

Reimplemented from QWidget.

### 57.2.1. URL

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

# 58. QDict Class Reference

`#include <qdict.h>`

Inherits QGDict.

## 58.1. Detailed Description

The QDict class is a template class that provides a dictionary based on `QString` keys.

QDict is implemented as a template 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 QString keys, which are Unicode strings. If you want to use non-Unicode, plain 8-bit `char*` keys, use the QAsciiDict template. A QDict has the same performace as a QAsciiDict.

The dictionary has very fast insertion and lookup.

Example:

```
#include <qdict.h>
#include <stdio.h>
void main()
{
  // Creates a dictionary that maps QString ==> 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 `QString` keys to `char*` items. Note that the mapping is case insensitive (specified in the constructor). QDict implements the [] operator to lookup an item.

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 QString ==> 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, QAsciiDict, QIntDict, QPtrDict and Collection Classes

## 58.2. 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 )

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. 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 comparison includes the whole Unicode alphabeth.

**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 QString & 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 QString & 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 QString & 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 QString & 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 QString & 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 qDebug().

**type * QDict::take ( const QString & 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().

### 58.2.1. URL

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

# 59. QDictIterator Class Reference

`#include <qdict.h>`

Inherits QGDictIterator.

## 59.1. Detailed Description

The QDictIterator class provides an iterator for QDict collections.

QDictIterator is implemented as a template 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 QString ==> 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.

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

## 59.2. 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.

**QString 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.

### 59.2.1. URL

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

# 60. QDir Class Reference

```
#include <qdir.h>
```

### 60.1. 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() )
    qWarning( "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"
    qWarning( "Cannot find the \"/tmp\" directory" );
} else {
    QFile f( d.filePath("ex1.txt") );       // "/tmp/ex1.txt"
    if ( !f.open(IO_ReadWrite) )
        qWarning( "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
    }
}
```

## 60.2. Member Function Documentation

### QDir::QDir ()

Constructs a QDir pointing to the current directory.

See also: currentDirPath().

**QDir::QDir ( const QString & path, const QString & nameFilter = QString::null, 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 ( const QDir & d )**

Constructs a QDir that is a copy of the given directory.

See also: operator=().

**QDir::~QDir ()** `[virtual]`

Destroys the QDir and cleans up.

**QString QDir::absFilePath ( const QString & fileName, bool acceptAbsPath = TRUE ) const** `[virtual]`

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** `[virtual]`

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** `[virtual]`

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 QString & dirName, bool acceptAbsPath = TRUE )** `[virtual]`

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() )
                qWarning( "Cannot cd into \"%s\".", (char*)d.absFilePath("c++") );
            else
                qWarning( "Cannot create directory \"%s\"\n"
                          "A file named \"c++\" already exists in \"%s\"",
                          (const char *)d.absFilePath("c++"),
                          (const char *)d.path() );
            return;
        } else {
            qWarning( "Creating directory \"%s\"",
                      (const char *) d.absFilePath("c++") );
            if ( !d.mkdir( "c++" ) ) {
                qWarning("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 ()** `[virtual]`

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 QString & 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 QString & 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 ()** `[virtual]`

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** `[virtual]`

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. "/").

**QStrList QDir::encodedEntryList ( const QString & nameFilter, int filterSpec = DefaultFilter, int sortSpec = DefaultSort ) const** `[virtual]`

This function is included to easy porting from Qt 1.x to Qt 2.0, it is the same as entryList(), but encodes the filenames as 8-bit strings using QFile::encodedName().

It is more efficient to use entryList().

**QStrList QDir::encodedEntryList ( int filterSpec = DefaultFilter, int sortSpec = DefaultSort ) const** `[virtual]`

This function is included to easy porting from Qt 1.x to Qt 2.0, it is the same as entryList(), but encodes the filenames as 8-bit strings using QFile::encodedName().

It is more efficient to use entryList().

**const QFileInfoList * QDir::entryInfoList ( const QString & nameFilter, int filterSpec = DefaultFilter, int sortSpec = DefaultSort ) const** `[virtual]`

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.

The returned pointer is a const pointer to a QFileInfoList. The list is owned by the QDir object and

will be reused on the next call to entryInfoList() for the same QDir instance. If you want to keep the

entries of the list after a subsequent call to this function you will need to copy them.

See also: entryList(), setNameFilter(), setSorting() and setFilter().

**const QFileInfoList * QDir::entryInfoList ( int filterSpec = DefaultFilter, int sortSpec = DefaultSort ) const** `[virtual]`

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.

The returned pointer is a const pointer to a QFileInfoList. The list is owned by the QDir object and

will be reused on the next call to entryInfoList() for the same QDir instance. If you want to keep the

entries of the list after a subsequent call to this function you will need to copy them.

See also: entryList(), setNameFilter(), setSorting() and setFilter().

**QStringList QDir::entryList ( const QString & nameFilter, int filterSpec = DefaultFilter, int sortSpec = DefaultSort ) const** `[virtual]`

Returns a list of the names of all files and directories in the directory indicated by the setSorting(), setFilter() and setNameFilter() specifications.

The the filter and sorting specifications can be overridden using the *nameFilter, filterSpec* and

*sortSpec* arguments.

Returns and empty list if the directory is unreadable or does not exist.

See also: entryInfoList(), setNameFilter(), setSorting(), setFilter() and encodedEntryList().

**QStringList QDir::entryList ( int filterSpec = DefaultFilter, int sortSpec = DefaultSort ) const** `[virtual]`

Returns a list of the names of all files and directories in the directory indicated by the setSorting(), setFilter() and setNameFilter() specifications.

The the filter and sorting specifications can be overridden using the *filterSpec* and *sortSpec*

arguments.

Returns an empty list if the directory is unreadable or does not exist.

See also: entryInfoList(), setNameFilter(), setSorting(), setFilter() and encodedEntryList().

**bool QDir::exists () const** `[virtual]`

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 QString & name, bool acceptAbsPath = TRUE )** `[virtual]`

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 QString & fileName, bool acceptAbsPath = TRUE ) const** `[virtual]`

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 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** `[virtual]`

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().

**bool QDir::isRelative () const** `[virtual]`

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 QString & path )** `[static]`

Returns TRUE if the path is relative, FALSE if it is absolute.

See also: isRelative().

**bool QDir::isRoot () const** `[virtual]`

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() )
    qWarning( "It IS a root link!" );
```

See also: root() and rootDirPath().

**bool QDir::match ( const QString & filter, const QString & 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 QString & dirName, bool acceptAbsPath = TRUE ) const** `[virtual]`

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().

**QString QDir::nameFilter () const**

Returns the string set by setNameFilter()

See also: setNameFilter().

**bool QDir::operator!= ( const QDir & d ) const** `[virtual]`

Returns TRUE if the *d* and this dir have different path or different sort/filter settings, otherwise FALSE.

**QDir & QDir::operator= ( const QString & path )**

Sets the directory path to be the given path.

**QDir & QDir::operator= ( const QDir & d )**

Makes a copy of d and assigns it to this QDir.

**bool QDir::operator== ( const QDir & d ) const** `[virtual]`

Returns TRUE if the *d* and this dir have the same path and all sort and filter settings are equal, otherwise FALSE.

**QString 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().

**QString QDir::path () const** `[virtual]`

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 QString & fileName, bool acceptAbsPath = TRUE )** `[virtual]`

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 QString & name, const QString & newName, bool acceptAbsPaths = TRUE )** `[virtual]`

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 QString & dirName, bool acceptAbsPath = TRUE ) const** `[virtual]`

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 QString & path )** `[static]`

Sets the the current directory. Returns TRUE if successful.

**void QDir::setFilter ( int filterSpec )** `[virtual]`

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 )** `[virtual]`

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 QString & nameFilter )** `[virtual]`

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 QString & path )** `[virtual]`

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 )** `[virtual]`

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.

### 60.2.1. URL

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

# 61. QDoubleValidator Class Reference

`#include <qvalidator.h>`

Inherits QValidator.

## 61.1. 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.

## 61.2. 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().

**QValidator::State QDoubleValidator::validate( QString & input, int & ) const**

Returns *Acceptable* if *input* contains a number in the legal range and format, *Valid* if it contains another number, a number with too many digits after the decimal point or is empty, and *Invalid* if *input* is not a number.

### 61.2.1. URL

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

# 62. QDragEnterEvent Class Reference

`#include <qevent.h>`

Inherits QDragMoveEvent.

## 62.1. Detailed Description

The event sent to widgets when a drag-and-drop first drags onto it.

This event is always immediate followed by a QDragMoveEvent, thus you need only respond to one or the other event. Note that this class inherits most of its functionality from QDragMoveEvent, which in turn inherits most of its functionality from QDropEvent.

See also: QDragLeaveEvent, QDragMoveEvent and QDropEvent.

Examples:

desktop/desktop.cpp

## 62.2. Member Function Documentation

### QDragEnterEvent::QDragEnterEvent ( const QPoint & pos )

Constructs a QDragEnterEvent entering at the given point. Note that QDragEnterEvent constructed outside of the Qt internals will not work - they currently rely on internal state.

#### 62.2.1. URL

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

# 63. QDragLeaveEvent Class Reference

`#include <qevent.h>`

Inherits QEvent.

## 63.1. Detailed Description

The event sent to widgets when a drag-and-drop leaves it.

This event is always preceded by a QDragEnterEvent and a series of QDragMoveEvent. It is not sent if a QDropEvent is sent instead.

See also: QDragEnterEvent, QDragMoveEvent and QDropEvent.

Examples:

desktop/desktop.cpp

## 63.2. Member Function Documentation

### QDragLeaveEvent::QDragLeaveEvent ()

Constructs a QDragLeaveEvent. Note that QDragLeaveEvent constructed outside of the Qt internals will not work - they currently rely on internal state.

#### 63.2.1. URL

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

# 64. QDragMoveEvent Class Reference

`#include <qevent.h>`

Inherits QDropEvent.

Inherited by QDragEnterEvent.

## 64.1. Detailed Description

Event sent as a drag-and-drop is in progress.

When a widget 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.

Note that this class inherits most of its functionality from QDropEvent.

Examples:

desktop/desktop.cpp

## 64.2. Member Function Documentation

### QDragMoveEvent::QDragMoveEvent ( const QPoint & pos, Type type=DragMove )

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 ( const QRect & r )

The same as accept(), but also notifies that future moves will also be acceptable if they remain within the rectangle *r* on the widget - this can improve performance, but may also be ignored by the underlying system.

If the rectangle is empty, then drag move events will be sent continuously. This is useful if the source is scrolling in a timer event.

**QRect QDragMoveEvent::answerRect () const**

{ return rect; }

Returns the rectangle for which the acceptance of the move event applies.

**void QDragMoveEvent::ignore ( const QRect & r )**

The opposite of accept(const QRect&).

### 64.2.1. URL

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

# 65. QDragObject Class Reference

`#include <qdragobject.h>`

Inherits QMimeSource and QObject.

Inherited by QImageDrag, QStoredDrag and QTextDrag.

## 65.1. Detailed Description

The QDragObject encapsulates MIME-based information transfer.

QDragObject is the base class for all data that needs to be transferred between and within applications, both for drag-and-drop and for the clipboard.

See the Drag-and-drop documentation for an overview of how to provide drag-and-drop in your application.

See the QClipboard documentation for an overview of how to provide cut-and-paste in your application.

## 65.2. 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 ()** `[virtual]`

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.

The function returns TRUE if the caller should delete the original copy of the dragged data (but also note target()).

Note that if the drag contains *references* to information (eg. file names is a QUriDrag are references) then the return value should always be ignored, as the target is expected to manipulate the referred-to content directly. On X11 the return value should always be correct anyway, but on Windows this is not necessarily the case (eg. the file manager starts a background process to move files, so the source *must not* delete the files!).

**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).

Normally one of simpler drag(), dragMove(), or dragCopy() functions would be used instead.

**Warning:** in Qt 1.x, drag operations all return FALSE. This will change in later versions - the functions are provided in this way to assist preemptive development - code both move and copy with Qt 1.x to be prepared.

**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.

**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.

**void QDragObject::setPixmap ( QPixmap pm, QPoint hotspot )** `[virtual]`

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 splendor. 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 )** `[virtual]`

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.

**void QDragObject::setTarget ( QWidget * t )** `[static]`

For internal use only.

**QWidget * QDragObject::source ()**

Returns a pointer to the drag source where this object originated.

**QWidget * QDragObject::target ()** `[static]`

After the drag completes, this function will return the QWidget which received the drop, or 0 if the data was dropped on some other program.

This can be useful for detecting the case where drag-and-drop is to and from the same widget.

### 65.2.1. URL

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

# 66. QDropEvent Class Reference

`#include <qevent.h>`

Inherits QEvent and QMimeSource.

Inherited by QDragMoveEvent.

## 66.1. Detailed Description

Event sent when a drag-and-drop is completed.

When a widget 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.

Examples:

desktop/desktop.cpp

## 66.2. Member Type Documentation

### QDropEvent::Action

This type describes the action which a source requests that a target perform with dropped data. The values are:

- `Copy` - the default action. The source simply users the data provided in the operation.
- `Link`. The source should somehow create a link to the location specified by the data.
- `Move`. The source should somehow move the object from the location specified by the data to a new location.
- `Private`. The target has special knowledge of the MIME type, which the source should respond to similar to a Copy.
- `UserAction`. The source and target can co-operate using special actions. This feature is not supported in Qt at this time.

The Link and Move actions only makes sense if the data is a reference, such as text/uri-list file lists

## 66.3. Member Function Documentation

**void QDropEvent::acceptAction ( bool y=TRUE=TRUE )**

Call this to indicate that the action described by action() is accepted, not merely the default copy action. If you call acceptAction(TRUE), there is no need to also call accept(TRUE).

**Action QDropEvent::action () const**

Returns the action which the target is requesting be performed with the data. If your application understands the action and can process the supplied data, call acceptAction(); if your application can process the supplied data but can only perform the Copy action, call accept().

**QByteArray QDropEvent::data ( const char * f ) const**

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

Use QDropEvent::encodedData().

**QByteArray QDropEvent::encodedData ( const char * format ) const** `[virtual]`

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: format() and QByteArray::size().

Reimplemented from QMimeSource.

**const char* QDropEvent::format ( int n = 0 ) const** `[virtual]`

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().

Reimplemented from QMimeSource.

**void QDropEvent::ignore ()**

The opposite of accept().

**bool QDropEvent::isAccepted () const**

Returns TRUE if the drop target accepts the event.

**const QPoint& QDropEvent::pos () const**

Returns the position where the drop was made.

**bool QDropEvent::provides ( const char * mimeType ) const** `[virtual]`

Returns TRUE if this event provides format *mimeType* or FALSE if it does not.

See also: data().

Reimplemented from QMimeSource.

**void QDropEvent::setAction ( Action a )**

Sets the action. This is used internally, you should not need to call this in your code - the *source* decides the action, not the target.

**QWidget* QDropEvent::source () const**

If the source of the drag operation is a widget in this application, this function returns that source, otherwise 0. The source of the operation is the first parameter to to drag object subclass.

This is useful if your widget needs special behavior when dragging to itelf, etc.

See QDragObject::QDragObject() and subclasses.

### 66.3.1. URL

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

# 67. QDropSite Class Reference

`#include <qdropsite.h>`

### 67.1. Detailed Description

Provides nothing and does nothing.

This class exists only so that old code will not break. It does nothing. If your code uses it, you can safely delete it.

It was used in Qt 1.x to do some drag and drop; that has since been folded into QWidget.

For detailed information about drag-and-drop, see the QDragObject class.

See also: QDragObject, QTextDrag and QImageDrag.

Examples:

desktop/desktop.cpp

### 67.2. Member Function Documentation

**QDropSite::QDropSite ( QWidget * self )**

Constructs a QDropSite to handle events for the widget *self*.

Pass `this` as the *parent* parameter. This enables dropping by calling QWidget::setAcceptDrops(TRUE).

**QDropSite::~QDropSite ()** `[virtual]`

Destructs the drop site.

#### 67.2.1. URL

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

# 68. QEvent Class Reference

`#include <qevent.h>`

Inherits Qt.

Inherited by QChildEvent, QCloseEvent, QCustomEvent, QDragLeaveEvent, QDragResponseEvent, QDropEvent, QFocusEvent, QHideEvent, QKeyEvent, QMouseEvent, QMoveEvent, QPaintEvent, QResizeEvent, QShowEvent, QTimerEvent and QWheelEvent.

## 68.1. 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 received 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 describe the particular event.

See also: QObject::event(), QObject::installEventFilter(), QWidget::event(), QApplication::sendEvent(), QApplcation::postEvent() and QApplication::processEvents().

## 68.2. Member Function Documentation

### QEvent::QEvent ( Type type )

Contructs an event object with a *type*. The file qevent.h lists all event types.

**QEvent::~QEvent** () `[virtual]`

Destroys the event. Reports an error if the event is posted for dispatch to an object.

### 68.2.1. URL

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

# 69. QFile Class Reference

`#include <qfile.h>`

Inherits QIODevice.

## 69.1. 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.

## 69.2. Member Function Documentation

**QFile::QFile ()**

Constructs a QFile with no name.

**QFile::QFile ( const QString & 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.

Some "write-behind" filesystems may report an unspecified error on closing the file. These errors only indiciate that something may have gone wrong since the previous open(). In such a case status() reports IO_UnspecifiedError after close(), otherwise IO_Ok.

See also: open() and flush().

Reimplemented from QIODevice.

**QString QFile::decodeName ( const QCString & localFileName )** `[static]`

This does the reverse of QFile::encodeName().

See also: setDecodingFunction().

**QCString QFile::encodeName ( const QString & fileName )** `[static]`

When you use QFile, QFileInfo, and QDir to access the filesystem with Qt, you can use Unicode filenames. On Unix, these filenames are converted to an 8-bit encoding. If you want to do your own file I/O on Unix, you should convert the filename using this function. On Windows NT, Unicode filenames are supported directly in the filesystem and this function should be avoided. On Windows 95, non-Latin1 locales are not supported at this time.

By default, this function converts to the local 8-bit encoding determined by the user's locale. This is sufficient for filenames that the user chooses. Filenames hard-coded into the application should only use 7-bit ASCII filename characters.

The conversion scheme can be changed using setEncodingFunction(). This might be useful if you wish to give the user an option to store in filenames in UTF-8, etc., but beware that such filenames would probably then be unrecognizable when seen by other programs.

See also: decodedName().

**bool QFile::exists () const**

Returns TRUE if this file exists, otherwise FALSE.

See also: name().

**bool QFile::exists ( const QString & 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.

**QString 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().

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 ( QString & s, uint maxlen )**

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.

Note that the string is read as plain Latin1 bytes, not Unicode.

See also: readBlock() and QTextStream::readLine().

**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 QString & fileName )** `[static]`

Removes the file *fileName*. Returns TRUE if successful, otherwise FALSE.

**void QFile::setDecodingFunction ( DecoderFn f )** `[static]`

Sets the function for decoding 8-bit filenames. The default uses the locale-specific 8-bit encoding.

See also: encodeName() and decodedName().

**void QFile::setEncodingFunction ( EncoderFn f )** `[static]`

Sets the function for encoding Unicode filenames. The default encodes in the locale-specific 8-bit encoding.

See also: encodeName().

**void QFile::setName ( const QString & 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.

### 69.2.1. URL

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

# 70. QFileDialog Class Reference

```
#include <qfiledialog.h>
```

Inherits QDialog.

### 70.1. 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(),

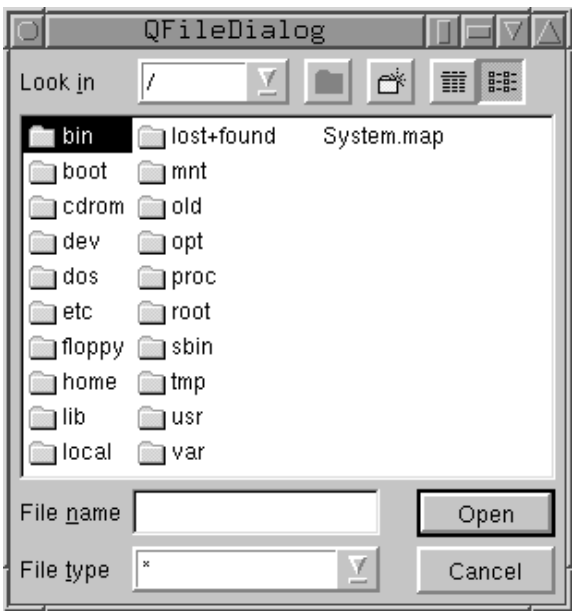
which may be used like this:

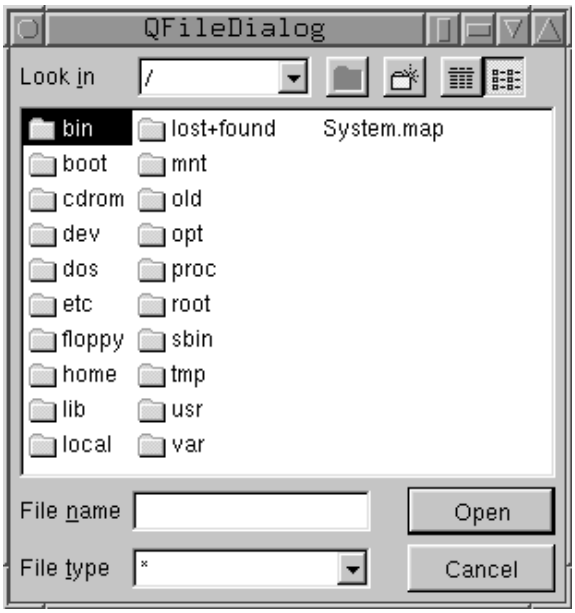
```
QString s( QFileDialog::getOpenFileName() );

if ( s.isNull() )

    return;

open( s ); // open() being your function to read the file
```






See also: QPrintDialog.

Examples:

qdir/qdir.cpp
movies/main.cpp

## 70.2. Member Type Documentation

### QFileDialog::Mode

This enum type is used to set and read QFileDialog's operating mode. The defined values are:

- `AnyFile` - Return the name of any file, whether existing or not.
- `ExistingFile` - Return the name of a single, existing, file.
- `Directory` - Return the name of a directory.
- `ExistingFiles` - Return the names of zero or more existing files.

Examples:

qdir/qdir.cpp

### 70.3. 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 QString & dirName, const QString & filter = QString::null, 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 QString & ) [signal]**

This signal is emitted when the user has selected a new directory.

**QString 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 QString & )** `[signal]`

This signal is emitted when the user highlights a file.

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

This signal is emitted when the user selects a file.

**QString QFileDialog::getExistingDirectory ( const QString & dir = QString::null, 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 QString & startWith = QString::null, const QString & filter= QString::null, 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 QString::null, 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( QString::null, "*.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().

**QStringList QFileDialog::getOpenFileNames ( const QString & filter= QString::null, const QString & dir = QString::null, 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.

Example:

```
QStringList s( QFileDialog::getOpenFileNames() );
// do something with the files in s.
```

**QString QFileDialog::getSaveFileName ( const QString & startWith = QString::null, const QString & filter= QString::null, 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 QString::null, 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( QString::null, "*.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().

Examples:

qdir/qdir.cpp

**void QFileDialog::setDir ( const QDir & dir )**

Sets a directory path for the file dialog.

See also: dir().

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

Sets a directory path string for the file dialog.

See also: dir().

**void QFileDialog::setFilter ( const QString & 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 QStringList & )** `[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), `AnyFile` (any valid file name is accepted) or `ExistingFiles` (like `ExistingFile`, but multple files may be selected)

See also: mode().

Examples:

qdir/qdir.cpp

**void QFileDialog::setSelection ( const QString & filename )**

Sets the default selection to *filename*. If *filename* is absolute, setDir() is also called.

Examples:

qdir/qdir.cpp

### 70.3.1. URL

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

# 71. QFileIconProvider Class Reference

`#include <qfiledialog.h>`

Inherits QObject.

### 71.1. 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.

### 71.2. 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.

#### 71.2.1. URL

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

# 72. QFileInfo Class Reference

```
#include <qfileinfo.h>
```

### 72.1. 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:

i18n/main.cpp
dirview/main.cpp

### 72.2. Member Function Documentation

**QFileInfo::QFileInfo ()**

Constructs a new empty QFileInfo.

**QFileInfo::QFileInfo ( const QDir & d, const QString & 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 QString & 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(QString, ), isRelative(), QDir::setCurrent() and QDir::isRelativePath().

**QFileInfo::QFileInfo ( const QFileInfo & fi )**

Constructs a new QFileInfo that is a copy of *fi*.

**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().

**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:

i18n/main.cpp

**QString QFileInfo::extension ( bool complete = TRUE ) const**

Returns the extension name of the file.

If *complete* is TRUE (the default), extension() returns the string of all characters in the file name after (but not including) the first '.' character. For a file named "archive.tar.gz" this returns "tar.gz".

If *complete* is FALSE, extension() returns the string of all characters in the file name after (but not including) the last '.' character. For a file named "archive.tar.gz" this returns "gz".

Example:

```
QFileInfo fi( "lex.yy.c" );
QString ext = fi.extension();              // ext = "yy.c"
QString ext = fi.extension( FALSE );       // ext = "c"
```

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().

**QString 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().

Examples:

dirview/main.cpp

**QString 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().

**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().

**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().

**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().

**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().

**QFileInfo & QFileInfo::operator= ( const QFileInfo & fi )**

Makes a copy of *fi* and assigns it to this QFileInfo.

**QString 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 ) )
    qWarning( "Tonsils can be changed by me, and the group can read them.");
if ( fi.permission( QFileInfo::WriteGroup | QFileInfo::WriteOther ) )
    qWarning( "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 QString & fileName )**

Sets the file to obtains 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 QString & 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()
{
    QString absolute = "/liver/aorta";
    QString 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().

**uint QFileInfo::size () const**

Returns the file size in bytes, or 0 if the file does not exist if the size cannot be fetched.

### 72.2.1. URL

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

# 73. QFocusData Class Reference

```
#include <qfocusdata.h>
```

### 73.1. 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().

### 73.2. 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.

### 73.2.1. URL

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

# 74. QFocusEvent Class Reference

`#include <qevent.h>`

Inherits QEvent.

### 74.1. 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:

qmag/qmag.cpp

### 74.2. Member Function Documentation

**QFocusEvent::QFocusEvent ( Type type )**

Constructs a focus event object.

The *type* parameter must be either *QEvent::FocusIn* or *QEvent::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.

#### 74.2.1. URL

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

# 75. QFont Class Reference

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

### 75.1. 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 application's 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 (also known as fixed-width) 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 setRawName() 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 using 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, QApplication::setFont() 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 for the current locale. \endlink 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 and for more general information on encodings, see or Roman Czyborra's font encodings.

See also: QFontMetrics, QFontInfo, QApplication::setFont(), QWidget::setFont(),

QPainter::setFont(), QFont::StyleHint, QFont::CharSet and QFont::Weight.

Examples:

showimg/main.cpp
grapher/grapher.cpp
xform/xform.cpp
drawdemo/drawdemo.cpp
desktop/desktop.cpp
scrollview/scrollview.cpp
movies/main.cpp
picture/picture.cpp
hello/main.cpp

## 75.2. Member Type Documentation

### QFont::CharSet

The following character set encodings are available:

- `QFont::ISO_8859_1` - Latin1 , common in much of Europe
- `QFont::ISO_8859_2` - Latin2, Central and Eastern European character set
- `QFont::ISO_8859_3` - Latin3, less common European character set
- `QFont::ISO_8859_4` - Latin4, less common European character set
- `QFont::ISO_8859_5`, Cyrillic
- `QFont::ISO_8859_6`, Arabic
- `QFont::ISO_8859_7`, Greek
- `QFont::ISO_8859_8`, Hebrew
- `QFont::ISO_8859_9`, Turkish
- `QFont::ISO_8859_10..15`, other ISO 8859 characters sets
- `QFont::KOI8R` - KOI8-R, Cyrillic, defined in RFC 1489.
- `QFont::AnyCharSet` - whatever is handiest.

### QFont::StyleHint

Style hints are 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 value of `AnyStyle` leaves the task of finding a good default family to the font

matching algorithm.

The other available style hints are `QFont::SansSerif, QFont::TypeWriter, QFont::OldEnglish, QFont::System`

**QFont::Weight**

Contains the predefined font weights:

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

### 75.3. Member Function Documentation

**QFont::QFont ()**

Constructs a font object that refers to the default font.

See also: QApplication::setFont() and QApplication::font().

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

Constructs a font object with the specified *family, pointSize, weight* and *italic* settings. 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 QString & 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 ( const QFont & font )**

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

**QFont::~QFont ()**

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 QApplication default font.

This function will be removed in a future version of Qt. Please use QApplication::font() instead.

**bool QFont::dirty () const** `[protected]`

Returns TRUE if the font attributes have been changed and the font has to be (re)loaded, or FALSE if no changes have been made.

**QString QFont::encodingName ( CharSet cs )** `[static]`

Returns the encoding name of a character set, e.g. QFont::ISO_8859_1 returns "iso8859-1" and QFont::Unicode returns "iso10646".

**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

**QString 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 );
qDebug( "Font family requested is    : \"%s\"", font.family() );
qDebug( "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().

**void QFont::initialize ()** `[static]`

Internal function that initializes the font system.

**void QFont::insertSubstitution ( const QString & familyName, const QString & 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, i.e. 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: QMap.

**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::locale_init () `[static]`**

Internal function that uses locale information to find the preferred character set of loaded fonts.

**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 rawMode() 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 rawMode() is enabled for both fonts, then only the family fields are compared.

See also: operator!=().

**int QFont::pixelSize () const**

Returns the logical pixel height of characters in the font if shown on the screen.

**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 );

qDebug( "Font size requested is    : %d", font.pointSize() );

qDebug( "Font size actually used is: %d", info.pointSize() );
```

See also: setPointSize() and deciPointSize().

**float QFont::pointSizeFloat () const**

Returns the height of characters in the font in points (1/72 inch).

See also: pointSize().

**bool QFont::rawMode () const**

Returns the value set by setRawMode().

See also: setRawMode().

**QString QFont::rawName () const**

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

See also: setRawName().

**void QFont::removeSubstitution ( const QString & 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
    qFatal( "Cannot find a Latin 1 Times font" );
```

See also: charSet(), QFontInfo and font matching

**void QFont::setDefaultFont ( const QFont & f )** `[static]`

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

Sets the QApplication default font.

This function will be removed in a future version of Qt. Please use QApplication::setFont() instead.

**void QFont::setFamily ( const QString & 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

**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

**void QFont::setPixelSize ( int pixelSize )**

Sets the logical height of characters in the font if shown on the screen.

**void QFont::setPixelSizeFloat ( float pixelSize )**

Sets the logical pixel height of characters in the font if shown on the screen.

**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

**void QFont::setPointSizeFloat ( float pointSize )**

Sets the point size to *pointSize*. The point size must be greater than zero. The requested precision may not be achieved on all platforms.

**void QFont::setRawName ( const QString & name )**

Sets a font by its system specific name. The function is in particular useful under X, where system font settings ( for example X resources) are usually available as XLFD (X Logical Font Description) only. You can pass an XLFD as *name* to this function.

The big advantage over Qt-1.x' concept of raw fonts is, that a font after setRawName() is still a full-featured QFont. It can be queried (for example with italic()) or modified (for example with setItalic() ) and is therefore also suitable as a basis font for rendering rich text.

If Qt's internal font database cannot resolve the raw name, the font becomes a raw font with *name* as family.

See also: rawName(), setRawMode() and setFamily().

Bugs and limitations:

- Doesn't handle wildcards in XLFDs well, only complete descriptions. Aliases (file `fonts.alias` in the font directory on X11) are not supported.

**void QFont::setStrikeOut ( bool enable )**

Sets strike out on or off.

See also: strikeOut(), QFontInfo and font matching

**void QFont::setStyleHint ( StyleHint hint )**

Sets the style hint.

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 the example below 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 );        // preferred family is Bavaria
```

```
    font.setStyleHint( QFont::Serif );  // can also use any serif font
    push.setFont( font );
    return app.exec( &push );
}
```

See also: QFont::StyleHint, 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.

**void QFont::setWeight ( int weight )**

Sets the weight (or boldness), which should be a value from the QFont::Weight enumeration.

Example:

```
QFont font( "courier" );
font.setWeight( QFont::Bold );
```

Strictly speaking you can use all values in the range [0,99] (where 0 is ultralight and 99 is extremely black), but there is perhaps asking too much of the underlying window system.

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

**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().

**QString QFont::substitute ( const QString & familyName )** `[static]`

Returns the font family name to be used whenever *familyName* is specified. 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().

**QStringList QFont::substitutions ()** `[static]`

Returns a sorted list of substituted family names.

See also: insertSubstitution(), removeSubstitution() and substitute().

**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.

### 75.4. 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.

#### 75.4.1. URL

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

# 76. QFontDialog Class Reference

`#include <qfontdialog.h>`

Inherits QDialog.

## 76.1. Detailed Description

The QFontDialog provides a dialog widget for selecting a text font

Used for allowing the user to select a font among the available fonts on the underlying window

system.

## 76.2. Member Function Documentation

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

Event filter to make up, down, pageup and pagedown work correctly in the line edits.

Reimplemented from QObject.

**QListBox * QFontDialog::familyListBox () const** `[protected]`

Returns a pointer to the "font family" list box. This is usable mainly if you reimplement updateFontFamilies();.

**QFont QFontDialog::getFont ( bool * ok, QWidget * parent = 0,const char * name = 0 )** `[static]`

Opens a modal font dialog and returns the font selected by the user.

The *ok* parameter is set to TRUE if the user clicked OK, and FALSE if the user clicked Cancel.

If the user clicks Cancel the Qt default font is returned.

This static function is less capable than the full QFontDialog object, but is convenient and easy to

use.

Example:

```
// start at the current working directory and with *.cpp as filter

bool ok;
```

```
QFont f = QFontDialog::getFont( &ok, this );
if ( ok ) {
    // the user selected a valid font
} else {
    // the user cancelled the dialog
}
```

**QFont QFontDialog::getFont ( bool * ok, const QFont & initial, QWidget * parent = 0, const char * name = 0 )** `[static]`

Opens a modal font dialog and returns the font selected by the user.

*initial* is the initial selected font.

The *ok* parameter is set to TRUE if the user clicked OK, and FALSE if the user clicked Cancel.

If the user clicks Cancel the *initial* font is returned.

This static function is less capable than the full QFontDialog object, but is convenient and easy to use.

Example:

```
// start at the current working directory and with *.cpp as filter
bool ok;
QFont f = QFontDialog::getFont( &ok, QFont( "Times", 12 ), this );
if ( ok ) {
    // the user selected a valid font
} else {
    // the user cancelled the dialog
}
```

Another example:

```
mywidget.setFont( QFontDialog::getFont( 0, mywidget.font() ) );
```

**QComboBox * QFontDialog::scriptCombo () const** `[protected]`

Returns a pointer to the "font style" list box. This is usable mainly if you reimplement updateFontStyles();.

**QListBox * QFontDialog::sizeListBox () const** `[protected]`

Returns a pointer to the "font size" list box. This is usable mainly if you reimplement updateFontSizes();.

**QListBox * QFontDialog::styleListBox () const** `[protected]`

Returns a pointer to the "font style" list box. This is usable mainly if you reimplement updateFontStyles();.

**void QFontDialog::updateFamilies ()** `[virtual protected]`

Update the contents of the "font family" list box. This function can be reimplemented if you have special requirements.

**void QFontDialog::updateScripts ()** `[virtual protected]`

Update the contents of the "font script" combo box. This function can be reimplemented if you have special requirements.

**void QFontDialog::updateStyles ()** `[virtual protected]`

Update the contents of the "font style" list box. This function can be reimplemented if you have special requirements.

### 76.2.1. URL

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

# 77. QFontInfo Class Reference

```
#include <qfontinfo.h>
```

### 77.1. 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 font. This is equivalent to QFontInfo(widget->font()). Setting a new font for the widget later will not affect the font info object.

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 (unlike the two above cases, which take a "snapshot" of a font).

Example:

```
QFont font("reykjavik",24);
QFontInfo fi(font);
if ( font.family() == fi.family() ) {
    ; // got this font
} else {
    ; // got fi.family() instead
}
```

See also: QFont and QFontMetrics.

## 77.2. 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().

### QString QFontInfo::family () const

Returns the family name of the matched window system font.

See also: QFont::family().

**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().

**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().

**bool QFontInfo::rawMode () const**

Returns TRUE if the font is a raw mode font.

If it is a raw mode font, all other functions in QFontInfo will return the same values set in the QFont,

regardless of the font actually used.

See also: QFont::rawMode().

**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().

### 77.2.1. URL

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

# 78. QFontMetrics Class Reference

```
#include <qfontmetrics.h>
```

### 78.1. 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 font. This is equivalent to QFontMetrics(widget->font()). Setting a new font for the widget later will not affect the font metrics object.

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 (unlike the two above cases, which take a "snapshot" of a font).

Example:

```
QFont font("times",24);
QFontMetrics fm(font);
int w = fm.width("What's the width of this text");
int h = fm.height();
```

See also: QFont and QFontInfo.

Examples:

grapher/grapher.cpp
xform/xform.cpp
drawdemo/drawdemo.cpp
scrollview/scrollview.cpp
movies/main.cpp

## 78.2. 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
scrollview/scrollview.cpp

**QRect QFontMetrics::boundingRect ( QChar 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().

Examples:

xform/xform.cpp

**QRect QFontMetrics::boundingRect ( const QString & str, int len = -1 ) const**

Returns the bounding rectangle of the first *len* characters of *str,* which is the set of pixels the text would cover if drawn at (0,0).

If *len* is negative (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().

**QRect QFontMetrics::boundingRect ( int x, int y, int w, int h, int flgs, const QString & str, int len=-1, int tabstops=0, int * tabarray=0, 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 the Qt namespace.

If `ExpandTabs` is set in *flgs,* 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

**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

**bool QFontMetrics::inFont ( QChar 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 ( QChar 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(QChar) for a graphical description of this metric.*

See also: rightBearing(QChar), 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:

scrollview/scrollview.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 ( QChar 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 flgs, const QString & 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 *flgs* 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 qnamespace.h.

If `ExpandTabs` is set in *flgs,* 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 ( QChar 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 QString & 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(stra+strb). This is almost never the case with boundingRect().

See also: boundingRect().

Examples:

drawdemo/drawdemo.cpp
scrollview/scrollview.cpp
movies/main.cpp

**int QFontMetrics::width ( char c ) const**

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

Provided to aid porting from Qt 1.x.

### 78.2.1. URL

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

# 79. QFrame Class Reference

`#include <qframe.h>`

Inherits QWidget.

Inherited by QGrid, QGroupBox, QHBox, QLCDNumber, QLabel, QMenuBar, QPopupMenu, QProgressBar, QScrollView, QSpinBox, QSplitter, QTableView and QWidgetStack.

## 79.1. Detailed Description

The QFrame class is the base class of widgets that (can) have a frame.

It draws a frame 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( QFrame::Panel | QFrame::Sunken );
emptyFrame->setLineWidth( 2 );
```

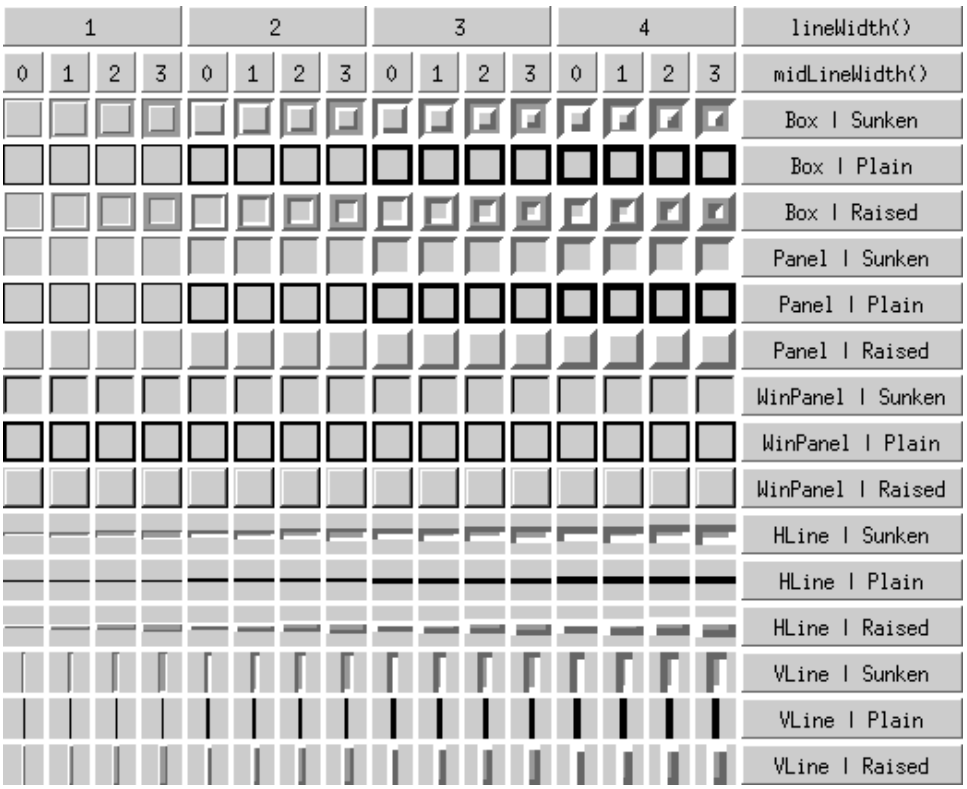
A frame widget has three attributes: frame style, a line width and a mid-line width.

The frame style is specified by a frame shape and a shadow style. The frame shapes are `NoFrame, Box, Panel, StyledPanel 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 in the illustration 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:

xform/xform.cpp
popup/popup.cpp
scrollview/scrollview.cpp

movies/main.cpp

## 79.2. Member Function Documentation

**QFrame::QFrame ( QWidget * parent=0, const char * name=0, WFlags f=0, bool = TRUE )**

Constructs a frame widget with frame style `NoFrame` and a 1 pixel frame width.

The last argument exists for compatibility with Qt 1.x; it no longer has any meaning. (In Qt 2.x, QFrame always allows the HLine and VLine styles.)

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(), QPainter::setClipRect() and drawContentsMask().

Reimplemented in QMenuBar, QLabel, QProgressBar and QLCDNumber.

**void QFrame::drawContentsMask ( QPainter * )** `[virtual protected]`

Virtual function that draws the mask of the frame's contents.

If you reimplemented drawContents(QPainter*) and your widget should support transparency you probably have to re-implement this function as well.

The default implementation is empty.

See also: drawContents(), updateMask(), QWidget::setAutoMask(), contentsRect() and QPainter::setClipRect().

Reimplemented in 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(), setPalette() and drawFrameMask().

**void QFrame::drawFrameMask ( QPainter * p )** `[virtual protected]`

Virtual function that draws the mask of the frame's frame.

If you reimplemented drawFrame(QPainter*) and your widget should support transparency you probably have to re-implement this function as well.

The default implementation is empty.

See also: drawFrame(), updateMask(), QWidget::setAutoMask() and QPainter::setClipRect().

**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.

The default implementation calls update().

Reimplemented in QHBox, QWidgetStack, QScrollView and QGrid.

**QRect QFrame::frameRect () const**

Returns the frame rectangle.

The default frame rectangle is equivalent to the widget rectangle.

See also: setFrameRect().

**Shadow QFrame::frameShadow () const**

Returns the frame shadow value from the frame style.

See also: setFrameShadow(), frameStyle() and frameShape().

**Shape QFrame::frameShape () const**

Returns the frame shape value from the frame style.

See also: setFrameShape(), frameStyle() and frameShadow().

**int QFrame::frameStyle () const**

Returns the frame style.

The default value is QFrame::NoFrame.

See also: setFrameStyle(), frameShape() and frameShadow().

Examples:

scrollview/scrollview.cpp

**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. The frame width also includes the margin.

See also: lineWidth(), midLineWidth(), frameStyle() and margin().

**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().

Examples:

scrollview/scrollview.cpp

**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().

Examples:

scrollview/scrollview.cpp

**int QFrame::midLineWidth () const**

Returns the width of the mid-line.

The default value is 0.

See also: setMidLineWidth(), lineWidth() and frameWidth().

Examples:

scrollview/scrollview.cpp

**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 )** `[virtual]`

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().

Reimplemented in QWidgetStack.

**void QFrame::setFrameStyle ( int style )** `[virtual]`

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.
- `StyledPanel` draws a rectangular panel with a look depending on the current GUI style. It can be raised or sunken.
- `PopupPanel` is used to draw a frame suitable for popup windows. Its look also depends on the current GUI style, usually the same as `StyledPanel.`
- `WinPanel` draws a rectangular panel that can be raised or sunken, very like those in Windows 95. Specifying this shape sets the line width to 2 pixels. WinPanel is provided for compatibility. For GUI style independence we recommend using StyledPanel instead.
- `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.

See also: Illustration, frameStyle(), colorGroup() and QColorGroup.

Examples:

tabdialog/tabdialog.cpp
xform/xform.cpp
cursor/cursor.cpp
layout/layout.cpp
popup/popup.cpp

scrollview/scrollview.cpp
customlayout/main.cpp

**void QFrame::setLineWidth ( int w )** `[virtual]`

Sets the line width to *w*.

See also: frameWidth(), lineWidth() and setMidLineWidth().

Examples:

xform/xform.cpp
scrollview/scrollview.cpp

**void QFrame::setMargin ( int w )** `[virtual]`

Sets the width of the margin to *w*.

See also: margin() and setLineWidth().

Examples:

tabdialog/tabdialog.cpp
scrollview/scrollview.cpp

**void QFrame::setMidLineWidth ( int w )** `[virtual]`

Sets the width of the mid-line to *w*.

See also: midLineWidth() and setLineWidth().

Examples:

scrollview/scrollview.cpp

**QSizePolicy QFrame::sizePolicy () const** `[virtual]`

If this is a line, it may stretch in the direction of the line, but it is fixed in the other direction. If this is a normal frame, use QWidget's default behavior.

Reimplemented from QWidget.

**void QFrame::updateMask ()** `[virtual protected]`

Reimplementation of QWidget::updateMask(). Draws the mask of the frame when transparency is required.

This function calls the virtual functions drawFrameMask() and drawContentsMask(). These are the ones you may want to reimplement in subclasses.

See also: QWidget::setAutoMask(), drawFrameMask() and drawContentsMask().

Reimplemented from QWidget.

### 79.2.1. URL

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

# 80. QGArray Class Reference

`#include <qgarray.h>`

Inherited by QArray.

### 80.1. 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.

### 80.2. 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 QArray.

**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.

**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.

### 80.2.1. URL

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

# 81. QGCache Class Reference

`#include <qgcache.h>`

Inherits QCollection.

Inherited by QAsciiCache, QCache and QIntCache.

## 81.1. Detailed Description

The QGCache class is an internal class for implementing QCache template classes.

QGCache is a strictly internal class that acts as a base class for the collection classes QCache and

QIntCache.

## 81.2. Member Function Documentation

**QGCache::QGCache ( const QGCache & )** `[protected]`

For internal use only.

**QGCache::QGCache ( int maxCost, uint size, KeyType kt, bool caseSensitive, bool copyKeys )** `[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.

**QCollection::Item QGCache::find_other( const char * key, bool ref=TRUE ) const** `[protected]`

For internal use only.

**QCollection::Item QGCache::find_string( const QString & key, bool ref=TRUE ) const** `[protected]`

For internal use only.

**bool QGCache::insert_other ( const char * key, QCollection::Item data, int cost, int priority )** `[protected]`

For internal use only.

**bool QGCache::insert_string ( const QString & key, QCollection::Item 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_other ( const char * key )** `[protected]`

For internal use only.

**bool QGCache::remove_string ( const QString & 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.

**QCollection::Item QGCache::take_other( const char * key )** `[protected]`

For internal use only.

**QCollection::Item QGCache::take_string( const QString & key )** `[protected]`

For internal use only.

**int QGCache::totalCost () const** `[protected]`

For internal use only.

### 81.2.1. URL

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

# 82. QGCacheIterator Class Reference

`#include <qgcache.h>`

Inherited by QAsciiCacheIterator, QCacheIterator and QIntCacheIterator.

## 82.1. 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.

## 82.2. 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.

**QCollection::Item QGCacheIterator::get() const** `[protected]`

For internal use only.

**const char * QGCacheIterator::getKeyAscii () const** `[protected]`

For internal use only.

**long QGCacheIterator::getKeyInt () const** `[protected]`

For internal use only.

**QString QGCacheIterator::getKeyString () const** `[protected]`

For internal use only.

**QCollection::Item QGCacheIterator::operator()()** `[protected]`

For internal use only.

**QCollection::Item QGCacheIterator::operator++()** `[protected]`

For internal use only.

**QCollection::Item QGCacheIterator::operator+=( uint jump )** `[protected]`

For internal use only.

**QCollection::Item QGCacheIterator::operator--()** `[protected]`

For internal use only.

**QCollection::Item QGCacheIterator::operator-=( uint jump )** `[protected]`

For internal use only.

**QGCacheIterator & QGCacheIterator::operator= ( const QGCacheIterator & ci )** `[protected]`

For internal use only.

**QCollection::Item QGCacheIterator::toFirst()** `[protected]`

For internal use only.

**QCollection::Item QGCacheIterator::toLast()** `[protected]`

For internal use only.

### 82.2.1. URL

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

# 83. QGDict Class Reference

`#include <qgdict.h>`

Inherits QCollection.

Inherited by QAsciiDict, QDict, QIntDict and QPtrDict.

## 83.1. Detailed Description

The QGDict class is an internal class for implementing QDict template classes.

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.

- 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.

## 83.2. Member Function Documentation

**QGDict::QGDict ( const QGDict & dict )** `[protected]`

For internal use only.

**QGDict::QGDict ( uint len, KeyType kt, bool caseSensitive, bool copyKeys )** `[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::hashKeyString ( const QString & key )** `[protected]`

For internal use only.

**QCollection::Item QGDict::look_ascii( const char * key, QCollection::Item d, int op )**

For internal use only.

**QCollection::Item QGDict::look_int( long key, QCollection::Item d, int op )**

For internal use only.

**QCollection::Item QGDict::look_ptr( void * key, QCollection::Item d, int op )**

For internal use only.

**QCollection::Item QGDict::look_string( const QString & key, QCollection::Item 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, QCollection::Item & 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_ascii ( const char * key, QCollection::Item item=0 )** `[protected]`

For internal use only.

**bool QGDict::remove_int ( long key, QCollection::Item item=0 )** `[protected]`

For internal use only.

**bool QGDict::remove_ptr ( void * key, QCollection::Item item=0 )** `[protected]`

For internal use only.

**bool QGDict::remove_string ( const QString & key, QCollection::Item item=0 )** `[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.

**QCollection::Item QGDict::take_ascii( const char * key )** `[protected]`

For internal use only.

**QCollection::Item QGDict::take_int( long key )** `[protected]`

For internal use only.

**QCollection::Item QGDict::take_ptr( void * key )** `[protected]`

For internal use only.

**QCollection::Item QGDict::take_string( const QString & key )** `[protected]`

For internal use only.

**QDataStream& QGDict::write ( QDataStream & s ) const**

For internal use only.

**QDataStream& QGDict::write ( QDataStream & s, QCollection::Item ) const** `[virtual protected]`

Writes a collection/dictionary item to the stream *s* and returns a reference to the stream.

See also: read().

### 83.2.1. URL

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

# 84. QGDictIterator Class Reference

```
#include <qgdict.h>
```

Inherited by QAsciiDictIterator, QDictIterator, QIntDictIterator and QPtrDictIterator.

## 84.1. 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.

## 84.2. 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.

**QCollection::Item QGDictIterator::operator()()**

For internal use only.

**QCollection::Item QGDictIterator::operator++()**

For internal use only.

**QCollection::Item QGDictIterator::operator+=( uint jumps )**

For internal use only.

**QGDictIterator & QGDictIterator::operator= ( const QGDictIterator & it )**

For internal use only.

**QCollection::Item QGDictIterator::toFirst()**

For internal use only.

### 84.2.1. URL

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

# 85. QGL Class Reference

```
#include <qgl.h>
```

Inherited by QGLContext, QGLFormat and QGLWidget.

### 85.1. 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 QGL class is a namespace for miscellaneous identifiers in the Qt OpenGL Extension.

Normally, you can ignore this class. QGLWidget and the other OpenGL Extension classes inherit it, so when you make your own QGLWidget subclass, you can use the identifiers in the QGL namespace without qualification.

However, occasionally you may find yourself in situations where you need ot refer to these identifiers from outside the QGL namespace scope, e.g. in static utility functions. In such cases, simply write e.g. `QGL::DoubleBuffer` instead just `DoubleBuffer.`

#### 85.1.1. URL

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

# 86. QGLayoutIterator Class Reference

`#include <qabstractlayout.h>`

Inherits QShared.

## 86.1. Detailed Description

The abstract base class of internal layout iterators.

To be subclassed by custom layout implementors. The functions that need to be implemented are next(), current() and takeCurrent().

The QGLayoutIterator implements the functionality of QLayoutIterator. Each subclass of QLayout needs a QGLayoutIterator subclass.

## 86.2. Member Function Documentation

**QGLayoutIterator::~QGLayoutIterator** () `[virtual]`

Destroys the iterator.

**QLayoutItem * QGLayoutIterator::current** () `[virtual]`

Implemented in subclasses to return the current item, or 0 if there is no current item.

**QLayoutItem * QGLayoutIterator::next** () `[virtual]`

Implemented in subclasses to move the iterator to the next item and return that item, or 0 if there is no next item.

**QLayoutItem * QGLayoutIterator::takeCurrent** () `[virtual]`

Implemented in subclasses to remove the current item from the layout without deleting it, move the iterator to the next item and return the removed item, or 0 if no item was removed.

### 86.2.1. URL

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

# 87. QGLContext Class Reference

`#include <qgl.h>`

Inherits QGL.

## 87.1. 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.

## 87.2. Member Function Documentation

**QGLContext::QGLContext ( const QGLFormat & format, QPaintDevice * device )**

Constructs an OpenGL context for the paint device *device,* which can be a widget or a pixmap. The *format* specifies several display options for this context.

If the underlying OpenGL/Window system cannot satisfy all the features requested in *format,* the

nearest subset of features will be used. After creation, the format() method will return the actual

format obtained.

The context will be invalid if it was not possible to obtain a GL context at all.

See also: format() and 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 as closely as possible.

**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 GL rendering context on the paint device specified in the constructor, otherwise FALSE is returned (the context is invalid).

After successful creation, format() returns the set of features of the created GL rendering context.

If *shareContext* points to a valid QGLContext, this method will try to establish OpenGL display list sharing between this context and *shareContext*. Note that this may fail if the two contexts have different formats. Use isSharing() to test.

**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(), format() and isValid().

**QPaintDevice* QGLContext::device () const**

Returns the paint device set for this context.

See also: QGLContext::QGLContext().

**bool QGLContext::deviceIsPixmap () const** `[protected]`

Returns TRUE if the paint device of this context is a pixmap, otherwise FALSE.

**void QGLContext::doneCurrent ()** `[virtual protected]`

Makes no GL context the current context. Normally, you do not need to call this function, QGLContext calls it as necessary.

**QGLFormat QGLContext::format () const**

Returns the format.

See also: setFormat().

**bool QGLContext::initialized () const** `[protected]`

Returns TRUE if this context has been initialized, i.e. if QGLWidget::initializeGL() has been performed on it.

See also: setInitialized().

**bool QGLContext::isSharing () const**

Returns TRUE if display list sharing with another context was requested in the create() call, and the GL system was able to fulfill this request. Note that display list sharing may possibly not be supported between contexts with different formats.

**bool QGLContext::isValid () const**

Returns TRUE if a GL rendering context has been successfully created.

**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 *format* for this context. The context is reset.

Call create() to create a new GL context that tries to match the new format.

```
QGLContext *cx;

  ...

QGLFormat f;
```

```
f.setStereo( TRUE );

cx->setFormat( f );

if ( !cx->create() )

    exit(); // no OpenGL support, or cannot render on specified paintdevice

if ( !cx->format().stereo() )

    exit(); // could not create stereo context
```

See also: format(), reset() and create().

**void QGLContext::setInitialized ( bool on )** `[protected]`

Tells the context whether it has been initialized, i.e. whether QGLWidget::initializeGL() has been performed on it.

See also: initialized().

**void QGLContext::setWindowCreated ( bool on )** `[protected]`

Tells the context whether a window has already been created for it.

See also: windowCreated().

**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().

**bool QGLContext::windowCreated () const** `[protected]`

Returns TRUE if a window has been created for this context, otherwise FALSE.

See also: setWindowCreated().

### 87.2.1. URL

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

# 88. QGLFormat Class Reference

`#include <qgl.h>`

Inherits QGL.

## 88.1. 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 buffers.
- Direct rendering..

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 some features the driver/hardware does not provide when you create a QGLWidget, you will get the a rendering context with the nearest subset of features.

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 );
```

Or you can specify the desired format when creating an object of your QGLWidget subclass:

```
QGLFormat f;
f.setDoubleBuffer( FALSE );
f.setDirectRendering( FALSE );
MyGLWidget* myWidget = new MyGLWidget( f, ... );
```

You can even set the format for an already existing QGLWidget:

```
MyGLWidget *w;
  ...
QGLFormat f;
f.setAlpha( TRUE );
f.setStereo( TRUE );
w->setFormat( f );
if ( !w->format().stereo() ) {
    // ok, goggles off
    if ( !w->format().alpha() ) {
        qFatal( "Cool hardware wanted" );
    }
}
```

See also: QGLContext and QGLWidget.

### 88.2. Member Function Documentation

**QGLFormat::QGLFormat ()**

Constructs a QGLFormat object with the factory default settings:

- Double buffer: Enabled.
- Depth buffer: Enabled.
- RGBA: Enabled (i.e. color index disabled).
- Alpha channel: Disabled.
- Accumulator buffer: Disabled.
- Stencil buffer: Disabled.
- Stereo: Disabled.
- Direct rendering: Enabled.

**QGLFormat::QGLFormat ( int options )**

Creates a QGLFormat object that is a copy of the current application default format.

If *options* is not 0, this copy will be modified by these format options. The *options* parameter must be FormatOption values OR'ed together.

This constructor makes it easy to specify a certain desired format in classes derived from QGLWidget, for example:

```
// The rendering in MyGLWidget depends on using
// stencil buffer and alpha channel
MyGLWidget::MyGLWidget( QWidget* parent, const char* name )
    : QGLWidget( QGLFormat( StencilBuffer | AlphaChannel ), parent, name )
{
  if ( !format().stencil() )
    qWarning( "Could not get stencil buffer; results will be suboptimal" );
  if ( !format().alphaChannel() )
    qWarning( "Could not get alpha channel; results will be suboptimal" );
  ...
}
```

Note that there exists FormatOption values for both turning on and off all format settings, e.g. DepthBuffer and NoDepthBuffer, DirectRendering and IndirectRendering, etc.

See also: defaultFormat() and setOption().

**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().

**QGLFormat QGLFormat::defaultFormat ()** `[static]`

Returns the default QGLFormat for the application. All QGLWidgets that are created use this format unless anything else is specified.

If no special default format has been set using setDefaultFormat(), the default format is the same as

that created with QGLFormat().

See also: setDefaultFormat().

**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::directRendering () const**

Returns TRUE if direct rendering is enabled, otherwise FALSE.

Direct rendering is enabled by default.

See also: setDirectRendering().

**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.

**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 );
```

See also: defaultFormat().

**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::setDirectRendering ( bool enable )**

Enables direct rendering if *enable* is TRUE, or disables it if *enable* is FALSE.

Direct rendering is enabled by default.

Enabling this option will make OpenGL bypass the underlying window system and render directly from hardware to the screen, if this is supported by the system.

See also: directRendering().

**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::setOption ( FormatOption opt )**

Sets the option *opt.*

See also: testOption().

**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().

**bool QGLFormat::testOption ( FormatOption opt ) const**

Returns TRUE if format option *opt* is set, otherwise FALSE.

See also: setOption().

### 88.3. Related Functions

**const char * qGLVersion ()**

Returns the version number string for the Qt OpenGL extension, e.g. "1.0".

#### 88.3.1. URL

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

# 89. QGList Class Reference

`#include <qglist.h>`

Inherits QCollection.

Inherited by QList, QQueue and QStack.

## 89.1. 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.

## 89.2. Member Function Documentation

**QGList::QGList ()** `[protected]`

For internal use only.

**QGList::QGList ( const QGList & list )** `[protected]`

For internal use only.

**QGList::~QGList ()** `[virtual protected]`

For internal use only.

**void QGList::append ( QCollection::Item d )** `[protected]`

For internal use only.

**QCollection::Item QGList::at( uint index )** `[protected]`

For internal use only.

**int QGList::at () const** `[protected]`

For internal use only.

**QCollection::Item QGList::cfirst() const** `[protected]`

For internal use only.

**QCollection::Item QGList::clast() const** `[protected]`

For internal use only.

**void QGList::clear ()** `[virtual protected]`

For internal use only.

Reimplemented from QCollection.

**int QGList::compareItems ( QCollection::Item item1, QCollection::Item 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 ( QCollection::Item d ) const** `[protected]`

For internal use only.

**uint QGList::containsRef ( QCollection::Item 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 ( QCollection::Item d, bool fromStart = TRUE )** `[protected]`

For internal use only.

**int QGList::findRef ( QCollection::Item d, bool fromStart = TRUE )** `[protected]`

For internal use only.

**QCollection::Item QGList::first()** `[protected]`

For internal use only.

**QCollection::Item QGList::get() const** `[protected]`

For internal use only.

**void QGList::inSort ( QCollection::Item d )** `[protected]`

For internal use only.

**bool QGList::insertAt ( uint index, QCollection::Item d )** `[protected]`

For internal use only.

**QCollection::Item QGList::last()** `[protected]`

For internal use only.

**QCollection::Item QGList::next()** `[protected]`

For internal use only.

**QGList& QGList::operator= ( const QGList & list )** `[protected]`

For internal use only.

**QCollection::Item QGList::prev()** `[protected]`

For internal use only.

**QDataStream & QGList::read ( QDataStream & s )**

For internal use only.

**QDataStream & QGList::read ( QDataStream & s, QCollection::Item & 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 ( QCollection::Item 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 ( QCollection::Item d = 0 )** `[protected]`

For internal use only.

**QCollection::Item QGList::take()** `[protected]`

For internal use only.

**QCollection::Item QGList::takeAt( uint index )** `[protected]`

For internal use only.

**QCollection::Item QGList::takeFirst()** `[protected]`

For internal use only.

**QCollection::Item QGList::takeLast()** `[protected]`

For internal use only.

**QCollection::Item 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, QCollection::Item ) const** `[virtual protected]`

Writes a collection/list item to the stream *s* and returns a reference to the stream.

See also: read().

### 89.2.1. URL

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

# 90. QGListIterator Class Reference

`#include <qglist.h>`

Inherited by QListIterator.

## 90.1. Detailed Description

The QGListIterator is an internal class for implementing QListIterator.

QGListIterator is a strictly internal class that does the heavy work for QListIterator.

## 90.2. 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.

**QCollection::Item QGListIterator::get() const** `[protected]`

For internal use only.

**QCollection::Item QGListIterator::operator()()** `[protected]`

For internal use only.

**QCollection::Item QGListIterator::operator++()** `[protected]`

For internal use only.

**QCollection::Item QGListIterator::operator+=( uint jumps )** `[protected]`

For internal use only.

**QCollection::Item QGListIterator::operator--()** `[protected]`

For internal use only.

**QCollection::Item QGListIterator::operator-=( uint jumps )** `[protected]`

For internal use only.

**QGListIterator & QGListIterator::operator= ( const QGListIterator & it )** `[protected]`

For internal use only.

**QCollection::Item QGListIterator::toFirst()** `[protected]`

For internal use only.

**QCollection::Item QGListIterator::toLast()** `[protected]`

For internal use only.

### 90.2.1. URL

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

# 91. QGLWidget Class Reference

`#include <qgl.h>`

Inherits QWidget and QGL.

## 91.1. 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.

## 91.2. 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 widget will be invalid if the system has no OpenGL support.

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: If this widget and *shareWidget* has different formats, display list sharing may fail. You can check whether display list sharing succeeded by using the isSharing() method.

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 desired rendering options. If the underlying OpenGL/Window system cannot satisfy all the features requested in *format,* the nearest subset of features will be used.

After creation, the format() method will return the actual format obtained.

The widget will be invalid if the system has no OpenGL support.

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: If this widget and *shareWidget* has different formats, display list sharing may fail. You can check whether display list sharing succeeded by using the isSharing() method.

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.

**bool QGLWidget::autoBufferSwap () const** `[protected]`

Returns TRUE if the widget is doing automatic GL buffer swapping.

See also: setAutoBufferSwap().

**const QGLContext * QGLWidget::context () const**

Returns the current context.

See also: setContext().

**bool QGLWidget::doubleBuffer () const**

Returns TRUE if the contained GL rendering context has double buffering.

See also: QGLFormat::doubleBuffer().

**QGLFormat QGLWidget::format () const**

Returns the format of the contained GL rendering context.

See also: setFormat().

**void QGLWidget::glDraw ()** `[virtual protected]`

Executes the virtual function paintGL(), initializing first as necessary.

**void QGLWidget::glInit ()** `[virtual 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::isSharing () const**

Returns TRUE if display list sharing with another QGLWidget was requested in the constructor, and the GL system was able to provide it. The GL system may fail to provide display list sharing if the two QGLWidgets use different formats.

See also: format().

**bool QGLWidget::isValid () const**

Returns TRUE if the widget has a valid GL rendering context. A widget will be invalid if the system has no OpenGL support

**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. Will cause the virtual paintGL() fucntion to be called, initializing first as necessary.

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::qglClearColor ( const QColor & c )**

Convenience function for specifying the clearing color to OpenGL. Calls glClearColor (in RGBA mode) or glClearIndex (in color-index mode) with the color *c*.

See also: qglColor() and QColor.

**void QGLWidget::qglColor ( const QColor & c )**

Convenience function for specifying a drawing color to OpenGL. Calls glColor3 (in RGBA mode) or glIndex (in color-index mode) with the color *c*.

See also: qglClearColor() and QColor.

**QPixmap QGLWidget::renderPixmap ( int w = 0, int h = 0, bool useContext = FALSE )** `[virtual]`

Renders the current scene on a pixmap and returns it.

You may use this method on both visible and invisible QGLWidgets.

This method will create a pixmap and a temporary QGLContext to render on it. Then, initializeGL(),

resizeGL(), and paintGL() are called on this context. Finally, the widget's original GL context is

restored.

The size of the pixmap will be width *w* and height *h*. If any of those are 0 (the default), the pixmap

will have the same size as the widget.

If *useContext* is TRUE, this method will try to be more efficient by using the existing GL context to render the pixmap. The default is FALSE. Use only if you know what you are doing.

Bugs and limitations:

- May give unexpected results if the depth of the GL rendering context is different from the depth of the desktop.

**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::setAutoBufferSwap ( bool on )** `[protected]`

Turns on or off the automatic GL buffer swapping. If on, and the widget is using a double-buffered format, the background and foreground GL buffers will automatically be swapped after each time the paintGL() function has been called.

The buffer auto-swapping is on by default.

See also: autoBufferSwap(), doubleBuffer() and swapBuffers().

**void QGLWidget::setContext ( QGLContext * context, const QGLContext * shareContext = 0, bool deleteOldContext = TRUE )** `[virtual]`

Sets a new context for this widget. The QGLContext *context* must be created using *new*. QGLWidget will delete *context* when another context is set or when the widget is destroyed.

If *context* is invalid, QGLContext::create() is performed on it. The initializeGL() function will then be executed for the new context before the first resizeGL() or paintGL().

If *context* is invalid, this method will try to keep any existing display list sharing with other QGLWidgets this widget currently has, or (if *shareContext* points to a valid context) start display list sharing with that context, but it may fail. Use isSharing() to test.

If *deleteOldContext* is TRUE (the default), the existing context will be deleted. You may use FALSE here if you have kept a pointer to the old context (as returned by context()), and want to restore that context later.

See also: context(), setFormat() and isSharing().

**void QGLWidget::setFormat ( const QGLFormat & format )** `[virtual]`

Sets a new format for this widget.

If the underlying OpenGL/Window system cannot satisfy all the features requested in *format,* the nearest subset of features will be used. After creation, the format() method will return the actual rendering context format obtained.

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().

This method will try to keep any existing display list sharing with other QGLWidgets, but it may fail. Use isSharing() to test.

See also: format(), setContext(), isSharing() 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.

Normally, there is no need to explicitly call this function, because it is done automatically after each widget repaint, i.e. after each time paintGL() has been executed.

See also: doubleBuffer(), setAutoBufferSwap() and QGLFormat::setDoubleBuffer().

**void QGLWidget::updateGL ()** `[virtual slot]`

Updates the widget by calling glDraw().

### 91.2.1. URL

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

# 92. QGrid Class Reference

`#include <qgrid.h>`

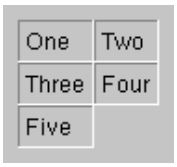
Inherits QFrame.

## 92.1. Detailed Description

The QGrid widget performs geometry management on its children

The number of rows or columns is defined in the constructor. All its children will be placed and sized according to their sizeHint() and sizePolicy() return values.

Use setMargin() to add space around the edge, and use addSpacing to add space between the widgets.




See also: QVBox and QHBox.

## 92.2. Member Function Documentation

**QGrid::QGrid ( int n, QWidget * parent=0, const char * name=0, WFlags f=0 )**

Constructs a grid widget with parent *parent* and name *name. n* specifies the number of columns.

**QGrid::QGrid ( int n, Direction dir, QWidget * parent=0, const char * name=0, WFlags f=0 )**

Constructs a grid widget with parent *parent* and name *name.* If *dir* is `Horizontal,` *n* specifies the number of columns. If *dir* is `Vertical,` *n* specifies the number of rows.

**void QGrid::frameChanged ()** `[virtual protected]`

Reimplemented for internal purposes.

Reimplemented from QFrame.

**void QGrid::setSpacing ( int space )**

Sets the spacing between children to *space*.

### 92.2.1. URL

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

# 93. QGridLayout Class Reference

```
#include <qlayout.h>
```

Inherits QLayout.

## 93.1. Detailed Description

The QGridLayout class lays out widgets in a grid.

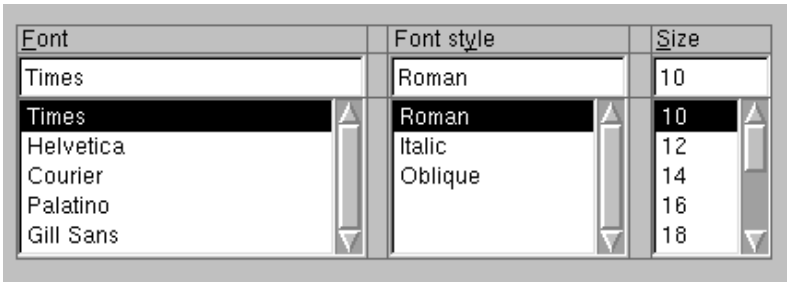
QGridLayout takes the space it gets (from its parent layout or from the mainWidget()), divides it up into rows and columns, and puts each of the widgets it manages into the correct cell(s).

Columns and rows behave identically; we will discuss columns but there are equivalent 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 managed widget or layout is put into a cell of its own using addWidget(), addLayout(), or by the auto-add facility, but you can also put widget into multiple cells using addMultiCellWidget(). If you do that, QGridLayout will make a guess at how to distribute the size over the columns/rows (based on the stretch factors). You can adjust the minimum width of each column/row using addColSpacing()/addRowSpacing().

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().

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.

See also the Layout Overview documentation.

Examples:

layout/layout.cpp
rot13/rot13.cpp

## 93.2. Member Function Documentation

**QGridLayout::QGridLayout ( QLayout * parentLayout, int nRows, int nCols, int space = -1, const char * name=0 )**

Constructs a new grid that is placed inside *parentLayout,* with *nRows* rows and *nCols* columns, If *space* is -1, this QGridLayout will inherits its parent's spacing(), otherwise *space* is used.

This grid is placed according to *parentLayout's* default placement rules.

**QGridLayout::QGridLayout ( QWidget * parent, int nRows, int nCols, int border=0, int space = -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. *space* is

the default number of pixels between cells. If *space* is -1 the value of *border* is used.

*name* is the internal object name.

**QGridLayout::QGridLayout ( int nRows, int nCols, int space = -1, const char * name=0 )**

Constructs a new grid with *nRows* rows and *nCols* columns, If *space* is -1, this QGridLayout will inherits its parent's spacing(), otherwise *space* is used.

You have to insert this grid into another layout. You can insert widgets and layouts in this layout at

any time, but layout will not be performed before it is inserted.

**QGridLayout::~QGridLayout ()**

Deletes this grid. Geometry management is terminated if this is a top-level grid.

**void QGridLayout::add ( QLayoutItem * item, int row, int col )** `[protected]`

Adds *item* at position *row, col.* The layout takes over ownership of *item.*

**void QGridLayout::addColSpacing ( int col, int minsize )**

Sets the minimum width of *col* to *minsize* pixels.

**void QGridLayout::addItem ( QLayoutItem * item )** `[virtual]`

Adds *item* to the next free position of this layout.

Reimplemented from QLayout.

**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).

**void QGridLayout::addMultiCell ( QLayoutItem * item, int fromRow, int toRow, int fromCol, int toCol, int alignmnt = 0 )**

Adds the *item* to the cell grid, spanning multiple rows/columns.

Alignment is specified by *alignmnt*

**void QGridLayout::addMultiCellWidget ( QWidget * w, int fromRow, int toRow, int fromCol, int toCol, int alignmnt = 0 )**

Adds the widget *w* to the cell grid, spanning multiple rows/columns.

Alignment is specified by *alignmnt* which takes the same arguments as QLabel::setAlignment(). The default alignment is 0, which means that the widget fills the entire cell.

A non-zero alignment indicates that the widget should not grow to fill the available space, but should be sized according to sizeHint().

Examples:

layout/layout.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 alignmnt = 0 )**

Adds the widget *w* to the cell grid at *row, col.* The top left position is (0,0)

Alignment is specified by *alignmnt* which takes the same arguments as QLabel::setAlignment(). The default alignment is 0, which means that the widget fills the entire cell.

Note 1: You should not call this if you have enabled the auto-add facility of the layout.

Note 2: The alignment parameter is interpreted more aggressively than in previous versions of Qt. A non-default alignment now indicates that the widget should not grow to fill the available space, but should be sized according to sizeHint().

Examples:

layout/layout.cpp
rot13/rot13.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. You should not need to call this function, as QGridLayout expands automatically as new items are inserted.

**bool QGridLayout::findWidget ( QWidget * w, int * row, int * col )** `[protected]`

Searches for *w* in this layout (not including child layouts). If *w* is found, it sets *row* and *col* to the row and column and returns TRUE. If *w* is not found, FALSE is returned.

**Warning:** If a widget spans multiple rows/columns, the top-left cell is returne.

**bool QGridLayout::hasHeightForWidth () const** `[virtual]`

Returns whether this layout's preferred height depends on its width.

Reimplemented from QLayoutItem.

**int QGridLayout::heightForWidth ( int w ) const** `[virtual]`

Returns the layout's preferred height when it is *w* pixels wide.

Reimplemented from QLayoutItem.

**void QGridLayout::invalidate ()** `[virtual]`

Resets cached information.

Reimplemented from QLayoutItem.

**QLayoutIterator QGridLayout::iterator ()** `[virtual]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QLayoutItem.

**QSize QGridLayout::maximumSize () const** `[virtual]`

Returns the maximum size needed by this grid.

Reimplemented from QLayoutItem.

**QSize QGridLayout::minimumSize () const** `[virtual]`

Returns the minimum size needed by this grid.

Reimplemented from QLayoutItem.

**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 )** `[virtual]`

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:

layout/layout.cpp

**void QGridLayout::setGeometry ( const QRect & s )** `[virtual]`

Resizes managed widgets within the rectangle *s*.

Reimplemented from QLayoutItem.

**void QGridLayout::setOrigin ( Corner c )**

Sets which of the four corners of the grid corresponds to (0,0).

**void QGridLayout::setRowStretch ( int row, int stretch )** `[virtual]`

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.

**QSize QGridLayout::sizeHint () const**

Returns the preferred size of this grid.

### 93.2.1. URL

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

# 94. QGroupBox Class Reference

`#include <qgroupbox.h>`

Inherits QFrame.

Inherited by QButtonGroup, QHGroupBox and QVGroupBox.

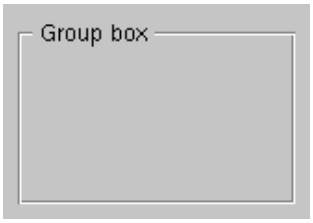
## 94.1. Detailed Description

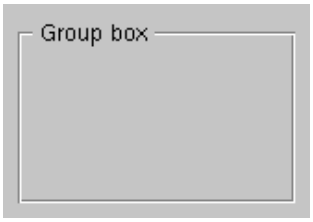
The QGroupBox widget provides a group box frame with a title.

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 contains, 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.




## 94.2. 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 QString & 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.

**QGroupBox::QGroupBox ( int strips, Orientation orientation, QWidget * parent=0, const char * name=0 )**

Constructs a group box with no title. Child widgets will be arranged in *strips* rows or columns (depending on *orientation).*

The *parent* and *name* arguments are passed to the QWidget constructor.

**QGroupBox::QGroupBox ( int strips, Orientation orientation, const QString & title, QWidget * parent=0, const char * name=0 )**

Constructs a group box with a *title.* Child widgets will be arranged in *strips* rows or columns (depending on *orientation).*

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 `AlignLeft`.

See also: setAlignment().

**void QGroupBox::paintEvent ( QPaintEvent * event )** `[virtual protected]`

Handles paint events for the group box.

Reimplemented from QWidget.

**void QGroupBox::resizeEvent ( QResizeEvent * e )** `[virtual protected]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QWidget.

**void QGroupBox::setAlignment ( int alignment )** `[virtual]`

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::setColumnLayout ( int columns, Orientation direction )** `[virtual]`

Changes the layout of the group box. This function is only useful in combination with the default constructor that does not take any layout information. This function will put all existing children in the new layout. Nevertheless is is not good programming style to call this function after children have been inserted.

**void QGroupBox::setTitle ( const QString & title )** `[virtual]`

Sets the group box title text to *title,* and add a focus-change accelerator if the *title* contains & followed by an appropriate letter. This produces "User information" with the U underscored and Alt-U moves the keyboard focus into the group.

```
g->setTitle( "&User information" );
```

### 94.2.1. URL

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

# 95. QHBox Class Reference

`#include <qhbox.h>`

Inherits QFrame.

Inherited by QVBox.

## 95.1. Detailed Description

The QHBox widget performs geometry management on its children

All its children will be placed beside each other and sized according to their sizeHint()s.

Use setMargin() to add space around the edge, and use setSpacing() to add space between the widgets.

See also: QVBox and QGrid.

## 95.2. Member Function Documentation

**QHBox::QHBox ( QWidget * parent=0, const char * name=0, WFlags f=0, bool allowLines=TRUE )**

Constructs an hbox widget with parent *parent* and name *name*

**QHBox::QHBox ( bool horizontal, QWidget * parent=0, const char * name=0, WFlags f=0, bool allowLines=TRUE )** `[protected]`

Constructs a horizontal hbox if *horizontal* is TRUE, otherwise constructs a vertical hbox (also known as a vbox).

This constructor is provided for the QVBox class. You should never need to use it directly.

**void QHBox::frameChanged ()** `[virtual protected]`

Reimplemented for internal purposes.

Reimplemented from QFrame.

**void QHBox::setSpacing ( int space )**

Sets the spacing between children to *space*.

Examples:

tabdialog/tabdialog.cpp

### 95.2.1. URL

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

# 96. QHBoxLayout Class Reference

`#include <qlayout.h>`

Inherits QBoxLayout.

## 96.1. Detailed Description

The QHBoxLayout class lines up widgets horizontally.

This class provides an easier way to construct horizontal box layout objects. See QBoxLayout for more details.

The simplest ways to use this class are:

```
QBoxLayout * l = new QHBoxLayout( widget );
l->setAutoAdd( TRUE );
new QSomeWidget( widget );
new QSomeOtherWidget( widget );
new QAnotherWidget( widget );
```

or

```
QBoxLayout * l = new QHBoxLayout( widget );
l->addWidget( existingChildOfWidget );
l->addWidget( anotherChildOfWidget );
```

See also: QVBoxLayout, QGridLayout and Layout overview documentation

Examples:

layout/layout.cpp
popup/popup.cpp

## 96.2. Member Function Documentation

**QHBoxLayout::QHBoxLayout ( QLayout * parentLayout, int space = -1, const char * name=0 )**

Creates a new horizontal box and adds it to *parentLayout.*

**QHBoxLayout::QHBoxLayout ( QWidget * parent, int border=0, int space = -1, const char * name=0 )**

Creates a new top-level horizontal box.

**QHBoxLayout::QHBoxLayout ( int space = -1, const char * name=0 )**

Creates a new horizontal box. You have to add it to another layout.

**QHBoxLayout::~QHBoxLayout ()**

Destroys this box.

### 96.2.1. URL

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

# 97. QHButtonGroup Class Reference

`#include <qhbuttongroup.h>`

Inherits QButtonGroup.

### 97.1. Detailed Description

The QHButtonGroup widget organizes QButton widgets in a group with one horizontal row.

QHButtonGroup is a convenience class that offers a thin layer on top of QButtonGroup. Think of it

as a QHBox that offers a frame with a title and is specifically designed for buttons.

### 97.2. Member Function Documentation

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

Constructs a horizontal button group with no title.

The *parent* and *name* arguments are passed to the QWidget constructor.

**QHButtonGroup::QHButtonGroup ( const QString & title, QWidget * parent=0, const char * name=0 )**

Constructs a horizontal button group with a title.

The *parent* and *name* arguments are passed to the QWidget constructor.

**QHButtonGroup::~QHButtonGroup ()**

Destroys the horizontal button group and its child widgets.

#### 97.2.1. URL

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

# 98. QHeader Class Reference

`#include <qheader.h>`

Inherits QWidget.

## 98.1. 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

## 98.2. 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 QIconSet & iconset, const QString & s, int size = -1 )**

Adds a new section, with icon set *iconset* and 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 is used.

**int QHeader::addLabel ( const QString & 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 widget position *c,* or -1 if outside.

**int QHeader::cellPos ( int i ) const**

Returns the position in pixels of section *i* of the header. The position is measured from the start of the first 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::count () const**

Returns the number of sections in the header.

**QIconSet * QHeader::iconSet ( int i ) const**

Returns the icon set set on logical section *i.*

**QString QHeader::label ( int i ) const**

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.

**Orientation QHeader::orientation () const**

Returns `Horizontal` if the header is horizontal, `Vertical` if the header is vertical.

**void QHeader::paintEvent ( QPaintEvent * e )** `[virtual protected]`

Paints the header.

Reimplemented from QWidget.

**void QHeader::paintSection ( QPainter * p, int id, QRect fr )** `[protected]`

Paints section *id* of the header, inside rectangle *fr* in widget coordinates.

**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 )** `[virtual]`

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 )** `[virtual]`

Enable clicking in logical column *i* if *enable* is TRUE, disable otherwise. If *i* is negative (as it is by default), 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 QString & s, int size = -1 )** `[virtual]`

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*.

Any icon set that has been defined for this section remains unchanged.

**void QHeader::setMovingEnabled ( bool enable )** `[virtual]`

Enable the user to exchange columns if *enable* is TRUE, disable otherwise.

See also: setClickEnabled() and setResizeEnabled().

**void QHeader::setOffset ( int x )** `[virtual slot]`

Scrolls the header such that *x* becomes the leftmost (or uppermost for vertical headers) visible pixel.

**void QHeader::setOrientation ( Orientation orientation )** `[virtual]`

Sets the header orientation. The *orientation* must be QHeader::Vertical or QHeader::Horizontal.

When adding labels without the size parameter, setOrientation should be called first, otherwise

labels will be sized incorrectly.

See also: orientation().

**void QHeader::setResizeEnabled ( bool enable, int i = -1 )** `[virtual]`

Enable user resizing of logical column *i* if *enable* is TRUE, disable otherwise. If *i* is negative (as it is by default), resizing is enabled/disabled for all columns.

See also: setMovingEnabled() and setClickEnabled().

**void QHeader::setTracking ( bool enable )** `[virtual]`

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::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.

**QSizePolicy QHeader::sizePolicy () const** `[virtual]`

A horizontal header may stretch horizontally, but is fixed vertically.

Reimplemented from QWidget.

**bool QHeader::tracking () const**

Returns TRUE if tracking is on, FALSE otherwise.

See also: setTracking().

### 98.2.1. URL

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

# 99. QHGroupBox Class Reference

`#include <qhgroupbox.h>`

Inherits QGroupBox.

### 99.1. Detailed Description

The QHGroupBox widget organizes QButton widgets in a group with one horizontal row.

QHGroupBox is a convenience class that offers a thin layer on top of QGroupBox. Think of it as a

QHBox that offers a frame with a title.

### 99.2. Member Function Documentation

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

Constructs a horizontal group box with no title.

The *parent* and *name* arguments are passed to the QWidget constructor.

**QHGroupBox::QHGroupBox ( const QString & title, QWidget * parent=0, const char * name=0 )**

Constructs a horizontal group box with a title.

The *parent* and *name* arguments are passed to the QWidget constructor.

**QHGroupBox::~QHGroupBox ()**

Destroys the horizontal group box and its child widgets.

#### 99.2.1. URL

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

# 100. QHideEvent Class Reference

`#include <qevent.h>`

Inherits QEvent.

### 100.1. Detailed Description

The event sent after a widget is hidden.

This event is sent just before QWidget::hide() returns, and also when a top-level window has been hidden (iconified) by the user.

See also: QShowEvent.

### 100.2. Member Function Documentation

**QHideEvent::QHideEvent ( bool spontaneous )**

Constructs a QHideEvent. *spontaneous* is TRUE if the event originated outside the application - ie. the user hid the window via the window manager controls.

**bool QHideEvent::spontaneous () const**

Returns TRUE if the event originated outside the application - ie. the user hid the window via the window manager controls, either by iconifying the window or by switching to another virtual desktop where the window isn't visible. The window will become hidden, but not withdrawn. If the window was iconified, QWidget::isMinimized() is TRUE.

#### 100.2.1. URL

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

# 101. QIconSet Class Reference

```
#include <qiconset.h>
```

## 101.1. 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.

When scaling icons, QIconSet uses smooth scaling, which can partially blend the color component of transparent images. If the results look poor, the best solution is to supply sizes that do not require scaling.

QIconSet provides a function, isGenerated(), that indicates whether an icon was set by the application programmer or computed by QIconSet itself.

See also: QPixmap, QLabel, QToolButton, QPopupMenu, GUI Design Handbook: Iconic Label,,

<q, href="http://www.microsoft.com/clipgallerylive/icons.asp">Microsoft, Icon and Gallery.</a>

### 101.2. 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.

See also: detach().

**QIconSet::~QIconSet ()** `[virtual]`

Destroys the icon set and frees any allocated resources.

**void QIconSet::detach ()**

Detaches this icon set from others with which it may share data. You should never need to call this function; reset() and setPixmap() call it as necessary. It exists merely so that the copy constructor and operator= can be faster.

**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.

See also: detach().

**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 an unspecified algorithm.

**void QIconSet::setPixmap ( const QPixmap & pm, Size s, Mode m = Normal )** `[virtual]`

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 QString & fileName, Size s, Mode m = Normal )** `[virtual]`

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.

### 101.2.1. URL

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

# 102. QImage Class Reference

```
#include <qimage.h>
```

### 102.1. 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
qmag/qmag.cpp
desktop/desktop.cpp

## 102.2. Member Type Documentation

### QImage::Endian

This enum type is used to describe the endianness of the CPU and graphics hardware.

The current values are:

- `IgnoreEndian` - does not matter. Useful for some operations that are independent of endianness
- `BigEndian` - network byte order, like on SPARC and Motorola CPUs.
- `LittleEndian` - PC/Alpha byte order.

### 102.3. Member Function Documentation

**QImage::QImage ()**

Constructs a null image.

See also: isNull().

**QImage::QImage ( const QString & fileName, const char * format=0 )**

Constructs an image from loading *fileName* and an optional *format.*

See also: load().

**QImage::QImage ( const QByteArray & array )**

Constructs an image from *array,* which must be in a supported image format image.

See also: loadFromData().

**QImage::QImage ( const QImage & image )**

Constructs a shallow copy of *image.*

**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=0 ) 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 QString & 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.

**void QImage::invertPixels ( bool invertAlpha = TRUE )**

Inverts all pixel values in the image.

For 32 bit (24 RGB + alpha buffer) images, set *invertAlpha* to FALSE if you want the alpha bits to be unchanged, otherwise they are inverted too.

**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 QString & 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.

Loads from a QByteArray.

**QImage QImage::mirror () const**

Returns a QImage which is a vertically mirrored copy of this image. The original QImage is left unchanged.

**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.

**bool QImage::operator!= ( const QImage & i ) const**

Returns TRUE if this image and *i* have different contents, and FALSE if they they have the same. This can be slow. Of course, this function returns quickly if e.g. the two images' widths are different.

See also: operator=().

**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().

**bool QImage::operator== ( const QImage & i ) const**

Returns TRUE if this image and *i* have the same contents, and FALSE if they differ. This can be slow. Of course, this function returns quickly if e.g. the two images' widths are different.

See also: operator=().

**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 QString & 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 QImage::swapRGB () const**

Returns a QImage where the values of the red and blue components of all pixels have been swapped, effectively converting a RGB image to a BGR image. The original QImage is left unchanged.

**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: height(), size() and rect().

### 102.4. Related Functions

**QDataStream & operator<< (QDataStream & s, const QImage & image)**

Writes an image to the stream as a PNG image.

See also: QImage::save().

**QDataStream & operator>> (QDataStream & s, QImage & image)**

Reads an image from the stream.

See also: QImage::load().

#### 102.4.1. URL

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

# 103. QImageConsumer Class Reference

```
#include <qasyncimageio.h>
```

### 103.1. 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.

### 103.2. 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 displayed 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.

### 103.2.1. URL

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

# 104. QImageDecoder Class Reference

```
#include <qasyncimageio.h>
```

### 104.1. 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.

### 104.2. 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 due 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.

### 104.2.1. URL

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

# 105. QImageDrag Class Reference

`#include <qdragobject.h>`

Inherits QDragObject.

## 105.1. Detailed Description

The QImageDrag provides a drag-and-drop object for transferring 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.

## 105.2. 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 ( const QMimeSource * e )** `[static]`

Returns TRUE if the information in *e* can be decoded into an image.

See also: decode().

Examples:

desktop/desktop.cpp

**bool QImageDrag::decode ( const QMimeSource * e, QImage & img )** `[static]`

Attempts to decode the dropped information in *e* into *img,* returning TRUE if successful.

See also: canDecode().

**bool QImageDrag::decode ( const QMimeSource * 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().

Examples:

desktop/desktop.cpp

**void QImageDrag::setImage ( QImage image )** `[virtual]`

Sets the image to be dragged. You will need to call this if you did not pass the image during construction.

### 105.2.1. URL

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

# 106. QImageFormat Class Reference

`#include <qasyncimageio.h>`

## 106.1. 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.

## 106.2. 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.

### 106.2.1. URL

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

# 107. QImageFormatType Class Reference

`#include <qasyncimageio.h>`

## 107.1. 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 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.

## 107.2. 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 definite 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.

### 107.2.1. URL

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

# 108. QImageIO Class Reference

```
#include <qimage.h>
```

## 108.1. 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: PNG, 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.

See also: QImage, QPixmap, QFile and QMovie.

## 108.2. 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 QString & 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.

**QString QImageIO::description () const**

Returns the image description string.

See also: setDescription().

**QString 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 QString & 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 QString & 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 QString & 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 extensions 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().

### 108.2.1. URL

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

# 109. QIntCache Class Reference

`#include <qintcache.h>`

Inherits QGCache.

## 109.1. Detailed Description

The QIntCache class is a template class that provides a cache based on `long` keys.

QIntCache is implemented as a template 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, accessed via `long` keys. 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 removed.

Apart from insert(), by far the most important function is find() (which also exists as operator[]). This function looks up an item, returns it, and by default marks it as being the most recently used item.

There are also methods to remove() or take() an object from the cache. Calling setAutoDelete(TRUE) for a cache tells it to delete items that are removed. The default is to not delete items when then are removed (i.e. remove() and take() are equivalent).

When inserting an item into the cache, 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 cache item, the virtual function QCollection::deleteItem() is called. It's default implementation in QIntCache is to delete the item if auto-deletion is enabled.

There is a QIntCacheIterator which may be used to traverse the items in the cache in arbitrary order.

See also: QIntCacheIterator, QCache, QAsciiCache and Collection Classes

## 109.2. Member Function Documentation

### QIntCache::QIntCache ( const QIntCache<type> & c )

For internal use only.

### QIntCache::QIntCache ( int maxCost=100, int size=17 )

Constructs a cache with the following properties:

Arguments:

- *maxCost* is the maximum allowed total cost.
- *size* is the size of the internal hash array. Each inserted item is associated with a cost. When inserting a new item, if the total cost of all items in the cache will exceeds *maxCost,* the cache will start throwing out the older (recently least used) items until there is room enough for the new item to be inserted.

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.

### QIntCache::~QIntCache ()

Removes all items from the cache and destroys it. All iterators that access this cache will be reset.

### void QIntCache::clear () [virtual]

Removes all items from the cache, and deletes them if auto-deletion has been enabled.

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, int 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 remove old, least-used items until there is room for this new item.

**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 )**

Removes the item associated with *k,* and returns TRUE if the item was present in the cache or FALSE if it was not.

The item is deleted if auto-deletion has been enabled, i.e. you have called setAutoDelete(TRUE).

If there are two or more items with equal keys, then the one that was inserted last is is removed.

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 removed 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().

### 109.2.1. URL

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

# 110. QIntCacheIterator Class Reference

```
#include <qcache.h>
```

Inherits QGCacheIterator.

### 110.1. 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 item 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.

### 110.2. 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().

### 110.2.1. URL

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

# 111. QIntDict Class Reference

`#include <qintdict.h>`

Inherits QGDict.

## 111.1. Detailed Description

The QIntDict class is a template class that provides a dictionary based on `long` keys.

QIntDict is implemented as a template 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.

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, QAsciiDict, QPtrDict and Collection Classes

## 111.2. 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 traversing order.

See also: remove(), clear() and setAutoDelete().

### 111.2.1. URL

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

# 112. QIntDictIterator Class Reference

```
#include <qintdict.h>
```

Inherits QGDictIterator.

## 112.1. Detailed Description

The QIntDictIterator class provides an iterator for QIntDict collections.

QIntDictIterator is implemented as a template 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.

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

## 112.2. 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.

### 112.2.1. URL

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

# 113. QIntValidator Class Reference

`#include <qvalidator.h>`

Inherits QValidator.

## 113.1. 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.

## 113.2. 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().

**QValidator::State QIntValidator::validate( QString & input, int & ) const**

Returns *Acceptable* if *input* contains a number in the legal range, *Valid* if it contains another integer or is empty, and *Invalid* if *input* is not an integer.

### 113.2.1. URL

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

# 114. QIODevice Class Reference

`#include <qiodevice.h>`

Inherited by QBuffer and QFile.

### 114.1. 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 maxlen );
    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.

## 114.2. 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 QFile and QBuffer.

**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 QFile and QBuffer.

**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 QFile and QBuffer.

**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 QFile and QBuffer.

**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 QFile and QBuffer.

**QByteArray QIODevice::readAll ()**

This convenience function returns all of the remaining data in the device. Note that this only works for direct access devices, such as QFile.

**int QIODevice::readBlock ( char * data, uint maxlen )** `[virtual]`

Reads at most *maxlen* 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 QFile and QBuffer.

**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.
- `IO_OnCloseError` An unspecified error happend on close.

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 QByteArray & data )**

This convenience function is the same as calling writeBlock( data.data(), data.size() ).

**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.

### 114.2.1. URL

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

# 115. QIODeviceSource Class Reference

`#include <qasyncio.h>`

Inherits QDataSource.

### 115.1. 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).

### 115.2. 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.

### 115.2.1. URL

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

# 116. QKeyEvent Class Reference

`#include <qevent.h>`

Inherits QEvent.

## 116.1. 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:

picture/picture.cpp

## 116.2. Member Function Documentation

**QKeyEvent::QKeyEvent ( Type type, int key, int ascii, int state, const QString & text=QString::null, bool autorep=FALSE, ushort count=1 )**

Constructs a key event object.

The *type* parameter must be `QEvent::KeyPress` or `QEvent::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).

*text* will be returned by text().

If *autorep* is TRUE then isAutoRepeat() will be TRUE.

*count* is the number of single keys.

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. Internationalized software should use text() instead.

See also: text().

Examples:

picture/picture.cpp

**int QKeyEvent::count () const**

Returns the number of single keys for this event. If text() is not empty, this is the length of the string. But Qt also compresses auto repeated key presses for non-visible keycodes, such as BackSpace or Delete. It is save to ignore count() and simply react on new key events, but if your application cannot keep up with the system's keyboard repeat rate, the users might feel it is a bit klundgy. Smarter applications therefore check count() and contain fancy code to optimize repetitive keysstrokes, even if the screen update might temporarily fell behind.

**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().

**bool QKeyEvent::isAccepted () const**

Returns TRUE if the receiver of the event wants to keep the key.

**bool QKeyEvent::isAutoRepeat () const**

Returns TRUE if the event was generated by the auto-repeat feature of the system. Some applications will choose to ignore events of this type.

**int QKeyEvent::key () const**

Returns the code of the key that was pressed or released.

The header file qnamespace.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).

**ButtonState QKeyEvent::state () const**

Returns the keyboard modifier flags that existed immediately before the event occurred.

The returned value is `ShiftButton, ControlButton` and `AltButton` OR'ed together.

See also: stateAfter().

**ButtonState QKeyEvent::stateAfter () const**

Returns the keyboard modifier flags that existed immediately after the event occurred.

See also: state().

**QString QKeyEvent::text () const**

Returns the Unicode text which this key generated.

See also: QWidget::setKeyCompression().

### 116.2.1. URL

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

# 117. QLabel Class Reference

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

Inherits QFrame.

### 117.1. 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.

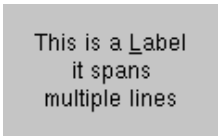
A label can be aligned in many different ways. The alignment setting specifies where to position the contents relative to the frame rectangle. See setAlignment() for a description of the alignment flags.

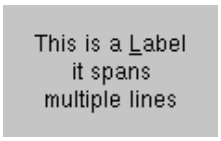
Enabling auto-resizing make the label resize itself whenever the contents change. The top left corner does not move.

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.




A label may also be used to provide a slightly larger amount of information, for example a few lines help text in a dialog. For this reason, QLabel also supports rich text rendering. The available styles are defined in the default stylesheet QStyleSheet::defaultSheet(). Usually the label autodetects from the set text whether rich text rendering is required. The format, however, can also be specified directly with setTextFormat(). Note that buddies will not work yet with rich text labels since there's not way to specify the accelerator (yet).

See also: QLineEdit, QMovie, QTextView and GUI Design Handbook: Label

Examples:

tabdialog/tabdialog.cpp
cursor/cursor.cpp
layout/layout.cpp
popup/popup.cpp
menu/menu.cpp
progress/progress.cpp
qmag/qmag.cpp
movies/main.cpp
customlayout/main.cpp

## 117.2. Member Function Documentation

**QLabel::QLabel ( QWidget * buddy, const QString & 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, const char * name=0, WFlags f=0 )**

Constructs an empty label which is left-aligned, vertically centered, has an automatic indent and with manual resizing.

The *parent, name* and *f* arguments are passed to the QFrame constructor.

See also: setAlignment(), setFrameStyle(), setIndent() and setAutoResize().

**QLabel::QLabel ( const QString & text, QWidget * parent, const char * name=0, WFlags f=0 )**

Constructs a label with a text. The label is left-aligned, vertically centered, has an automatic indent and with manual resizing.

The *parent, name* and *f* arguments are passed to the QFrame constructor.

See also: setAlignment(), setFrameStyle(), setIndent() 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.

**void QLabel::drawContentsMask ( QPainter * p )** `[virtual protected]`

Draws the label contents mask using the painter *p*. Used only in transparent mode.

See also: QWidget::setAutoMask();.

Reimplemented from QFrame.

**int QLabel::heightForWidth ( int w ) const** `[virtual]`

Reimplemented for rich text labels and WordBreak.

Reimplemented from QWidget.

**int QLabel::indent () const**

Returns the indent of the label.

See also: setIndent().

**QMovie* QLabel::movie () const**

Returns the QMovie currently displaying in the label, or 0 if none has been set.

See also: setMovie() and unsetMovie().

**QPixmap * QLabel::pixmap () const**

Returns the label pixmap.

See also: setPixmap().

**void QLabel::setAlignment ( int alignment )** `[virtual]`

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
popup/popup.cpp
qmag/qmag.cpp
customlayout/main.cpp

**void QLabel::setAutoResize ( bool enable )** `[virtual]`

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().

**void QLabel::setBuddy ( QWidget * buddy )** `[virtual]`

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:

layout/layout.cpp

**void QLabel::setIndent ( int indent )**

Sets the indent of the label to *indent* pixels.

The indent 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 *indent* is negative (as it is by default), the label computes the indent as follows: If the frame width is zero, the effective indent becomes 0. If the frame style is greater than zero, the effective indent becomes half the width of the "x" character (of the widget's current font.

Setting a non-negative indent gives the specified indent in pixels.

See also: indent(), frameWidth() and font().

Examples:

movies/main.cpp

**void QLabel::setMovie ( const QMovie & movie )** `[virtual 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.

See also: unsetMovie().

**void QLabel::setNum ( double num )** `[virtual 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 )** `[virtual 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 )** `[virtual 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(), setTextFormat() and setAutoResize().

**void QLabel::setText ( const QString & text )** `[virtual 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.

*text* may be interpreted either as plain text or as rich text, depending on the textFormat(). The default setting is `AutoText,` i.e. the label autodetects the format from *text.* If the format is `RichText,` the label uses the default mime source factory to resolve references to images. See QMimeSourceFactory::defaultFactory() for details.

Note that a label is only useful for rather small documents with one or maximal two lines of text. If you need to display larger documents, a QTextView is the widget of choice. It will flicker less on resize and can also provide a scrollbar if necessary.

See also: text(), setTextFormat(), setPixmap(), setAutoResize() and QTextView.

Examples:

cursor/cursor.cpp
layout/layout.cpp

popup/popup.cpp
qmag/qmag.cpp
customlayout/main.cpp

**void QLabel::setTextFormat ( Qt::TextFormat format )**

Sets the text format to *format.* Possible choices are

- `PlainText` - all characters are displayed verbatimely, including all blanks and linebreaks. Word wrap is availbe with the `WordBreak` alignment flag (see setAlignment() for details).
- `RichText` - rich text rendering. The available styles are defined in the default stylesheet QStyleSheet::defaultSheet().
- `AutoText` - this is also the default. The label autodetects which rendering style suits best, `PlainText` or `RichText`. Technically, this is done by using the QStyleSheet::mightBeRichText() heuristic.

**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, use heightForWidth().

Examples:

layout/layout.cpp

Reimplemented from QWidget.

**QSizePolicy QLabel::sizePolicy () const** `[virtual]`

Specifies that this widget may stretch horizontally and vertically beyond the sizeHint().

Reimplemented from QWidget.

**QString QLabel::text () const**

Returns the label text. This may be either plain text or a small rich text document.

See also: textFormat(), setText() and setTextFormat().

**Qt::TextFormat QLabel::textFormat() const**

Returns the current text format.

See also: setTextFormat().

### 117.2.1. URL

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

# 118. QLayout Class Reference

`#include <qabstractlayout.h>`

Inherits QLayoutItem and QObject.

Inherited by QBoxLayout and QGridLayout.

## 118.1. 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.

Most users of Q*Layout are likely to use some of the basic functions provided by QLayout, such as 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.

To make your own layout manager, make a subclass of QGLayoutIterator and implement the functions addItem(), sizeHint(), setGeometry() and iterator(). You should also implement minimumSize(), otherwise your layout will be resized to zero size if there is little space. To support children whose height depend on their widths, implement hasHeightForWidth() and heightForWidth(). See the custom layout page for an in-depth description and the layout overview page for an introduction.

Geometry management stops when the layout manager is deleted.

## 118.2. Member Type Documentation

### QLayout::ResizeMode

The possible values are are:

- `Fixed` - the main widget's size is set to sizeHint(), it cannot be resized at all.
- `Minimum` - The main widget's minimum size is set to minimumSize(), it cannot be smaller.
- `FreeResize` - the widget is not constrained.

### 118.3. Member Function Documentation

**QLayout::QLayout ( QWidget * parent, int border=0, int space=-1, const char * name=0 )**

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. *space* sets the value of spacing(), which gives the spacing between widgets. The default value for *space* is -1, which means that the value of *border* is used.

*name* is the internal object name

There can only be one top-level layout for a widget. It is returned by QWidget::layout().

**QLayout::QLayout ( QLayout * parentLayout, int space=-1, const char * name=0 )**

Constructs a new child QLayout, and places it inside *parentLayout,* using the default placement defined by addItem().

If *space* is -1, this QLayout inherits *parentLayout's* spacing(), otherwise *space* is used.

**QLayout::QLayout ( int space=-1, const char * name=0 )**

Constructs a new child QLayout, If *space* is -1, this QLayout inherits its parent's spacing(), otherwise *space* is used.

This layout has to be inserted into another layout before geometry management will work.

**QLayout::~QLayout ()**

Deletes all layout children. Geometry management stops when a toplevel layout is deleted.

**bool QLayout::activate ()**

Redoes the layout for mainWidget(). You should generally not need to call this, as it is automatically called at most appropriate times.

However, if you set up a QLayout for a visible widget without resizing that widget, you need to call this function in order to lay it out.

See also: QWidget::updateGeometry().

Examples:

layout/layout.cpp
popup/popup.cpp
scrollview/scrollview.cpp

**void QLayout::add ( QWidget * w )**

Adds *w* to this layout in a manner specific to the layout. This function uses addItem.

Examples:

customlayout/main.cpp

**void QLayout::addChildLayout ( QLayout * l )** `[protected]`

This function is called from addLayout functions in subclasses, to add *l* layout as a sublayout.

**void QLayout::addItem ( QLayoutItem * item )** `[virtual]`

Implemented in subclasses to add *item.* How it is added is specific to each subclass.

Note that the ownership of *item* is transferred to the layout, and it is the layout's responsibility to

delete it.

Reimplemented in QGridLayout and QBoxLayout.

**bool QLayout::autoAdd () const**

Returns TRUE if this layout automatically grabs all new mainWidget()'s new children and adds them as defined by addItem(). This only has effect for top-level layouts, ie. layouts that are direct children of their mainWidget().

autoAdd() is disabled by default.

See also: setAutoAdd().

**void QLayout::deleteAllItems ()** `[protected]`

Removes and deletes all items in this layout.

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

Performs child widget layout when the parent widget is resized. Also handles removal of widgets and child layouts.

Reimplemented from QObject.

**QRect QLayout::geometry () const** `[virtual]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QLayoutItem.

**bool QLayout::isEmpty () const** `[virtual]`

Returns TRUE if this layout is empty. The default implementation returns FALSE.

Reimplemented from QLayoutItem.

**bool QLayout::isTopLevel () const**

Returns TRUE if this layout is a top level layout, i.e. not a child of another layout.

**QLayout * QLayout::layout ()** `[virtual]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QLayoutItem.

**QWidget * QLayout::mainWidget ()**

Returns the main widget (parent widget) of this layout, or 0 if this layout is a sub-layout which is not yet inserted.

**int QLayout::margin () const**

returns the width of the outside border of the layout.

See also: spacing() and setMargin().

**QSize QLayout::maximumSize () const** `[virtual]`

Returns the maximum size of this layout. This is the largest size that the layout can have, while still respecting the specifications. Does not include what's needed by margin() or menuBar().

The default implementation allows unlimited resizing.

Reimplemented from QLayoutItem.

**QMenuBar* QLayout::menuBar () const**

Returns the menu bar set for this layout, or a null pointer if no menu bar is set.

**QSize QLayout::minimumSize () const** `[virtual]`

Returns the minimum size of this layout. This is the smallest size that the layout can have, while still respecting the specifications. Does not include what's needed by margin() or menuBar().

The default implementation allows unlimited resizing.

Reimplemented from QLayoutItem.

**QLayout::ResizeMode QLayout::resizeMode() const**

Returns the resize mode.

**void QLayout::setAutoAdd ( bool b )** `[virtual]`

Sets autoAdd() if *b* is TRUE.

See also: autoAdd().

**void QLayout::setGeometry ( const QRect & r )** `[virtual]`

This function is reimplemented in subclasses to perform layout.

The default implementation maintains the geometry() information. Reimplementors must call this

function.

Reimplemented from QLayoutItem.

**void QLayout::setMargin ( int border )** `[virtual]`

Sets the outside border of the layout to *border*.

See also: margin() and setSpacing().

**void QLayout::setMenuBar ( QMenuBar * w )** `[virtual]`

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 QMenuBar.

Examples:

scrollview/scrollview.cpp

**void QLayout::setResizeMode ( ResizeMode mode )**

Sets the resize mode to *mode*.

The default mode is `Minimum` for top level widgets, and `FreeResize` for all others.

See also: QLayout::ResizeMode.

**void QLayout::setSpacing ( int space )** `[virtual]`

Sets the internal spacing of the layout to *space*.

See also: spacing() and setMargin().

**int QLayout::spacing () const**

Returns the spacing between widgets inside the layout.

See also: margin() and setSpacing().

**int QLayout::totalHeightForWidth ( int w ) const**

For internal use only.

**QSize QLayout::totalMaximumSize () const**

For internal use only.

**QSize QLayout::totalMinimumSize () const**

For internal use only.

**QSize QLayout::totalSizeHint () const**

For internal use only.

### 118.3.1. URL

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

# 119. QLayoutItem Class Reference

`#include <qabstractlayout.h>`

Inherited by QLayout, QSpacerItem and QWidgetItem.

## 119.1. Detailed Description

The abstract items which a QLayout manipulates.

For custom layouts.

See also: QLayout.

## 119.2. Member Function Documentation

**QLayoutItem::QLayoutItem ( int alignment = 0 )**

Constructs a layout item with alignment *alignment.* Alignment may not be supported by all subclasses.

**QLayoutItem::~QLayoutItem ()** `[virtual]`

Destructs the QLayoutItem.

**int QLayoutItem::alignment () const**

Returns the alignment of this item.

**QSizePolicy::ExpandData QLayoutItem::expanding() const** `[virtual]`

Implemented in subclasses to return whether this item "wants" to expand.

Reimplemented in QWidgetItem, QBoxLayout, QGridLayout, QSpacerItem and QLayout.

**QRect QLayoutItem::geometry () const** `[virtual]`

Returns the rectangle covered by this layout item.

Reimplemented in QWidgetItem, QLayout and QSpacerItem.

**bool QLayoutItem::hasHeightForWidth () const** `[virtual]`

Returns TRUE if this layout's preferred height depends on its width. The default implementation returns FALSE;

Reimplement this function in layout managers that support height for width.

See also: heightForWidth() and QWidget::heightForWidth().

Reimplemented in QBoxLayout, QGridLayout and QWidgetItem.

**int QLayoutItem::heightForWidth ( int ) const** `[virtual]`

Returns the preferred height for this layout item, given the width *w*.

The default implementation returns -1, indicating that the preferred height is independent of the width of the item. Using the function hasHeightForWidth() will typically be much faster than calling this function and testing for -1.

Reimplement this function in layout managers that support height for width. A typical implementation will look like this:

```
int MyLayout::heightForWidth( int w ) const
{
    if ( cache_dirty || cached_width != w ) {
        //Not all C++ compilers support "mutable" yet:
        MyLayout * mthis = (MyLayout*)this;
        int h = calculateHeightForWidth( w );
        mthis->cached_hfw = h;
        return h;
    }
    return cached_hfw;
}
```

Caching is strongly recommended, without it layout will take exponential time.

See also: hasHeightForWidth().

Reimplemented in QWidgetItem, QBoxLayout and QGridLayout.

**void QLayoutItem::invalidate ()** `[virtual]`

Invalidates any cached information in this layout item.

Reimplemented in QBoxLayout and QGridLayout.

**virtual bool QLayoutItem::isEmpty () const** `[virtual]`

Implemented in subclasses to return whether this item is empty, i.e. whether it contains any widgets.

Reimplemented in QWidgetItem, QSpacerItem and QLayout.

**QLayoutIterator QLayoutItem::iterator ()** `[virtual]`

Returns an iterator over this item's QLayoutItem children. The default implementation returns an empty iterator.

Reimplement this function in subclasses that can have children.

Reimplemented in QGridLayout and QBoxLayout.

**QLayout * QLayoutItem::layout ()** `[virtual]`

If this item is a QLayout, return it as a QLayout, otherwise return 0. This function provides type-safe casting.

Reimplemented in QLayout.

**QSize QLayoutItem::maximumSize () const** `[virtual]`

Implemented in subclasses to return the maximum size of this item.

Reimplemented in QBoxLayout, QLayout, QSpacerItem, QGridLayout and QWidgetItem.

**QSize QLayoutItem::minimumSize () const** `[virtual]`

Implemented in subclasses to return the minimum size of this item.

Reimplemented in QSpacerItem, QLayout, QWidgetItem, QGridLayout and QBoxLayout.

**void QLayoutItem::setAlignment ( int a )** `[virtual]`

Sets the alignment of this item to *a*.

**void QLayoutItem::setGeometry ( const QRect & r )** `[virtual]`

Implemented in subclasses to set this item's geometry to *r*.

Reimplemented in QLayout, QSpacerItem, QWidgetItem, QBoxLayout and QGridLayout.

**QSize QLayoutItem::sizeHint () const** `[virtual]`

Implemented in subclasses to return the preferred size of this item.

Reimplemented in QSpacerItem and QWidgetItem.

**QSpacerItem * QLayoutItem::spacerItem ()** `[virtual]`

If this item is a QSpacerItem, return it as a QSpacerItem, otherwise return 0. This function provides type-safe casting.

Reimplemented in QSpacerItem.

**QWidget * QLayoutItem::widget ()** `[virtual]`

If this item is a QWidgetItem, the managed widget is returned. The default implementation returns

0;.

### 119.2.1. URL

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

# 120. QLayoutIterator Class Reference

```
#include <qabstractlayout.h>
```

### 120.1. Detailed Description

The QLayoutIterator class provides iterators over QLayoutItem

Use QLayoutItem::iterator() to create an iterator over a layout.

QLayoutIterator uses explicit sharing with a reference count. If an iterator is copied, and one of the copies is modified, both iterators will be modified.

A QLayoutIterator is not protected against changes in its layout. If the layout is modified or deleted, the iterator will become invalid. It is not posible to test for validity. It is safe to delete an invalid layout. Any other access may lead to an illegal memory reference, and the abnormal termination of the program.

Calling takeCurrent() or deleteCurrent() leaves the iterator in a valid state, but may invalidate any other iterators that access the same layout.

The following code will draw a rectangle for each layout item in the layout structure of the widget.

```
static void paintLayout( QPainter *p, QLayoutItem *lay )
{
    QLayoutIterator it = lay->iterator();
    QLayoutItem *child;
    while ( (child = it.current() ) ) {
        paintLayout( p, child );
        it.next();
    }
    p->drawRect( lay->geometry() );
}
void ExampleWidget::paintEvent( QPaintEvent * )
{
    QPainter p( this );
    if ( layout() )
```

```
        paintLayout( &p, layout() );
}
```

All the functionality of QLayoutIterator is implemented by subclasses of QGLayoutIterator. Note that there is not much point in subclassing QLayoutIterator, since none of the functions are virtual.

## 120.2. Member Function Documentation

### QLayoutIterator::QLayoutIterator ( QGLayoutIterator * gi )

Constructs an iterator based on *gi.* The constructed iterator takes ownership of *gi,* and will delete it.

This constructor is provided for layout implementors. Application programmers should use QLayoutItem::iterator() to create an iterator over a layout.

### QLayoutIterator::QLayoutIterator ( const QLayoutIterator & i )

Creates a shallow copy of *i;* if the copy is modified, then the original will also be modified.

### QLayoutIterator::~QLayoutIterator ()

Destroys the iterator.

### QLayoutItem * QLayoutIterator::current ()

Returns the current item, or 0 if there is no current item.

### void QLayoutIterator::deleteCurrent ()

Removes and deletes the current child item from the layout and moves the iterator to the next item. This iterator will still be valid, but any other iterator over the same layout may become invalid.

### QLayoutItem * QLayoutIterator::operator++ ()

Moves the iterator to the next child item, and returns that item, or 0 if there is no such item.

**QLayoutIterator & QLayoutIterator::operator= ( const QLayoutIterator & i )**

Assigns *i* to this iterator and returns a reference to this iterator.

**QLayoutItem * QLayoutIterator::takeCurrent ()**

Removes the current child item from the layout without deleting it and moves the iterator to the next item. Returns the removed item, or 0 if there was no item to be removed. This iterator will still be valid, but any other iterator over the same layout may become invalid.

### 120.2.1. URL

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

# 121. QLCDNumber Class Reference

```
#include <qlcdnumber.h>
```

Inherits QFrame.

### 121.1. 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 not possible 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.

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:

xform/xform.cpp

### 121.2. 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:

xform/xform.cpp

**void QLCDNumber::display ( const QString & 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(QString ) was the last called, 0 is returned.

See also: value() and display().

**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::setBinMode ()** `[virtual 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 ()** `[virtual 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 ()** `[virtual 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 )** `[virtual]`

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 )** `[virtual]`

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 ()** `[virtual 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 )** `[virtual]`

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 )** `[virtual 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.

**QSizePolicy QLCDNumber::sizePolicy () const** `[virtual]`

Specifies that this widget may stretch horizontally and vertically beyond the sizeHint().

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(QString ) was the last called, 0 is returned.

See also: intValue() and display().

### 121.2.1. URL

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

# 122. QLineEdit Class Reference

```
#include <qlineedit.h>
```

Inherits QWidget.

Inherited by QRenameEdit.

### 122.1. 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.

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:

tabdialog/tabdialog.cpp
xform/xform.cpp
layout/layout.cpp
popup/popup.cpp

## 122.2. Member Type Documentation

### QLineEdit::EchoMode

This enum type describes the ways in which QLineEdit can display its contents. The currently defined values 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.

See also: setEchoMode(), echoMode() and QMultiLineEdit::EchoMode.

## 122.3. Member Function Documentation

### QLineEdit::QLineEdit ( QWidget * parent, 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 ( const QString & contents, QWidget * parent, const char * name=0 )

Constructs a line editor whose edit buffer starts out as *contents.*

The cursor position is set to the end of the line and the maximum buffer size to 32767 characters.

The *parent* and *name* arguments are sent to the QWidget constructor.

### QLineEdit::~QLineEdit ()

Destroys the line editor.

### int QLineEdit::alignment () const

Returns the current alignment of the line editor. Possible Values are Qt::AlignLeft, Qt::AlignRight and Qt::AlignCenter.

See also: setAlignment().

**void QLineEdit::backspace ()**

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 will be put at the beginning of the marked text and the marked text will be removed.

See also: del().

**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 ).

**void QLineEdit::copy () const**

Copies the marked text to the clipboard, if there is any and echoMode() is Normal.

See also: cut() and paste().

**void QLineEdit::cursorLeft ( bool mark, int steps = 1 )**

Moves the cursor leftwards one or more characters.

See also: cursorRight().

**int QLineEdit::cursorPosition () const**

Returns the current cursor position for this line edit.

See also: setCursorPosition().

**void QLineEdit::cursorRight ( bool mark, int steps = 1 )**

Moves the cursor rightwards one or more characters.

See also: cursorLeft().

**void QLineEdit::cursorWordBackward ( bool mark )**

Moves the cursor one word to the left. If *mark* is TRUE, the text is marked.

See also: cursorWordForward().

**void QLineEdit::cursorWordForward ( bool mark )**

Moves the cursor one word to the right. If *mark* is TRUE, the text is marked.

See also: cursorWordBackward().

**void QLineEdit::cut ()**

Copies the marked text to the clipboard and deletes it, if there is any.

If the current validator disallows deleting the marked text, cut() will copy it but not delete it.

See also: copy() and paste().

**void QLineEdit::del ()**

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 will be put at the beginning of the marked text and the marked text will be removed.

See also: backspace().

**void QLineEdit::deselect ()** `[slot]`

Deselects all text (i.e. removes marking) and leaves the cursor at the current position.

**QString QLineEdit::displayText () const**

Returns the text that's currently displayed. This is normally the same as text(), but can be e.g. "*****" if EchoMode is Password or "" if it is NoEcho.

See also: setEchoMode(), text() and EchoMode.

**QLineEdit::EchoMode QLineEdit::echoMode() const**

Returns the current echo mode of the line edit.

See also: setEchoMode() and EchoMode.

**bool QLineEdit::edited () const**

Returns the edited flag of the line edit. If this returns FALSE, the line edit's contents have not been changed since the construction of the QLineEdit (or the last call to setEdited( FALSE ), if any). If it returns true, the contents have been edited, or setEdited( TRUE ) has been called.

See also: setEdited().

**void QLineEdit::end ( bool mark )**

Moves the text cursor to the right end of the line. If mark is TRUE text will be marked towards the last position, if not any marked text will be unmarked if the cursor is moved.

See also: home().

**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**

Returns TRUE if part of the text has been marked by the user (e.g. by clicking and dragging).

**void QLineEdit::home ( bool mark )**

Moves the text cursor to the left end of the line. If mark is TRUE text will be marked towards the first position, if not any marked text will be unmarked if the cursor is moved.

See also: end().

**void QLineEdit::insert ( const QString & 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**

Returns the text marked by the user (e.g. by clicking and dragging), or a null string 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().

**QSize QLineEdit::minimumSizeHint () const** `[virtual]`

Returns a minimum size for the line edit.

The width returned tends to be enough for one character.

Reimplemented from QWidget.

**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::paste ()**

Inserts the clipboard's text at the cursor position, deleting any previous marked text.

If the end result is not acceptable for the current validator, nothing happens.

See also: copy() and cut().

**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::setAlignment ( int flag )**

Sets the alignment of the line editor. Possible Values are Qt::AlignLeft, Qt::AlignRight and Qt::AlignCenter.

See also: alignment().

**void QLineEdit::setCursorPosition ( int newPos )** `[virtual]`

Set the cursor position for this line edit to *newPos* and repaint accordingly.

See also: cursorPosition().

**void QLineEdit::setEchoMode ( EchoMode mode )** `[virtual]`

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(), EchoMode and displayText().

**void QLineEdit::setEdited ( bool on )**

Sets the edited flag of this line edit to *on*. The edited flag is never read by QLineEdit, and is changed to TRUE whenever the user changes its contents.

This is useful e.g. for things that need to provide a default value, but cannot find the default at once.

Just open the line edit without the best default and when the default is known, check the edited()

return value and set the line edit's contents if the user has not started editing the line edit.

See also: edited().

**void QLineEdit::setEnabled ( bool e )** `[virtual]`

Reimplemented for internal reasons; the API is not affected.

**void QLineEdit::setFont ( const QFont & f )** `[virtual]`

Reimplemented for internal reasons; the API is not affected.

**void QLineEdit::setFrame ( bool enable )** `[virtual]`

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 )** `[virtual]`

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.

**void QLineEdit::setSelection ( int start, int length )** `[virtual]`

Sets the marked area of this line edit to start at *start* and be *length* characters long.

**void QLineEdit::setText ( const QString & text )** `[virtual 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 ( const QValidator * v )** `[virtual]`

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.

Reimplemented from QWidget.

**QSizePolicy QLineEdit::sizePolicy () const** `[virtual]`

Specifies that this widget can use more, but is able to survive on less, horizontal space; and is fixed vertically.

Reimplemented from QWidget.

**QString QLineEdit::text () const**

Returns the text currently in the line.

See also: setText().

**void QLineEdit::textChanged ( const QString & )** `[signal]`

This signal is emitted every time the text has changed. The argument is the new text.

**bool QLineEdit::validateAndSet ( const QString & 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.

Linebreaks in *newText* are converted to spaces, and it is truncated to maxLength() if necessary before

testing its validity.

Repaints and emits textChanged() if appropriate.

**const QValidator * QLineEdit::validator () const**

Returns a pointer to the current input validator, or 0 if no validator has been set.

### 122.3.1. URL

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

# 123. QList Class Reference

```
#include <qlist.h>
```

Inherits QGList.

Inherited by QStrList.

### 123.1. Detailed Description

The QList class is a template class that provides doubly linked lists.

In Qt 2.0 QList is only implemented as a template 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 QString& name, int salary ) { n=name; s=salary; }
    QString     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().ascii(), emp->salary() );
}
```

Program output:

```
Bill earns 50000
Steve earns 80000
Ron earns 60000
```

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

### 123.2. 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

**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.

Please note that inSort is slow. If you want to insert lots of items in a list and sort after inserting then you should use \link sort(). inSort() takes up to O(n) compares. That means inserting n items in your list will need O(n^2) compares while sort() only needs O(n*logn) for the same task. So you inSort() only if you already have a presorted list and want to insert only few additional items.

See also: insert(), compareItems(), current() and sort().

**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 item 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

**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 item 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 occurrence 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 occurrence 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.

### 123.2.1. URL

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

# 124. QListBox Class Reference

```
#include <qlistbox.h>
```

Inherits QScrollView.

Inherited by QFileListBox.

### 124.1. Detailed Description

The QListBox widget provides a list of selectable, read-only items.

This is typically a single-column list where zero or one item items are selected at once, but can also be used in many other ways.

QListBox will add scroll bars as necessary, but isn't intended for *really* big lists. If you want more than a few thousand items, it's probably better to use a different widget, chiefly because the scroll bars won't provide very good navigation, but also because QListBox may become slow at larger sizes.

There is a variety of selection modes. The default is single-selection, but you can change it using setSelectionMode().

The list box normally arranges its items in a single column with a vertical scroll bar if necessary, but it is also possible to have a different fixed number of columns (setColumnMode()), or as many columns as will fit in the list box' assigned screen space (setColumnMode( FitToWidth )), or to have a fixed number of rows (setRowMode()), or as many rows as will fit in the list box' assigned screen space (setRowMode( FitToHeight )). In all these cases, QListBox will add scroll bars as appropriate in at least one direction.

If multiple rows is used, each row can be as high as necessary (the normal setting), or you can request that all items will have the same height by calling setVariableHeight( FALSE ). Of course there is a similar setVariableWidth().

The items discussed are QListBoxItem objects. QListBox provides methods to insert new items as a string, as pixmaps, and as QListBoxItem * (insertItem() with various arguments), and to replace an

existing item with a new string, pixmap or QListBoxItem (changeItem() with various arguments). You can also remove items (surprise: removeItem()) and clear() the entire list box. Note that if you create a QListBoxItem yourself and insert it, it becomes the property of QListBox and you may not delete it. (QListBox will delete it when appropriate.)

You can also create a QListBoxItem such as QListBoxText or QListBoxPixmap with the list box as first parameter. The item will then append itself. When deleteting an item, it will automatically be remove from the listbox again.

The list of items can be arbitrarily big; if necessary, QListBox adds scroll bars. It can be single-column (as most list boxes are) or multi-column, and offers both single and multiple selection. (QListBox does however not support multiple-column items; QListView does that job.)

The list box items can be accessed both as QListBoxItem objects (recommended) and using integer indexes (the original QListBox implementation used an array of strings internally, and the API still supports this mode of operation). Everything can be done using the new objects; most things can be done using the indexes too but unfortunately not everything.

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 `WheelFocus` as a default focusPolicy(), i.e. it can get keyboard focus both by tabbing, clicking and the mouse wheel.

New items may be inserted using either insertItem(), insertStrList() and inSort().

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 is does not need them any more.

See also: QListView, QComboBox, QButtonGroup and GUI Design Handbook: List Box (two sections)

Examples:

tabdialog/tabdialog.cpp

## 124.2. Member Type Documentation

### QListBox::LayoutMode

This enum type decides how QListBox lays out its rows and columns. The two modes interact, of course.

The possible values for each mode are:

- `FixedNumber` - there is a fixed number of rows (or columns).
- `FitToHeight` - there are as many rows as will fit on-screen. (Ditto with `FitToWidth` and columns.)

- `Variable` - there are as many rows as are required by the column mode. (Or as many columns as required by the row mode.)

Example: When you call setRowMode( FitToHeight ), columnMode() automatically becomes `Variable` to accomodate the row mode you've set.

**QListBox::SelectionMode**

This enumerated type is used by QListBox to indicate how it reacts to selection by the user. It has three values:

- `Single` - When the user selects an item, any already-selected item becomes unselected, and the user cannot unselect the selected item. This means that the user can never clear the selection, even though the selection may be cleared by the application programmer using QListBox::clear().
- `Extended` - When the user selects an item in the most ordinary way, the selection is cleared and the new item selected. However, if the user then selects more items *in the same operation* (by dragging the mouse), these other items are also selected. There may also be other ways to select multiple items.
- `Multi` - When the user selects an item in the most ordinary way, the selection status of that item is toggled and the other items are left alone.

In other words, `Single` is a real single-selection list box, `Multi` a real multi-selection list box, and `Extended` list box where users can select multiple items but usually want to select either just one or a range of contiguous items.

## 124.3. 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 QScrollView constructor.

**QListBox::~QListBox ()**

Destroys the list box. Deletes all list box items.

**void QListBox::changeItem ( const QListBoxItem * lbi, int index )**

Replaces the item at position *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 QString & 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::clearSelection ()** `[virtual slot]`

Deselects all items, if possible.

Note that a single-selection listbox will automatically select an item if it has keyboard focus.

**QListBox::LayoutMode QListBox::columnMode() const**

Returns the column layout mode for this list box. This is normally `FixedNumber,` but can be changed by calling setColumnMode()

See also: rowMode(), setColumnMode() and numColumns().

**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().

**void QListBox::doLayout () const** `[protected]`

This function does the hard layout work. You should never need to call it.

**void QListBox::ensureCurrentVisible ()** `[virtual slot]`

Ensures that the current item is visible.

**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 ( QListBoxItem * )** `[signal]`

This signal is emitted when the user highlights a new current item. The argument is a pointer to the new current item.

See also: selected(), currentItem() and selectionChanged().

**void QListBox::highlighted ( const QString & )** `[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().

**int QListBox::index ( const QListBoxItem * lbi ) const**

Returns the index of *lbi,* or -1 if the item is not in this list box.

See also: item().

**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().

**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 QString & 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().

Examples:

tabdialog/tabdialog.cpp

**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.

**Warning:** This function uses `const char *` rather than QString, so we recommend against using it. It is provided so that legacy code will continue to work, and so that programs that certainly will not need to handle code outside a single 8-bit locale can use it. See insertStringList() - it uses real QStrings.

**Warning:** This function is never significantly faster than a loop around insertItem().

See also: insertItem() and inSort().

**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.

**Warning:** This function uses `const char *` rather than QString, so we recommend against using it. It is provided so that legacy code will continue to work, and so that programs that certainly will not need to handle code outside a single 8-bit locale can use it. See insertStringList() - it uses real QStrings.

**Warning:** This function is never significantly faster than a loop around insertItem().

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.

**Warning:** This function uses `const char *` rather than QString, so we recommend against using it. It is provided so that legacy code will continue to work, and so that programs that certainly will not need to handle code outside a single 8-bit locale can use it. See insertStringList() - it uses real QStrings.

**Warning:** This function is never significantly faster than a loop around insertItem().

See also: insertItem() and inSort().

**void QListBox::insertStringList ( const QStringList & 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.

**Warning:** This function is never significantly faster than a loop around insertItem().

See also: insertItem() and inSort().

**bool QListBox::isMultiSelection () const**

Returns TRUE if the listbox is in multi-selection mode or extended selection mode, and FALSE if it is in single-selection mode.

See also: selectionMode() and setSelectionMode().

**bool QListBox::isSelected ( const QListBoxItem * i ) const**

Returns TRUE if item *i* is selected. Returns FALSE if it is not selected or if there is an error.

**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**

Returns a pointer to the item at position *index,* or 0 if *index* is out of bounds.

See also: index().

**QListBoxItem * QListBox::itemAt ( QPoint p ) const**

Returns a pointer to the item at *p,* which is in on-screen coordinates, or a null pointer if there is no item at *p.*

**int QListBox::itemHeight ( int index = 0 ) const**

Returns the height in pixels of the item with index *index. index* defaults to 0.

If *index* is too large, this function returns 0.

**QRect QListBox::itemRect ( QListBoxItem * item ) const**

Returns the rectangle on the screen *item* occupies in viewport()'s coordinates, or an invalid rectangle if *i* is a null pointer or is not currently visible.

**bool QListBox::itemVisible ( const QListBoxItem * item )**

Returns TRUE if *item* is at least partly visible, or else FALSE.

**bool QListBox::itemVisible ( int index )**

Returns TRUE if the item at position *index* is at least partly visible.

**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.

**QSize QListBox::minimumSizeHint () const** `[virtual]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QWidget.

**void QListBox::mouseDoubleClickEvent ( QMouseEvent * )** `[virtual protected]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QWidget.

**void QListBox::mouseMoveEvent ( QMouseEvent * e )** `[virtual protected]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QWidget.

**void QListBox::mousePressEvent ( QMouseEvent * e )** `[virtual protected]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QWidget.

**void QListBox::mouseReleaseEvent ( QMouseEvent * e )** `[virtual protected]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QWidget.

**int QListBox::numColumns () const**

Returns the number of columns in the list box. This is normally 1, but can be different if setColumnMode() or setRowMode() has been called.

See also: setColumnMode(), setRowMode() and numRows().

**int QListBox::numItemsVisible () const**

Returns the number of visible items. Both partially and entirely visible items are counted.

**int QListBox::numRows () const**

Returns the number of rows in the list box. This is equal to the number of items in the default single-column layout, but can be different.

See also: setRowMode() and numColumns().

**void QListBox::paintCell ( QPainter * p, int row, int col )** `[virtual protected]`

Provided for compatibility with the old QListBox. We recommend using QListBoxItem::paint().

**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 and deletes 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]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QWidget.

**QListBox::LayoutMode QListBox::rowMode() const**

Returns the row layout mode for this list box. This is normally `Variable,` but can be changed by calling setRowMode().

See also: columnMode(), setRowMode() and numRows().

**void QListBox::selected ( QListBoxItem * )** `[signal]`

This signal is emitted when the user double-clicks on an item or presses return when an item is highlighted. The argument is a pointer to the new selected item.

See also: highlighted() and selectionChanged().

**void QListBox::selected ( const QString & )** `[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().

**QListBox::SelectionMode QListBox::selectionMode() const**

Returns the selection mode of the list box. The initial mode is `Single.`

See also: setSelectionMode().

**void QListBox::setBottomItem ( int index )** `[virtual]`

Scrolls the list box so the item at position *index* in the list is displayed in the bottom row of the list box.

See also: setTopItem().

**void QListBox::setColumnMode ( LayoutMode mode )** `[virtual]`

Sets the column layout mode to *mode,* and the number of displayed columns accordingly.

The row layout mode implicitly becomes `Variable.`

If *mode* is `Variable,` this function returns without doing anything.

**void QListBox::setColumnMode ( int columns )** `[virtual]`

Sets the column layout mode for this list box to `FixedNumber,` and sets the number of displayed columns accordingly.

See also: setRowMode(), columnMode() and numColumns().

**void QListBox::setCurrentItem ( QListBoxItem * i )** `[virtual]`

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().

Examples:

tabdialog/tabdialog.cpp

**void QListBox::setCurrentItem ( int index )** `[virtual]`

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

This is a bit slower than the QListBoxItem * version.

**void QListBox::setFont ( const QFont & font )** `[virtual]`

Reimplemented for internal reasons; the API is not affected.

**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. We recommend using setSelectionMode() instead; that function also offers a third mode of selection.

See also: setSelectionMode() and selectionMode().

**void QListBox::setRowMode ( LayoutMode mode )** `[virtual]`

Sets the row layout mode to *mode,* and the number of displayed rows accordingly.

The column layout mode implicitly becomes `Variable.`

If *mode* is `Variable,` this function returns without doing anything.

**void QListBox::setRowMode ( int rows )** `[virtual]`

Sets the row layout mode for this list box to `FixedNumber` and sets the number of displayed rows accordingly.

See also: setColumnMode(), rowMode() and numRows().

**void QListBox::setSelected ( QListBoxItem * item, bool select )** `[virtual]`

Selects *item* if *select* is TRUE, or unselects it if *select* is FALSE, and repaints the item appropriately.

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::setSelected ( int index, bool select )**

Selects the item at position *index* if *select* is TRUE, or unselects it if *select* is FALSE, and repaints the item appropriately.

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::setSelectionMode ( SelectionMode mode )** `[virtual]`

Sets the list box to selection mode *mode,* which may be one of `Single` (the default), *Extended* and `Multi`.

See also: selectionMode().

**void QListBox::setTopItem ( int index )** `[virtual]`

Scrolls the list box so the item at position *index* in the list is displayed in the top row of the list box.

See also: topItem() and ensureCurrentVisible().

**void QListBox::setVariableHeight ( bool enable )** `[virtual]`

Sets this list box to have variable-height rows if *enable* is TRUE, and equal-height rows if *enable* is FALSE.

When the list box has variable-height rows, each row is as high as the highest item in that row.

When it has same-sized rows, all rows are as high as the highest item in the list box.

The default is TRUE.

See also: setVariableWidth() and variableHeight().

**void QListBox::setVariableWidth ( bool enable )** `[virtual]`

Sets this list box to have variable-width columns if *enable* is TRUE, and equal-width columns if *enable* is FALSE.

When the list box has variable-width columns, each column is as wide as the widest item in that

column. When it has same-sized columns, all columns are as wide as the widest item in the list box.

The default is FALSE.

See also: setVariableHeight() and variableWidth().

**void QListBox::showEvent ( QShowEvent * )** `[virtual protected]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QWidget.

**QSize QListBox::sizeHint () const** `[virtual]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QWidget.

**void QListBox::takeItem ( const QListBoxItem * item )**

Removes *item* from the listbox and causes an update of the screen display. The item is not deleted. You should normally not need to call this function, as QListBoxItem::~QListBoxItem() calls it. The normal way to delete an item is `delete`.

See also: QListBox::insertItem().

**QString QListBox::text ( int index ) const**

Returns the text at position *index,* or a null string if there is no text at that position.

See also: pixmap().

**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 the index of an item at the top of the screen. If there are more than one of them, an arbitrary item is selected and returned.

**void QListBox::triggerUpdate ( bool doLayout )**

Ensures that a single paint event will occur at the end of the current event loop iteration. If *doLayout* is TRUE, the layout is also redone.

**void QListBox::updateItem ( QListBoxItem * i )** `[protected]`

Repaints *i.*

**void QListBox::updateItem ( int index )** `[protected]`

Repaints the item at position *index* in the list.

**bool QListBox::variableHeight () const**

Returns TRUE if this list box has variable-height rows, and FALSE if all the rows have the same height.

See also: setVariableHeight() and setVariableWidth().

**bool QListBox::variableWidth () const**

Returns TRUE if this list box has variable-width columns, and FALSE if all the columns have the same width.

See also: setVariableHeight() and setVariableWidth().

**void QListBox::viewportMouseDoubleClickEvent ( QMouseEvent * e ) [virtual protected]**

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QScrollView.

**void QListBox::viewportMouseMoveEvent ( QMouseEvent * e ) [virtual protected]**

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QScrollView.

**void QListBox::viewportMousePressEvent ( QMouseEvent * e ) [virtual protected]**

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QScrollView.

**void QListBox::viewportMouseReleaseEvent ( QMouseEvent * e ) [virtual protected]**

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QScrollView.

**void QListBox::viewportPaintEvent ( QPaintEvent * e ) [virtual]**

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QScrollView.

### 124.3.1. URL

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

# 125. QListBoxItem Class Reference

```
#include <qlistbox.h>
```

Inherited by QListBoxPixmap and QListBoxText.

## 125.1. 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( QString 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.

### 125.2. Member Function Documentation

**QListBoxItem::QListBoxItem ( QListBox * listbox = 0 )**

Constructs an empty list box item in the listbox *listbox*

**QListBoxItem::~QListBoxItem ()** `[virtual]`

Destroys the list box item.

**int QListBoxItem::height ( const QListBox * lb ) const** `[virtual]`

Implement this function to return the height of your item. The *lb* parameter is the same as listBox() and is provided for convenience and compatibility.

See also: paint() and width().

Reimplemented in QListBoxText and QListBoxPixmap.

**QListBox * QListBoxItem::listBox () const**

Returns a pointer to the listbox containing this item.

**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 QString & text )** `[virtual 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().

**QString QListBoxItem::text () const** `[virtual]`

Returns the text of the item, which is used for sorting.

See also: setText().

### 125.2.1. URL

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

# 126. QListBoxPixmap Class Reference

`#include <qlistbox.h>`

Inherits QListBoxItem.

### 126.1. Detailed Description

The QListBoxPixmap class provides list box items with a pixmap.

See also: QListBox and QListBoxItem.

### 126.2. Member Function Documentation

**QListBoxPixmap::QListBoxPixmap ( QListBox * listbox, const QPixmap & pixmap )**

Creates a new list box item in listbox *listbox* showing the pixmap *pixmap.*

**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]`

Returns the height of the pixmap.

See also: paint() and width().

Reimplemented from QListBoxItem.

**void QListBoxPixmap::paint ( QPainter * painter )** `[virtual protected]`

Draws the pixmap using *painter.*

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**

Returns the width of the pixmap, plus some margin.

See also: paint() and height().

### 126.2.1. URL

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

# 127. QListBoxText Class Reference

`#include <qlistbox.h>`

Inherits QListBoxItem.

## 127.1. 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.

## 127.2. Member Function Documentation

**QListBoxText::QListBoxText ( QListBox * listbox, const QString & text=QString::null )**

Constructs a list box item in listbox *listtbox* showing the text *text.*

**QListBoxText::QListBoxText ( const QString & text=QString::null )**

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 * painter )** `[virtual protected]`

Draws the text using *painter.*

Reimplemented from QListBoxItem.

**int QListBoxText::width ( const QListBox * lb ) const**

Returns the width of this line.

See also: paint() and height().

### 127.2.1. URL

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

# 128. QListIterator Class Reference

```
#include <qlist.h>
```

Inherits QGListIterator.

## 128.1. Detailed Description

The QListIterator class provides an iterator for QList collections.

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
```

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

Examples:

dirview/main.cpp

## 128.2. 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* ()**

Asterix operator. Returns a pointer to the current iterator item. Same as current().

**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().

### 128.2.1. URL

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

# 129. QListView Class Reference

`#include <qlistview.h>`

Inherits QScrollView.

Inherited by QFileListView.

### 129.1. 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.

To handle events such as mouse-presses on the listview, derived classes can override the QScrollView functions contentsMousePressEvent, contentsMouseReleaseEvent, contentsMouseDoubleClickEvent, contentsMouseMoveEvent, contentsDragEnterEvent, contentsDragMoveEvent, contentsDragLeaveEvent, contentsDropEvent, and contentsWheelEvent.

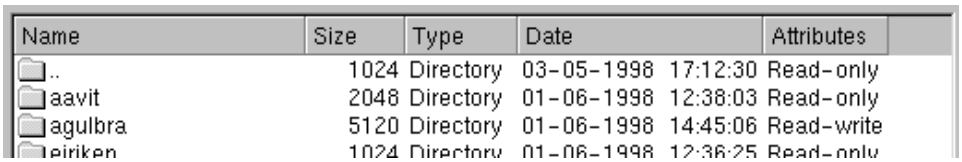
There are also several functions for mapping between items and coordinates. itemAt() returns the item at a position on-screen, itemRect() returns the rectangle an item occupies on the screen and itemPos() returns the position of any item (not on-screen, in the list view). firstChild() returns the item 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.

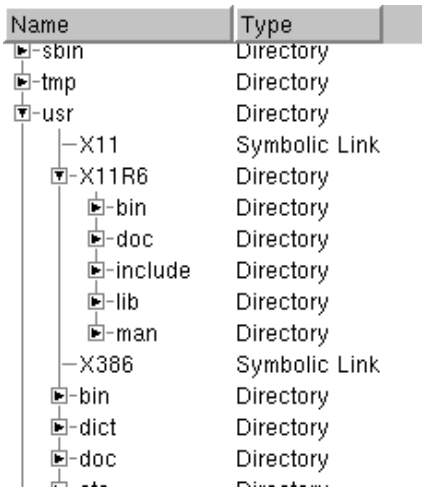
Since QListView offers multiple selection it has to display keyboard focus and selection state separately. Therefore there are functions both to set the selection state of an item, setSelected(), and to select which item displays keyboard focus, setCurrentItem().

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, hierarchical (from the dirview/dirview.cpp example).

## 129.2. Member Type Documentation

### QListView::WidthMode

This enum type describes how the width of a column in the view changes. The currently defined modes are:

- `Manual` - the column width does not change automatically
- `Maximum` - the column is automatically sized according to the widths of all items in the column. (Note: The column never shrinks in this case.)

See also: setColumnWidth(), setColumnWidthMode() and columnWidth().

## 129.3. 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 QIconSet & iconset, const QString & label, int width = -1 )** `[virtual]`

Adds a new column at the right end of the widget, with the header *label* and *iconset,* 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().

Examples:

dirview/main.cpp

**int QListView::addColumn ( const QString & 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().

**void QListView::clearSelection () [virtual]**

Sets all items to be not selected, updates the list view as necessary and emits the selectionChanged() signals. Note that for multi-selection list views, this function needs to iterate over *all* items.

See also: setSelected() and setMultiSelection().

**int QListView::columnAlignment ( int column ) const**

Returns the alignment of logical column *column*. The default is `AlignLeft`.

**QString QListView::columnText ( int c ) const**

Returns the text of column *c*.

**int QListView::columnWidth ( int c ) const**

Returns the width of column *c*.

**QListView::WidthMode QListView::columnWidthMode( int c ) const**

Returns the currently set WidthMode for column *c*.

See also: setColumnWidthMode().

**void QListView::contentsMouseDoubleClickEvent ( QMouseEvent * e ) [virtual protected]**

Processes mouse double-click events on behalf of the viewed widget.

Reimplemented from QScrollView.

**void QListView::contentsMouseMoveEvent ( QMouseEvent * e )** `[virtual protected]`

Processes mouse move events on behalf of the viewed widget.

Reimplemented from QScrollView.

**void QListView::contentsMousePressEvent ( QMouseEvent * e )** `[virtual protected]`

Processes mouse move events on behalf of the viewed widget.

Reimplemented from QScrollView.

**void QListView::contentsMouseReleaseEvent ( QMouseEvent * e )** `[virtual protected]`

Processes mouse move events on behalf of the viewed widget.

Reimplemented from QScrollView.

**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, or 0 if the change was to make no item current. This can

happen e.g. if all items in the list view are deleted.

Note that you may not delete any QListViewItem objects in slots connected to this signal.

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::doAutoScroll ()** `[protected slot]`

This slot handles auto-scrolling when the mouse button is pressed and the mouse is outside the widget.

**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::enabledChange ( bool e )** `[virtual protected]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QWidget.

**void QListView::ensureItemVisible ( const QListViewItem * i )**

Ensures that *i* is made 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]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QWidget.

**void QListView::focusOutEvent ( QFocusEvent * )** `[virtual protected]`

Reimplemented for internal reasons; the API is not affected.

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.

**void QListView::insertItem ( QListViewItem * i )** `[virtual]`

Inserts *i* into the list view as a top-level item. You do not need to call this unless you've called removeItem( *i* ) or QListViewItem::removeItem( i ) and need to reinsert *i* elsewhere.

See also: QListViewItem::removeItem(), (important) and removeItem().

**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 ( const QListViewItem * item ) const**

Identical to *item->isOpen()*. Provided for completeness.

See also: setOpen().

**bool QListView::isSelected ( const 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 & viewPos ) const**

Returns a pointer to the QListViewItem at *viewPos*. Note that *viewPos* 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.

This is a thin wrapper around QListViewItem::itemPos().

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]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QWidget.

**QSize QListView::minimumSizeHint () const** `[virtual]`

Reimplemented for internal reasons; the API is not affected.

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::removeItem ( QListViewItem * i )** `[virtual]`

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

Removes *i* from the list view; *i* must be a top-level item. The warnings regarding

QListViewItem::removeItem( i ) apply to this function too.

See also: QListViewItem::removeItem(), (important) and insertItem().

**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.

**QListViewItem * QListView::selectedItem () const**

Returns a pointer to the selected item, if the list view is in single-selection mode and an item is selected.

If no items are selected or the list view is in multi-selection mode this function returns 0.

See also: setSelected() and setMultiSelection().

**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.

Note that you may not delete any QListViewItem objects in slots connected to this signal.

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, or 0 if the change was to unselect the selected item.

There is another signal which is more useful in multi-selection mode.

Note that you may not delete any QListViewItem objects in slots connected to this signal.

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]`

Configures the logical column *column* to have alignment *align*. The alignment is ultimately passed to QListViewItem::paintCell() for each item in the view.

**void QListView::setColumnText ( int column, const QIconSet & iconset, const QString & label )** `[virtual]`

Sets the heading text of column *column* to *iconset* and *label*. The leftmost colum is number 0.

**void QListView::setColumnText ( int column, const QString & 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*. The default depends on whether the width argument to addColumn was positive or negative.

See also: QListViewItem::width().

**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.

**void QListView::setItemMargin ( int m )** `[virtual]`

Sets the advisory item margin which list items may use to *m.*

The item margin defaults to one pixel 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.

**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 currently selected item is

unselected and *item* 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::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().

Examples:

dirview/main.cpp

**void QListView::show ()** `[virtual]`

Reimplemented to setx the correct background mode and viewed area size.

Examples:

dirview/main.cpp

Reimplemented from QWidget.

**void QListView::showEvent ( QShowEvent * )** `[virtual protected]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QWidget.

**QSize QListView::sizeHint () const** `[virtual]`

Returns a size suitable for this scroll view. This is as wide as QHeader's sizeHint() recommends and tall enough for perhaps 10 items.

Reimplemented from QWidget.

**void QListView::styleChange ( QStyle & old )** `[virtual protected]`

Reimplemented to let the list view items update themselves. *s* is the new GUI style.

Reimplemented from QWidget.

**void QListView::takeItem ( QListViewItem * i )** `[virtual]`

Removes *i* from the list view; *i* must be a top-level item. The warnings regarding QListViewItem::takeItem( i ) apply to this function too.

See also: QListViewItem::takeItem(), (important) and insertItem().

**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 content 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.

### 129.3.1. URL

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

# 130. QListViewItem Class Reference

```
#include <qlistview.h>
```

Inherits Qt.

Inherited by QCheckListItem.

### 130.1. 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();
}
```

Also there is now an interator class to traverse a tree of list view items. To iterate over all items of a list view, do:

```
QListViewItemIterator it( listview );
for ( ; it.current(); ++it )
  do_something_with_the_item( it.current() );
```

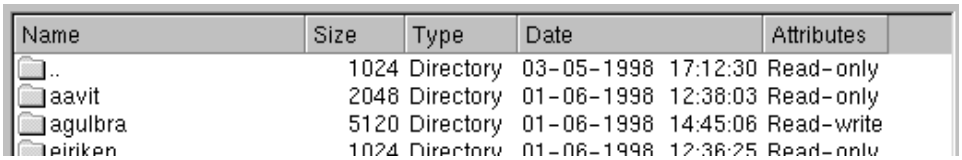
Note that the order of the children will change when the sorting order changes, and is undefined if the items are not visible. You can however call enforceSortOrder() at any time, and QListView will always call it before it needs to show an item.

Many programs will need to reimplement QListViewItem. The most commonly reimplemented functions are:

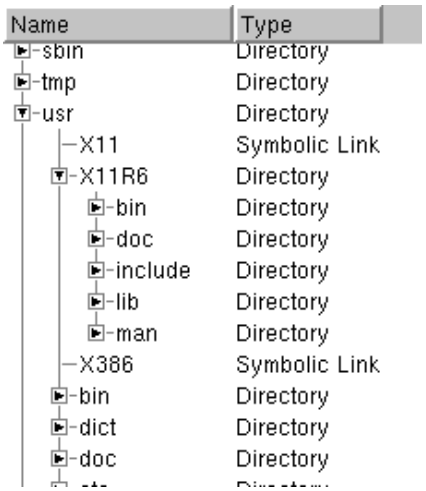
- text() returns the text in a column. Many subclasses will compute that on the fly.
- key() is used for sorting. The default key() simply calls text(), but judicious use of key can be used to sort by e.g. date (as QFileDialog does).
- setup() is called before showing the item, and whenever e.g. the 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() or setOpen( TRUE ) 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.




## 130.2. Member Function Documentation

**QListViewItem::QListViewItem ( QListView * parent )**

Creates a new top-level list view item in the QListView *parent.*

**QListViewItem::QListViewItem ( QListView * parent, QString label1, QString label2 = QString::null, QString label3 = QString::null, QString label4 = QString::null, QString label5 = QString::null, QString label6 = QString::null, QString label7 = QString::null, QString label8 = QString::null )**

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" );
```

See also: setText().

**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, QString label1, QString label2 = QString::null, QString label3 = QString::null, QString label4 = QString::null, QString label5 = QString::null, QString label6 = QString::null, QString label7 = QString::null, QString label8 = QString::null )**

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 ).

See also: setText().

**QListViewItem::QListViewItem ( QListViewItem * parent )**

Creates a new list view item which is a child of *parent* and first in the parent's list of children.

**QListViewItem::QListViewItem ( QListViewItem * parent, QString label1, QString label2 = QString::null, QString label3 = QString::null, QString label4 = QString::null, QString label5 = QString::null, QString label6 = QString::null, QString label7 = QString::null, QString label8 = QString::null )**

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 );
```

See also: setText().

**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, QString label1, QString label2 = QString::null, QString label3 = QString::null, QString label4 = QString::null, QString label5 = QString::null, QString label6 = QString::null, QString label7 = QString::null, QString label8 = QString::null )**

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 ).

See also: setText().

**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. You should not need to call this function; it is called automatically by the constructor of *newChild.*

This function works even if this item is not contained in a list view.

**void QListViewItem::invalidateHeight ()** `[virtual]`

Invalidates the cached total height of this item including all open children.

This function works even if this item is not contained in a list view.

See also: setHeight(), height() and totalHeight().

**bool QListViewItem::isExpandable () const**

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, grandparent'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().

**int QListViewItem::itemPos () const**

Returns the y coordinate of *item* in the list view's coordinate system. This functions is normally much slower than QListView::itemAt(), but it works for all items, while QListView::itemAt() normally works only for items on the screen.

See also: QListView::itemAt(), QListView::itemRect() and QListView::itemPos().

**QString 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.

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; the update rectangle is at 0, 0 and has size *w, h.* The top of the rectangle you own is at *y* (which is never greater than 0 but can be outside the window system's allowed coordinate range).

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 honor 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. The painter *p* has the right font on entry.

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.*

This function works even if this item is not contained in a list view, but reimplementations of it are not required to work properly in that case.

See also: setText() and setPixmap().

**void QListViewItem::removeItem ( QListViewItem * item )** `[virtual]`

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

Removes *tbg* from this object's list of children and causes an update of the screen display. You should normally not need to call this function, as QListViewItem::~QListViewItem() calls it. The normal way to delete an item is `delete`.

**Warning: This function leaves *tbg* and its children in a state where most member functions are unsafe. Only the few functions that are explicitly documented to work in this state may be used then.**

See also: QListViewItem::insertItem().

**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().

**void QListViewItem::setHeight ( int height )** `[virtual protected]`

Sets this item's own height to *height* pixels. This implicitly changes totalHeight() too.

Note that e.g. a font change causes this height to be overwritten unless you reimplement setup().

For best results in Windows style, we suggest using an even number of pixels.

See also: height(), 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 bookkeeping.

See also: height() and totalHeight().

Examples:

dirview/main.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: height() and totalHeight().

**void QListViewItem::setText ( int column, const QString & 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 to the height of a single line of text in the list view's font. (If you use icons, multi-line text etc. you will probably need to call setHeight() yourself or reimplement this.).

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().

**void QListViewItem::takeItem ( QListViewItem * item )** `[virtual]`

Removes *item* from this object's list of children and causes an update of the screen display. You should normally not need to call this function, as QListViewItem::~QListViewItem() calls it. The normal way to delete an item is `delete`.

**Warning: This function leaves *item* and its children in a state where most member functions are unsafe. Only the few functions that are explicitly documented to work in this state may be used then.**

See also: QListViewItem::insertItem().

**QString QListViewItem::text ( int column ) const** `[virtual]`

Returns the text in column *column,* or a null string if there is no text in that column.

This function works even if this item is not contained in a list view, but reimplementations of it are not required to work properly in that case.

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.

setHeight() 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().

### 130.2.1. URL

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

# 131. QListViewItemIterator Class Reference

```
#include <qlistview.h>
```

## 131.1. Detailed Description

The QListViewItemIterator class provides an iterator for collections of QListViewItems

Construct an instance of a QListViewItemIterator with either a QListView* or a QListViewItem* as argument, to operate on the tree of QListViewItems.

A QListViewItemIterator iterates over all items of a listview. This means ++it makes always the first child of the current item the new current one. If there is no child, the next sibling gets the new current item, and if there is no next sibling, the next sibling of the parent is set to current.

Example:

Often you want to get all items, which were selected by a user. Here is an example which does this and stores the pointers to all selected items in a QList.

```
// Somewhere a listview is generated like this
QListView *lv = new QListView(this);
// Enable multiselection
lv->setMultiSelection( TRUE );
// Insert the items here
// ...
// This function is called to get a list of the selected items of a listview
QList<QListViewItem> * getSelectedItems( QListView *lv ) {
  if ( !lv )
    return 0;
  // Create the list
  QList<QListViewItem> *lst = new QList<QListViewItem>;
  lst->setAutoDelete( FALSE );
  // Create an iterator and give the listview as argument
  QListViewItemIterator it( lv );
  // iterate through all items of the listview
  for ( ; it->current(); ++it ) {
```

```
        if ( it->current()->isSelected() )
          lst->append( it->current() );
      }
      return lst;
    }
```

Using a QListViewItemIterator is a convinient way to traverse the tree of QListViewItems of a QListView. It makes especially operating on a hirarchical QListView easy.

Also, multiple QListViewItemIterators can operate on the tree of QListViewItems. A QListView knows about all iterators which are operating on its QListViewItems. So when a QListViewItem gets removed, all iterators that point to this item get updated and point to the new current item after that.

See also: QListView and QListViewItem.

## 131.2. Member Function Documentation

### QListViewItemIterator::QListViewItemIterator ()

Constructs an empty iterator.

### QListViewItemIterator::QListViewItemIterator ( QListView * lv )

Constructs an iterator for the QListView *lv*. The current iterator item is set to point on the first child ( QListViewItem ) of *lv*.

### QListViewItemIterator::QListViewItemIterator ( QListViewItem * item )

Constructs an iterator for the QListView of the *item*. The current iterator item is set to point on the *item*.

### QListViewItemIterator::QListViewItemIterator ( const QListViewItemIterator & it )

Constructs an iterator for the same QListView as *it*. The current iterator item is set to point on the current item of *it*.

**QListViewItemIterator::~QListViewItemIterator ()**

Destroys the iterator.

**QListViewItem * QListViewItemIterator::current () const**

Returns a pointer to the current item of the iterator.

**QListViewItemIterator & QListViewItemIterator::operator++ ()**

Prefix ++ makes the next item in the QListViewItem tree of the QListView of the iterator the current item and returns it. If the current item was the last item in the QListView or null, null is returned.

**const QListViewItemIterator QListViewItemIterator::operator++ ( int )**

Postfix ++ makes the next item in the QListViewItem tree of the QListView of the iterator the current item and returns the item, which was the current one before.

**QListViewItemIterator & QListViewItemIterator::operator+= ( int j )**

Sets the current item to the item *j* positions after the current item in the QListViewItem hirarchie. If this item is beyond the last item, the current item is set to null.

The new current item (or null, if the new current item is null) is returned.

**QListViewItemIterator & QListViewItemIterator::operator-- ()**

Prefix -- makes the previous item in the QListViewItem tree of the QListView of the iterator the current item and returns it. If the current item was the last first in the QListView or null, null is returned.

**const QListViewItemIterator QListViewItemIterator::operator-- ( int )**

Postfix -- makes the previous item in the QListViewItem tree of the QListView of the iterator the current item and returns the item, which was the current one before.

**QListViewItemIterator & QListViewItemIterator::operator-= ( int j )**

Sets the current item to the item *j* positions before the current item in the QListViewItem hirarchie. If this item is above the first item, the current item is set to null. The new current item (or null, if the new current item is null) is returned.

**QListViewItemIterator & QListViewItemIterator::operator= ( const QListViewItemIterator & it )**

Assignment. Makes a copy of *it* and returns a reference to its iterator.

### 131.2.1. URL

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

# 132. QLNode Class Reference

```
#include <qglist.h>
```

### 132.1. 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().

### 132.2. Member Function Documentation

#### QCollection::Item QLNode::getData()

Returns a pointer (`void*`) to the actual data in the list node.

#### 132.2.1. URL

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

# 133. QMainWindow Class Reference

`#include <qmainwindow.h>`

Inherits QWidget.

## 133.1. 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 example (an editor) sets a QMultiLineEdit to be the central widget.

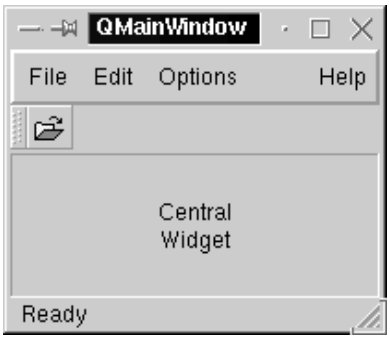
QMainWindow automatically detects the creation of a menu bar or status bar if you specify the QMainWindow as parent, or you can use the provided menuBar() and statusBar() functions. menuBar() and statusBar() create a suitable widget if one doesn't exist, and updates the window's layout to make space.

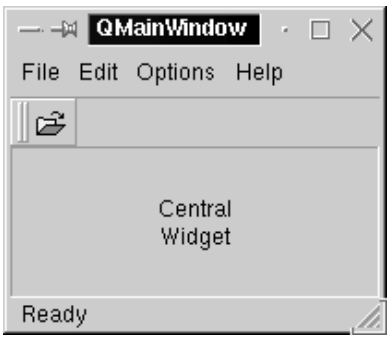
QMainWindow also provides a QToolTipGroup connected to the status bar. toolTipGroup() provides access to the QToolTipGroup, but there is no way to set the tool tip group.

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:

mainlyQt/editor.cpp

## 133.2. Member Type Documentation

### QMainWindow::ToolBarDock

Each toolbar can be in one of the following positions:

- `Top` - above the central widget, below the menubar.
- `Bottom` - below the central widget, above the status bar.
- `Left` - to the left of the central widget.
- `Right` - to the left of the central widget.

Other values are also defined for future expansion.

### 133.3. Member Function Documentation

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

Constructs an empty main window.

**QMainWindow::~QMainWindow ()**

Destroys the object and frees any allocated resources.

**void QMainWindow::addToolBar ( QToolBar * toolBar, ToolBarDock edge = Top, bool newLine = FALSE )**

Adds *toolBar* to this the end of *edge* and makes it start a new line of tool bars if *nl* is TRUE.

If *toolBar* is already managed by some main window, it is first removed from that window.

**void QMainWindow::addToolBar ( QToolBar * toolBar, const QString & label, ToolBarDock edge = Top, bool newLine = FALSE )**

Adds *toolBar* to this the end of *edge,* labelling it *label* and makes it start a new line of tool bars if *newLine* is TRUE.

If *toolBar* is already managed by some main window, it is first removed from that 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().

Examples:

qfd/qfd.cpp

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

Monitors events to ensure layout is updated.

Reimplemented from QObject.

**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]`

Reimplemented for internal reasons; the API is not affected.

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().

**void QMainWindow::moveToolBar ( QToolBar * toolBar, ToolBarDock edge = Top )**

Moves *toolBar* to this the end of *edge.*

If *toolBar* is already managed by some main window, it is moved from that window to this and set to

*not* start a new line.

**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.

**void QMainWindow::resizeEvent ( QResizeEvent * )** `[virtual protected]`

Monitors events to ensure layout is updated.

Reimplemented from QWidget.

**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 central widget is the one around which the toolbars etc. are arranged.

**void QMainWindow::setDockEnabled ( ToolBarDock dock, bool enable )** `[virtual]`

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 )** `[virtual 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 ()** `[virtual 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 )** `[virtual 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:

themes/main.cpp
mainlyQt/editor.cpp
application/main.cpp
i18n/main.cpp
scribble/main.cpp
qfd/qfd.cpp
addressbook/main.cpp
qfileiconview/main.cpp

Reimplemented from QWidget.

**QStatusBar * QMainWindow::statusBar () const**

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

See also: menuBar() and toolTipGroup().

Examples:

qfd/qfd.cpp

**void QMainWindow::styleChange ( QStyle & old )** `[virtual protected]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QWidget.

**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();.

**void QMainWindow::whatsThis ()** `[slot]`

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

This is the same as QWhatsThis::enterWhatsThisMode(), but as a slot of of a main window object.

This way it can be easily used for popup menus as in the code fragment:

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

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

See also: QWhatsThis::enterWhatsThisMode().

### 133.3.1. URL

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

# 134. QMap Class Reference

```
#include <qmap.h>
```

### 134.1. Detailed Description

The QMap class is a value based template class that provides a dictionary

Define a template instance QMap<Key,Data> to create a dictionary with keys of type X and values of type Y. Please notice that QMap does not store pointers to the members of the map. It holds a copy of every member. That is the reason why this kind of classes are called "value based" while QList and QDict are "reference based".

Some classes can not be used within a QMap. For example everything derived from QObject and as a reason of this that extends to all classes which implement widgets. You can only put stuff in a QMap which can be easily copied. That means that the class X and Y must implement a copy constructor.

Example:

```
#include <qmap.h>
#include <qstring.h>
#include <stdio.h>
class Employee
{
public:
    Employee( const QString& name, int salary ) { n=name; s=salary; }
    Employee( const Employee& _e ) { n = _e.name; s = _e.s; }
    QString     name()   const               { return n; }
    int         salary() const               { return s; }
    void        setSalary( int _s ) const    { s = _s; }
private:
    QString     n;
    int         s;
};
void main()
```

```
{
    typedef QMap<QString,Employee> EmployeeMap;
    EmployeeMap map;                // map of Employee
    map.insert( "Gates", Employee("Bill", 50000) );
    map.insert( "Ballmer", ( Employee("Steve",80000) );
    map.insert( "Sommer," ( Employee("Ron",  60000) );
    Employee joe( "Joe", 50000 );
    map.insert( "Doe", joe );
    joe.setSalary( 4000 );
    EmployeeMap::Iterator it;
    for( it = map.begin(); it != map.end(); ++it )
        printf( "%s, %s earns %d\n", it.key(), it.data().name(), it.data().salary() );
}
```

Program output:

```
Ballmer, Steve earns 80000
Doe, Joe earns 50000
Gates, Bill earns 50000
Sommer, Ron earns 60000
```

As you can see, the latest changes to Joe's salary did not affect the value in the list because the map created a copy of Joe's entry. In addition you should notice that the items are alphabetically sorted when iterating over the map.

There are two ways to find values in the list. The first one is to use the find() function. It returns an iterator pointing to the desired item or the end() iterator it no such element exists.

The second approach uses the operator[]. But be warned: If you don't know that the element you are searching for is really in the list, then you should not use operator[]. The following example illustrates that.

```
QMap<QString,QString> map;
map.insert( "Weis", "Torben" );
str << map["Weis"] << map["Ettrich"] << endl;
const QMap<QString,QString>& map2 = map;
```

```
    str << map["Weis"] << map["Reggie"] << endl;
```

The code fragment will print out "Weis", "" and the second part will print "Weis", "". In addition the first fragment inserted an empty entry with key "Ettrich". The second one did not insert an empty entry with key "Reggie" because the const operator[] was used which can not do insertion. So if you are not sure whether a certain element is in the map you should use find() and iterators.

If you just want to know whether a certain key is contained in the map, the the contains() function is what you are looking for. In addition count() tells you how many keys there are currently in the map.

Another method for traversing a map is to use the functions begin() and end(). With a simple for loop as shown in the example you can iterate over the complete map. It is save to have multiple iterators at the same time. If some member of the map is removed then only iterators pointing to the removed member become invalid. Inserting in the map does not invalidate any iterator.

Since QMap is value based there is no need to care about deleting elements in the list. The list holds its own copies and will free them if the corresponding member or the list itself is deleted. You can force the list to free all of its item with clear().

QMap is implicitly shared. This means you can just make copies of the map in time O(1). If multiple QMap instances share the same data and one is modifying the map's data then this modifying instance makes a copy and modifies its private copy - thus it does not affect other instances. From a developer's point of view you can think that a QMap and a copy of this map have nothing to do with each other.

There are two ways of inserting new elements in a list. One uses the insert() method while the other one uses operator[] like this:

```
    QMap<QString,QString> map;
    map["Weis"] = "Torben";
;
```

Items can be removed from the map in two ways. The first is to pass an iterator to the remove(). The other possibility is to pass a key value to remove() which will delete the entry with the requested

key. In addition you can clear the entire map using the clear() method.

See also: QMapIterator.

## 134.2. Member Function Documentation

**QMap::QMap ()**

Constructs an empty map.

**QMap::QMap ( const QMap<Key,T> & m )**

Constructs a copy of *m*.

This operation costs O(1) time since QMap is implicit shared. The first instance applying

modifications to a shared list will create a copy which takes in turn O(n) time. However returning a

QMap from a function is very fast.

**QMap::~QMap ()**

Destroys the map. References to the values in the map and all iterators of this map become invalidated. Since QMap is highly tuned for performance you won't see warnings if you use invalid iterators, because it is impossible for an iterator to check whether it is valid or not.

**ConstIterator QMap::begin () const**

Returns an iterator pointing to the first element in the map. This iterator equals end() if the map is empty;

See also: end(), \sa and QMapConstIterator.

**Iterator QMap::begin ()**

Returns an iterator pointing to the first element in the map. This iterator equals end() if the map is empty;

See also: end(), \sa and QMapIterator.

**void QMap::clear ()**

Removes all items from the map.

See also: remove().

**bool QMap::contains ( const Key & k ) const**

Returns TRUE if the key \k is contained in the map.

**uint QMap::count () const**

Returns the number of items in the ap.

See also: isEmpty().

**void QMap::detach ()** `[protected]`

If the map does not share its data with another QMap instance, then nothing happens, otherwise the function creates a new copy of this data and detaches from the shared one. This function is called whenever the map is modified. The implicit sharing mechanism is implemented this way.

**ConstIterator QMap::end () const**

Returns an iterator pointing behind the last element in the map. This iterator equals begin() if the map is empty.

See also: begin(), \sa and QMapConstIterator.

**Iterator QMap::end ()**

Returns an iterator pointing behind the last element in the map. This iterator equals begin() if the map is empty.

See also: begin(), \sa and QMapIterator.

**ConstIterator QMap::find ( const Key & k ) const**

Finds the key *k* in the map.

Returns end() if no key did match.

See also: QMapConstIterator.

**Iterator QMap::find ( const Key & k )**

Finds the key *k* in the map.

Returns end() if no key did match.

See also: QMapIterator.

### Iterator QMap::insert ( const Key & key, const T & value )

Inserts the *value* with key *k*.

Returns an iterator pointing at the inserted value.

See also: QMapIterator.

### bool QMap::isEmpty () const

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

See also: count().

### T& QMap::operator[] ( const Key & k )

Returns the value associated with the key *k*. If no such key is present then en empty item is inserted with this key and a reference to the item is returned.

You can use this operator in two directions: For reading and for writing:

```
QMap<QString,QString> map;

map[ "Weis" ] = "Torben";

stream << map[ "Weis" ];
```

### const T& QMap::operator[] ( const Key & k ) const

Returns the value associated with the key *k*. If no such key is present then a reference to an empty item is returned.

### void QMap::remove ( Iterator it )

Removes the item at position *it* in the map.

See also: clear(), \sa and QMapIterator.

### void QMap::remove ( const Key & k )

Removes the item with the key *k*.

See also: clear().

**Iterator QMap::replace ( const Key & k, const T & v )**

Replaces the value with key *k* from the map if possible and inserts the new value *v* with key \k in the map.

See also: insert(), \sa, remove(), \sa and QMapIterator.

### 134.3. Related Functions

**QDataStream& operator>> (QDataStream & s, QMap<Key,T> & m)**

Reads a map from the stream. The types *Key* and *T* must implement the streaming operator, too.

**QDataStream& operator<< (QDataStream & s, const QMap<Key,T> & m)**

Writes a map to the stream. The types *Key* and *T* must implement the streaming operator, too.

#### 134.3.1. URL

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

# 135. QMapConstIterator Class Reference

```
#include <qmap.h>
```

### 135.1. Detailed Description

The QMapConstIterator class provides an iterator for QMap.

In contrast to QMapIterator this class is used to iterate over a const map. It does not allow to modify the values of the map since this would break the const semantics.

For more informations on QMap iterators see QMapIterator.

See also: QMap, \sa and QMapIterator.

### 135.2. Member Function Documentation

**QMapConstIterator::QMapConstIterator ()**

Creates an uninitialized iterator.

**QMapConstIterator::QMapConstIterator ( const QMapIterator<K,T> & it )**

Constructs a copy of the iterator.

**QMapConstIterator::QMapConstIterator ( const QMapConstIterator<K,T> & it )**

Constructs a copy of the iterator.

**const T& QMapConstIterator::data () const**

Returns a const reference to the data of the current item.

**const K& QMapConstIterator::key () const**

Returns a const reference to the current key.

**bool QMapConstIterator::operator!= ( const QMapConstIterator<K,T> & it ) const**

Compares both iterators and returns TRUE if they point to different items.

**const T& QMapConstIterator::operator* () const**

Asterix operator. Returns a const reference to the current item. The same as data().

**bool QMapConstIterator::operator== ( const QMapConstIterator<K,T> & it ) const**

Compares both iterators and returns TRUE if they point to the same item.

### 135.2.1. URL

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

# 136. QMapIterator Class Reference

```
#include <qmap.h>
```

### 136.1. Detailed Description

The QMapIterator class provides an iterator for QMap.

You can not create an iterator by yourself. Instead you have to ask a map to give you one. An iterator has only the size of a pointer. On 32 bit machines that means 4 bytes otherwise 8 bytes. That makes them very fast. In fact they resemble the semantics of pointers as good as possible and they are almost as fast as usual pointers.

```
#include <qmap.h>
#include <qstring.h>
#include <stdio.h>
class Employee
{
public:
    Employee( const QString& name, int salary ) { n=name; s=salary; }
    Employee( const Employee& _e ) { n = _e.name; s = _e.s; }
    QString     name()   const               { return n; }
    int         salary() const               { return s; }
    void        setSalary( int _s ) const    { s = _s; }
private:
    QString     n;
    int         s;
};
void main()
{
    typedef QMap<QString,Employee> EmployeeMap;
    EmployeeMap map;                // map of Employee
    map.insert( "Gates", Employee("Bill", 50000) );
    map.insert( "Ballmer", ( Employee("Steve",80000) );
    map.insert( "Sommer," ( Employee("Ron",  60000) );
```

```
        Employee joe( "Joe", 50000 );
        map.insert( "Doe", joe );
        joe.setSalary( 4000 );
        EmployeeMap::Iterator it;
        for( it = map.begin(); it != map.end(); ++it )
            printf( "%s, %s earns %d\n", it.key(), it.data().name(), it.data().salary() );
    }
```

Program output:

```
Ballmer, Steve earns 80000
Doe, Joe earns 50000
Gates, Bill earns 50000
Sommer, Ron earns 60000
```

The only way to traverse a map is to use iterators. QMap is highly optimized for performance and memory usage. On the other hand that means that you have to be a bit more careful by what you are doing. QMap does not know about all its iterators and the iterators don't even know to which map they belong. That makes things fast and slim but a bit dangerous because it is up to you to make sure that iterators you are using are still valid. QDictIterator will be able to give warnings while QMapIterator may end up in an undefined state.

For every Iterator there is a ConstIterator. When accessing a QMap in a const environment or if the reference or pointer to the map is itself const, then you have to use the ConstIterator. Its semantics are the same, but it returns only const references to the item it points to.

See also: QMap.

### 136.2. Member Function Documentation

**QMapIterator::QMapIterator ()**

Creates an uninitialized iterator.

**QMapIterator::QMapIterator ( const QMapIterator<K,T> & it )**

Constructs a copy of the iterator.

**T& QMapIterator::data ()**

Returns a reference to the current item.

**const T& QMapIterator::data () const**

Returns a const reference to the data of the current item.

**const K& QMapIterator::key () const**

Returns a const reference to the data of the current key.

**bool QMapIterator::operator!= ( const QMapIterator<K,T> & it ) const**

Compares both iterators and returns TRUE if they point to different items.

**T& QMapIterator::operator* ()**

Asterix operator. Returns a reference to the current item. The same as data().

**const T& QMapIterator::operator* () const**

Asterix operator. Returns a const reference to the current item. The same as data().

**bool QMapIterator::operator== ( const QMapIterator<K,T> & it ) const**

Compares both iterators and returns TRUE if they point to the same item.

### 136.2.1. URL

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

# 137. QMenuBar Class Reference

```
#include <qmenubar.h>
```

Inherits QFrame and QMenuData.

### 137.1. 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
layout/layout.cpp
menu/menu.cpp
progress/progress.cpp
scrollview/scrollview.cpp

### 137.2. Member Type Documentation

#### QMenuBar::Separator

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

- `Never` - in many applications, there already is a separator, and two looks stupid.

- `InWindowsStyle` - in some other applications, a separator looks good in Windows style but not else.

### 137.3. 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().

**bool QMenuBar::customWhatsThis () const** `[virtual]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QWidget.

**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::focusOutEvent ( QFocusEvent * )** `[virtual protected]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QWidget.

**void QMenuBar::fontChange ( const QFont & f )** `[virtual protected]`

For internal use only.

Reimplemented from QWidget.

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

Returns the height that the menu would resize itself to if its parent (and hence itself) resized to the given 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.

Reimplemented from QWidget.

**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.

See also: Separator and setSeparator().

**void QMenuBar::setSeparator ( Separator when )** `[virtual]`

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.

See also: Separator and separator().

Examples:

layout/layout.cpp
scrollview/scrollview.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::styleChange ( QStyle & old )** `[virtual protected]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QWidget.

**void QMenuBar::updateItem ( int )** `[virtual]`

For internal use only.

Reimplemented from QMenuData.

### 137.3.1. URL

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

# 138. QMenuData Class Reference

`#include <qmenudata.h>`

Inherited by QMenuBar and QPopupMenu.

## 138.1. 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.

## 138.2. 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 qnamespace.h.

### void QMenuData::changeItem ( const QIconSet & icon, const QString & text, int id )

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

Changes the icon and text of the menu item *id*.

See also: pixmap().

**void QMenuData::changeItem ( const QPixmap & pixmap, int id )**

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

Changes the pixmap of the menu item *id*. If the item has an icon, the icon remains unchanged.

See also: pixmap().

**void QMenuData::changeItem ( const QString & text, int id )**

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

Changes the text of the menu item *id*. If the item has an icon, the icon remains unchanged.

See also: text().

**void QMenuData::changeItem ( int id, const QIconSet & icon, const QPixmap & pixmap )**

Changes the icon and pixmap of the menu item *id*.

See also: pixmap().

**void QMenuData::changeItem ( int id, const QIconSet & icon, const QString & text )**

Changes the icon and text of the menu item *id*.

See also: pixmap().

**void QMenuData::changeItem ( int id, const QPixmap & pixmap )**

Changes the pixmap of the menu item *id*. If the item has an icon, the icon remains unchanged.

See also: pixmap().

**void QMenuData::changeItem ( int id, const QString & text )**

Changes the text of the menu item *id*. If the item has an icon, the icon remains unchanged.

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.

**QIconSet* QMenuData::iconSet ( int id ) const**

Returns the icon set that has been set for menu item *id,* or 0 if no icon set has been set.

See also: changeItem(), text() and pixmap().

**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().

Examples:

scrollview/scrollview.cpp

**int QMenuData::insertItem ( const QIconSet & icon, const QPixmap & pixmap, QPopupMenu * popup, int id=-1, int index=-1 )**

Inserts a menu item with an icon, a pixmap and a sub menu. The icon will be displayed to the left of the pixmap in the item. Returns the menu item identifier.

The *popup* must be deleted by the programmer or by its parent widget. It is not deleted when this menu item is removed or when the menu is deleted.

The menu item is assigned the identifier *id* or an automatically generated identifier if *id* is < 0. The generated identifiers (negative integers) are guaranteed to be unique within the entire application.

The *index* specifies the position in the menu. The 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 QIconSet & icon, const QPixmap & pixmap, const QObject * receiver, const char * member, int accel = 0, int id = -1, int index = -1 )**

Inserts a menu item with an icon, a pixmap, an accelerator key, an id and an optional index and connects it to an object/slot. The icon will be displayed to the left of the pixmap in the item.

The menu item is assigned the identifier *id* or an automatically generated identifier if *id* is < 0. The generated identifiers (negative integers) are guaranteed to be unique within the entire application.

The *index* specifies the position in the menu. The 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 qnamespace.h.

**int QMenuData::insertItem ( const QIconSet & icon, const QPixmap & pixmap, int id=-1, int index=-1 )**

Inserts a menu item with an icon and a pixmap. The icon will be displayed to the left of the pixmap in the item. Returns the menu item identifier.

The menu item is assigned the identifier *id* or an automatically generated identifier if *id* is < 0. The generated identifiers (negative integers) are guaranteed to be unique within the entire application.

The *index* specifies the position in the menu. The 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 QIconSet & icon, const QString & text, QPopupMenu * popup, int id=-1, int index=-1 )**

Inserts a menu item with an icon, a text and a sub menu. The icon will be displayed to the left of the text in the item. Returns the menu item identifier.

The *popup* must be deleted by the programmer or by its parent widget. It is not deleted when this menu item is removed or when the menu is deleted.

The menu item is assigned the identifier *id* or an automatically generated identifier if *id* is < 0. The generated identifiers (negative integers) are guaranteed to be unique within the entire application.

The *index* specifies the position in the menu. The 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 QIconSet & icon, const QString & text, const QObject * receiver, const char * member, int accel = 0, int id = -1, int index = -1 )**

Inserts a menu item with an icon, a text, an accelerator key, an id and an optional index and connects it to an object/slot. The icon will be displayed to the left of the text in the item.

The menu item is assigned the identifier *id* or an automatically generated identifier if *id* is < 0. The generated identifiers (negative integers) are guaranteed to be unique within the entire application.

The *index* specifies the position in the menu. The 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 qnamespace.h.

**int QMenuData::insertItem ( const QIconSet & icon, const QString & text, int id=-1, int index=-1 )**

Inserts a menu item with an icon and a text. The icon will be displayed to the left of the text in the item. Returns the menu item identifier.

The menu item is assigned the identifier *id* or an automatically generated identifier if *id* is < 0. The generated identifiers (negative integers) are guaranteed to be unique within the entire application.

The *index* specifies the position in the menu. The 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, QPopupMenu * popup, int id=-1, int index=-1 )**

Inserts a menu item with a pixmap and a sub menu. The icon will be displayed to the left of the pixmap in the item. Returns the menu item identifier.

The *popup* must be deleted by the programmer or by its parent widget. It is not deleted when this menu item is removed or when the menu is deleted.

The menu item is assigned the identifier *id* or an automatically generated identifier if *id* is < 0. The generated identifiers (negative integers) are guaranteed to be unique within the entire application.

The *index* specifies the position in the menu. The 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 = 0, int id = -1, 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 if *id* is < 0. The generated identifiers (negative integers) are guaranteed to be unique within the entire application.

The *index* specifies the position in the menu. The 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 qnamespace.h.

**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 if *id* is < 0. The generated identifiers (negative integers) are guaranteed to be unique within the entire application.

The *index* specifies the position in the menu. The 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 QString & 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 or by its parent widget. It is not deleted when this menu item is removed or when the menu is deleted.

The menu item is assigned the identifier *id* or an automatically generated identifier if *id* is < 0. The generated identifiers (negative integers) are guaranteed to be unique within the entire application.

The *index* specifies the position in the menu. The 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().

Examples:

qtimage/qtimage.cpp
grapher/grapher.cpp
mainlyQt/editor.cpp
layout/layout.cpp
menu/menu.cpp
progress/progress.cpp
scrollview/scrollview.cpp

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

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 );
```

If you will need to translate accelerators, use QAccel::stringToKey() to calculate the accelerator key:

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

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

The menu item is assigned the identifier *id* or an automatically generated identifier if *id* is < 0. The generated identifiers (negative integers) are guaranteed to be unique within the entire application.

The *index* specifies the position in the menu. The 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 qnamespace.h.

**int QMenuData::insertItem ( const QString & 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 if *id* is < 0. The generated identifiers (negative integers) are guaranteed to be unique within the entire application.

The *index* specifies the position in the menu. The 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
menu/menu.cpp
progress/progress.cpp
scrollview/scrollview.cpp

**bool QMenuData::isItemChecked ( int id ) const**

Returns TRUE if the menu item has been checked, otherwise FALSE.

See also: setItemChecked().

Examples:

progress/progress.cpp

**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(), text() and iconSet().

**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, ALT,` or `UNICODE_ACCEL` (OR'ed or added). The header file qnamespace.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 );     // Control and O to open
fileMenu->insertItem( "Quit", 69 );         // add "Quit" item
```

```
    fileMenu->setAccel( CTRL + ALT + Key_Delete, 69 );

    mainMenu->insertItem( "File", fileMenu );   // add the file menu
```

If you will need to translate accelerators, use QAccel::stringToKey():

```
    fileMenu->setAccel( QAccel::stringToKey(tr("Ctrl+O")), 67 );
```

You can also specify the accelerator in the insertItem() function.

See also: setAccel(), accel(), insertItem(), QAccel and qnamespace.h.

**void QMenuData::setId ( int index, int id )** `[virtual]`

Sets the menu identifier of the item at *index* to *id.*

If index is out of range the operation is ignored.

See also: idAt().

**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
scrollview/scrollview.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

**void QMenuData::setWhatsThis ( int id, const QString & text )**

Sets a Whats This help for a certain menu item.

Arguments:

- *id* is the menu item id.
- *text* is the Whats This help text in QML format

See also: whatsThis().

**QString QMenuData::text ( int id ) const**

Returns the text that has been set for menu item *id,* or a null string if no text has been set.

See also: changeItem(), pixmap() and iconSet().

**void QMenuData::updateItem ( int )** `[virtual]`

Virtual function; notifies subclasses about an item that has been changed.

Reimplemented in QMenuBar.

**QString QMenuData::whatsThis ( int id ) const**

Returns the Whats This help text for the specified item *id* or QString::null if no text has been defined yet.

See also: setWhatsThis().

### 138.2.1. URL

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

# 139. QMessageBox Class Reference

`#include <qmessagebox.h>`

Inherits QDialog.

## 139.1. 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.

The text part of all message box messages can be either rich text or plain text. If you specify a rich text formatted string, it will be rendered using the default stylesheet. See QStyleSheet::defaultSheet() for details.

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.

## 139.2. Member Type Documentation

### QMessageBox::Icon

This type includes the following values:

- `QMessageBox::NoIcon`
- `QMessageBox::Information`
- `QMessageBox::Warning`
- `QMessageBox::Critical`

## 139.3. 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 QString & caption, const QString & text, Icon icon, int button0, int button1, int button2, QWidget * parent=0, const char * name=0, bool modal=TRUE, WFlags f=WStyle_DialogBorder )**

Constructs a message box with a *caption,* a *text,* an *icon* and up to three buttons.

The *icon* must be one of:

- `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 QString & caption, const QString & 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 QString & caption=QString::null )** `[static]`

Displays a simple message box about Qt, with window caption *caption* and optionally centered over *parent.* The message includes the version number of Qt being used by the application.

This is neat for inclusion into the Help menu. See the menu.cpp example.

Examples:

menu/menu.cpp
trivial/trivial.cpp

**void QMessageBox::adjustSize ()** `[virtual]`

Adjusts the size of the message box to fit the contents just before QDialog::exec() or QDialog::show() is called.

This function will not be called if the message box has been explicitly resized before showing it.

Reimplemented from QWidget.

**QString 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.

See also: setButtonText().

**int QMessageBox::critical ( QWidget * parent, const QString & caption, const QString & text, const QString & button0Text = QString::null, const QString & button1Text = QString::null, const QString & button2Text = QString::null, int defaultButtonNumber = 0, int escapeButtonNumber = -1 )** `[static]`

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 QString & caption, const QString & 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.

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 QString & caption, const QString & text, const QString & button0Text = QString::null, const QString & button1Text = QString::null, const QString & button2Text = QString::null, int defaultButtonNumber = 0, int escapeButtonNumber = -1 )** `[static]`

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().

Examples:

picture/picture.cpp

**int QMessageBox::information ( QWidget * parent, const QString & caption, const QString & text, int button0, int button1=0, int button2=0 )** `[static]`

Opens an information message box with 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().

**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 QString & text )**

Sets the text of the message box button *button* to *text.* Setting the text of a button that is not in the message box is quietly 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::setIcon ( const QPixmap & pix )** `[virtual]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QWidget.

**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::setText ( const QString & 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.

**void QMessageBox::styleChanged ( QStyle & )** `[protected]`

Reimplemented for internal reasons; the API is not affected.

**QString QMessageBox::text () const**

Returns the message box text currently set, or a null string if no text has been set.

See also: setText().

**int QMessageBox::warning ( QWidget * parent, const QString & caption, const QString & text, const QString & button0Text = QString::null, const QString & button1Text = QString::null, const QString & button2Text = QString::null, int defaultButtonNumber = 0, int escapeButtonNumber = -1 )** `[static]`

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().

Examples:

i18n/main.cpp

**int QMessageBox::warning ( QWidget * parent, const QString & caption, const QString & text, int button0, int button1, int button2=0 )** `[static]`

Opens a warning message box with 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().

### 139.3.1. URL

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

# 140. QMimeSource Class Reference

`#include <qmime.h>`

Inherited by QDragObject and QDropEvent.

### 140.1. Detailed Description

An abstract piece of formatted data.

Drag-and-drop and clipboard use this abstraction.

See also: IANA list of MIME media types

Examples:

qiconview/main.cpp

### 140.2. Member Function Documentation

**QMimeSource::~QMimeSource** () `[virtual]`

Provided to ensure subclasses destruct correctly.

**QByteArray QMimeSource::encodedData ( const char * ) const** `[virtual]`

Returns the encoded payload of this object, in the specified MIME format.

Subclasses must override this function.

Reimplemented in QDropEvent.

**const char * QMimeSource::format ( int i = 0 ) const** `[virtual]`

Returns the *ith* format, or NULL.

Examples:

qiconview/main.cpp

Reimplemented in QDropEvent.

**bool QMimeSource::provides ( const char * mimeType ) const** `[virtual]`

Returns TRUE if the object can provide the data in format *mimeType*. The default implementation iterates over format().

Note that it is often better to use the more-abstract canDecode() functions such as

QTextDrag::canDecode() and QImageDrag::canDecode().

Reimplemented in QDropEvent.

### 140.2.1. URL

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

# 141. QMimeSourceFactory Class Reference

```
#include <qmime.h>
```

## 141.1. Detailed Description

An extensible supply of MIME-typed data.

A QMimeSourceFactory provides an abstract interface to a collection of information. Each piece of information is represented by a QMimeSource object, which can be examined and converted to concrete data types by functions like QImageDrag::canDecode() and QImageDrag::decode().

The base QMimeSourceFactory can be used in two ways: as an abstraction of a collection of files, or as specifically stored data. For it to access files, call setFilePath() before accessing data. For stored data, call setData() for each item (there are also convenience functions setText(), setImage(), and setPixmap() that simply call setData() with massaged parameters).

The rich text widgets QTextView and QTextBrowser use QMimeSourceFactory to resolve references such as images or links within rich text documents. They either access the default factory ( see defaultFactory() ) or their own ( see QTextView::setMimeSourceFactory() ). Other classes that are capable of displaying rich text like QLabel, QWhatsThis or QMessageBox always use the default factory.

As mentioned earlier, a factory can also be used as container to store data associated with a name. This technique is useful whenever rich text contains images that are stored in the program itself, not loaded from the harddisk. Your program may for example define some image data as

```
static const char* myimage_xpm[]={
"...",
...
"..."};
```

To be able to use this image within some rich text, for example inside a QLabel, you have to create a QImage from the raw data and insert it into the factory with a unique name:

```
QMimeSourceFactory::defaultFactory()->setImage( "myimage", QImage(myimage_data) );
```

Now you can create a rich text QLabel with

```
QLabel* label = new QLabel(
    "Rich text with embedded image:<img source=\"myimage\">"
    "Isn't that <em>cute</em>?" );
```

## 141.2. Member Function Documentation

### QMimeSourceFactory::QMimeSourceFactory ()

Constructs a QMimeSourceFactory which has no filepath and no stored content.

### QMimeSourceFactory::~QMimeSourceFactory () [virtual]

Destructs the QMimeSourceFactory, deleting all stored content.

### const QMimeSource* QMimeSourceFactory::data ( const QString & abs_name ) const [virtual]

Returns a reference to the data associated with *abs_name*. The return value only remains valid until a subsequent call to this function for the same object, and only if setData() is not called to modify the data, so you should immediately decode the result.

If there is no data associated with *abs_name* in the factory's store, the factory tries to access the local filesystem. If *abs_name* isn't an absolute filename, the factory will search for it on all defined paths ( see setFilePath() ).

The factory understands all image formats supported by QImageIO. Any other mime types are determined by the filename extension. The default settings are

```
setExtensionType("html", "text/html;charset=iso8859-1");
setExtensionType("htm", "text/html;charset=iso8859-1");
setExtensionType("txt", "text/plain");
setExtensionType("xml", "text/xml;charset=utf8");
```

The effect of these is that filenames ending in "html" or "htm" will be treated as text encoded in the iso8859-1 encoding, those ending in "txt" will be treated as text encoded in the local encoding; those ending in "xml" will be treated as text encoded in UTF8 encoding. The text subtype ("html",

"plain", or "xml") does not affect the factory, but users of the factory may behave differently. We recommend creating "xml" files where practical as such files can be viewed regardless of the run-time encoding, and can encode any Unicode characters without resorting to encoding definitions inside the file.

Any file data that is not recognized will be retreived as a QMimeSource providing the "application/octet-stream" MIME type, which is just uninterpreted binary data. You can add further extensions or change existing ones with subsequent calls to setExtensionType(). If the extension mechanism is not sufficient for you problem domain, you may inherit QMimeSourceFactory and reimplement this function to perform some more clever mime type detection. The same applies if you want to use the mime source factory for accessing URL referenced data over a network.

**const QMimeSource* QMimeSourceFactory::data ( const QString & abs_or_rel_name, const QString & context ) const**

A convenience function. See data(const QString& abs_name).

**QMimeSourceFactory* QMimeSourceFactory::defaultFactory ()** `[static]`

Returns the application-wide default mime source factory. This factory is used by rich text rendering classes such as QSimpleRichText, QWhatsThis and also QMessageBox to resolve named references within rich text documents. It serves also as initial factory for the more complex render widgets QTextView and QTextBrowser.

See also: setDefaultFactory().

**QString QMimeSourceFactory::makeAbsolute ( const QString & abs_or_rel_name, const QString & context ) const** `[virtual]`

Converts the absolute or relative data item name *abs_or_rel_name* to an absolute name, interpretted within the context of the data item named *context* (this must be an absolute name).

**void QMimeSourceFactory::setData ( const QString & abs_name, QMimeSource * data )** `[virtual]`

Sets *data* to be the data item associated with the absolute name *abs_name*. Note that the ownership of *data* is transferred to the factory - do not delete or access the pointer after passing it to this function.

**void QMimeSourceFactory::setDefaultFactory ( QMimeSourceFactory * factory )** `[static]`

Sets the default *factory,* destroying any previously set mime source provider. The ownership of the factory is transferred.

See also: defaultFactory().

**void QMimeSourceFactory::setExtensionType ( const QString & ext, const char * mimetype )** `[virtual]`

Sets the MIME-type to be associated with a filename extension. This determines the MIME-type for files found via a path set by setFilePath().

**void QMimeSourceFactory::setFilePath ( const QStringList & path )** `[virtual]`

Sets a list of directories which will be searched when named data is requested.

**void QMimeSourceFactory::setImage ( const QString & abs_name, const QImage & image )** `[virtual]`

Sets *image* to be the data item associated with the absolute name *abs_name.*

Equivalent to setData(abs_name, new QImageDrag(image)).

**void QMimeSourceFactory::setPixmap ( const QString & abs_name, const QPixmap & pixmap )** `[virtual]`

Sets *pixmap* to be the data item associated with the absolute name *abs_name.*

**void QMimeSourceFactory::setText ( const QString & abs_name, const QString & text )** `[virtual]`

Sets *text* to be the data item associated with the absolute name *abs_name.*

Equivalent to setData(abs_name, new QTextDrag(text)).

### 141.2.1. URL

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

# 142. QMotifStyle Class Reference

`#include <qmotifstyle.h>`

Inherits QCommonStyle.

Inherited by QCDEStyle.

## 142.1. Detailed Description

Motif Look and Feel

This class implements the Motif look and feel. It almost completely resembles the original Motif look as defined by the Open Group, but also contains minor improvements. The Motif style is Qt's default GUI style on UNIX plattforms.

## 142.2. Member Function Documentation

**QMotifStyle::QMotifStyle ( bool useHighlightCols = FALSE )**

Constructs a QMotifStyle.

If useHighlightCols is FALSE (default value), then the style will polish the application's color palette to emulate the Motif way of highlighting, which is a simple inversion between the base and the text color.

**void QMotifStyle::drawArrow ( QPainter * p, ArrowType type, bool down, int x, int y, int w, int h, const QColorGroup & g, bool, const QBrush * = 0 )**

Reimplemented for internal reasons; the API is not affected.

**void QMotifStyle::drawBevelButton ( QPainter * p, int x, int y, int w, int h, const QColorGroup & g, bool sunken = FALSE, const QBrush * fill = 0 )** `[virtual]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QStyle.

**void QMotifStyle::drawButton ( QPainter * p, int x, int y, int w, int h, const QColorGroup & g, bool sunken = FALSE, const QBrush * fill = 0 )** `[virtual]`

Draws a press-sensitive shape.

Reimplemented from QStyle.

**void QMotifStyle::drawCheckMark ( QPainter * p, int x, int y, int w, int h, const QColorGroup & g, bool act, bool dis )**

Reimplemented for internal reasons; the API is not affected.

**void QMotifStyle::drawExclusiveIndicator ( QPainter * p, int x, int y, int w, int h, const QColorGroup & g, bool on, bool down = FALSE, bool = TRUE )** `[virtual]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QStyle.

**void QMotifStyle::drawExclusiveIndicatorMask ( QPainter * p, int x, int y, int, int, bool )** `[virtual]`

Draws the mask of a mark indicating the state of an exclusive choice.

Reimplemented from QStyle.

**void QMotifStyle::drawFocusRect ( QPainter * p, const QRect & r, const QColorGroup & g, const QColor * bg =0, bool atBorder = FALSE )** `[virtual]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QStyle.

**void QMotifStyle::drawIndicator ( QPainter * p, int x, int y, int w, int h, const QColorGroup & g, int s, bool down = FALSE, bool = TRUE )** `[virtual]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QStyle.

**void QMotifStyle::drawPushButton ( QPushButton * btn, QPainter * p )** `[virtual]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QStyle.

**void QMotifStyle::drawScrollBarControls ( QPainter * p, const QScrollBar * sb, int sliderStart, uint controls, uint activeControl )** `[virtual]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QStyle.

**void QMotifStyle::drawSlider ( QPainter * p, int x, int y, int w, int h, const QColorGroup & g, Orientation orient, bool, bool )** `[virtual]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QStyle.

**void QMotifStyle::drawSliderGroove ( QPainter * p, int x, int y, int w, int h, const QColorGroup & g, QCOORD, Orientation )** `[virtual]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QStyle.

**void QMotifStyle::drawSplitter ( QPainter * p, int x, int y, int w, int h, const QColorGroup & g, Orientation orient )** `[virtual]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QStyle.

**void QMotifStyle::drawTab ( QPainter * p, const QTabBar * tb, QTab * t, bool selected )** `[virtual]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QStyle.

**void QMotifStyle::drawTabMask ( QPainter * p, const QTabBar * tb, QTab * t, bool selected )** `[virtual]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QStyle.

**QSize QMotifStyle::exclusiveIndicatorSize () const** `[virtual]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QStyle.

**QSize QMotifStyle::indicatorSize () const** `[virtual]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QStyle.

**void QMotifStyle::polish ( QApplication * a )** `[virtual]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QStyle.

**void QMotifStyle::polish ( QPalette & pal )**

Reimplemented for internal reasons; the API is not affected.

**void QMotifStyle::polish ( QWidget * w )**

Reimplemented for internal reasons; the API is not affected.

**void QMotifStyle::polishPopupMenu ( QPopupMenu * p )**

Reimplemented for internal reasons; the API is not affected.

**void QMotifStyle::scrollBarMetrics ( const QScrollBar * sb, int & sliderMin, int & sliderMax, int & sliderLength, int & buttonDim )** `[virtual]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QStyle.

**void QMotifStyle::setUseHighlightColors ( bool arg )**

If the argument is FALSE, then the style will polish the application's color palette to emulate the Motif way of highlighting, which is a simple inversion between the base and the text color.

The effect will show up the next time a application palette is set via QApplication::setPalette(). The current color palette of the application remains unchanged.

See also: QStyle::polish(, QPalette&, ) and selectionOnlyInverse().

**int QMotifStyle::sliderLength () const** `[virtual]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QStyle.

**int QMotifStyle::splitterWidth () const** `[virtual]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QStyle.

**void QMotifStyle::tabbarMetrics ( const QTabBar * t, int & hframe, int & vframe, int & overlap )** `[virtual]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QStyle.

**bool QMotifStyle::useHighlightColors () const**

Returns whether the style treats the highlight colors of the palette Motif-like, which is a simple inversion between the base and the text color. The default is FALSE.

See also: setSelectionOnlyInverse().

### 142.2.1. URL

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

# 143. QMouseEvent Class Reference

`#include <qevent.h>`

Inherits QEvent.

### 143.1. 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
drawlines/connect.cpp
popup/popup.cpp
qmag/qmag.cpp
scrollview/scrollview.cpp
trivial/trivial.cpp
movies/main.cpp

### 143.2. Member Function Documentation

**QMouseEvent::QMouseEvent ( Type type, const QPoint & pos, const QPoint & globalPos, int button, int state )**

Constructs a mouse event object.

The type parameter must be `QEvent::MouseButtonPress`, `QEvent::MouseButtonRelease`, `QEvent::MouseButtonDblClick` or `QEvent::MouseMove`.

**QMouseEvent::QMouseEvent ( Type type, const QPoint & pos, int button, int state )**

Constructs a mouse event object.

The type parameter must be `QEvent::MouseButtonPress`, `QEvent::MouseButtonRelease`, `QEvent::MouseButtonDblClick` or `QEvent::MouseMove`.

**ButtonState 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:

movies/main.cpp

**const QPoint & QMouseEvent::globalPos () const**

Returns the global position of the mouse pointer *at the time* of the event. This is important on asynchronous window systems like X11: Whenever you move your widgets around in response to mouse evens, globalPos() can differ a lot from the current pointer position QCursor::pos().

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
drawlines/connect.cpp
qmag/qmag.cpp

**ButtonState 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 `QEvent::MouseButtonPress` and

`QEvent::MouseButtonDblClick`, the flag for the button() itself will not be set in the state;

while for `QEvent::MouseButtonRelease`, it will.

The returned value is `LeftButton, RightButton, MidButton, ShiftButton,`

`ControlButton` and `AltButton` OR'ed together.

See also: button() and stateAfter().

Examples:

movies/main.cpp

**ButtonState QMouseEvent::stateAfter () const**

Returns the state of buttons after the event.

See also: state().

**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().

Examples:

scrollview/scrollview.cpp

**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().

Examples:

scrollview/scrollview.cpp

### 143.2.1. URL

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

# 144. QMoveEvent Class Reference

```
#include <qevent.h>
```

Inherits QEvent.

### 144.1. 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().

### 144.2. 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().

#### 144.2.1. URL

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

# 145. QMovie Class Reference

```
#include <qmovie.h>
```

### 145.1. 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.

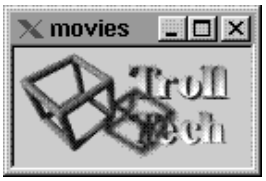
In Qt 1.30, the decoder factory interface is not available for adding support for new formats. Only GIF support is installed. The GIF decoder supports interlaced images, transparency, looping, image-restore disposal, local color maps, and background colors. The Netscape looping extension is obeyed.

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

## 145.2. 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 ( QByteArray data, int bufsize=1024 )**

Creates a QMovie which reads an image sequence from given data.

**QMovie::QMovie ( const QString & fileName, int bufsize=1024 )**

Creates a QMovie which reads an image sequence from the named file.

**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 ()**

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.

**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.

**void QMovie::disconnectResize ( QObject * receiver, const char * member=0 )**

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=0 )**

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.

**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.

**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().

### 145.2.1. URL

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

# 146. QMultiLineEdit Class Reference

```
#include <qmultilineedit.h>
```

Inherits QTableView.

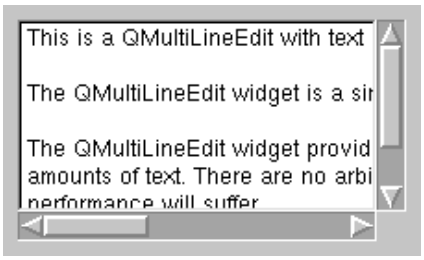
### 146.1. Detailed Description

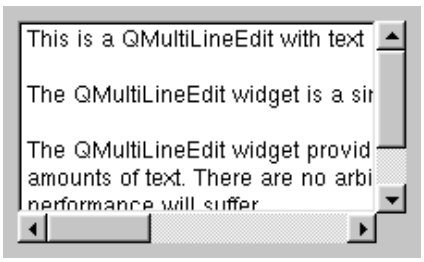
The QMultiLineEdit widget is a simple editor for inputting text.

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:

mainlyQt/editor.cpp
layout/layout.cpp
mainlyXt/editor.cpp
mainlyMotif/editor.cpp
rot13/rot13.cpp

scrollview/scrollview.cpp
customlayout/main.cpp

### 146.2. Member Function Documentation

**void QMultiLineEdit::append ( const QString & s )** `[slot]`

Appends *s* to the text.

**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::copy () const** `[slot]`

Copies the marked text to the clipboard. Will only copy if echoMode() is Normal.

**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::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::dragLeaveEvent ( QDragLeaveEvent * )** `[virtual protected]`

Handles drag leave events, cancelling any auto-scrolling.

Reimplemented from QWidget.

**void QMultiLineEdit::dragMoveEvent ( QDragMoveEvent * event )** `[virtual protected]`

Handles drag motion events, accepting text and positioning the cursor.

Reimplemented from QWidget.

**void QMultiLineEdit::dropEvent ( QDropEvent * event )** `[virtual protected]`

Handles drop events, pasting text.

Reimplemented from QWidget.

**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.

**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::insert ( const QString & s )** `[slot]`

Inserts *s* at the current cursor position.

**void QMultiLineEdit::insert ( const QString & str, bool mark )** `[virtual protected]`

Inserts *c* at the current cursor position.

**void QMultiLineEdit::insertChar ( QChar c )** `[protected]`

Inserts *c* at the current cursor position. (this function is provided for backward compatibility - it simply calls insert()).

**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 * e )** `[virtual protected]`

Handles mouse release events.

Reimplemented from QWidget.

**void QMultiLineEdit::newLine ()** `[virtual protected]`

Makes a line break at the current cursor position.

**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::resizeEvent ( QResizeEvent * e )** `[virtual protected]`

Reimplemented for internal purposes.

Reimplemented from QWidget.

**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::setOverwriteMode ( bool on )** `[virtual 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 )** `[virtual 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 QString & s )** `[virtual slot]`

Sets the text to *s,* removing old text, if any.

Examples:

mainlyQt/editor.cpp
layout/layout.cpp
mainlyXt/editor.cpp
mainlyMotif/editor.cpp
scrollview/scrollview.cpp
customlayout/main.cpp

**QString QMultiLineEdit::stringShown ( int row ) const** `[protected]`

Returns the string shown at line *row,* including processing of the echoMode().

**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().

**int QMultiLineEdit::textWidth ( const 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.

**void QMultiLineEdit::wheelEvent ( QWheelEvent * e )** `[virtual protected]`

An override - pass wheel events to the vertical scrollbar.

Reimplemented from QWidget.

### 146.2.1. URL

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

# 147. QNPInstance Class Reference

`#include <qnp.h>`

Inherits QObject.

## 147.1. 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

## 147.2. 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() and 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.

Examples:

grapher/grapher.cpp

**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.

**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.

**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.

**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.

**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.

**void QNPInstance::status ( const char * msg )**

Sets the status message in the browser containing this instance.

**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 behavior.

See also: newStreamCreated(), newStream(), Note, that, the, AsFileOnly, method, is, not,

supported, by, Netscape, 2.0, MS-Explorer and 3.0.

**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.

**const char* QNPInstance::userAgent () const**

Returns the user agent (browser name) containing this instance.

**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.

**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.

### 147.2.1. URL

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

# 148. QNPlugin Class Reference

```
#include <qnp.h>
```

## 148.1. 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

## 148.2. 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() and QNPInstance::getJavaPeer().

**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() and QNPInstance::getJavaPeer().

**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.

**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.

**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

### 148.2.1. URL

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

# 149. QNPStream Class Reference

```
#include <qnp.h>
```

### 149.1. 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

### 149.2. 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.

**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.

### 149.2.1. URL

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

# 150. QNPWidget Class Reference

`#include <qnp.h>`

Inherits QWidget.

## 150.1. 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

## 150.2. 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 behavior.

### 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.

### 150.2.1. URL

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

# 151. QObject Class Reference

`#include <qobject.h>`

Inherits Qt.

Inherited by QAccel, QApplication, QClipboard, QDataPump, QDragManager, QDragObject, QFileIconProvider, QLayout, QNPInstance, QSenderObject, QSessionManager, QSignal, QSignalMapper, QSocketNotifier, QStyle, QStyleSheet, QTimer, QToolTipGroup, QTranslator, QValidator and QWidget.

## 151.1. 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()).

## 151.2. 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.

**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.

**QObject* QObject::child ( const char * name, const char * type = 0 )**

Returns the pointer to a child widget with the required name and type or 0 if no child matches. This function works recursive. That means it traverses the entire object tree to find the child. That in turn means that names have to be unique with regard to their toplevel window.

If multiple widgets with the same name and type are found then it is undefined which one of them is

returned.

If *type* is set to 0 then the only criterion is the objects name.

This function is useful if you need a widget of a dialog that was created from a ressource file.

**void QObject::childEvent ( QChildEvent * )** `[virtual protected]`

This event handler can be reimplemented in a subclass to receive child events.

Child events are sent to objects when children are inserted or removed.

See also: event() and QChildEvent.

Reimplemented in QMainWindow, QWidgetStack and QSplitter.

**const QObjectList * QObject::children () const**

Returns a list of child objects, or 0 if this object has no children.

The QObjectList class is defined in the 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().

**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();
...
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:

showimg/main.cpp
grapher/grapher.cpp
xform/xform.cpp
application/main.cpp
qiconview/main.cpp
drawdemo/drawdemo.cpp
popup/popup.cpp
menu/menu.cpp
qmag/qmag.cpp
qwerty/main.cpp
forever/forever.cpp
rot13/rot13.cpp
scrollview/scrollview.cpp
addressbook/main.cpp
movies/main.cpp
hello/main.cpp
qbrowser/main.cpp
customlayout/main.cpp

**bool QObject::connect ( const QObject * sender, const char * signal, const char * member ) const**

This is an overloaded member function, provided for convenience. It 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.

**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 QSplitter, QToolBar, QClipboard, QSocketNotifier, QTimer, QMainWindow and

QWidget.

**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 QScrollView, QAccel, QToolBar, QMainWindow, QTabWidget, QFontDialog,

QListView, QComboBox, QFileDialog, QSpinBox, QLayout, QMenuBar and QWizard.

**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 )** `[virtual]`

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::reparent().

See also: removeChild() and QWidget::reparent().

**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 parent widget
        parent->installEventFilter( this ); // then install filter
}
bool MyWidget::eventFilter( QObject *, QEvent *e )
{
    if ( e->type() == QEvent::KeyPress ) {  // key press
        QKeyEvent *k = (QKeyEvent*)e;
        qDebug( "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:

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:

xform/xform.cpp
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. If the object does not have a name, it will return "unnamed", so that printf() (used in debug()) will not be asked to output a null pointer. If you want a null pointer to be returned for unnamed objects, you can call name(0).

```
qDebug( "MyClass::setPrecision(): (%s) unable to set precision to %f",
        name(), newPrecision );
```

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().

**const char * QObject::name ( const char * defaultName ) const**

Returns the name of this object, or *defaultName* if the object does not have a name.

**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 )** `[virtual]`

Removes the child object *obj* from the list of children.

**Warning:** This function will not remove a child widget from the screen. It will only remove it from the parent widget's list of children.

See also: insertChild() and QWidget::reparent().

Reimplemented in QScrollView.

**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 )** `[virtual]`

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().

Reimplemented in QWidget and QSignal.

**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 the timer event occurs once every time there are no more window system events to process.

The virtual timerEvent() function is called with the QTimerEvent event parameter class when a timer event occurs. Reimplement this function to get timer events.

If multiple timers are running, the QTimerEvent::timerId() can be used to find out which timer was activated.

Example:

```
class MyObject : public QObject
{
public:
    MyObject( QObject *parent=0, const char *name=0 );
protected:
    void  timerEvent( QTimerEvent * );
};
MyObject::MyObject( QObject *parent, const char *name )
    : QObject( parent, name )
{
    startTimer( 50 );                       // 50 millisecond timer
    startTimer( 1000 );                     // 1 second timer
    startTimer( 60000 );                    // 1 minute timer
}
void MyObject::timerEvent( QTimerEvent *e )
{
    qDebug( "timer event, id=%d", e->timerId() );
}
```

There is practically no upper limit for the interval value (more than one year). 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:

xform/xform.cpp
qmag/qmag.cpp
forever/forever.cpp

**void QObject::staticMetaObject ()** `[static protected]`

The functionality of initMetaObject(), provided as a static function.

**QStringList QObject::superClasses ( bool includeThis = FALSE ) const**

Returns a list of all super classes. If *includeThis* is TRUE then the most derived class name of this object is added, too. The most derived classes appear first in the list. That means that QObject is always the last entry in the list.

**void QObject::timerEvent ( QTimerEvent * )** `[virtual protected]`

This event handler can be reimplemented in a subclass to receive timer events for the object.

QTimer provides a higher-level interface to the timer functionality, and also more general

information about timers.

See also: startTimer(), killTimer(), killTimers() and event().

Reimplemented in QPopupMenu and QMultiLineEdit.

**QString QObject::tr ( const char * text )** `[static]`

Returns a translated version of *text,* or *text* if there is no appropriate translated version. All QObject subclasses which use the Q_OBJECT macro have an overridden version of this.

See also: QApplication::translate().

Examples:

xform/xform.cpp

### 151.3. 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.

```
QListBox * c = (QListBox *)::qt_find_obj_child(myWidget,QListBox,
                                               "listboxname");
```

```
    if ( c )
        c->insertItem( "another string" );
```

### 151.3.1. URL

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

# 152. QPaintDevice Class Reference

```
#include <qpaintdevice.h>
```

Inherited by QPicture, QPixmap, QPrinter and QWidget.

## 152.1. 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.

### 152.2. 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 QPrinter and QPicture.

**int QPaintDevice::devType () const**

Returns the device type identifier: `QInternal::Widget`, `QInternal::Pixmap`, `QInternal::Printer`, `QInternal::Picture` or `QInternal::UndefinedDevice`.

**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.

**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 QWidget, QPrinter, QPicture and QPixmap.

**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().

## 152.3. Related Functions

**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*.

Examples:

xform/xform.cpp
desktop/desktop.cpp

**void bitBlt (QPaintDevice * dst, int dx, int dy, const QPaintDevice * src, int sx, int sy, int sw, int sh, Qt::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 = src OR dst.
- `XorROP:` dst = src XOR dst.
- `NotAndROP:` dst = (NOT src) AND dst
- `NotCopyROP:` dst = NOT src
- `NotOrROP:` dst = (NOT src) OR dst
- `NotXorROP:` dst = (NOT src) XOR dst
- `AndROP` dst = src AND dst
- `NotROP:` dst = NOT dst
- `ClearROP:` dst = 0
- `SetROP:` dst = 1
- `NopROP:` dst = dst
- `AndNotROP:` dst = src AND (NOT dst)
- `OrNotROP:` dst = src OR (NOT dst)
- `NandROP:` dst = NOT (src AND dst)
- `NorROP:` dst = NOT (src OR 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.

### 152.3.1. URL

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

# 153. QPaintDeviceMetrics Class Reference

```
#include <qpaintdevicemetrics.h>
```

### 153.1. 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 );
double aspect = (double)pdm.widthMM / (double)pdm.heightMM();
```

### 153.2. 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).

**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).

**int QPaintDeviceMetrics::widthMM () const**

Returns the width of the paint device, measured in millimeters.

### 153.2.1. URL

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

# 154. QPainter Class Reference

```
#include <qpainter.h>
```

Inherits Qt.

### 154.1. 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( Qt::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( Qt::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:** The range of acceptable coordinate values to QPainter's various drawing functions is limited by the capabilities of the drawing engine of the underlying window system. Currently, only Windows NT is able to handle the full 32 bit range; on other platforms, the output may be incorrect when the absolute value of any coordinate (after any set world and/or view transforms have been applied) exceeds some system-dependent value. All systems we have tested were able to correctly handle coordinates up to +/- 2000.

**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
grapher/grapher.cpp
drawlines/connect.cpp
xform/xform.cpp
drawdemo/drawdemo.cpp
progress/progress.cpp
qmag/qmag.cpp
splitter/splitter.cpp
forever/forever.cpp
desktop/desktop.cpp
scrollview/scrollview.cpp
trivial/trivial.cpp
movies/main.cpp
picture/picture.cpp

## 154.2. 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.

**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
picture/picture.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(size());
    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 QString & 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 qnamespace.h.

See also: drawText() and fontMetrics().

**QRect QPainter::boundingRect ( const QRect & r, int tf, const QString &, 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:

grapher/grapher.cpp
drawlines/connect.cpp
progress/progress.cpp
splitter/splitter.cpp
scrollview/scrollview.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
qmag/qmag.cpp
scrollview/scrollview.cpp
movies/main.cpp
picture/picture.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:

drawlines/connect.cpp
desktop/desktop.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

**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:

grapher/grapher.cpp
drawdemo/drawdemo.cpp
forever/forever.cpp
trivial/trivial.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 QString & 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.

**void QPainter::drawText ( int x, int y, int w, int h, int tf, const QString & 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.

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 qnamespace.h.

See also: boundingRect().

Examples:

grapher/grapher.cpp
drawdemo/drawdemo.cpp
progress/progress.cpp
desktop/desktop.cpp
scrollview/scrollview.cpp
trivial/trivial.cpp
movies/main.cpp
picture/picture.cpp

**void QPainter::drawText ( const QPoint & p, const QString &, 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 QString &, 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)* specify 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
picture/picture.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() )`

**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
scrollview/scrollview.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:

drawdemo/drawdemo.cpp
desktop/desktop.cpp
scrollview/scrollview.cpp
movies/main.cpp

**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::restoreWorldMatrix ()**

Restores the current world matrix (pops a saved matrix off the stack).

See also: saveWorldMatrix().

**void QPainter::rotate ( double 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::saveWorldMatrix ()**

Saves the current world matrix (pushes the matrix onto a stack).

In sane code a save() has a corresponding restoreWorldMatrix().

See also: restoreWorldMatrix().

**void QPainter::scale ( double sx, double 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
drawdemo/drawdemo.cpp
forever/forever.cpp
desktop/desktop.cpp
picture/picture.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
progress/progress.cpp
splitter/splitter.cpp
trivial/trivial.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
scrollview/scrollview.cpp
movies/main.cpp
picture/picture.cpp

**void QPainter::setPen ( PenStyle style )**

Sets a new painter pen with style `style`, width 0 and black color.

See also: pen() and QPen.

Examples:

grapher/grapher.cpp
drawlines/connect.cpp
drawdemo/drawdemo.cpp
progress/progress.cpp
forever/forever.cpp
desktop/desktop.cpp
scrollview/scrollview.cpp
movies/main.cpp

**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().

**void QPainter::setRasterOp ( RasterOp r )**

Sets the raster operation to *r*.

The *r* parameter must be one of:

- `Qt::CopyROP:` dst = src.
- `Qt::OrROP:` dst = src OR dst.
- `Qt::XorROP:` dst = src XOR dst.
- `Qt::NotAndROP:` dst = (NOT src) AND dst
- `Qt::NotCopyROP:` dst = NOT src
- `Qt::NotOrROP:` dst = (NOT src) OR dst
- `Qt::NotXorROP:` dst = (NOT src) XOR dst
- `Qt::AndROP` dst = src AND dst
- `Qt::NotROP:` dst = NOT dst
- `Qt::ClearROP:` dst = 0
- `Qt::SetROP:` dst = 1
- `Qt::NopROP:` dst = dst
- `Qt::AndNotROP:` dst = src AND (NOT dst)
- `Qt::OrNotROP:` dst = src OR (NOT dst)
- `Qt::NandROP:` dst = NOT (src AND dst)
- `Qt::NorROP:` dst = NOT (src OR dst)

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:

drawdemo/drawdemo.cpp
forever/forever.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( double a )
{
    QWMatrix m;
    m.rotate( a );
    setWorldMatrix( m, TRUE );
}
```

Note that you should always use combine when you are drawing into a QPicture. Otherwise the picture may not be completely encapsulated and cannot be replayed with additional transformations. Using the translate(), scale(), etc. functions is safe.

Furthermore, you can easily save and restore the current world transformation matrix with the convenient functions saveWorldMatrix() and restoreWorldMatrix(), respectively. If you need to draw some top-to-bottom text at the position (x y), for instance, but everything else shall remain unrotated, you can easily achieve this with:

```
void MyWidget::paintEvent()
{
    QPainter paint( this );
    ...
    paint.saveWorldMatrix();
    paint.translate( x, y );
    paint.rotate( 90 );
    paint.drawText( 0, 0, "top-to-bottom text" );
    paint.restoreWorldMatrix();
    ....
}
```

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

**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 ( double sh, double 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 ( double dx, double 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().

### 154.3. Related Functions

**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().

**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 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 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().

**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().

### 154.3.1. URL

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

# 155. QPaintEvent Class Reference

```
#include <qevent.h>
```

Inherits QEvent.

## 155.1. 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
grapher/grapher.cpp
drawlines/connect.cpp
xform/xform.cpp
drawdemo/drawdemo.cpp
progress/progress.cpp
qmag/qmag.cpp
splitter/splitter.cpp
forever/forever.cpp
desktop/desktop.cpp
trivial/trivial.cpp
picture/picture.cpp

## 155.2. Member Function Documentation

**QPaintEvent::QPaintEvent ( const QRect & paintRect, bool erased=TRUE = TRUE )**

Constructs a paint event object with the rectangle that should be updated.

**QPaintEvent::QPaintEvent ( const QRegion & paintRegion, bool erased=TRUE = TRUE )**

Constructs a paint event object with the region that should be updated.

**bool QPaintEvent::erased () const**

Returns whether the paint event region (or rectangle) has been erased with the widget's background.

**const QRect & QPaintEvent::rect () const**

Returns the rectangle that should be updated.

See also: region() and QPainter::setClipRect().

Examples:

qtimage/qtimage.cpp
grapher/grapher.cpp
progress/progress.cpp
splitter/splitter.cpp
trivial/trivial.cpp

**const QRegion & QPaintEvent::region () const**

Returns the region that should be updated.

See also: rect() and QPainter::setClipRegion().

### 155.2.1. URL

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

# 156. QPalette Class Reference

```
#include <qpalette.h>
```

## 156.1. 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.

## 156.2. Member Function Documentation

### QPalette::QPalette ()

Constructs a palette that consists of color groups with only black colors.

### QPalette::QPalette ( const QColor & button )

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

Constructs a palette from the *button* color. The other colors are automatically calculated, based on this color. Background will be the button color as well.

**QPalette::QPalette ( const QColor & button, const QColor & background )**

Constructs a palette from a *button* color and a background. 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().

**const QBrush & QPalette::brush ( ColorGroup gr, QColorGroup::ColorRole r ) const**

Returns the brush in *gr* used for color role *r.*

**const QColor & QPalette::color ( ColorGroup gr, QColorGroup::ColorRole r ) const**

Returns the color in *gr* used for color role *r.*

See also: brush().

**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::setBrush ( ColorGroup gr,QColorGroup::ColorRole r,const QBrush & b )**

Sets the brush in *gr* used for color role *r* to *b*.

**void QPalette::setBrush ( QColorGroup::ColorRole r, const QBrush & b )**

Sets the brush in all color groups (Normal, Disabled, and Active) that is used for color role *r* to *b*.

**void QPalette::setColor ( ColorGroup gr,QColorGroup::ColorRole r,const QColor & c )**

Sets the brush in *gr* used for color role *r* to the solid color *c*.

**void QPalette::setColor ( QColorGroup::ColorRole r, const QColor & c )**

Sets the color of the brush in *gr* used for color role *r* to *c*.

**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().

### 156.3. Related Functions

**QDataStream & operator>> (QDataStream & s, QPalette & p)**

Reads a palette from the stream and returns a reference to the stream.

**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.

### 156.3.1. URL

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

# 157. QPen Class Reference

`#include <qpen.h>`

Inherits Qt.

### 157.1. 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 `Qt::SolidLine`. 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 precision 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:

progress/progress.cpp
desktop/desktop.cpp

### 157.2. 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::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().

### 157.3. 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.

#### 157.3.1. URL

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

# 158. QPicture Class Reference

```
#include <qpicture.h>
```

Inherits QPaintDevice.

## 158.1. 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:

xform/xform.cpp
picture/picture.cpp

## 158.2. Member Function Documentation

**QPicture::QPicture ( int formatVer = 0 )**

Constructs an empty picture.

The *formatVersion* parameter may be used to create a QPicture that can be read by programs compiled with earlier versions of Qt. Currently supported is *formatVersion* == 1, which will make a QPicture which is stored in a format that is binary compatible with Qt 1.x programs.

Reading of Qt 1.x format pictures is supported and needs no special coding; the format is automatically detected.

**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 QString & fileName )**

Loads a picture from the file specified by *fileName* and returns TRUE if successful, otherwise FALSE.

See also: save().

Examples:

xform/xform.cpp
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, dpi = 72,

numcolors=16777216 and depth=24.

Reimplemented from QPaintDevice.

**QPicture& QPicture::operator= ( const QPicture & p )**

Makes this picture be a deep copy of *p*.

**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 QString & 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 )** `[virtual]`

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().

### 158.2.1. URL

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

# 159. QPixmap Class Reference

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

Inherits QPaintDevice and Qt.

Inherited by QBitmap.

## 159.1. 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 QPixmap; 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.

**Note about Windows 95 and 98:** Because of internal limitations in the operating system, Windows 9x easily crashes if you create around 1000 pixmaps or more. Qt will work around this problem if you set the pixmap optimization to `QPixmap::MemoryOptim`. Read more about this in the setOptimization() documentation.

See also: QBitmap, QImage, QImageIO and Shared Classes

Examples:

qtimage/qtimage.cpp
grapher/grapher.cpp
xform/xform.cpp
menu/menu.cpp
qmag/qmag.cpp
desktop/desktop.cpp
scrollview/scrollview.cpp
movies/main.cpp
picture/picture.cpp

## 159.2. Member Function Documentation

### QPixmap::QPixmap ()

Constructs a null pixmap.

See also: isNull().

### QPixmap::QPixmap ( const QString & 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 QString & 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 QByteArray & img_data )**

Constructs a pixmaps by loading from *img_data.* The data can be in any image format supported by Qt.

See also: loadFromData().

**QPixmap::QPixmap ( const QPixmap & pixmap )**

Constructs a pixmap which is a copy of *pixmap.*

**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, Optimization optimization = DefaultOptim )**

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, Optimization optimization = DefaultOptim )**

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 colors 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.

If a pixmap with depth 1 is painted with color0 and color1 and converted to an image, the pixels painted with color0 will produce pixel index 0 in the image and those painted with color1 will produce pixel index 1.

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 & image, 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().

**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 = Qt::white )**

Fills the pixmap with the color *fillColor*.

**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 );
}
```

Examples:

grapher/grapher.cpp
xform/xform.cpp
desktop/desktop.cpp

**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 arguments *(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
xform/xform.cpp
desktop/desktop.cpp
scrollview/scrollview.cpp
movies/main.cpp

**const char* QPixmap::imageFormat ( const QString & 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
scrollview/scrollview.cpp

**bool QPixmap::isQBitmap () const**

Returns TRUE if this is a QBitmap, otherwise FALSE.

**bool QPixmap::load ( const QString & 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:

xform/xform.cpp
scrollview/scrollview.cpp
picture/picture.cpp

**bool QPixmap::load ( const QString & 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 ( const 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.

**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 QString & 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 & newmask )**

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::DefaultOptim`, sets the default optimization, i.e. QPixmap::defaultOptimization().
- `QPixmap::NoOptim`, equivalent with MemoryOptim.
- `QPixmap::MemoryOptim`, use as little memory/resource as possible, with little or no caching. The tradeoff may be unnoticably slower pixmap operations. Read the important note about Windows 95/98 below.
- `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, but may consume more memory. Use this option for pixmap drawn extremely frequently, e.g. masked pixmaps/sprites.

Use the setDefaultOptimization() to change the default optimization for all new pixmaps.

**Note about Windows 95 and 98:** On Windows 9x, the system crashes if you create around 1000 pixmaps or more, independent of the size of the pixmaps or installed RAM. The reason is that Windows 9x has a serious resource limitation that comes from the old, 64k segmented memory model. Windows NT does not have this limitation.

Qt tries to work around the resource limitation. If you set the pixmap optimization to `QPixmap::MemoryOptim` and the width of your pixmap is less than or equal to 128 pixels, Qt will create a logical pixmap which refers an internal pixmap. This internal pixmap is shared by hundreds of other pixmaps which have the optimization set to `QPixmap::MemoryOptim`. The result is that your application may use thousands of pixmaps on Windows 9x and spend only a few percent of the system resources.

When should you set the memory optimization? If your application uses hundreds of pixmaps, e.g. on tool bar buttons, in popup menus, and you plan to run it on Windows 95 or Windows 98.

Here is an example:

```
QPixmap::setDefaultOptimization( QPixmap::MemoryOptim );
while ( ... ) {
  // load tool bar pixmaps etc.
  QPixmap *pixmap = new QPixmap(fileName);
}
QPixmap::setDefaultOptimization( QPixmap::NormalOptim );
```

See also: optimization(), setDefaultOptimization() and defaultOptimization().

Examples:

desktop/desktop.cpp

**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
xform/xform.cpp
desktop/desktop.cpp
scrollview/scrollview.cpp
movies/main.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 baseline 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
xform/xform.cpp
qmag/qmag.cpp
desktop/desktop.cpp
movies/main.cpp

### 159.3. Related Functions

**QDataStream & operator>> (QDataStream & s, QPixmap & pixmap)**

Reads a pixmap from the stream.

See also: QPixmap::load().

**QDataStream & operator<< (QDataStream & s, const QPixmap & pixmap)**

Writes a pixmap to the stream as a PNG image.

See also: QPixmap::save().

#### 159.3.1. URL

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

# 160. QPixmapCache Class Reference

```
#include <qpixmapcache.h>
```

## 160.1. 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.

## 160.2. 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 QString & 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 QString&, QPixmap&) instead.**

Example:

```
QPixmap* pp;
QPixmap p;
if ( (pp=QPixmapCache::find("my_previous_copy", pm)) ) {
    p = *pp;
} else {
    p.load("bigimage.png");
    QPixmapCache::insert("my_previous_copy", new QPixmap(p));
}
painter->drawPixmap(0, 0, p);
```

**bool QPixmapCache::find ( const QString & 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.png");
```

```
        QPixmapCache::insert("my_previous_copy", pm);
    }
    painter->drawPixmap(0, 0, p);
```

**bool QPixmapCache::insert ( const QString & 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 QString&, const QPixmap&) instead.

**void QPixmapCache::insert ( const QString & 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 your own 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().

### 160.2.1. URL

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

# 161. QPlatinumStyle Class Reference

`#include <qplatinumstyle.h>`

Inherits QWindowsStyle.

## 161.1. Detailed Description

Platinum Look and Feel

This class implements the Platinum look and feel. It's an experimental class that tries to resemble a Macinosh-like GUI style with the QStyle system. The emulation is, however, far from being perfect yet.

## 161.2. Member Function Documentation

**QPlatinumStyle::QPlatinumStyle ()**

Constructs a QPlatinumStyle.

**QRect QPlatinumStyle::buttonRect ( int x, int y, int w, int h )** `[virtual]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QStyle.

**QRect QPlatinumStyle::comboButtonFocusRect ( int x, int y, int w, int h )** `[virtual]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QStyle.

**QRect QPlatinumStyle::comboButtonRect ( int x, int y, int w, int h )** `[virtual]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QStyle.

**void QPlatinumStyle::drawBevelButton ( QPainter * p, int x, int y, int w, int h, const QColorGroup & g, bool sunken = FALSE, const QBrush * fill = 0 )** `[virtual]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QStyle.

**void QPlatinumStyle::drawButton ( QPainter * p, int x, int y, int w, int h, const QColorGroup & g, bool sunken = FALSE, const QBrush * fill = 0 )** `[virtual]`

Draws a press-sensitive shape.

Reimplemented from QStyle.

**void QPlatinumStyle::drawCheckMark ( QPainter * p, int x, int y, int w, int h, const QColorGroup & g, bool act, bool dis )**

Reimplemented for internal reasons; the API is not affected.

**void QPlatinumStyle::drawComboButton ( QPainter * p, int x, int y, int w, int h, const QColorGroup & g, bool = FALSE, bool editable = FALSE, bool = TRUE, const QBrush * fill = 0 )** `[virtual]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QStyle.

**void QPlatinumStyle::drawExclusiveIndicator ( QPainter * p, int x, int y, int w, int h, const QColorGroup & g, bool on, bool down = FALSE, bool = TRUE )** `[virtual]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QStyle.

**void QPlatinumStyle::drawIndicator ( QPainter * p, int x, int y, int w, int h, const QColorGroup & g, int s, bool down = FALSE, bool = TRUE )** `[virtual]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QStyle.

**void QPlatinumStyle::drawIndicatorMask ( QPainter * p, int x, int y, int w, int h, int s )** `[virtual]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QStyle.

**void QPlatinumStyle::drawPopupMenuItem ( QPainter * p, bool checkable, int maxpmw, int tab, QMenuItem * mi, const QPalette & pal, bool act, bool enabled, int x, int y, int w, int h )**

Reimplemented for internal reasons; the API is not affected.

**void QPlatinumStyle::drawPopupPanel ( QPainter * p, int x, int y, int w, int h, const QColorGroup & g, int lineWidth = 2, const QBrush * fill = 0 )** `[virtual]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QStyle.

**void QPlatinumStyle::drawPushButton ( QPushButton * btn, QPainter * p )** `[virtual]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QStyle.

**void QPlatinumStyle::drawPushButtonLabel ( QPushButton * btn, QPainter * p )** `[virtual]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QStyle.

**void QPlatinumStyle::drawRiffles ( QPainter * p, int x, int y, int w, int h, const QColorGroup & g, bool horizontal )** `[protected]`

draw the nifty Macintosh decoration used on sliders.

**void QPlatinumStyle::drawScrollBarBackground ( QPainter * p, int x, int y, int w, int h, const QColorGroup & g, bool horizontal, const QBrush * fill = 0 )** `[protected]`

Reimplemented for internal reasons; the API is not affected.

**void QPlatinumStyle::drawScrollBarControls ( QPainter * p, const QScrollBar * sb, int sliderStart, uint controls, uint activeControl )** `[virtual]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QStyle.

**void QPlatinumStyle::drawSlider ( QPainter * p, int x, int y, int w, int h, const QColorGroup & g, Orientation orient, bool, bool )** `[virtual]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QStyle.

**void QPlatinumStyle::drawSliderGroove ( QPainter * p, int x, int y, int w, int h, const QColorGroup & g, QCOORD c, Orientation orient )** `[virtual]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QStyle.

**QSize QPlatinumStyle::exclusiveIndicatorSize () const** `[virtual]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QStyle.

**int QPlatinumStyle::extraPopupMenuItemWidth ( bool checkable, int maxpmw, QMenuItem * mi, const QFontMetrics & fm )** `[virtual]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QStyle.

**QSize QPlatinumStyle::indicatorSize () const** `[virtual]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QStyle.

**int QPlatinumStyle::maximumSliderDragDistance () const** `[virtual]`

Reimplementation from QWindowsStyle to disable the Windows typical jump back when dragging controls.

See also: QStyle.

Reimplemented from QStyle.

**QColor QPlatinumStyle::mixedColor ( const QColor & c1, const QColor & c2 )** `[protected]`

mixes two colors to a new colors.

**void QPlatinumStyle::polishPopupMenu ( QPopupMenu * p )**

Reimplemented for internal reasons; the API is not affected.

**int QPlatinumStyle::popupMenuItemHeight ( bool checkable, QMenuItem * mi, const QFontMetrics & fm )**

Reimplemented for internal reasons; the API is not affected.

**void QPlatinumStyle::scrollBarMetrics ( const QScrollBar * sb, int & sliderMin, int & sliderMax, int & sliderLength, int & buttonDim )** `[virtual]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QStyle.

**int QPlatinumStyle::sliderLength () const** `[virtual]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QStyle.

### 161.2.1. URL

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

# 162. QPNGImagePacker Class Reference

```
#include <qpngio.h>
```

Inherits QPNGImageWriter.

### 162.1. Detailed Description

Creates well-compressed PNG animations.

By using transparency, QPNGImagePacker allows you to build a PNG image from a sequence of

QImages.

### 162.2. Member Function Documentation

**QPNGImagePacker::QPNGImagePacker ( QIODevice * iod, int storage_depth, int conversionflags )**

Create an image packer that writes PNG data to *iod,* using a *storage_depth* bit encoding (use 8 or 32, depending on the desired quality and compression requirements).

**bool QPNGImagePacker::packImage ( const QImage & img )**

Add the image *img* to the PNG animation, analyzing the differences between this and the previous image to improve compression.

**void QPNGImagePacker::setPixelAlignment ( int x )**

Align pixel differences to *x* pixels. For example, using 8 can improve playback on certain hardware.

Normally the default of 1-pixel alignment (ie. no alignment) gives bets compression and

performance.

#### 162.2.1. URL

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

# 163. QPoint Class Reference

```
#include <qpoint.h>
```

### 163.1. 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 `int)`. The minimum value of QCOORD is QCOORD_MIN (-2147483648) and the maximum value is QCOORD_MAX (2147483647).

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:

drawlines/connect.cpp
xform/xform.cpp
drawdemo/drawdemo.cpp
forever/forever.cpp

### 163.2. Member Function Documentation

**QPoint::QPoint ()**

Constructs a point (0,0) (a null point).

**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.

**int QPoint::manhattanLength () const**

Returns the sum of the absolute values of x() and y(), traditionally known as the "Manhattan length" of the vector from the origin to the point. The tradition arises since such distances apply to travelers who can only travel on a rectangular grid, like the streets of Manhattan.

This is a useful approximation to the true length, sqrt(pow(x(),2)+pow(y(),2)).

**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:

drawdemo/drawdemo.cpp

**int QPoint::y () const**

Returns the y coordinate of the point.

See also: x().

Examples:

drawdemo/drawdemo.cpp

### 163.3. Related Functions

**QPoint operator+ (const QPoint & p1, const QPoint & p2)**

Returns the sum of *p1* and *p2;* each component is added separately.

**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.

**QDataStream & operator<< (QDataStream & s, const QPoint & p)**

Writes a QPoint to the stream and returns a reference to the stream.

Serialization format: [x (Q_INT32), y (Q_INT32)].

**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)**

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.

**QDataStream & operator>> (QDataStream & s, QPoint & p)**

Reads a QPoint from the stream and returns a reference to the stream.

**bool operator!= (const QPoint & p1, const QPoint & p2)**

Returns TRUE if *p1* and *p2* are different, or FALSE if they are equal.

**bool operator== (const QPoint & p1, const QPoint & p2)**

Returns TRUE if *p1* and *p2* are equal, or FALSE if they are different.

**QPoint operator* (const QPoint & p, int c)**

Multiplies both of *p's* components by *c* and returns the result.

**QPoint operator- (const QPoint & p1, const QPoint & p2)**

Returns *p2* subtracted from *p1;* each component is subtracted separately.

**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.

### 163.3.1. URL

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

# 164. QPointArray Class Reference

```
#include <qpointarray.h>
```

Inherits QArray.

## 164.1. Detailed Description

The QPointArray class provides an array of points.

The QPointArray is an array of QPoint objects. In addition to the functions provided by QArray, QPointArray provides some handy methods:

For convenient reading and writing of the point data: setPoints(), putPoints(), point(), and setPoint().

For geometry operations: boundingRect() and translate(). As for the latter, note that QWMatrix provides a map() function for more general transformation of QPointArrays.

QPointArray is used by the QPainter to draw line segments, polylines, polygons and Bezier curves.

Note that since this class is a QArray, it is explicitly shared and works with shallow copies by default.

Examples:

drawdemo/drawdemo.cpp
desktop/desktop.cpp
picture/picture.cpp

## 164.2. 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().

**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.

**void QPointArray::cleanBuffers ()** `[static]`

For internal use only.

**QPointArray QPointArray::copy () const**

Creates a deep copy of the array.

**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*. The resulting array has sufficient resolution for pixel accuracy (see the overloaded function which takes an additional QWMatrix parameter).

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::makeArc ( int x, int y, int w, int h, int a1, int a2, const QWMatrix & xf )**

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,* transformed by the matrix *xf.* The resulting array has sufficient resolution for pixel accuracy.

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* ).

The returned array has sufficient resolution for use as pixels (see the overloaded function which

takes an additional QWMatrix parameter).

**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::point ( uint index ) const**

Returns the point at position *index* in the array.

Examples:

drawdemo/drawdemo.cpp

**void QPointArray::point ( uint index, int * x, int * y ) const**

Returns the point at position *index* in the array in **x* and **y*.

**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() ).

Examples:

drawdemo/drawdemo.cpp

**void QPointArray::setPoint ( uint index, int x, int y )**

Sets the point at position *index* in the array to *(x,y).*

**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().

**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::shortPoints ( int index = 0, int nPoints = -1 ) const**

For internal use only.

**void QPointArray::translate ( int dx, int dy )**

Translates all points in the array *(dx,dy)*.

### 164.3. Related Functions

**QDataStream & operator>> (QDataStream & s, QPointArray & a)**

Reads a point array from the stream and returns a reference to the stream.

**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)

### 164.3.1. URL

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

# 165. QPopupMenu Class Reference

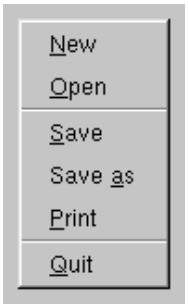
```
#include <qpopupmenu.h>
```

Inherits QFrame and QMenuData.

### 165.1. Detailed Description

The QPopupMenu class provides a popup menu 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
mainlyQt/editor.cpp
layout/layout.cpp
menu/menu.cpp
progress/progress.cpp
scrollview/scrollview.cpp

### 165.2. Member Function Documentation

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

Constructs a popup menu with a parent and a widget name.

Although a popup menu is always a top level widget, if a parent is passed, the popup menu will be deleted on destruction of that parent (as with any other QObject).

**QPopupMenu::~QPopupMenu ()**

Destroys the popup menu.

**void QPopupMenu::aboutToShow ()** `[signal]`

This signal is emitted just before the popup menu is displayed. You can connect it to any slot that sets up the menu contents (e.g. to ensure that the right items are enabled).

See also: 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.

**void QPopupMenu::closeEvent ( QCloseEvent * e )** `[virtual protected]`

Handles close events for the popup menu.

Reimplemented from QWidget.

**bool QPopupMenu::customWhatsThis () const** `[virtual]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QWidget.

**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 in either the popup menu or one of its submenus, or -1 if no item is selected (normally because the user presses Escape).

Note that all signals are emitted as usual. If you connect a menu item to a slot and call the menu's exec(), you get the result both via the signal-slot connection and in the return value of exec().

Common usage is to position the popup at the current mouse position:

```
exec(QCursor::pos());
```

or aligned to a widget:

```
exec(somewidget.mapToGlobal(QPoint(0,0)));
```

When positioning a popup with exec() or popup(), keep in mind that you cannot rely on the popup menu's current size(). For performance reasons, the popup adapts its size only when actually needed. So in many cases, the size before and after the show is different. Instead, use sizeHint(). It calculates the proper size depending on the menu's current contents.

See also: popup() and sizeHint().

**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.

**int QPopupMenu::idAt ( const QPoint & pos ) const**

Return the id of the item at *pos,* or -1 if there is no item there, or if it is a separator item.

**int QPopupMenu::idAt ( int index ) const**

Returns the identifier of the menu item at position *index* in the internal list, or -1 if *index* is out of range.

See also: QMenuData::setId() and QMenuData::indexOf().

Examples:

scrollview/scrollview.cpp

**bool QPopupMenu::isCheckable () const**

Returns whether display of check marks by the menu items is enabled.

See also: setCheckable() and QMenuData::setItemChecked().

**int QPopupMenu::itemHeight ( QMenuItem * mi ) const** `[protected]`

Calculates the height in pixels of the item *mi*.

**int QPopupMenu::itemHeight ( int row ) const** `[protected]`

Calculates the height in pixels of the item in row *row*.

**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::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().

When positioning a popup with exec() or popup(), keep in mind that you cannot rely on the popup menu's current size(). For performance reasons, the popup adapts its size only when actually needed. So in many cases, the size before and after the show is different. Instead, use sizeHint(). It calculates the proper size depending on the menu's current contents.

Examples:

qtimage/qtimage.cpp

**void QPopupMenu::setActiveItem ( int i )** `[virtual]`

Sets the currently active item to *i* and repaints as necessary.

**void QPopupMenu::setCheckable ( bool enable )** `[virtual]`

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
menu/menu.cpp
progress/progress.cpp
scrollview/scrollview.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.

**QSize QPopupMenu::sizeHint () const** `[virtual]`

Returns the size the popupmenu would use shall it become visible now. (##### is that english??)

Note that this size may be different from the popup's actual size. It changes all the time a new item is added or an existing one is modified. For performance reasons, QPopupMenu doesn't change its physical size each time this happens but only once before it is shown.

See also: exec() and show().

Reimplemented from QWidget.

**void QPopupMenu::styleChange ( QStyle & old )** `[virtual protected]`

Reimplemented for internal purposes.

Reimplemented from QWidget.

**void QPopupMenu::timerEvent ( QTimerEvent * e )** `[virtual protected]`

Handles timer events for the popup menu.

Reimplemented from QObject.

### 165.2.1. URL

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

# 166. QPrinter Class Reference

`#include <qprinter.h>`

Inherits QPaintDevice.

## 166.1. Detailed Description

The QPrinter class is a paint device that paint on a printer.

On Windows, it uses the built-in printer drivers. On X11, it generates postscript and sends that to lpr, lp or another print command.

QPrinter is used much the same way that QWidget and QPixmap are used. The big difference is that you must keep track of the pages.

QPrinter supports a number of settable parameters, mostly related to pages, and most can be changed by the end user in when the application calls QPrinter::setup().

The most important ones are:

- setOrientation() tells QPrinter to turn the page (virtual).
- setPageSize() tells QPrinter what page size to expect from the printer.
- setFullPage() tells QPrinter whether you want to deal with the full page (so you can have accurate margins etc.) or with jus with the part the printer can draw on. The default is FALSE: You can probably paint on (0,0) but the document's margins are unknown.
- setNumCopies() tells QPrinter how many copies of the document it should print.
- setMinMax() tells QPrinter and QPrintDialog what the allowed range for fromPage() and toPage() are.

There are also some settings that the user sets (through QPrintDialog) and that applications are expected to obey:

- pageOrder() tells the application program whether to print first-page-first or last-page-first.
- colorMode() tells the application program whether to print in color or grayscale. (If you print in color and the printer does not support color, Qt will try to approximate. The document may take

longer to print, but the quality should not be made visibly poorer.)

- fromPage() and toPage() indicate what pages the application program should print.

You can of course call e.g. setPageOrder() to establish a default before you ask the user through setup().

Once you've started printing, newPage() is essential. You will probably also need to look at the QPaintDeviceMetrics for the printer (see the simple print function in the Application walk-through).

If you want high-quality printing with accurate margins, setFullPage( TRUE ) is a must.

If you want to abort the print job, abort() will make a best effort. It may cancel the entire job, or just some of it.

Examples:

drawdemo/drawdemo.cpp

## 166.2. Member Type Documentation

### QPrinter::Orientation

This enum type (not to be confused with Qt::Orientation) is used to decide how Qt should print on each sheet.

- `Portrait` - (the default) means to print such that the page's height is greater than its width.
- `Landscape` - means to print such that the page's width is greater than its height.

This type interacts with QPrinter::PageSize and QPrinter::setFullPage() to determine the final size of the page available to the application.

### QPrinter::PageOrder

This enum type is used by QPrinter/QPrintDialog to tell the application program how to print. The possible values are

- `QPrinter::FirstPageFirst` - the lowest-numbered page should be printed first.
- `QPrinter::LastPageFirst` - the highest-numbered page should be printed first.

### QPrinter::PageSize

This enum type decides what paper size QPrinter is to use. QPrinter does not check that the paper size is available; it just uses this information together with Orientation and QPrinter::setFullPage() to determine the printable area (see QPaintDeviceMetrics).

The defined sizes (with setFullPage( TRUE )) are

- `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)

With setFullPage( FALSE ) (the default), the metrics will be a bit smaller. How much depends on the printer in use.

### 166.3. 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.

**QString QPrinter::creator () const**

Returns the creator name.

See also: setCreator().

**QString 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().

**bool QPrinter::fullPage () const**

**QSize QPrinter::margins () const**

**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().

**QString 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 system-dependent.

See also: setPageSize().

**QString QPrinter::printProgram () const**

Returns the name of the program that sends the print output to the printer.

The default is to return a null string; meaning that QPrinter will try to be smart in a system-

dependent way. On X11 only, you can set it to something different to use a specific print program.

See also: setPrintProgram() and setPrinterSelectionOption().

**QString QPrinter::printerName () const**

Returns the printer name. This value is initially set to the name of the default printer.

See also: setPrinterName().

**QString QPrinter::printerSelectionOption () const**

Returns the printer options selection string. This is only useful if the print command has been explicitly set.

The default value (a null string) implies to select printer in a system-dependent manner.

Any other value implies to use that value.

See also: setPrinterSelectionOption().

**void QPrinter::setColorMode ( ColorMode newColorMode )** `[virtual]`

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 QString & creator )** `[virtual]`

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" with some version number.

See also: creator().

**void QPrinter::setDocName ( const QString & name )** `[virtual]`

Sets the document name.

**void QPrinter::setFromTo ( int fromPage, int toPage )** `[virtual]`

Sets the from page and to page.

The from-page and to-page settings specify what pages to print.

This function is useful mostly for setting a default value that the user can override in the print dialog when you call setup().

See also: fromPage(), toPage(), setMinMax() and setup().

**void QPrinter::setFullPage ( bool fp )** `[virtual]`

**void QPrinter::setMinMax ( int minPage, int maxPage )** `[virtual]`

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 outside the range specified by min and max pages.

See also: minPage(), maxPage(), setFromTo() and setup().

**void QPrinter::setNumCopies ( int numCopies )** `[virtual]`

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 )** `[virtual]`

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 QString & fileName )** `[virtual]`

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 )** `[virtual]`

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 )** `[virtual]`

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.

This function is useful mostly for setting a default value that the user can override in the print dialog when you call setup().

**void QPrinter::setPageSize ( PageSize newPageSize )** `[virtual]`

Sets the printer page size to *newPageSize*.

The default page size is system-dependent.

This function is useful mostly for setting a default value that the user can override in the print dialog when you call setup().

See also: pageSize(), PageSize and setFullPage().

**void QPrinter::setPrintProgram ( const QString & printProg )** `[virtual]`

Sets the name of the program that should do the print job.

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 QString & name )** `[virtual]`

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().

**void QPrinter::setPrinterSelectionOption ( const QString & option )** `[virtual]`

Sets the printer to use *option* to select printer. *option* is null by default, meaning to be a little smart, but can be set to other values to use a specific printer selection option.

If the printer selection option is changed while the printer is active, the current print job may or may not be affected.

**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.

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().

### 166.3.1. URL

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

# 167. QProgressBar Class Reference

```
#include <qprogressbar.h>
```

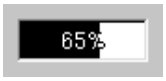
Inherits QFrame.

## 167.1. 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

## 167.2. 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().

**bool QProgressBar::centerIndicator () const**

Returns where the indicator string should be displayed if indicatorFollowsStyle() is TRUE.

See also: setCenterIndicator(), indicatorFollowsStyle(), setIndicatorFollowsStyle() and setIndicator().

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

Handles paint events for the progress bar. In WindowsStyle, colorGroup().text() and colorGroup::highlight() are used. In MotifStyle, colorGroup().base() is also used.

Reimplemented from QFrame.

**void QProgressBar::drawContentsMask ( QPainter * p )** `[virtual protected]`

Draws the progressbar contents mask using the painter *p*. Used only in transparent mode.

See also: QWidget::setAutoMask();.

Reimplemented from QFrame.

**bool QProgressBar::indicatorFollowsStyle () const**

Returns whether the display of the indicator string should follow the GUI style or not.

See also: setIndicatorFollowsStyle(), setCenterIndicator(), centerIndicator() and setIndicator().

**QSize QProgressBar::minimumSizeHint () const** `[virtual]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QWidget.

**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'.

**void QProgressBar::setCenterIndicator ( bool on )**

If set to TRUE (the default) the progress bar always shows the indicator text at the center of the progress bar, regardless of the GUI style currently set. If set to FALSE the progress bar always shows the indicator text outside the progress bar, regardless of the GUI style currently set.

Calling this function always sets indicatorFollowsStyle() to FALSE.

See also: centerIndicator(), indicatorFollowsStyle(), setIndicatorFollowsStyle() and setIndicator().

**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::setIndicatorFollowsStyle ( bool on )**

When set to TRUE (the default) the positioning of the indicator string follows the GUI style. When set to FALSE the indicator posion is decided by the value of indicatorFollowsStyle().

See also: indicatorFollowsStyle(), centerIndicator(), setCenterIndicator() and setIndicator().

**void QProgressBar::setProgress ( int progress )** `[virtual 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 )** `[virtual 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.

**QSizePolicy QProgressBar::sizePolicy () const** `[virtual]`

Specifies that this widget can use more, but is able to survive on less, horizontal space; and is fixed vertically.

Reimplemented from QWidget.

**void QProgressBar::styleChange ( QStyle & old )** `[virtual protected]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QWidget.

**int QProgressBar::totalSteps () const**

Returns the total number of steps.

See also: setTotalSteps().

### 167.2.1. URL

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

# 168. QProgressDialog Class Reference

```
#include <qprogressdialog.h>
```

Inherits QSemiModal.

## 168.1. Detailed Description

Provides feedback on the progress of a slow operation.

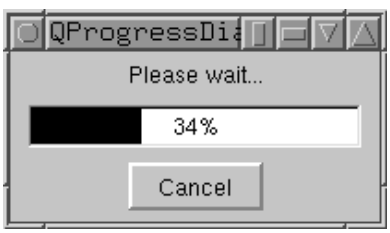
A progress dialog is used to give the user an indication of how long an operation is going to take to perform, and to reassure them that the application has not frozen.

QProgressDialog contains a label, a progress bar and a cancellation button.

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, "progress", TRUE );
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

### 168.2. 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 QString & labelText, const QString & 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().

**void QProgressDialog::closeEvent ( QCloseEvent * e )** `[virtual protected]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QWidget.

**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 QString & 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 QString & 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.

Examples:

progress/progress.cpp

**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 ( QStyle & 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

### 168.2.1. URL

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

# 169. QPtrDict Class Reference

`#include <qptrdict.h>`

Inherits QGDict.

## 169.1. Detailed Description

The QPtrDict class is a template class that provides a dictionary based on `void*` keys.

QPtrDict is implemented as a template 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]" );  // describe 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.

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, QAsciiDict, QIntDict and Collection Classes

## 169.2. 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 traversing order.

See also: remove(), clear() and setAutoDelete().

### 169.2.1. URL

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

# 170. QPtrDictIterator Class Reference

```
#include <qptrdict.h>
```

Inherits QGDictIterator.

## 170.1. Detailed Description

The QPtrDictIterator class provides an iterator for QPtrDict collections.

QPtrDictIterator is implemented as a template 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]" );  // describe 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.

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

## 170.2. 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.

### 170.2.1. URL

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

# 171. QPushButton Class Reference

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

Inherits QButton.

## 171.1. Detailed Description

The QPushButton widget provides a push button with a text or pixmap label.

The push button (command button in Windows parlance) is perhaps the most central widget: Push it to command the computer to perform some action.

Being so central, the widget has grown to accomodate a great many variations in the past decade, and by now the "comand button" chapter of the Microsoft style guide shows about ten different states of Windows push buttons, and the text implies that there are dozens more when all the combinations of features are taken into consideration.

The most important modes or states are, sorted roughly by importance:

- Available or not ("grayed out", disabled).
- Standard push button or toggling push button.
- On or off (only for toggling push buttons).
- Default or normal. The default button in a dialog can generally be "clicked" using the Enter or Return key.
- Auto-repeat or not.
- Pressed down or not.

In Qt, the QButton class provides most of the modes and other API, and QPushButton provides GUI logic. See QButton for more information about the API.

See also: QRadioButton, QToolButton, GUI Design Handbook: Push Button and Microsoft Style Guide

Examples:

xform/xform.cpp
layout/layout.cpp
drawdemo/drawdemo.cpp
popup/popup.cpp
qmag/qmag.cpp
rot13/rot13.cpp
scrollview/scrollview.cpp
customlayout/main.cpp

## 171.2. Member Function Documentation

**QPushButton::QPushButton ( QWidget * parent, 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 QString & text, QWidget * parent, 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.

**void QPushButton::focusOutEvent ( QFocusEvent * e )** `[virtual protected]`

Handles focus out 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 )** `[virtual]`

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 )** `[virtual]`

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.

**void QPushButton::setGeometry ( const QRect & r )** `[virtual]`

Reimplements QWidget::setGeometry() for internal purposes.

Examples:

popup/popup.cpp

qmag/qmag.cpp

**void QPushButton::setGeometry ( int x, int y, int w, int h )** `[virtual]`

Reimplements QWidget::setGeometry() for internal purposes.

**void QPushButton::setIsMenuButton ( bool enable )** `[virtual]`

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 )** `[virtual 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

**QSize QPushButton::sizeHint () const** `[virtual]`

Returns a size which fits the contents of the push button.

Reimplemented from QWidget.

**QSizePolicy QPushButton::sizePolicy () const** `[virtual]`

Specifies that this widget may stretch horizontally, but is fixed vertically.

Reimplemented from QWidget.

**void QPushButton::toggle ()** `[slot]`

Toggles the state of a toggle button.

See also: isOn(), setOn(), toggled() and isToggleButton().

### 171.2.1. URL

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

# 172. QQueue Class Reference

```
#include <qqueue.h>
```

Inherits QGList.

### 172.1. Detailed Description

The QQueue class is a template class that provides a queue.

QQueue is implemented as a template class. Define a template instance QQueue<X> to create a queue that operates on pointers to X, or X*.

A queue is a First In, First Out (FIFO) structure. Items are added to the back of the queue with enqueue() and retrieved from the front with dequeue().

See also: Collection Classes

### 172.2. 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 (shallow copy).

**QQueue::~QQueue ()**

Destroys the list. Items in the list are destroyed if autoDelete() is TRUE.

**QQueue::operator type* () const**

Returns a reference to the front item in the queue. The queue is not changed.

**type* QQueue::current () const**

Returns a reference to the front item in the queue. The queue is not changed.

**type* QQueue::dequeue ()**

Takes the front item from the queue. First in, first out.

**void QQueue::enqueue ( const type * d )**

Adds an item to the back of the queue. First in, first out.

**type* QQueue::head () const**

Returns a reference to the front 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. First in, first out. The item is deleted if autoDelete() is TRUE. Returns TRUE if there was an item to remove.

### 172.2.1. URL

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

# 173. QRadioButton Class Reference

```
#include <qradiobutton.h>
```

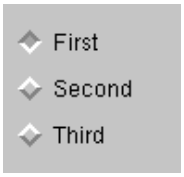
Inherits QButton.

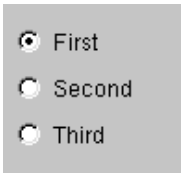
## 173.1. Detailed Description

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:

xform/xform.cpp
drawdemo/drawdemo.cpp

## 173.2. Member Function Documentation

**QRadioButton::QRadioButton ( QWidget * parent, 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 QString & text, QWidget * parent, 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.

**void QRadioButton::focusInEvent ( QFocusEvent * e )** `[virtual protected]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QWidget.

**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::setChecked ( bool check )** `[virtual]`

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:

xform/xform.cpp
drawdemo/drawdemo.cpp

**QSize QRadioButton::sizeHint () const** `[virtual]`

Returns a size which fits the contents of the radio button.

Reimplemented from QWidget.

**QSizePolicy QRadioButton::sizePolicy () const** `[virtual]`

Specifies that this widget may stretch horizontally, but is fixed vertically.

Reimplemented from QWidget.

### 173.2.1. URL

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

# 174. QRangeControl Class Reference

```
#include <qrangecontrol.h>
```

Inherited by QScrollBar, QSlider and QSpinBox.

### 174.1. Detailed Description

The QRangeControl class provides an integer value within a range.

It was originally designed for the QScrollBar widget, but it can also be practical for other purposes. QSlider, QSpinBox and QDial also inherit QRangeControl. Here are the five main concepts the class has:

- The current value. This is the bounded integer that QRangeControl maintains. value() returns this, and several functions including setValue() set it.
- The minimum. This is the lowest value value() can ever return. Returned by minValue(), set by setRange() or one of the constructors.
- The maximum. This is the highest value value() can ever return. Returned by maxValue(), set by setRange() or one of the constructors.
- The line step. This is the smaller of two natural steps QRangeControl provides, and typically corresponds to the user pressing an arrow key. Returned by lineStep(), set using setSteps(), and the addLine() and subtractLine() allow easy movement of the current value by lineStep().
- The page step. This is the larger of two natural steps QRangeControl provides, and typically corresponds to the user pressing one of the PageUp and PageDown keys. Returned by pageStep(), set using setSteps(), and the addPage() and substractPage() allow easy movement of the current value by pageStep().

Note that unity (1) may be viewed as a third step size. setValue() lets you set the current value to any integer in the allowed range, not just minValue()+n*lineStep() for integer values of n. Some widgets may allow the user to set any value at all, others may just provide multiples of lineStep()/pageStep(). The choice is up to you.

QRangeControl provides three virtual functions that are well-suited e.g. to updating the on-screen representation of range controls and emitting signals, namely valueChange(), rangeChange() and stepChange().

Finally, QRangeControl provides a function called bound() to let you force arbitrary integers to within the range of a range control.

We recommend that all widgets provide at least a signal called valueChanged(), and many widgets will want to provide addStep(), addPage(), substractStep() and substractPage() as slots.

## 174.2. 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 whose value can never be smaller than *minValue* or greater than *maxValue,* whose line step size is *lineStep* and page step size is *pageStep,* and whose value is initially *value.*

*value* is forced to be within the legal range.

### void QRangeControl::addLine ()

Equivalent to

```
setValue( value()+lineStep() )
```

plus a test for numerical overflow

See also: subtractLine().

### void QRangeControl::addPage ()

Equivalent to

```
setValue( value()+pageStep() )
```

plus a test for numerical overflow.

See also: subtractPage().

**int QRangeControl::bound ( int v ) const**

Forces *v* within the range from minValue() to maxValue() inclusive, and returns the result.

This function is provided so that you can easily force other numbers than value() into the allowed range. You do not need to call it in order to use QRangeControl itself.

See also: setValue(), value(), minValue() and maxValue().

**void QRangeControl::directSetValue ( int value )** `[protected]`

Sets the range control value directly without calling valueChange().

Forces the new value to be within the legal range.

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::positionFromValue ( int logical_val, int span ) const** `[protected]`

Converts *logical_val* to a pixel position. minValue() maps to 0, maxValue() maps to *span,* and other values are distributed evenly in between.

This function can handle the entire integer range without overflow.

**int QRangeControl::prevValue () const** `[protected]`

Returns the previous value of the range control. When the range control is initially created, this is the same as value().

Note that prevValue() can be outside the legal range if a call to setRange() causes the current value

to change. (For example if the range was 0-1000 and the current value 500, setRange( 0, 400 )

makes value() return 400 and prevValue() 500.)

See also: value() and setRange().

**void QRangeControl::rangeChange ()** `[virtual protected]`

This virtual function is called whenever the range control range changes. You can reimplement 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 one or both of the new min and max values are different

from the previous setting. Calls the virtual valueChange() function if the current value is adjusted

because or was is outside the new range.

If *maxValue* is smaller than *minValue, minValue* becomes the only legal value.

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: lineStep(), pageStep() and setRange().

**void QRangeControl::setValue ( int value )**

Sets the range control value to *value* and forces it to be within the legal range.

Calls the virtual valueChange() function if the new 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 reimplement 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() )
```

plus a test for numerical underflow

See also: addLine().

**void QRangeControl::subtractPage ()**

Equivalent to

```
setValue( value()-pageStep() )
```

plus a test for numerical underflow

See also: addPage().

**int QRangeControl::value () const**

Returns the current range control value. This is guaranteed to be within the range [ minValue() ... maxValue() ].

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 reimplement 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.

**int QRangeControl::valueFromPosition ( int pos, int span ) const** `[protected]`

Converts the pixel position *pos* to a value. 0 maps to minValue(), *span* maps to maxValue(), and other values are distributed evenly in between.

This function can handle the entire integer range without overflow.

### 174.2.1. URL

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

# 175. QRect Class Reference

```
#include <qrect.h>
```

## 175.1. 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 `int)`. The minimum value of QCOORD is QCOORD_MIN (-2147483648) and the maximum value is QCOORD_MAX (2147483647).

Note that the size (width and height) of a rectangle 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:

xform/xform.cpp
forever/forever.cpp
desktop/desktop.cpp
movies/main.cpp

## 175.2. 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().

**QPoint QRect::bottomRight () const**

Returns the bottom right position of the rectangle.

See also: moveBottomRight(), bottomLeft(), topLeft(), topRight(), bottom() and right().

**QPoint QRect::center () const**

Returns the center point of the rectangle.

See also: moveCenter(), topLeft(), topRight(), bottomLeft() and bottomRight().

**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:

xform/xform.cpp
desktop/desktop.cpp
movies/main.cpp

**QRect QRect::intersect ( const QRect & r ) const**

```
r.intersect(s)
```

is equivalent to

```
r&s
```

**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().

**int QRect::left () const**

Returns the left coordinate of the rectangle. Identical to x().

See also: x(), top(), right(), setLeft(), topLeft() and bottomLeft().

Examples:

xform/xform.cpp
desktop/desktop.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().

**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().

**QRect QRect::operator& ( const QRect & r ) const**

Returns the intersection of this rectangle and *r.*

Returns an empty rectangle if there is no intersection.

See also: operator&=(), operator|(), isEmpty(), intersects() and contains().

**QRect& QRect::operator&= ( const QRect & r )**

Intersects this rectangle with *r.*

**QRect QRect::operator| ( const QRect & r ) const**

Returns the bounding rectangle of this and *r.*

The bounding rectangle of a nonempty rectangle and an empty or invalid rectangle is defined to be the nonempty rectangle.

See also: operator|=(), operator&(), intersects() and contains().

**QRect& QRect::operator|= ( const QRect & r )**

Unites this rectangle with *r*.

**QCOORD & QRect::rBottom ()**

Returns the reference to the bottom coordinate of the rectangle.

See also: rLeft(), rTop() and rRight().

**QCOORD & QRect::rLeft ()**

Returns the reference to the left coordinate of the rectangle.

See also: rTop(), rRight() and rBottom().

**QCOORD & QRect::rRight ()**

Returns the reference to the right coordinate of the rectangle.

See also: rLeft(), rTop() and rBottom().

**QCOORD & QRect::rTop ()**

Returns the reference to the top coordinate of the rectangle.

See also: rLeft(), rRight() and rBottom().

**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:

xform/xform.cpp
desktop/desktop.cpp

**QPoint QRect::topLeft () const**

Returns the top left position of the rectangle.

See also: moveTopLeft(), topRight(), bottomLeft(), bottomRight(), left() and top().

**QPoint QRect::topRight () const**

Returns the top right position of the rectangle.

See also: moveTopRight(), topLeft(), bottomLeft(), bottomRight(), top() and right().

**QRect QRect::unite ( const QRect & r ) const**

```
r.unite(s)
```

is equivalent to

```
r|s
```

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:

xform/xform.cpp
desktop/desktop.cpp
movies/main.cpp

**int QRect::x () const**

Returns the left coordinate of the rectangle. Identical to left().

See also: left(), y() and setX().

Examples:

xform/xform.cpp
desktop/desktop.cpp
movies/main.cpp

**int QRect::y () const**

Returns the top coordinate of the rectangle. Identical to top().

See also: top(), x() and setY().

Examples:

xform/xform.cpp
desktop/desktop.cpp
movies/main.cpp

### 175.3. Related Functions

**bool operator== (const QRect & r1, const QRect & r2)**

Returns TRUE if *r1* and *r2* are equal, or FALSE if they are different.

**QDataStream & operator>> (QDataStream & s, QRect & r)**

Reads a QRect from the stream and returns a reference to the stream.

**QDataStream & operator<< (QDataStream & s, const QRect & r)**

Writes a QRect to the stream and returns a reference to the stream.

Serialization format: [left (Q_INT32), top (Q_INT32), right (Q_INT32), bottom (Q_INT32)].

**bool operator!= (const QRect & r1, const QRect & r2)**

Returns TRUE if *r1* and *r2* are different, or FALSE if they are equal.

### 175.3.1. URL

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

# 176. QRegExp Class Reference

```
#include <qregexp.h>
```

### 176.1. Detailed Description

The QRegExp class provides pattern matching using regular expressions or 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 (defined as any character where QChar::isSpace() returns TRUE. This includes ASCII characters 9 (TAB), 10 (LF), 11 (VT), 12(FF), 13 (CR), and 32 (Space)).
- *x12* matches the character 0x12 (18 decimal, 12 hexadecimal).
- *022* matches the character 022 (18 decimal, 22 octal).

In wildcard mode, it only knows four primitives:

- *c* matches the character 'c'
- *?* 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.

QRegExp supports Unicode both in the pattern strings and in the strings to be matched.

When writing regular expressions in C++ code, remember that the C++ preprocessor processes \ characters. So in order to match e.g. a "." character, you must write "\\." in C++ source, not "\.".

Bugs and limitations:

- Case insensitive matching is not supported for non-ASCII (non-8bit) characters. Any charcter with a non-zero QChar.row() is matched case sensitively even if the QRegExp is in case insensitive mode.

Examples:

qmag/qmag.cpp

## 176.2. Member Function Documentation

**QRegExp::QRegExp ()**

Constructs an empty regular expression.

**QRegExp::QRegExp ( const QString & 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 ( const QRegExp & r )**

Constructs a regular expression which is a copy of *r*.

See also: operator=(const and QRegExp&).

**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 QString & str, int index=0, int * len=0, bool indexIsStart = TRUE ) 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*.

If *indexIsStart* is TRUE (the default), the position *index* in the string will match the start-of-input

primitive (^) in the regexp, if present. Otherwise, position 0 in *str* will match.

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

**const QChar * QRegExp::matchstr ( uint * rxd, const QChar * str, uint strlength, const QChar * 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 QString & pattern )**

Sets the pattern string to *pattern* and returns a reference to this regexp. The case sensitivity or wildcard options do not change.

**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.

**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.

**QString 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().

### 176.2.1. URL

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

# 177. QRegion Class Reference

```
#include <qregion.h>
```

## 177.1. 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().

## 177.2. Member Function Documentation

### QRegion::QRegion ()

Constructs an null region.

See also: isNull().

**QRegion::QRegion ( const QBitmap & bm )**

Constructs a region from a bitmap.

The pixels in *bm* that are color1 will be part of the region as if each was a 1 by 1 rectangle.

**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 rectangle 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 ) const**

Equivalent to intersect(r).

**QRegion& QRegion::operator&= ( const QRegion & r )**

Equivalent to *this = *this & r.

**QRegion QRegion::operator+ ( const QRegion & r ) const**

Equivalent to unite(r) and operator|.

**QRegion& QRegion::operator+= ( const QRegion & r )**

Equivalent to *this = *this + r.

**QRegion QRegion::operator- ( const QRegion & r ) const**

Equivalent to subtract(r).

**QRegion& QRegion::operator-= ( const QRegion & r )**

Equivalent to *this = *this - r.

**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.

**QRegion QRegion::operator^ ( const QRegion & r ) const**

Equivalent to eor(r).

**QRegion& QRegion::operator^= ( const QRegion & r )**

Equivalent to *this = *this ^ r.

**QRegion QRegion::operator| ( const QRegion & r ) const**

Equivalent to unite(r) and operator+.

**QRegion& QRegion::operator|= ( const QRegion & r )**

Equivalent to *this = *this | r.

**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.*

### 177.3. Related Functions

**QDataStream & operator>> (QDataStream & s, QRegion & r)**

Reads a region from the stream and returns a reference to the stream.

**QDataStream & operator<< (QDataStream & s, const QRegion & r)**

Writes a region to the stream and returns a reference to the stream.

### 177.3.1. URL

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

# 178. QResizeEvent Class Reference

```
#include <qevent.h>
```

Inherits QEvent.

### 178.1. 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:

mainlyQt/editor.cpp
xform/xform.cpp
mainlyXt/editor.cpp
drawdemo/drawdemo.cpp
menu/menu.cpp
qmag/qmag.cpp
mainlyMotif/editor.cpp

### 178.2. 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().

### 178.2.1. URL

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

# 179. QScrollBar Class Reference

```
#include <qscrollbar.h>
```

Inherits QRangeControl and QWidget.

## 179.1. 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.)

Some GUI styles, for example the provided Windows and Motif styles, also use the pageStep() value to calculate the size of the sliding thumb (scroll indicator).

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

## 179.2. Member Function Documentation

**QScrollBar::QScrollBar ( Orientation orientation, QWidget * parent, 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 ( QWidget * parent, const char * name=0 )**

Constructs a vertical scroll bar.

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, 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. It is also used to calculate the size of the sliding thumb (scroll indicator).
- *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 currently 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 * )** `[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 )** `[virtual]`

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.

**void QScrollBar::setTracking ( bool enable )** `[virtual]`

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 releases 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.

**QSizePolicy QScrollBar::sizePolicy () const** `[virtual]`

Specifies that this widget can use more, but is able to survive on less, space in the orientation() direction; and is fixed in the other direction.

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::styleChange ( QStyle & old )** `[virtual protected]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QWidget.

**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.

**void QScrollBar::wheelEvent ( QWheelEvent * e )** `[virtual protected]`

Handles wheel events for the scroll bar.

Reimplemented from QWidget.

### 179.2.1. URL

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

# 180. QScrollView Class Reference

```
#include <qscrollview.h>
```

Inherits QFrame.

Inherited by QListBox, QListView and QTextView.

## 180.1. Detailed Description

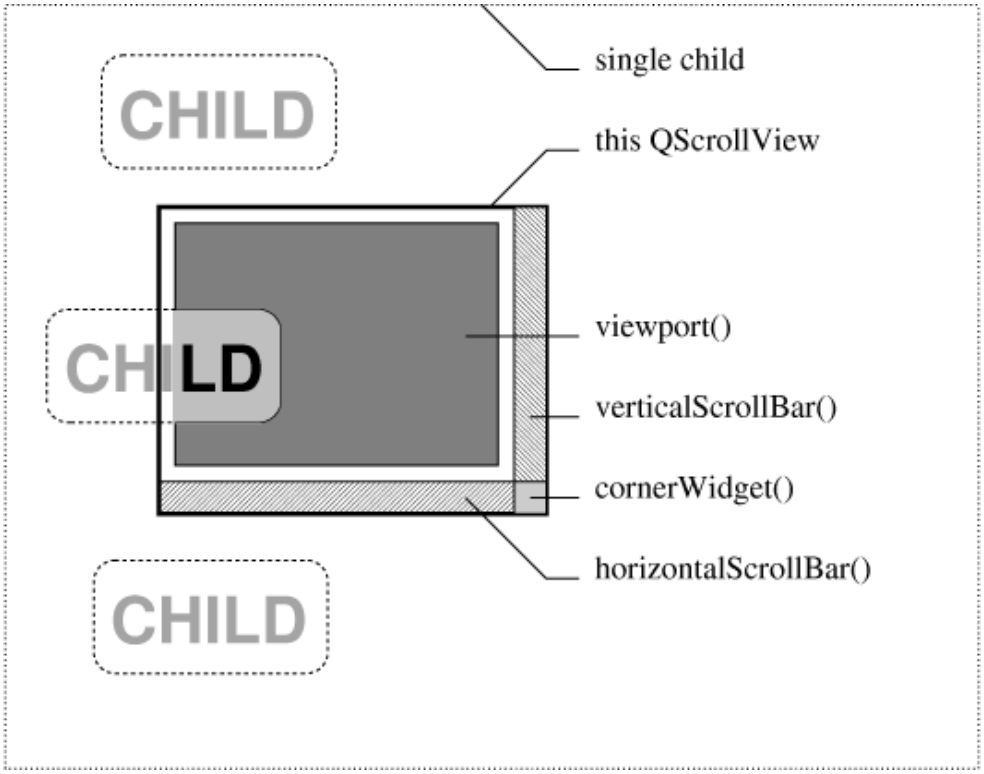
The QScrollView widget provides a scrolling area with on-demand scrollbars.

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() to position widgets on the view. For large numbers of such child widgets, consider using packChildWidgets() to improve performance.

To use QScrollView effectively, it is important to understand its widget structure in the three styles of usage: a single large child widget, a large panning area with some widgets, a large panning area with many widgets.

**One Big Widget**




The first, simplest usage of QScrollView depicated above is appropriate for scrolling areas which are *never* more than about 4000 pixels in either dimension (this is about the maximum reliable size on X11 servers). In this usage, you just make one large child in the QScrollView. The child should be a child of the viewport() of the scrollview, and be added with addChild():

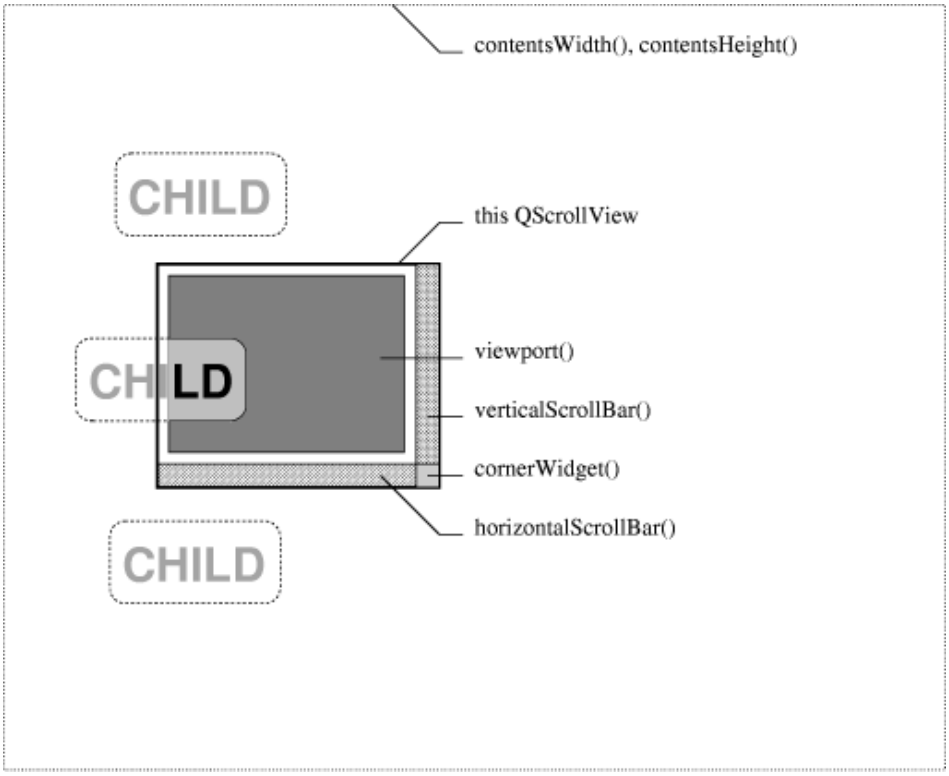
```
QScrollView* sv = new QScrollView(...);
QVBox* big_box = new QVBox(sv->viewport());
sv->addChild(big_box);
```

You may go on to add arbitrary child widgets to the single child in the scrollview, as you would with any widget:

```
QLabel* child1 = new QLabel("CHILD", big_box);
QLabel* child2 = new QLabel("CHILD", big_box);
QLabel* child3 = new QLabel("CHILD", big_box);
...
```

Here, the QScrollView has 4 children - the viewport(), the verticalScrollBar(), the horizontalScrollBar(), and a small cornerWidget(). The viewport() has 1 child, the big QVBox. The QVBox has the three labels as child widgets. When the view is scrolled, the QVBox is moved, and its children move with it as child widgets normally do.

**Very Big View, some Widgets**



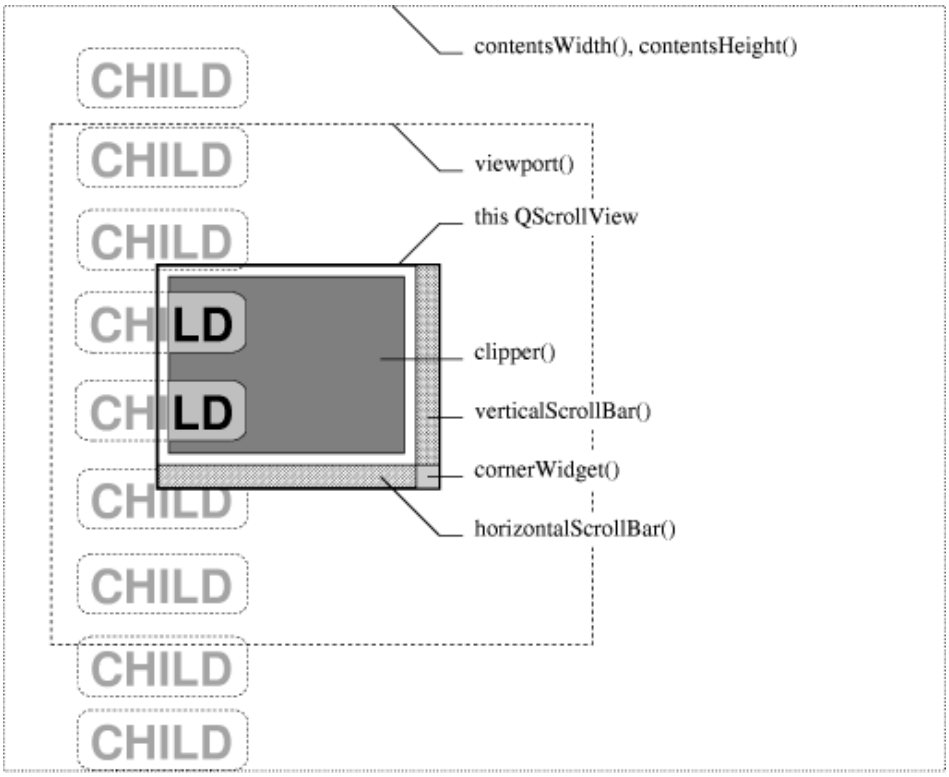

The second usage of QScrollView depicated above is appropriate when few, if any, widgets are on a very large scrolling area that is potentially larger than 4000 pixels in either dimension. In this usage, you call resizeContents() to set the size of the area, and override drawContents() to paint the contents. You may also add some widgets, by making them children of the viewport() and adding them with addChild() (this is the same as the process for the single large widget in the previous example):

```
QScrollView* sv = new QScrollView(...);
QLabel* child1 = new QLabel("CHILD", sv->viewport());
```

```
sv->addChild(child1);
QLabel* child2 = new QLabel("CHILD", sv->viewport());
sv->addChild(child2);
QLabel* child3 = new QLabel("CHILD", sv->viewport());
sv->addChild(child3);
```

Here, the QScrollView has the same 4 children - the viewport(), the verticalScrollBar(), the horizontalScrollBar(), and a small cornerWidget(). The viewport() has the three labels as child widgets. When the view is scrolled, the scrollview moves the child widgets individually.

**Very Big View, many Widgets**




The final usage of QScrollView depicated above is appropriate when many widgets are on a very large scrolling area that is potentially larger than 4000 pixels in either dimension. In this usage, you call resizeContents() to set the size of the area, and override drawContents() to paint the contents. You then call enableClipper(TRUE) and add widgets, again by making them children of the viewport() and adding them with addChild():

```
QScrollView* sv = new QScrollView(...);
sv->enableClipper(TRUE);
QLabel* child1 = new QLabel("CHILD", sv->viewport());
sv->addChild(child1);
QLabel* child2 = new QLabel("CHILD", sv->viewport());
sv->addChild(child2);
QLabel* child3 = new QLabel("CHILD", sv->viewport());
sv->addChild(child3);
```

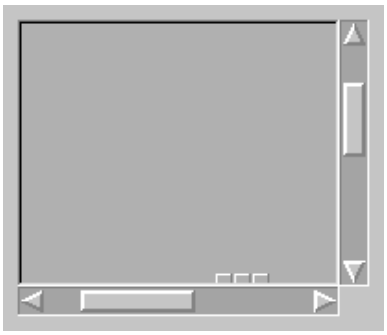
Here, the QScrollView has 4 children - the clipper() (*not* the viewport() this time), the verticalScrollBar(), the horizontalScrollBar(), and a small cornerWidget(). The clipper() has 1 child - the viewport(). The viewport() has the three labels as child widgets. When the view is scrolled, the viewport() is moved, and its children move with it as child widgets normally do.

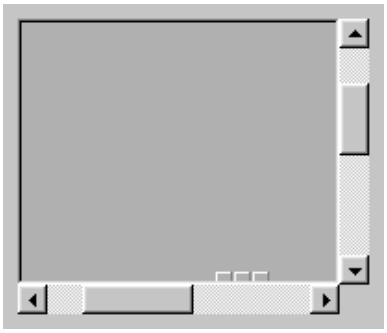
Normally you will use the first or third method if you want any child widgets in the view.

Note that the widget you see in the scrolled area is the viewport() widget, not the QScrollView itself. So, to turn mouse tracking on for example, use viewport()->setMouseTracking(TRUE).

To enable drag-and-drop, you would setAcceptDrops(TRUE) on the QScrollView (since drag-and-drop events propagate to the parent), but to work out what logical position in the view, you would need to map the drop co-ordinate from being relative to the QScrollView to being relative to the contents - use the function mapToContents() for this.

To handle mouse events on the scrolling area, subclass scrollview as you would subclass other widgets, but rather than overriding mousePressEvent(), override viewportMousePressEvent() instead (if you override mousePressEvent() you'll only get called when part of the QScrollView is clicked - and the only such part is the "corner" (if you don't set a cornerWidget()) and the frame, everything else being covered up by the viewport, clipper, or scrollbars.





Examples:

scrollview/scrollview.cpp

### 180.2. Member Type Documentation

#### QScrollView::ResizePolicy

This enum type is used to control QScrollView's reaction to resize events. There are three possible settings:

- `Default` - QScrollView selects one of the other settings automatically when it has to. At the time of writing, QScrollView changs to `Manual` if you resize the contents with resizeContents(), and to `AutoOne` if a child is added.
- `Manual` - the view stays the size set by resizeContents().
- `AutoOne` - if there is only only child widget, the view stays the size of that widget. Otherwise, the behaviour is undefined.

### QScrollView::ScrollBarMode

This enum type describes the various modes of QScrollView's scroll bars. The defined modes are:

- `Auto` - QScrollView shows a scrollbar when the content is too tall to fit and not else. This is the default.
- `AlwaysOff` - QScrollView never shows a scrollbar.
- `AlwaysOn` - QScrollView always shows a scrollbar.

(The modes for the horizontal and vertical scroll bars are independent.).

Examples:

scrollview/scrollview.cpp

## 180.3. 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` as well as `WNorthWestGravity` is propagated to the viewport() widget.

### QScrollView::~QScrollView ()

Destructs the QScrollView. Any children added with addChild() will be destructed.

### void QScrollView::addChild ( QWidget * child, int x=0, int y=0 ) [virtual]

Inserts *child* into the scrolled area positioned at *(x, y)*. The position defaults to (0,0). If the child is already in the view, it is just moved.

You may want to call packChildWidgets() after a large number of such additions is complete.

Examples:

scrollview/scrollview.cpp

**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 )**

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

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.

**QWidget* QScrollView::clipper () const**

Returns the clipper widget. Contents in the scrollview is ultimately clipped to be inside the clipper widget.

You should not need to access this.

See also: visibleWidth() and visibleHeight().

**void QScrollView::contentsDragEnterEvent ( QDragEnterEvent * )** `[virtual protected]`

This event handler is called whenever the QScrollView receives a dragEnterEvent() - the drag position is translated to be a point on the contents.

**void QScrollView::contentsDragLeaveEvent ( QDragLeaveEvent * )** `[virtual protected]`

This event handler is called whenever the QScrollView receives a dragLeaveEvent() - the drag position is translated to be a point on the contents.

**void QScrollView::contentsDragMoveEvent ( QDragMoveEvent * )** `[virtual protected]`

This event handler is called whenever the QScrollView receives a dragMoveEvent() - the drag position is translated to be a point on the contents.

**void QScrollView::contentsDropEvent ( QDropEvent * )** `[virtual protected]`

This event handler is called whenever the QScrollView receives a dropEvent() - the drop position is translated to be a point on the contents.

**int QScrollView::contentsHeight () const**

Returns the height of the contents area.

**void QScrollView::contentsMouseDoubleClickEvent ( QMouseEvent * )** `[virtual protected]`

This event handler is called whenever the QScrollView receives a mouseDoubleClickEvent() - the click position is translated to be a point on the contents.

Reimplemented in QListView.

**void QScrollView::contentsMouseMoveEvent ( QMouseEvent * )** `[virtual protected]`

This event handler is called whenever the QScrollView receives a mouseMoveEvent() - the mouse position is translated to be a point on the contents.

Reimplemented in QListView.

**void QScrollView::contentsMousePressEvent ( QMouseEvent * )** `[virtual protected]`

This event handler is called whenever the QScrollView receives a mousePressEvent() - the press position is translated to be a point on the contents.

Reimplemented in QListView.

**void QScrollView::contentsMouseReleaseEvent ( QMouseEvent * )** `[virtual protected]`

This event handler is called whenever the QScrollView receives a mouseReleaseEvent() - the release position is translated to be a point on the contents.

Reimplemented in QListView.

**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().

**QPoint QScrollView::contentsToViewport ( const QPoint & p )**

Returns the point *p* translated to a point on the viewport() widget.

**void QScrollView::contentsToViewport ( int x, int y, int & vx, int & vy )**

Translates a point *(x, y)* in the contents to a point *(vx, vy)* on the viewport() widget.

**void QScrollView::contentsWheelEvent ( QWheelEvent * )** `[virtual protected]`

This event handler is called whenever the QScrollView receives a wheelEvent() - the mouse position is translated to be a point on the contents.

**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.

Examples:

scrollview/scrollview.cpp

**void QScrollView::drawContents ( QPainter * p, int clipx, int clipy, int clipw, int cliph )** `[virtual protected]`

Reimplement this method if you are viewing a drawing area rather than a widget.

The function should draw the rectangle (*clipx, clipy, clipw, cliph* ) of the contents, using painter *p*.

The clip rectangle is in the scroll views's coordinates.

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;
  // Paint using the small coordinates...
  if ( x2 >= x1 && y2 >= y1 )
      p->fillRect(x1, y1, x2-x1+1, y2-y1+1, red);
}
```

The clip rectangle and translation of the painter *p* is already set appropriately.

**void QScrollView::drawContentsOffset ( QPainter * p, int offsetx, int offsety, int clipx, int clipy, int clipw, int cliph )** `[virtual protected]`

For backward compatibility only. It is easier to use drawContents(QPainter*,int,int,int,int).

The default implementation translates the painter appropriately and calls

drawContents(QPainter*,int,int,int,int).

Reimplemented in QListView and QTextView.

**void QScrollView::enableClipper ( bool y )**

When large numbers of child widgets are in a scrollview, especially if they are close together, the scrolling performance can suffer greatly. If you call enableClipper(TRUE), the scrollview will use an extra widget to group child widgets.

Note that you may only call enableClipper() prior to adding widgets.

For a full discussion, see the overview documentation of this class.

Examples:

scrollview/scrollview.cpp

**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().

Examples:

scrollview/scrollview.cpp

**QScrollBar* QScrollView::horizontalScrollBar () const**

Returns the component horizontal scrollbar. It is made available to allow accelerators, autoscrolling, etc., and to allow changing of arrow scroll rates: bar->setSteps( rate, bar->pageStep() ).

It should not be otherwise manipulated.

This function never returns 0.

**int QScrollView::leftMargin () const** `[protected]`

Returns the current left margin.

See also: setMargins().

**QSize QScrollView::minimumSizeHint () const** `[virtual]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QWidget.

**void QScrollView::moveChild ( QWidget * child, int x, int y )** `[virtual]`

Repositions *child* to *(x, y)*. This functions the same as addChild().

**void QScrollView::removeChild ( QObject * child )** `[virtual]`

Just an override to keep QObject::removeChild() accessible.

Reimplemented from QObject.

**void QScrollView::removeChild ( QWidget * child )**

Removes a child from the scrolled area. Note that this happens automatically if the child is deleted.

**void QScrollView::repaintContents ( int x, int y, int w, int h, bool erase=TRUE )**

Calls repaint() on rectangle defined by *x, y, w, h,* translated appropriately. If the rectangle in not visible, nothing is repainted.

See also: updateContents().

**void QScrollView::resize ( const QSize & s )**

An override - ensures scrollbars are correct size upon resize.

Examples:

qiconview/main.cpp
dirview/main.cpp

**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.

See also: setResizePolicy() and 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 )** `[virtual 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 separately

reparent 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().

Examples:

scrollview/scrollview.cpp

**void QScrollView::setHBarGeometry ( QScrollBar & hbar, int x, int y, int w, int h )** `[virtual protected]`

Called when the horizontal scrollbar geometry changes. This is provided as a protected function so that subclasses can do interesting things like providing extra buttons in some of the space normally used by the scrollbars.

The default implementation simply gives all the space to *hbar*.

See also: setVBarGeometry().

**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().

Examples:

scrollview/scrollview.cpp

**void QScrollView::setMargins ( int left, int top, int right, int bottom )** `[virtual 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*.

See also: resizePolicy() and ResizePolicy.

**void QScrollView::setVBarGeometry ( QScrollBar & vbar, int x, int y, int w, int h )** `[virtual protected]`

Called when the vertical scrollbar geometry changes. This is provided as a protected function so that subclasses can do interesting things like providing extra buttons in some of the space normally used by the scrollbars.

The default implementation simply gives all the space to *vbar*.

See also: setHBarGeometry().

**void QScrollView::setVScrollBarMode ( ScrollBarMode mode )** `[virtual]`

Sets the mode for the vertical scrollbar.

See also: vScrollBarMode() and setHScrollBarMode().

Examples:

scrollview/scrollview.cpp

**void QScrollView::show ()** `[virtual]`

An override - ensures scrollbars are correct size upon showing.

Examples:

qiconview/main.cpp

Reimplemented from QWidget.

**void QScrollView::showChild ( QWidget * child, bool y=TRUE )**

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

Sets the visibility of *child*. Equivalent to QWidget::show() or QWidget::hide().

**QSizePolicy QScrollView::sizePolicy () const** `[virtual]`

Specifies that this widget can use additional space, and that it can survive on less than sizeHint().

Reimplemented from QWidget.

**int QScrollView::topMargin () const** `[protected]`

Returns the current top margin.

See also: setMargins().

**void QScrollView::updateContents ( int x, int y, int w, int h )**

Calls update() on rectangle defined by *x, y, w, h,* translated appropriately. If the rectangle in not visible, nothing is repainted.

See also: repaintContents().

**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().

Examples:

scrollview/scrollview.cpp

**QScrollBar* QScrollView::verticalScrollBar () const**

Returns the component vertical scrollbar. It is made available to allow accelerators, autoscrolling, etc., and to allow changing of arrow scroll rates: bar->setSteps( rate, bar->pageStep() ).

It should not be otherwise manipulated.

This function never returns 0.

**QWidget* QScrollView::viewport () const**

Returns the viewport widget of the scrollview. This is the widget containing the contents widget or which is the drawing area.

Examples:

scrollview/scrollview.cpp

**void QScrollView::viewportDragEnterEvent ( QDragEnterEvent * e )** `[virtual protected]`

To provide simple processing of events on the contents, this method receives all drag enter events sent to the viewport.

The default implementation translates the event and calls contentsDragEnterEvent().

See also: QWidget::dragEnterEvent().

**void QScrollView::viewportDragLeaveEvent ( QDragLeaveEvent * e )** `[virtual protected]`

To provide simple processing of events on the contents, this method receives all drag leave events sent to the viewport.

The default implementation calls contentsDragLeaveEvent().

See also: QWidget::dragLeaveEvent().

**void QScrollView::viewportDragMoveEvent ( QDragMoveEvent * e )** `[virtual protected]`

To provide simple processing of events on the contents, this method receives all drag move events sent to the viewport.

The default implementation translates the event and calls contentsDragMoveEvent().

See also: QWidget::dragMoveEvent().

**void QScrollView::viewportDropEvent ( QDropEvent * e )** `[virtual protected]`

To provide simple processing of events on the contents, this method receives all drop events sent to the viewport.

The default implementation translates the event and calls contentsDropEvent().

See also: QWidget::dropEvent().

**void QScrollView::viewportMouseDoubleClickEvent ( QMouseEvent * e )** `[virtual protected]`

To provide simple processing of events on the contents, this method receives all mouse double click events sent to the viewport.

The default implementation translates the event and calls contentsMouseDoubleClickEvent().

See also: QWidget::mouseDoubleClickEvent().

Reimplemented in QListBox.

**void QScrollView::viewportMouseMoveEvent ( QMouseEvent * e ) [virtual protected]**

To provide simple processing of events on the contents, this method receives all mouse move events sent to the viewport.

The default implementation translates the event and calls contentsMouseMoveEvent().

See also: QWidget::mouseMoveEvent().

Reimplemented in QTextBrowser, QListBox and QTextView.

**void QScrollView::viewportMousePressEvent ( QMouseEvent * e ) [virtual protected]**

To provide simple processing of events on the contents, this method receives all mouse press events sent to the viewport.

The default implementation translates the event and calls contentsMousePressEvent().

See also: contentsMousePressEvent() and QWidget::mousePressEvent().

Reimplemented in QTextView, QListBox and QTextBrowser.

**void QScrollView::viewportMouseReleaseEvent ( QMouseEvent * e ) [virtual protected]**

To provide simple processing of events on the contents, this method receives all mouse release events sent to the viewport.

The default implementation translates the event and calls contentsMouseReleaseEvent().

See also: QWidget::mouseReleaseEvent().

Reimplemented in QTextView, QListBox and QTextBrowser.

**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).

Reimplemented in QListBox.

**void QScrollView::viewportResizeEvent ( QResizeEvent * )** `[virtual protected]`

To provide simple processing of events on the contents, this method receives all resize events sent to the viewport.

See also: QWidget::resizeEvent().

Reimplemented in QTextView.

**QSize QScrollView::viewportSize ( int x, int y ) const**

Returns the viewport size for size (*x, y*).

The viewport size depends on *x,y* (the size of the contents), the size of this widget, the modes of the horizontal and vertical scroll bars.

This function permits widgets that can trade vertical and horizontal space for each other to control scroll bar appearance better. For example, a word processor or web browser can control the width of the right margin accurately, whether there needs to be a vertical scroll bar or not.

**QPoint QScrollView::viewportToContents ( const QPoint & vp )**

Returns the point on the viewport *vp* translated to a point in the contents.

**void QScrollView::viewportToContents ( int vx, int vy, int & x, int & y )**

Translates a point (*vx, vy*) on the viewport() widget to a point (*x, y*) in the contents.

**void QScrollView::viewportWheelEvent ( QWheelEvent * e )** `[virtual protected]`

To provide simple processing of events on the contents, this method receives all wheel events sent to the viewport.

The default implementation translates the event and calls contentsWheelEvent().

See also: QWidget::wheelEvent().

**int QScrollView::visibleHeight () const**

Returns the vertical amount of the content that is visible.

**int QScrollView::visibleWidth () const**

Returns the horizontal amount of the content that is visible.

**void QScrollView::wheelEvent ( QWheelEvent * e )** `[virtual protected]`

An override - pass wheel events to the vertical scrollbar.

Reimplemented from QWidget.

### 180.3.1. URL

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

# 181. QSemiModal Class Reference

`#include <qsemimodal.h>`

Inherits QWidget.

Inherited by QProgressDialog.

## 181.1. 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.

## 181.2. 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 behavior 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 )** `[virtual]`

Reimplements QWidget::setGeometry() for internal purposes.

**void QSemiModal::setGeometry ( int x, int y, int w, int h )** `[virtual]`

Reimplements QWidget::setGeometry() for internal purposes.

**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.

### 181.2.1. URL

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

# 182. QSessionManager Class Reference

```
#include <qsessionmanager.h>
```

Inherits QObject.

## 182.1. Detailed Description

The QSessionManager class provides access to the session manager.

The QSessionManager class provides access to the session manager during a session management action. In Qt, session management actions are handled in the two virtual functions QApplication::commitData() and QApplication::saveState(). Both functions provide a reference to a session manager object as argument, thus allowing the application to communicate with the session manager.

During a session management action, i.e. within one of the two mentioned functions, no user interaction is possible, *unless* the application got explicit permission from the session manager. An ordinary permission can be requested with allowsInteraction(). With allowsErrorInteraction(), applications can ask for a higher priority interaction permission in case an error occured.

Another important function is cancel(), which asks the session manager to cancel the shutdown process.

For sophisticated session managers as provided on Unix, QSessionManager offers further possibilites to finetune an application's session management behaviour: setRestartCommand(), setDiscardCommand(), setRestartHint(), setProperty(), requestPhase2(). Please see the respective function descriptions for further details.

## 182.2. Member Function Documentation

**bool QSessionManager::allowsErrorInteraction ()**

Like allowsInteraction() but tells the session manager in addition that an error occured. Session managers may treat error interaction requests with higher priority. That means it is more likely that an error interaction is granted. However, you are still not guaranteed that the session manager will follow your request.

See also: allowsInteraction(), release() and cancel().

**bool QSessionManager::allowsInteraction ()**

Asks the session manager for permission to interact with the user. Returns TRUE if the interaction was granted, FALSE otherwise.

The rationale behind this is to make it possible to synchronize user interaction during a shutdown.

Fancy session managers on multitasking operating systems may ask all applications simultaniously to commit their data, which results in a much faster shutdown.

After the interaction is is recommended (though not necessary) to release the session manager with a call to release(). This way, other applications may get the chance to interact with the user while your application is still busy saving data.

If the user decides to cancel the shutdown process during the interaction phase, inform the session manager with a call to cancel().

Here's an example usage of the mentioned functions that may occur in the QApplication::commitData() function of an application:

```
void MyApplication::commitData( QSessionManager& sm ) {
    if ( sm.allowsInteraction() ) {
        switch ( QMessageBox::warning( yourMainWindow, "Application Name",
                                       "Save changes to Document Foo?",
                                       tr("&Yes"),
                                       tr("&No"),
                                       tr("Cancel"),
                                       0, 2) ) {
        case 0: // yes
            sm.release();
            // save document here. If saving fails, call sm.cancel()
            break;
        case 1: // no
            break;
        default: // cancel
```

```
            sm.cancel();
            break;
        }
    } else {
        // we did not get permission to interact, do something reasonable instead.
    }
}
```

If an error occured within the application while saving its data, you may want to use

allowsErrorInteraction() instead.

See also: QApplication::commitData(), release() and cancel().

**void QSessionManager::cancel ()**

Tells the session manager to cancel the shutdown process. Applications should not call this function without asking the user first.

See also: allowsInteraction() and allowsErrorInteraction().

**QStringList QSessionManager::discardCommand () const**

Returns the currently set discard command.

See also: setDiscardCommand(), restartCommand() and setRestartCommand().

**bool QSessionManager::isPhase2 () const**

Returns whether the session manager is currently performing a second session management phase.

See also: requestPhase2().

**void QSessionManager::release ()**

Releases the session manager after an interaction phase.

See also: allowsInteraction() and allowsErrorInteraction().

**void QSessionManager::requestPhase2 ()**

Requests a second session management phase for the application. The application may then simply return from the QApplication::commitData() or QApplication::saveState() function. The respective function will be called again after the first session management phase has been finished, this time with isPhase2() returning TRUE.

The two phases are useful for applications like X11 window manager, that need to store

informations about other application's windows and therefore have to wait until these applications

finished their respective session management tasks.

See also: isPhase2().

**QStringList QSessionManager::restartCommand () const**

Returns the currently set restart command.

See also: setRestartCommand() and restartHint().

**QString QSessionManager::sessionId () const**

Returns the identifier of the current session.

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

previous session.

See also: QApplication::sessionId().

**void QSessionManager::setDiscardCommand ( const QStringList & )**

See also: discardCommand() and setRestartCommand().

**void QSessionManager::setProperty ( const QString & name, const QString & value )**

Low-level write access to the application's identification and state record kept in the session manager.

**void QSessionManager::setProperty ( const QString & name, const QStringList & value )**

Low-level write access to the application's identification and state record kept in the session manager.

**void QSessionManager::setRestartCommand ( const QStringList & command )**

If the session manager is capable of restoring sessions, it will execute *command* in order to restore the application. The command defaults to

```
appname -session  id
```

The `-session` option is mandatory, otherwise QApplication can not tell whether it has been restored or what the current session identifier is. See QApplication::isSessionRestored() and QApplication::sessionId() for details. If your application is very simply, it may be possible to store the entire application state in additional command line options. In general, this is a very bad idea, since command lines are often limited to a few hundred bytes. Instead, use temporary files or a database for this purpose. By marking the data with the unique sessionId(), you will be able to restore the application in a future session.

See also: restartCommand(), setDiscardCommand() and setRestartHint().

**void QSessionManager::setRestartHint ( RestartHint hint )**

Sets the application's restart hint to *hint*. The restart hint defines, under what circumstances an application should be restarted by the session manager in the next session.

The following values are possible:

- `RestartIfRunning` - This is the default hint. If the application still runs by the time the session is shut down, it shall be restarted.
- `RestartAnyway` - Restart the application in the next session, regardless whether it runs at the end of this session or not. This is in particulary useful for configuration utilities that just run during startup and quit themselves after finishing their task.
- `RestartImmediately` - Ensure that the application runs all the time. If it exits during the session for whatever reason, restart it immediately. This might be useful for desktop utilities that are supposed to run permanentely, for example a file manager.
- `RestartNever` - Do not restart this application under any circumstances.

The default hint is `RestartIfRunning`. Note that these flags are only hints, a session manager may or may not obey them.

Preferrably, the restart hint shall be defined in QApplication::saveState() since most session managers perform a checkpoint immediately after an application's startup.

See also: restartHint().

### 182.2.1. URL

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

# 183. QShared Class Reference

```
#include <qshared.h>
```

Inherited by QGLayoutIterator, QMapPrivateBase, QStringData and QValueListPrivate.

### 183.1. 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

#### 183.1.1. URL

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

# 184. QShowEvent Class Reference

```
#include <qevent.h>
```

Inherits QEvent.

### 184.1. Detailed Description

The event sent when a widget is shown.

There are two kind of show events: spontaneous show events by the window system and internal show events. Spontaneous show events are sent just after the window system shows the window, including after a top-level window has been shown (un-iconified) by the user. Internal show events are delivered just before the widget becomes visible.

See also: QHideEvent.

### 184.2. Member Function Documentation

**QShowEvent::QShowEvent ( bool spontaneous )**

Constructs a QShowEvent. *spontaneous* is TRUE if the event originated outside the application - ie. the user revealed the window via the window manager controls.

**bool QShowEvent::spontaneous () const**

Returns TRUE if the event originated outside the application - ie. the user revealed the window via the window manager controls.

#### 184.2.1. URL

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

# 185. QSignal Class Reference

```
#include <qsignal.h>
```

## 185.1. 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 );

}
```

### 185.2. 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 )** `[virtual]`

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().

Reimplemented from QObject.

### 185.2.1. URL

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

# 186. QSignalMapper Class Reference

`#include <qsignalmapper.h>`

Inherits QObject.

## 186.1. 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.

## 186.2. 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 QString & )** `[signal]`

This signal is emitted when map() is signalled from an object which has a string mapping set.

See also: setMapping(QString).

**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 QString & identifier )** `[virtual]`

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 )** `[virtual]`

Adds a mapping such that when map() is signalled from the given sender, the signal mapped(identifier) is emitted.

There may be at most one integer identifier for each object.

### 186.2.1. URL

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

# 187. QSimpleRichText Class Reference

```
#include <qsimplerichtext.h>
```

## 187.1. Detailed Description

A small displayable piece of rich text.

This class encapsulates simple richt text usage where a string is interpretted as richt text and can be drawn. This is in particular useful if you want to display some rich text in a custom widget.

Once created, the rich text object can be queried for its width(), height() and the actual width used (see widthUsed()). Most importantly, it can be drawn on any given QPainter with draw().

QSimpleRichText can also be used to implement hypertext or active text facilities by using anchor().

Once constructed from a string, the contents cannot be changed, only resized. If the contents should change, just throw the rich text object away and make a new one with the new contents.

For large documents, see QTextView or QTextBrowser.

## 187.2. Member Function Documentation

### QSimpleRichText::QSimpleRichText ( const QString & text, const QFont & fnt, const QString & context = QString::null, const QStyleSheet * s = 0 )

Constructs a QSimpleRichText from the rich text string *text* and the font *fnt.*

The font is used as basis for the text rendering. When using rich text rendering on a certain widget *w,* you would regularily specify the widget's font as shown in the following code example:

```
QSimpleRichText* t = new QSimpleRichText( contents, w->font() );
```

*context* is the optional context of the document. This becomes important if *text* contains relative references, for example within image tags. QSimpleRichText always uses the default mime source factory (see QMimeSourceFactory::defaultFactory() ) to resolve those references. The context will then be used to calculate the absolute path. See QMimeSourceFactory::makeAbsolute() for details.

*s* is an optional stylesheet. If it is 0, the default style sheet will be used (see QStyleSheet::defaultSheet() ).

**QSimpleRichText::~QSimpleRichText ()**

Destructs the document, freeing memory.

**QString QSimpleRichText::anchor ( QPainter * p, const QPoint & pos )**

Returns the anchor at the requested position. The QPainter is needed for font size calculations. An empty string is returned if no anchor is specified for this certain position.

**QString QSimpleRichText::context () const**

Returns the context of the rich text document. If no context has been specified in the constructor, a null string is returned.

**void QSimpleRichText::draw ( QPainter * p, int x, int y, const QRegion & clipRegion, const QColorGroup & cg, const QBrush * paper = 0 ) const**

Draws the formatted text with *p,* at position (*x,\a* y), clipped to *clipRegion.* Colors from the *cg* are used as needed, and if not 0, **paper* is used as the background brush.

Note that the display code is highly optimized to reduce flicker, so passing a brush for *paper* is

preferrable to simply clearing the area to be painted and then calling this without a brush.

This is a convenience function if there's no palette but just a cologroup available. If you have a

palette, pass this instead of *cg.*

**void QSimpleRichText::draw ( QPainter * p, int x, int y, const QRegion & clipRegion, const QPalette & pal, const QBrush * paper = 0 ) const**

Draws the formatted text with *p,* at position (*x,\a* y), clipped to *clipRegion.* Colors from the palette *pal* are used as needed, and if not 0, **paper* is used as the background brush.

Note that the display code is highly optimized to reduce flicker, so passing a brush for *paper* is

preferrable to simply clearing the area to be painted and then calling this without a brush.

**int QSimpleRichText::height () const**

Returns the height of the document, in pixels.

See also: setWidth().

**void QSimpleRichText::setWidth ( QPainter * p, int w )**

Sets the width of the document to *w* pixels, recalculating the layout as if it were to be drawn with *p*.

See also: height().

**int QSimpleRichText::width () const**

Returns the set width of the document, in pixels.

See also: widthUsed().

**int QSimpleRichText::widthUsed () const**

Returns the width in pixels that is actually used by the document. This can be smaller or wider than the set width.

It may be wider, for example, if the text contains images or non-breakable words that are already

wider than the available space. It's smaller when the document only consists of lines that do not fill

the width completely.

See also: width().

### 187.2.1. URL

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

# 188. QSize Class Reference

```
#include <qsize.h>
```

## 188.1. 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 `int)`. The minimum value of QCOORD is QCOORD_MIN (-2147483648) and the maximum value is QCOORD_MAX (2147483647).

See also: QPoint and QRect.

Examples:

xform/xform.cpp
progress/progress.cpp
qfd/qfd.cpp

## 188.2. 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:

qfd/qfd.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*= ( double 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/= ( double 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().

**void QSize::setWidth ( int w )**

Sets the width to *w*.

See also: width() and setHeight().

**void QSize::transpose ()**

Swaps the values of width and height.

**int QSize::width () const**

Returns the width.

See also: height().

Examples:

qfd/qfd.cpp

### 188.3. Related Functions

**QSize operator* (const QSize & s, int c)**

Multiplies *s* by *c* and returns the result.

**QSize operator* (const QSize & s, double c)**

Multiplies *s* by *c* and returns the result.

**QSize operator* (int c, const QSize & s)**

Multiplies *s* by *c* and returns the result.

**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 & s, int c)**

Divides *s* by *c* and returns the result.

**bool operator== (const QSize & s1, const QSize & s2)**

Returns TRUE if *s1* and *s2* are equal, or FALSE if they are different.

**QSize operator/ (const QSize & s, double 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 (Q_INT32), height (Q_INT32)].

**QDataStream & operator>> (QDataStream & s, QSize & sz)**

Reads the size from the stream and returns a reference to the stream.

**QSize operator- (const QSize & s1, const QSize & s2)**

Returns *s2* subtracted from *s1;* each component is subtracted separately.

**QSize operator* (double c, const QSize & s)**

Multiplies *s* by *c* and returns the result.

**QSize operator+ (const QSize & s1, const QSize & s2)**

Returns the sum of *s1* and *s2;* each component is added separately.

### 188.3.1. URL

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

# 189. QSizeGrip Class Reference

`#include <qsizegrip.h>`

Inherits QWidget.

## 189.1. Detailed Description

The QSizeGrip class provides corner-grip for resizeing a top level window.

This widget works like the standard Windows resize handle. In the X11 version this resize handle generally works differently than the one provided by the system; we hope to reduce this difference in the future.

Put this widget anywhere in a tree and the user can use it to resize the top-level window. Generally this should be in the lower right-hand corner. Note that QStatusBar already uses this widget, so if you have a status bar (eg. you are using QMainWindow), then you don't need to use this widget explicitly.

[Image Omitted]

[Image Omitted]

See also: QStatusBar.

Examples:

scrollview/scrollview.cpp

## 189.2. Member Function Documentation

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

Construct a resize corner as a child widget of *parent.*

**void QSizeGrip::mouseMoveEvent ( QMouseEvent * e )** `[virtual protected]`

Resizes the top-level widget containing this widget.

Reimplemented from QWidget.

**void QSizeGrip::mousePressEvent ( QMouseEvent * e )** `[virtual protected]`

Primes the resize operation.

Reimplemented from QWidget.

**void QSizeGrip::paintEvent ( QPaintEvent * e )** `[virtual protected]`

Paints the resize grip - small diagonal textured lines in the lower righthand corner.

Reimplemented from QWidget.

**QSize QSizeGrip::sizeHint () const** `[virtual]`

Returns a small size.

Reimplemented from QWidget.

**QSizePolicy QSizeGrip::sizePolicy () const** `[virtual]`

The size grip has a fixed height, but can grow horizontally.

Reimplemented from QWidget.

### 189.2.1. URL

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

# 190. QSizePolicy Class Reference

```
#include <qsizepolicy.h>
```

## 190.1. Detailed Description

A layout attribute describing horizontal and vertical resizing.

Widgets which override QWidget::sizePolicy() return a QSizePolicy describing the horizontal and vertical resizing policy best used when laying out the widget.

Only the constructor is of interest in most applications.

## 190.2. Member Type Documentation

### QSizePolicy::SizeType

The sizing types used when constructing a QSizePolicy are:

- `Fixed` - the sizeHint() is the only acceptable alternative, so never grow or shrink (eg. the vertical direction of a pushbutton)
- `Minimum` - the sizeHint() is minimal, and sufficient. The widget can be expanded, but there is no advantage to it being larger. (eg. the horizontal direction of a pushbutton)
- `Maximum` - the sizeHint() is a maximum, the widget can be shrunk any amount without detriment if other widgets need the space (eg. a separator line)
- `Preferred` - the sizeHint() is best, but the widget can be shrunk below that and still be useful. The widget can be expanded, but there is no advantage to it being larger than sizeHint() (the default QWidget policy)
- `MinimumExpanding` - the sizeHint() is a minimum, the widget can make use of extra space, so it should get as much space as possible. (not currently used by any standard Qt widgets)
- `Expanding` - the sizeHint() is a sensible size, but the widget can be shrunk below that and still be useful. The widget can make use of extra space, so it should get as much space as possible. (eg. the horizontal direction of a slider)

## 190.3. Member Function Documentation

### QSizePolicy::QSizePolicy ()

Default constructor, produces a minimally initialized QSizePolicy.

**QSizePolicy::QSizePolicy ( SizeType hor, SizeType ver, bool hfw = FALSE )**

This is the constructor normally used to return a value in the overridden sizeHint() function of a QWidget subclass.

It constructs a QSizePolicy with independent horizontal and vertical sizing types, *hor* and *ver* respectively. These sizing types affect how the widget is treated by the *link* QLayout layout engine\endlink.

If *hfw* is TRUE, the preferred height of the widget is dependent on the width of the widget (for example, a QLabel with automatic word-breaking).

**bool QSizePolicy::hasHeightForWidth () const**

Returns TRUE if the widget's preferred height depends on its width.

**QSizePolicy::SizeType QSizePolicy::horData() const**

Returns the horizontal component of the size policy.

**bool QSizePolicy::mayGrowHorizontally () const**

Returns TRUE if the widget can sensibly be wider than its sizeHint().

**bool QSizePolicy::mayGrowVertically () const**

Returns TRUE if the widget can sensibly be taller than its sizeHint().

**bool QSizePolicy::mayShrinkHorizontally () const**

Returns TRUE if the widget can sensibly be narrower than its sizeHint().

**bool QSizePolicy::mayShrinkVertically () const**

Returns TRUE if the widget can sensibly be lower than its sizeHint().

**void QSizePolicy::setHeightForWidth ( bool b )**

Sets the hasHeightForWidth() flag to *b*.

**void QSizePolicy::setHorData ( SizeType d )**

Sets the horizontal component of the size policy to *d*.

**void QSizePolicy::setVerData ( SizeType d )**

Sets the vertical component of the size policy to *d*.

**QSizePolicy::SizeType QSizePolicy::verData() const**

Returns the vertical component of the size policy.

### 190.3.1. URL

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

# 191. QSlider Class Reference

```
#include <qslider.h>
```

Inherits QRangeControl and QWidget.

### 191.1. Detailed Description

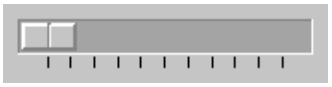
The QSlider widget provides a vertical or horizontal slider.

The slider is the classic widget for controlling a bounded value. It lets the user move a slider along a horizontal or vertical groove, and translates the slider's position into an integer value in the legal range.

QSlider inherits QRangeControl, which provides the "integer" side of the slider. setRange() and value() are likely to be used by practically all slider users; see the QRangeControl documentation for information about the many other functions that class provides.

The main functions offered by the slider itself are tickmark and orientation control; you can use setTickmarks() to indicate where you want the tickmarks to be, setTickInterval() to indicate how many of them you want, and setOrientation() to indicate whether the slider is to e horizontal or vertical.

A slider has a default focusPolicy() of *TabFocus,* and a suitable keyboard interface.



See also: QScrollBar, QSpinBox and GUI Design Handbook: Slider

Examples:

xform/xform.cpp

## 191.2. Member Type Documentation

### QSlider::TickSetting

This enum specifies where the tick marks are to be drawn, relative to the slider's groove and the handle the user moves. The possible values are `NoMarks` (no tickmarks are drawn), `Above, Below, Left, Right` and `Both.`

`NoMarks` means to not draw any tickmarks; `Both` means to draw tickmarks on both sides of the groove. `Above` and `Below` mean to draw tickmarks above and below the (horizontal) slider. `Left` and `Right` mean to draw tickmarks to the left and right of the (vertical) slider.

## 191.3. Member Function Documentation

### QSlider::QSlider ( Orientation orientation, QWidget * parent, 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 ( QWidget * parent, const char * name=0 )

Constructs a vertical slider.

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, 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, const QColorGroup & g, 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.*

Setting the colorgroup is useful to reuse the code to draw a mask if the slider supports transparceny.

See also: setAutoMask() and updateMask().

**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.*

Do not reimplement this function, it's mainly there for compatibility reasons. It simply calls

drawTicks( QPainter *p, const QColorGroup& g, int d, int w, int i ) const

which is the right function to overload.

See also: colorGroup(), drawTicks(, QPainter, *p, const, QColorGroup&, g, int, d, int, w, int, i, ),

const and paintSlider.

**void QSlider::drawWinGroove ( QPainter * p, QCOORD c )** `[protected]`

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

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::focusOutEvent ( QFocusEvent * )** `[virtual protected]`

Handles focus out events for the slider.

Reimplemented from QWidget.

**void QSlider::keyPressEvent ( QKeyEvent * e )** `[virtual protected]`

Handles key press events for the slider.

Reimplemented from QWidget.

**QSize QSlider::minimumSizeHint () const** `[virtual]`

Reimplemented for internal reasons; the API is not affected.

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 * )** `[virtual protected]`

Handles paint events for the slider.

Reimplemented from QWidget.

**void QSlider::paintSlider ( QPainter * p, const QColorGroup & g, const QRect & r )** `[virtual protected]`

Paints the slider button using painter *p* with size, a colorgroup and position given by *r*. Reimplement this function to change the look of the slider button.

Setting the colorgroup is useful to reuse the code to draw a mask if the slider supports transparceny.

See also: setAutoMask() and updateMask().

**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 )** `[virtual]`

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.

**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().

**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().

**void QSlider::setTracking ( bool enable )** `[virtual]`

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().

**void QSlider::setValue ( int value )** `[virtual slot]`

Makes QRangeControl::setValue() available as a slot.

Examples:

xform/xform.cpp

**QSize QSlider::sizeHint () const** `[virtual]`

Returns the recommended size of the slider.

Reimplemented from QWidget.

**QSizePolicy QSlider::sizePolicy () const** `[virtual]`

Specifies that this widget can use more, but is able to survive on less, space in the orientation() direction; and is fixed in the other direction.

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::updateMask ()** `[virtual protected]`

Reimplementation of QWidget::updateMask(). Draws the mask of the slider when transparency is required.

See also: QWidget::setAutoMask().

Reimplemented from QWidget.

**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.

**void QSlider::wheelEvent ( QWheelEvent * e )** `[virtual protected]`

Handles wheel events for the slider.

Reimplemented from QWidget.

### 191.3.1. URL

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

# 192. QSocketNotifier Class Reference

`#include <qsocketnotifier.h>`

Inherits QObject.

## 192.1. 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 occured (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.

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.)

For write notifiers, after the activated() signal has been received and you have sent the data to be written on the socket, immediately disable the notifier. When you have more data to be written, enable it again to get a new activated() signal. The exception is if the socket data writing operation (send() or equivalent) fails with a "Would block" error, meaning that some buffer is full and you must wait before sending more data. In this case, you do not need to disable and re-enable the write notifier, it will fire again as soon as the system allows more data may be sent.

The behaviour of a write notifier that is left in enabled state after having emitting the first activated() signal (and no "would block" error has occured) is undefined. Depending on the operating system, it may fire on every pass of the event loop, or not at all.

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.

See also: QSocket, QServerSocket and QSocketDevice.

## 192.2. 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 occured (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 `QEvent::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 )** `[virtual]`

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).

Write notifiers should normally be disabled immediately after the activated() signal has been emitted; see discussion of write notifiers in the class description above.

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().

### 192.2.1. URL

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

# 193. QSortedList Class Reference

```
#include <qsortedlist.h>
```

Inherits QList.

### 193.1. Detailed Description

The QSortedList class allows sorting of elements which implement operator<

If you want to sort a QList you have to reimplement the QGList::compareItems method. If the elements of your list support the operator< and operator== then you can use QSortedList instead. It does not force you to reimplement QGList::compareItems to sort the list.

See also: QList and Collection Classes

### 193.2. Member Function Documentation

**QSortedList::QSortedList ()**

Constructs an empty list.

**QSortedList::QSortedList ( const QSortedList<type> & list )**

Constructs a copy of *list.*

Each item in *list* is copied to this new list.

**QSortedList::~QSortedList ()**

Removes all items from the list and destroys the list.

All list iterators that access this list will be reset.

#### 193.2.1. URL

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

# 194. QSpacerItem Class Reference

`#include <qabstractlayout.h>`

Inherits QLayoutItem.

## 194.1. Detailed Description

A QLayoutItem that represents blank space.

For custom layouts.

See also: QLayout.

## 194.2. Member Function Documentation

**QSpacerItem::QSpacerItem ( int w, int h, QSizePolicy::SizeType hData=QSizePolicy::Minimum=QSizePolicy::Minimum, QSizePolicy::SizeType vData= QSizePolicy::Minimum= QSizePolicy::Minimum )**

Constructs a spacer item with preferred width *w,* preferred height *h,* horizontal size policy *hData* and vertical size policy *vData.*

The default values gives a gap that is able to stretch, if nothing else wants the space.

**void QSpacerItem::changeSize ( int w, int h, QSizePolicy::SizeType hData=QSizePolicy::Minimum, QSizePolicy::SizeType vData=QSizePolicy::Minimum )**

Changes this spacer item to have preferred width *w,* preferred height *h,* horizontal size policy *hData* and vertical size policy *vData.*

The default values gives a gap that is able to stretch, if nothing else wants the space.

**QRect QSpacerItem::geometry () const** `[virtual]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QLayoutItem.

**bool QSpacerItem::isEmpty () const** `[virtual]`

Returns TRUE, since a space item never contains widgets.

Reimplemented from QLayoutItem.

**QSize QSpacerItem::maximumSize () const** `[virtual]`

Returns the maximum size of this space item.

Reimplemented from QLayoutItem.

**QSize QSpacerItem::minimumSize () const** `[virtual]`

Returns the minimum size of this space item.

Reimplemented from QLayoutItem.

**void QSpacerItem::setGeometry ( const QRect & r )** `[virtual]`

This function stores *r*, so it can be returned by geometry().

Reimplemented from QLayoutItem.

**QSize QSpacerItem::sizeHint () const** `[virtual]`

Returns the preferred size of this space item.

Reimplemented from QLayoutItem.

**QSpacerItem * QSpacerItem::spacerItem ()** `[virtual]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QLayoutItem.

### 194.2.1. URL

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

# 195. QSpinBox Class Reference

```
#include <qspinbox.h>
```

Inherits QFrame and QRangeControl.

## 195.1. 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 an integer 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 arbitrary 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

### 195.2. 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 resources.

**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.

**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::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().

**QString 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::setEnabled ( bool on )** `[virtual slot]`

Reimplemented for internal reasons; the API is not affected.

**void QSpinBox::setPrefix ( const QString & 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 QString & text )** `[virtual]`

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().

**void QSpinBox::setSuffix ( const QString & 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().

**void QSpinBox::setValidator ( const QValidator * v )** `[virtual]`

Sets the validator to *v*. The validator controls what keyboard input is accepted when the user is editing in the value field. The default is to use a suitable QIntValidator.

Use setValidator(0) to turn off input validation (entered input will still be clamped to the range of the spinbox).

**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 )** `[virtual]`

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.

Reimplemented from QWidget.

**QSizePolicy QSpinBox::sizePolicy () const** `[virtual]`

Specifies that this widget may stretch horizontally, but is fixed vertically.

Reimplemented from QWidget.

**QString QSpinBox::specialValueText () const**

Returns the currently special-value text, or a null string 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 ( QStyle & old )** `[virtual protected]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QWidget.

**QString QSpinBox::suffix () const** `[virtual]`

Returns the currently set suffix, or 0 if no suffix is currently set.

See also: setSuffix(), setPrefix() and suffix().

**QString 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().

**const QValidator * QSpinBox::validator () const**

Returns the validator which constrains editing for this spin box if there is any, or else 0.

See also: setValidator() and QValidator.

**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 QString & 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().

**void QSpinBox::wheelEvent ( QWheelEvent * e )** `[virtual protected]`

Handles wheel events for the spinbox.

Reimplemented from QWidget.

**bool QSpinBox::wrapping () const**

Returns the current setWrapping() value.

### 195.2.1. URL

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

# 196. QSplitter Class Reference

```
#include <qsplitter.h>
```

Inherits QFrame.

## 196.1. 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. Any number of widgets may be controlled.

To show a QListBox, a QListView and a QMultiLineEdit side by side:

```
QSplitter *split = new QSplitter( parent );
QListBox *lb = new QListBox( split );
QListView *lv = new QListView( split );
QMultiLineEdit *ed = 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, all widgets can be as large or as small as the user wishes, down to minimumSizeHint(). 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.

The initial distribution of size between the widgets is determined by the initial size of each widget. You can also use setSizes() to set the sizes of all the widgets. The function sizes() returns the sizes set by the user.

If you hide() a child, its space will be distributed among the other children. When you show() it again, it will be reinstated.

See also: QTabBar.

Examples:

mainlyQt/editor.cpp
splitter/splitter.cpp

## 196.2. Member Type Documentation

### QSplitter::ResizeMode

This enum type describes how QSplitter will resize each of its child widgets. The currently defined values are:

- `Stretch` - the widget will be resized when the splitter itself is resized.
- `KeepSize` - QSplitter will try to keep this widget's size unchanged.

## 196.3. Member Function Documentation

### QSplitter::QSplitter ( Orientation o, QWidget * parent=0, const char * name=0 )

Creates splitter with orientation *o*.

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

Creates a horizontal splitter.

### QSplitter::~QSplitter ()

Destructs the splitter.

### int QSplitter::adjustPos ( int p, int id ) `[protected]`

Returns the legal position closest to *p* of the splitter with id *id*.

See also: idAfter().

**void QSplitter::childEvent ( QChildEvent * c )** `[virtual protected]`

Tells the splitter that a child widget has been inserted/removed.

Reimplemented from QObject.

**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*.

See also: QStyle::drawSplitter.

**bool QSplitter::event ( QEvent * e )** `[virtual protected]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QObject.

**void QSplitter::getRange ( int id, int * min, int * max )** `[protected]`

Returns the valid range of the splitter with id *id* in *min* and *max*.

See also: idAfter().

**int QSplitter::idAfter ( QWidget * w ) const** `[protected]`

Returns the id of the splitter to the right of or below the widget *w*, or 0 if there is no such splitter. (ie. it is either not in this QSplitter, or it is at the end).

**QSize QSplitter::minimumSizeHint () const** `[virtual]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QWidget.

**void QSplitter::moveSplitter ( QCOORD p, int id )** `[protected]`

Moves the left/top edge of the splitter handle with id *id* as close as possible to *p* which is the distance from the left (or top) edge of the widget.

See also: idAfter().

**void QSplitter::moveToFirst ( QWidget * w )**

Moves *w* to the leftmost/top position.

Examples:

splitter/splitter.cpp

**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().

**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 use of the splitter.

**void QSplitter::setOpaqueResize ( bool on = TRUE )** `[virtual]`

Sets opaque resize to *on*. Opaque resize is initially turned off.

See also: opaqueResize().

Examples:

mainlyQt/editor.cpp
splitter/splitter.cpp

**void QSplitter::setOrientation ( Orientation o )** `[virtual]`

Sets the orientation to *o*. By default the orientation is horizontal (the widgets are side by side).

See also: orientation().

**void QSplitter::setResizeMode ( QWidget * w, ResizeMode mode )** `[virtual]`

Sets resize mode of *w* to *mode.*

See also: ResizeMode.

Examples:

splitter/splitter.cpp

**void QSplitter::setRubberband ( int p )** `[virtual protected]`

Shows a rubber band at position *p.* If *p* is negative, the rubber band is removed.

**void QSplitter::setSizes ( QValueList<int> list )**

Sets the size parameters to the values given in *list.* Extra values in *list* are ignored.

If *list* contains to few values, the result is undefined but the program will still be well-behaved.

**QSize QSplitter::sizeHint () const** `[virtual]`

Returns a size based on the child widgets.

Reimplemented from QWidget.

**QSizePolicy QSplitter::sizePolicy () const** `[virtual]`

Says that this widget wants to grow in both height and width.

Reimplemented from QWidget.

**QValueList<int> QSplitter::sizes () const**

Returns a list of the size parameters of all the widgets in this splitter.

Giving the values to setSizes() will give a splitter with the same layout as this one.

**void QSplitter::styleChange ( QStyle & old )** `[virtual protected]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QWidget.

### 196.3.1. URL

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

# 197. QStack Class Reference

```
#include <qstack.h>
```

Inherits QGList.

### 197.1. Detailed Description

The QStack class is a template class that provides a stack.

QStack is implemented as a template class. Define a template instance QStack<X> to create a stack that operates on pointers to X, or X*.

A stack is a Last In, First Out (LIFO) structure. Items are added to the top of the stack with push() and retrieved from the top with pop().

See also: Collection Classes

### 197.2. 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.

### 197.2.1. URL

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

# 198. QStatusBar Class Reference

```
#include <qstatusbar.h>
```

Inherits QWidget.

## 198.1. 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` - 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.

## 198.2. 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 QString & message )** `[slot]`

Hide the normal status indications and display *message,* until clear() or another message() is called.

See also: clear().

**void QStatusBar::message ( const QString & 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().

### 198.2.1. URL

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

# 199. QStoredDrag Class Reference

`#include <qdragobject.h>`

Inherits QDragObject.

Inherited by QUriDrag.

## 199.1. 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.

## 199.2. 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 )** `[virtual]`

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.

### 199.2.1. URL

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

# 200. QStrIList Class Reference

`#include <qstrlist.h>`

Inherits QStrList.

### 200.1. 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 comparison. The inSort() function will

insert strings in a sorted order.

The QStrListIterator class is an iterator for QStrList.

### 200.2. 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.

#### 200.2.1. URL

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

# 201. QString Class Reference

```
#include <qstring.h>
```

Inherited by QConstString.

### 201.1. Detailed Description

The QString class provides an abstraction of Unicode text and the classic C zero-terminated char array (*char**).

QString uses implicit sharing, and so it is very efficient and easy to use.

In all QString methods that take *const char** parameters, the *const char** is interpreted as a classic C-style 0-terminated ASCII string. It is legal for the *const char** parameter to be 0. The results are undefined if the *const char** string is not 0-terminated. Functions that copy classic C strings into a QString will not copy the terminating 0-character. The QChar array of the QString (as returned by unicode()) is not terminated by a null.

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.

Note that if you find that you are mixing usage of QCString, QString, and QByteArray, this causes lots of unnecessary copying and might indicate that the true nature of the data you are dealing with is uncertain. If the data is NUL-terminates 8-bit data, use QCString; if it is unterminated (ie. contains NULs) 8-bit daya, use QByteArray; if it is text, use QString.

See also: Shared classes

Examples:

qdir/qdir.cpp
showimg/main.cpp
tabdialog/tabdialog.cpp
grapher/grapher.cpp
xform/xform.cpp
layout/layout.cpp

i18n/main.cpp
drawdemo/drawdemo.cpp
popup/popup.cpp
progress/progress.cpp
qmag/qmag.cpp
table/main.cpp
forever/forever.cpp
rot13/rot13.cpp
desktop/desktop.cpp
scrollview/scrollview.cpp
movies/main.cpp
picture/picture.cpp
hello/main.cpp
qbrowser/main.cpp

## 201.2. Member Function Documentation

**QString::QString ()**

Constructs a null string.

See also: isNull().

**QString::QString ( QChar ch )**

Constructs a string containing the one character *ch.*

**QString::QString ( const QChar * unicode, uint length )**

Constructs a string that is a deep copy of the first *length* QChar in the array *unicode.*

See also: QString::make().

**QString::QString ( const QByteArray & ba )**

Constructs a string that is a deep copy of *ba* interpreted as a classic C string.

**QString::QString ( const QString & s )**

Constructs an implicitly-shared copy of *s.*

**QString& QString::append ( char ch )**

Appends *ch* to the string and returns a reference to the result. Equivalent to operator+=().

**QString& QString::append ( const QString & str )**

Appends *str* to the string and returns a reference to the result. Equivalent to operator+=().

**QString QString::arg ( QChar a, int fieldwidth=0 ) const**

Returns a string equal to this one, but with the first occurrence of `%`*digit replaced by the character a.*

See arg(const QString&,int) for more details.

**QString QString::arg ( char a, int fieldwidth=0 ) const**

Returns a string equal to this one, but with the first occurrence of `%`*digit replaced by the character a.*

See arg(const QString&,int) for more details.

**QString QString::arg ( const QString & a, int fieldwidth=0 ) const**

Returns a string equal to this one, but with the first occurrence of `%`*digit replaced by the text a. This is particularly useful for translations, as it allows the order of the replacements to be controlled by the translator. For example:*

```
label.setText( tr("I have %1 to your %2").arg(mine).arg(yours) );
```

If there is no `%`*digit pattern, a warning message (qWarning()) is printed and the text as appended*

*with a space at the end of the string. This is error-recovery and should not be occur in correct code.*

*fieldwidth* is the minimum amount of space the text will be padded to. A positive value produces

right-aligned text, while a negative value produces left aligned text.

See also: QObject::tr().

**QString QString::arg ( double a, int fieldwidth=0, char fmt='g', int prec=-1 ) const**

Returns a string equal to this one, but with the first occurrence of `%`*digit replaced by the value a.*

See arg(const QString&,int) for more details.

**QString QString::arg ( int a, int fieldwidth=0, int base=10 ) const**

See QString::arg(long a, int fieldwidth, int base).

**QString QString::arg ( long a, int fieldwidth=0, int base=10 ) const**

Returns a string equal to this one, but with the first occurrence of *%digit replaced by the integer value in base base (defaults to decimal).*

The value is converted to *base* notation (default is decimal). The base must be a value from 2 to 36.

See arg(const QString&,int) for more details.

**QString QString::arg ( short a, int fieldwidth=0, int base=10 ) const**

See QString::arg(long a, int fieldwidth, int base).

**QString QString::arg ( uint a, int fieldwidth=0, int base=10 ) const**

See QString::arg(ulong a, int fieldwidth, int base).

**QString QString::arg ( ulong a, int fieldwidth=0, int base=10 ) const**

Returns a string equal to this one, but with the first occurrence of *%digit replaced by the unsigned integer value in base base (defaults to decimal).*

The value is converted to *base* notation (default is decimal). The base must be a value from 2 to 36.

See arg(const QString&,int) for more details.

**QString QString::arg ( ushort a, int fieldwidth=0, int base=10 ) const**

See QString::arg(ulong a, int fieldwidth, int base).

**QChar QString::at ( uint ) const**

Returns the character at *i,* or 0 if *i* is beyond the length of the string.

Note: If this QString is not const or const&, the non-const at() will be used instead, which will

expand the string if *i* is beyond the length of the string.

**QCharRef QString::at ( uint i )**

Returns a reference to the character at *i,* expanding the string with QChar::null if necessary. The resulting reference can then be assigned to, or otherwise used immediately, but becomes invalid once further modifications are made to the string.

**QChar::Direction QString::basicDirection()**

This function returns the basic directionality of the string (QChar::DirR for right to left and QChar::DirL for left to right). Useful to find the right alignment.

**int QString::compare ( const QString & s ) const**

Compares this string to *s,* returning an integer less than, equal to, or greater than zero if it is, respectively, lexically less than, equal to, or greater than *s.*

**int QString::compare ( const QString & s1, const QString & s2 )** `[static]`

Compare *s1* to *s2* returning an integer less than, equal to, or greater than zero if s1 is, respectively, lexically less than, equal to, or greater than s2.

**void QString::compose ()**

Note that this function is not supported in Qt 2.0, and is merely for experimental and illustrative purposes. It is mainly of interest to those experimenting with Arabic and other composition-rich texts.

Applies possible ligatures to a QString, useful when composition-rich text requires rendering with

glyph-poor fonts, but also makes compositions such as QChar(0x0041) ('A') and QChar(0x0308)

(Unicode accent diaresis) giving QChar(0x00c4) (German A Umlaut).

**QChar QString::constref ( uint i ) const**

Equivalent to at(i), this returns the QChar at *i* by value.

See also: ref().

**int QString::contains ( QChar 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 QString & 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().

**int QString::contains ( char c, bool cs=TRUE ) const**

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

**QString QString::copy () const**

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

Returns a deep copy of this string.

Doing this is redundant in Qt 2.x, since QString is implicitly shared, and so will automatically be deeply copied as necessary.

**void QString::fill ( QChar c, int len = -1 )**

Fills the string with *len* characters of value *c*.

If *len* is negative, the current string length is used.

**int QString::find ( QChar 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 QString & 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::find ( char c, int index=0, bool cs=TRUE ) const**

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

**int QString::findRev ( QChar c, int index=-1, bool cs=TRUE ) const**

Finds the first occurrence of the character *c,* starting at position *index* and searching backwards. If *index* is negative, the search starts at the end.

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. If *index* is negative, the search starts at the end of this string.

Returns the position of the next match (backwards), or -1 if *rx* was not found.

**int QString::findRev ( const QString & str, int index=-1, bool cs=TRUE ) const**

Finds the first occurrence of the string *str,* starting at position *index* and searching backwards. If *index* is negative, the search starts at the end.

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**

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

**QString QString::fromLatin1 ( const char * chars, int len=-1 )** `[static]`

Creates a QString from Latin1 text. This is the same as the QString(const char*) constructor, but you can make that constructor invisible if you compile with the define QT_NO_CAST_ASCII, in which case you can explicitly create a QString from Latin-1 text using this function.

**QString QString::fromLocal8Bit ( const char * local8Bit, int len=-1 )** `[static]`

Returns the unicode string decoded from the first *len* bytes of *local8Bit.* If *len* is -1 (the default), the length of *local8Bit* is used. If trailing partial characters are in *local8Bit,* they are ignored.

*local8Bit* is assumed to be encoded in a locale-specific format.

See QTextCodec for more diverse coding/decoding of Unicode strings.

**QString QString::fromUtf8 ( const char * utf8, int len=-1 )** `[static]`

Returns the unicode string decoded from the first *len* bytes of *utf8.* If *len* is -1 (the default), the length of *utf8* is used. If trailing partial characters are in *utf8,* they are ignored.

See QTextCodec for more diverse coding/decoding of Unicode strings.

**QString & QString::insert ( uint index, QChar 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 = "Ys";
s.insert( 1, 'e' );                         // s == "Yes"
s.insert( 3, '!');                          // s == "Yes!"
```

See also: remove() and replace().

Examples:

xform/xform.cpp

**QString & QString::insert ( uint index, const QChar * s, uint len )**

Insert *len* units of QChar data from *s* into the string before position *index.*

**QString & QString::insert ( uint index, const QString & 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"
```

**QString& QString::insert ( uint index, char c )**

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

**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() and length().

Examples:

qmag/qmag.cpp
hello/main.cpp

**bool QString::isNull () const**

Returns TRUE if the string is null. A null string is also an empty string.

Example:

```
QString a;          // a.unicode() == 0,  a.length() == 0

QString b == "";    // b.unicode() == "", b.length() == 0

a.isNull();         // TRUE, because a.unicode() == 0

a.isEmpty();        // TRUE, because a.length() == 0

b.isNull();         // FALSE, because b.unicode() != 0

b.isEmpty();        // TRUE, because b.length() == 0
```

See also: isEmpty() and length().

Examples:

qdir/qdir.cpp

**const char* QString::latin1 () const**

Returns a Latin-1 representation of the string. Note that the returned value is undefined if the string contains non-Latin-1 characters. If you want to convert strings into formats other than Unicode, see the QTextCodec classes.

This function is mainly useful for boot-strapping legacy code to use Unicode.

The result remains valid so long as one unmodified copy of the string exists.

See also: utf8() and local8Bit().

**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(), mid() and isEmpty().

**QString QString::leftJustify ( uint width, QChar fill=' ', bool truncate=FALSE ) const**

Returns a string of length *width* 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.

Null strings and empty strings have zero length.

See also: isNull() and isEmpty().

**QCString QString::local8Bit () const**

Returns the string encoded in a locale-specific format. On X11, this is the QTextCodec::codecForLocale(). On Windows, it is a system-defined encoding.

See QTextCodec for more diverse coding/decoding of Unicode strings.

See also: QString::fromLocal8Bit(), local8Bit() and utf8().

**QString QString::lower () const**

Returns a new string that is the string converted to lower case.

Example:

```
QString s("TeX");
QString t = s.lower();                      // t == "tex"
```

See also: upper().

**QString QString::mid ( uint index, uint len=0xffffffff ) 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::number ( double n, char f='g', int prec=6 )** `[static]`

A convenience factory function that returns a string representation of the number *n.*

Arguments:

- *f* is the format specifier: 'f', 'F', 'e', 'E', 'g', 'G' (same as sprintf()).
- *prec* is the precision.

See also: setNum().

**QString QString::number ( int n, int base=10 )** `[static]`

A convenience factory function that returns a string representation of the number *n.*

See also: setNum().

**QString QString::number ( long n, int base=10 )** `[static]`

A convenience factory function that returns a string representation of the number *n.*

See also: setNum().

**QString QString::number ( uint n, int base=10 )** `[static]`

A convenience factory function that returns a string representation of the number *n*.

See also: setNum().

**QString QString::number ( ulong n, int base=10 )** `[static]`

A convenience factory function that returns a string representation of the number *n*.

See also: setNum().

**bool QString::operator! () const**

Returns TRUE if it is a null string, otherwise FALSE. Thus you can write:

```
QString name = getName();
if ( !name )
  name = "Rodney";
```

Note that if you say:

```
QString name = getName();
if ( name )
  doSomethingWith(name);
```

Then this will call `operator const char*()`, which will do what you want, but rather inefficiently - you may wish to define the macro QT_NO_ASCII_CAST when writing code which you wish to strictly remain Unicode-clean.

When you want the above semantics, use `!isNull()` or even `!!`:

```
QString name = getName();
if ( !!name )
  doSomethingWith(name);
```

**QString & QString::operator+= ( QChar c )**

Appends *c* to the string and returns a reference to the string.

**QString & QString::operator+= ( char c )**

Appends *c* to the string and returns a reference to the string.

**QString& QString::operator+= ( const QString & str )**

Appends *str* to the string and returns a reference to the string.

**QString & QString::operator= ( const QCString & cs )**

Assigns a deep copy of *cs,* interpretted a classic C string, to this string and returns a reference to this 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= ( QChar c )**

Sets the string to contain just the single character *c*.

**QString& QString::operator= ( char c )**

Sets the string to contain just the single character *c*.

**QChar QString::operator[] ( int ) const**

Returns the character at *i,* or QChar::null if *i* is beyond the length of the string.

Note: If this QString is not const or const&, the non-const operator[] will be used instead, which will expand the string if *i* is beyond the length of the string.

**QCharRef QString::operator[] ( int )**

Returns an object that references the character at *i.* This reference can then be assigned to, or otherwise used immediately, but becomes invalid once further modifications are made to the string. The QCharRef internal class can be used much like a constant QChar, but if you assign to it, you change the original string (which enlarges and detaches itself). You will get compilation errors if you try to use the result as anything but a QChar.

**QString & QString::prepend ( const QString & s )**

Prepend *s* to the string. Equivalent to insert(0,s).

See also: insert().

**QString& QString::prepend ( char ch )**

Prepends *ch* to the string and returns a reference to the result.

See also: insert().

**QChar& QString::ref ( uint i )**

Returns the QChar at *i* by reference.

See also: constref().

**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 QString & str )**

Replaces every occurrence of *rx* in the string with *str*. Returns a reference to the string.

Examples:

```
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 ""
```

**QString & QString::replace ( uint index, uint len, const QChar * s, uint slen )**

Replaces *len* characters starting at position *index* by *slen* units ot QChar data from *s,* and returns a reference to the string.

See also: insert() and remove().

**QString & QString::replace ( uint index, uint len, const QString & 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().

Examples:

qmag/qmag.cpp

**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(), mid() and isEmpty().

**QString QString::rightJustify ( uint width, QChar fill=' ', bool truncate=FALSE ) const**

Returns a string of length *width* 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().

**void QString::setExpand ( uint index, QChar c )**

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

Sets the character at position *index* to *c* and expands the string if necessary, filling with spaces.

This method is redundant in Qt 2.x, because operator[] will expand the string as necessary, if you are assigning a value.

**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.

Examples:

table/main.cpp

**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, int base=10 )**

Sets the string to the printed value of *n* and returns a reference to the string.

**QString & QString::setNum ( long n, int base=10 )**

Sets the string to the printed value of *n* and returns a reference to the string.

The value is converted to *base* notation (default is decimal). The base must be a value from 2 to 36.

**QString & QString::setNum ( short n, int base=10 )**

Sets the string to the printed value of *n* and returns a reference to the string.

**QString & QString::setNum ( uint n, int base=10 )**

Sets the string to the printed unsigned value of *n* and returns a reference to the string.

**QString & QString::setNum ( ulong n, int base=10 )**

Sets the string to the printed unsigned value of *n* and returns a reference to the string.

The value is converted to *base* notation (default is decimal). The base must be a value from 2 to 36.

**QString & QString::setNum ( ushort n, int base=10 )**

Sets the string to the printed unsigned value of *n* and 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 character for which QChar::isSpace() returns TRUE. This includes ASCII

characters 9 (TAB), 10 (LF), 11 (VT), 12 (FF), 13 (CR), and 32 (Space).

```
QString s = "  lots\t of\nwhite    space ";
QString t = s.simplifyWhiteSpace();         // t == "lots of white space"
```

See also: stripWhiteSpace().

**QString & QString::sprintf ( const char * cformat, ... )**

Safely builds a formatted string from a format string and an arbitrary list of arguments. The format string supports all the escape sequences of printf() in the standard C library.

The %s escape sequence expects a UTF-8 encoded string. For typesafe string building, with full

Unicode support, you can use QTextOStream like this:

```
QString str;

QString s = ...;

int x = ...;

QTextOStream(&str) << s << " : " << x;
```

For translations, especially if the strings contains more than one escape sequence, you should

consider using the arg() function instead. This allows the order of the replacements to be controlled

by the translator, and has Unicode support.

See also: arg(const and QString&,int).

Examples:

xform/xform.cpp
layout/layout.cpp
drawdemo/drawdemo.cpp
popup/popup.cpp
progress/progress.cpp
qmag/qmag.cpp
forever/forever.cpp
scrollview/scrollview.cpp
movies/main.cpp
picture/picture.cpp

**QString QString::stripWhiteSpace () const**

Returns a new string that has white space removed from the start and the end.

White space means any character for which QChar::isSpace() returns TRUE. This includes ASCII

characters 9 (TAB), 10 (LF), 11 (VT), 12 (FF), 13 (CR), and 32 (Space).

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, int base=10 ) 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, int base=10 ) 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, int base=10 ) 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, int base=10 ) 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, int base=10 ) 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, int base=10 ) 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.

**void QString::truncate ( uint newLen )**

Truncates the string at position *newLen.* If newLen is less than the current length, this is equivalent to setLength( newLen ). Otherwise, nothing happens.

Example:

```
QString s = "truncate this string";
s.truncate( 5 );                            // s == "trunc"
```

In Qt 1.x, it was possible to "truncate" a string to a longer length. This is no longer possible.

See also: setLength().

**const QChar* QString::unicode () const**

Returns the Unicode representation of the string. The result remains valid until the string is modified.

**QString QString::upper () const**

Returns a new string that is the string converted to upper case.

Example:

```
QString s("TeX");
QString t = s.upper();                        // t == "TEX"
```

See also: lower().

**QCString QString::utf8 () const**

Returns the string encoded in UTF8 format.

See QTextCodec for more diverse coding/decoding of Unicode strings.

See also: QString::fromUtf8() and local8Bit().

**QString QString::visual ( int index = 0, int len = -1 )**

This function returns the QString ordered visually. Useful for painting the string or when transforming to a visually ordered encoding.

### 201.3. Related Functions

**QDataStream & operator>> (QDataStream & s, QString & str)**

Reads a string from the stream.

**QString operator+ (const char * s1, const QString & s2)**

Returns the concatenated string of s1 and s2.

**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`

**QString operator+ (const QString & s1, const char * s2)**

Returns the concatenated string of s1 and s2.

**QString operator+ (const QString & s1, const QString & s2)**

Returns the concatenated string of s1 and s2.

**QDataStream & operator<< (QDataStream & s, const QString & str)**

Writes a string to the stream.

Output format: [length (Q_UINT32) data...].

**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 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 *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 greater than or equal to *s2,* otherwise FALSE.

Equivalent to `strcmp(s1,s2) >= 0`

**QString operator+ (char c, const QString & s)**

Returns the concatenated string of c and s.

**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`

**QString operator+ (const QString & s, char c)**

Returns the concatenated string of s and c.

**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 less than *s2,* otherwise FALSE.

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 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 char * 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`

### 201.3.1. URL

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

# 202. QStringList Class Reference

```
#include <qstringlist.h>
```

Inherits QValueList<QString>.

## 202.1. Detailed Description

A list of strings.

This class is a list of QString objects. Like QValueList it is value based unlike QList. In contrast to QStrList it deals with real QString objects instead of character pointers. That makes QStringList the class of choice if you have to deal with unicode strings.

Like QString itself, a QStringList provides implicit share. Therefore string lists can be passed around as value-parameters both fast and safe.

Example:

```
QStringList list;
// three different ways of appending values:
list.append( "Torben");
list += "Warwick";
list << "Matthias" << "Arnt" << "Paul";
// sort the list, Arnt's now first
list.sort();
// print it out
for ( QStringList::Iterator it = list.begin(); it != list.end(); ++it ) {
    printf( "%s \n", it->latin1() );
}
```

## 202.2. Member Function Documentation

### QStringList::QStringList ()

Creates an empty list.

**QStringList::QStringList ( const QString & i )**

Constructs a string list consisting of the single string *i*. To make longers lists easily, use:

```
QString s1,s2,s3;
...
QStringList mylist = QStringList() << s1 << s2 << s3;
```

**QStringList::QStringList ( const QStringList & l )**

Creates a copy of the list. This function is very fast since QStringList is implicit shared. However, for the programmer this is the same as a deep copy. If this list or the original one or some other list referencing the same shared data is modified, then the modifying list makes a copy first.

**void QStringList::sort ()**

Sorts the list of strings in ascending order. The sorting algorithm used is HeapSort which operates

in O(n*logn).

### 202.2.1. URL

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

# 203. QStrList Class Reference

`#include <qstrlist.h>`

Inherits QList.

Inherited by QStrIList.

### 203.1. 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 comparison.

The inSort() function will insert strings in a sorted order.

The QStrListIterator class is an iterator for QStrList.

### 203.2. 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.

### 203.2.1. URL

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

# 204. QStyle Class Reference

`#include <qstyle.h>`

Inherits QObject.

Inherited by QCommonStyle.

## 204.1. Detailed Description

Encapsulates common Look and Feel of a GUI.

While it is not possible to fully enumerate the look of graphic elements and the feel of widgets in a GUI, a large number of elements are common to many widgets. The QStyle class allows the look of these elements to be modified across all widgets that use the QStyle methods. It also provides two feel options - Motif and Windows.

In previous versions of Qt, the look and feel option for widgets was specified by a single value - the GUIStyle. Starting with Qt 2.0, this notion has been expanded to allow the look to be specified by virtual drawing functions.

Derived classes may override some or all of the drawing functions to modify the look of all widgets which utilize those functions.

Examples:

themes/main.cpp

## 204.2. Member Function Documentation

**QStyle::~QStyle** () `[virtual]`

Destructs the style.

**QRect QStyle::buttonRect ( int x, int y, int w, int h )** `[virtual]`

Returns the rectangle available for contents in a push button. Usually this is the entire rectangle but it may also be smaller when you think about rounded buttons.

Reimplemented in QPlatinumStyle.

**QRect QStyle::comboButtonFocusRect ( int x, int y, int w, int h )** `[virtual]`

Returns the rectangle used to draw the the focus rectangle in a combo box.

Reimplemented in QPlatinumStyle, QCommonStyle and QWindowsStyle.

**QRect QStyle::comboButtonRect ( int x, int y, int w, int h )** `[virtual]`

Returns the rectangle available for contents in a combo box button. Usually this is the entire rectangle without the nifty menu indicator, but it may also be smaller when you think about rounded buttons.

Reimplemented in QWindowsStyle, QCommonStyle and QPlatinumStyle.

**int QStyle::defaultFrameWidth () const** `[virtual]`

The default frame width, usually 2.

Reimplemented in QCommonStyle and QCDEStyle.

**void QStyle::drawArrow ( QPainter * p, Qt::ArrowType type, bool down, int x, int y, int w, int h, const QColorGroup & g, bool enabled, const QBrush * fill = 0 )** `[virtual]`

Draws a button indicating direction.

**void QStyle::drawBevelButton ( QPainter *, int, int, int, int, const QColorGroup &, bool = FALSE, const QBrush * = 0 )** `[virtual]`

Draws a press-sensitive shape in the style of a bevel button.

Reimplemented in QMotifStyle, QWindowsStyle and QPlatinumStyle.

**void QStyle::drawButton ( QPainter *, int, int, int, int, const QColorGroup &, bool = FALSE, const QBrush * = 0 )** `[virtual]`

Draws a press-sensitive shape in the style of a full featured push button.

Reimplemented in QPlatinumStyle, QMotifStyle and QWindowsStyle.

**void QStyle::drawButtonMask ( QPainter *, int, int, int, int )** `[virtual]`

Draw the mask of a pushbutton. Useful if a rounded pushbuttons needs to be transparent because the style uses a fancy background pixmap.

**void QStyle::drawComboButton ( QPainter * p, int x, int y, int w, int h, const QColorGroup & g, bool sunken = FALSE, bool editable = FALSE, bool enabled = TRUE, const QBrush * fill = 0 )** `[virtual]`

Draws a press-sensitive shape in the style of a combo box or menu button.

Reimplemented in QPlatinumStyle, QWindowsStyle and QCommonStyle.

**void QStyle::drawComboButtonMask ( QPainter * p, int x, int y, int w, int h )** `[virtual]`

Draw the mask of a combo box button. Useful if a rounded buttons needs to be transparent because the style uses a fancy background pixmap.

Reimplemented in QCommonStyle.

**void QStyle::drawExclusiveIndicator ( QPainter *, int x, int y, int w, int h, const QColorGroup &, bool on, bool down = FALSE, bool enabled = TRUE )** `[virtual]`

Draws a mark indicating the state of an exclusive choice.

Reimplemented in QWindowsStyle, QMotifStyle, QPlatinumStyle and QCDEStyle.

**void QStyle::drawExclusiveIndicatorMask ( QPainter * p, int x, int y, int w, int h, bool )** `[virtual]`

Draws the mask of a mark indicating the state of an exclusive choice.

Reimplemented in QMotifStyle and QWindowsStyle.

**void QStyle::drawFocusRect ( QPainter * p, const QRect & r, const QColorGroup & g, const QColor * = 0, bool atBorder = FALSE )** `[virtual]`

Draws a mark indicating keyboard focus is on *r*. *atBorder* indicates whether the focus rectangle is at the border of an item (for example an item in a listbox). Certain styles (Motif style as the most prominent example) might have to shrink the rectangle a bit in that case to ensure that the focus rectangle is visible at all.

Reimplemented in QWindowsStyle and QMotifStyle.

**void QStyle::drawIndicator ( QPainter *, int, int, int, int, const QColorGroup &, int state, bool = FALSE, bool = TRUE )** `[virtual]`

Draws a mark indicating the state of a choice.

Reimplemented in QCDEStyle, QMotifStyle, QPlatinumStyle and QWindowsStyle.

**void QStyle::drawIndicatorMask ( QPainter * p, int x, int y, int w, int h, int )** `[virtual]`

Draws the mask of a mark indicating the state of a choice.

Reimplemented in QPlatinumStyle.

**void QStyle::drawItem ( QPainter * p, int x, int y, int w, int h, int flags, const QColorGroup & g, bool enabled, const QPixmap * pixmap, const QString & text, int len=-1, const QColor * penColor = 0 )** `[virtual]`

Draw text or a pixmap in an area.

**void QStyle::drawPanel ( QPainter * p, int x, int y, int w, int h, const QColorGroup & g, bool sunken=FALSE, int lineWidth = 1, const QBrush * fill = 0 )** `[virtual]`

Draws a panel to separate parts of the visual interface.

Reimplemented in QWindowsStyle.

**void QStyle::drawPopupPanel ( QPainter * p, int x, int y, int w, int h, const QColorGroup & cg, int lineWidth = 2, const QBrush * fill = 0 )** `[virtual]`

Draws a panel suitable as frame for popup windows.

Reimplemented in QWindowsStyle and QPlatinumStyle.

**void QStyle::drawPushButton ( QPushButton *, QPainter * )** `[virtual]`

Draws a pushbutton. This function will normally call drawButton() with arguments according to the current state of the pushbutton.

See also: drawPushButtonLabel() and QPushButton::drawButton().

Reimplemented in QPlatinumStyle, QMotifStyle and QWindowsStyle.

**void QStyle::drawPushButtonLabel ( QPushButton *, QPainter * )** `[virtual]`

Draws the label of a pushbutton. This function will normally call drawItem() with arguments according to the current state of the pushbutton.

See also: drawPushButton() and QPushButton::drawButtonLabel().

Reimplemented in QCommonStyle and QPlatinumStyle.

**void QStyle::drawRect ( QPainter * p, int x, int y, int w, int h, const QColor & c, int lineWidth = 1, const QBrush * fill = 0 )** `[virtual]`

Draws a simple rectangle to separate parts of the visual interface.

**void QStyle::drawRectStrong ( QPainter * p, int x, int y, int w, int h, const QColorGroup & g, bool sunken=FALSE, int lineWidth = 1, int midLineWidth = 0, const QBrush * fill = 0 )** `[virtual]`

Draws an emphasized rectangle to strongly separate parts of the visual interface.

**void QStyle::drawScrollBarControls ( QPainter *, const QScrollBar *, int sliderStart, uint controls, uint activeControl )** `[virtual]`

Draws the given scrollbar. Used internally by QScrollbar.

The controls are either ADD_LINE, SUB_LINE, ADD_PAGE, SUB_PAGE, FIRST, LAST,

SLIDER or NONE

Controls is a combination of these, activeControl is the control currently pressed down.

Reimplemented in QPlatinumStyle, QMotifStyle and QWindowsStyle.

**void QStyle::drawSeparator ( QPainter * p, int x1, int y1, int x2, int y2, const QColorGroup & g, bool sunken = TRUE, int lineWidth = 1, int midLineWidth = 0 )** `[virtual]`

Draws a line to separate parts of the visual interface.

**void QStyle::drawSlider ( QPainter * p, int x, int y, int w, int h, const QColorGroup & g, Orientation, bool tickAbove, bool tickBelow )** `[virtual]`

Draws a slider.

Reimplemented in QPlatinumStyle, QMotifStyle and QWindowsStyle.

**void QStyle::drawSliderGroove ( QPainter * p, int x, int y, int w, int h, const QColorGroup & g, QCOORD c, Orientation )** `[virtual]`

Draws a slider groove.

Reimplemented in QWindowsStyle, QPlatinumStyle and QMotifStyle.

**void QStyle::drawSliderGrooveMask ( QPainter * p, int x, int y, int w, int h, QCOORD c, Orientation )** `[virtual]`

Draws the mask of a slider groove.

Reimplemented in QCommonStyle.

**void QStyle::drawSliderMask ( QPainter * p, int x, int y, int w, int h, Orientation, bool, bool )** `[virtual]`

Draws the mask of a slider.

Reimplemented in QWindowsStyle and QCommonStyle.

**void QStyle::drawSplitter ( QPainter * p, int x, int y, int w, int h, const QColorGroup & g, Orientation orient )** `[virtual]`

Draws a splitter handle in the rectangle described by *x, y, w, h* using painter *p* and color group *g*. The orientation is *orient.*

See also: splitterWidth().

Reimplemented in QMotifStyle and QWindowsStyle.

**void QStyle::drawTab ( QPainter * p, const QTabBar * tb, QTab * t, bool selected )** `[virtual]`

TODO.

Reimplemented in QMotifStyle, QWindowsStyle and QCommonStyle.

**void QStyle::drawTabMask ( QPainter * p, const QTabBar * tb, QTab * t, bool selected )** `[virtual]`

TODO.

Reimplemented in QWindowsStyle, QMotifStyle and QCommonStyle.

**void QStyle::drawToolButton ( QPainter * p, int x, int y, int w, int h, const QColorGroup & g, bool sunken = FALSE, const QBrush * fill = 0 )** `[virtual]`

Draws a press-sensitive shape in the style of a toolbar button

The default implementation calls drawBevelButton()

See also: drawBevelButton().

**QSize QStyle::exclusiveIndicatorSize () const** `[virtual]`

Returns the size of the mark used to indicate exclusive choice.

Reimplemented in QPlatinumStyle, QMotifStyle and QWindowsStyle.

**int QStyle::extraPopupMenuItemWidth ( bool checkable, int maxpmw, QMenuItem * mi, const QFontMetrics & fm )** `[virtual]`

Returns the extra width of a menu item *mi,* that means all extra pixels besides the space the menu item text requires. *checkable* defines, whether the menu has a check column. *maxpmw* is the maximium width of all iconsets within a check column and *fm* defines the font metrics used to draw the label. This is particulary useful to calculate a suitable size for a submenu indicator or the column separation, including the tab column used to indicate item accelerators.

Reimplemented in QPlatinumStyle and QWindowsStyle.

**void QStyle::getButtonShift ( int & x, int & y )** `[virtual]`

Some GUI styles shift the contents of a button when the button is down. The default implementation returns 0 for both x and y.

Reimplemented in QWindowsStyle and QCommonStyle.

**GUIStyle QStyle::guiStyle () const**

Returns an indicator to the additional "feel" component of a style. Current supported values are Windows and Motif.

**QSize QStyle::indicatorSize () const** `[virtual]`

Returns the size of the mark used to indicate choice.

Reimplemented in QPlatinumStyle, QWindowsStyle and QMotifStyle.

**QRect QStyle::itemRect ( QPainter * p, int x, int y, int w, int h, int flags, bool enabled, const QPixmap * pixmap, const QString & text, int len=-1 )** `[virtual]`

Returns the appropriate area within a rectangle in which to draw text or a pixmap.

**int QStyle::maximumSliderDragDistance () const** `[virtual]`

Some feels require the scrollbar or other sliders to jump back to the original position when the mouse pointer is too far away while dragging.

This behavior can be customized with this function. The default is -1 (no jump back) while

Windows requires 20 (weird jump back).

Reimplemented in QPlatinumStyle, QCommonStyle and QWindowsStyle.

**void QStyle::polish ( QApplication * )** `[virtual]`

Late initialization of the QApplication object.

See also: unPolish.

Reimplemented in QMotifStyle.

**void QStyle::polish ( QPalette & )** `[virtual]`

The style may have certain requirements for color palettes. In this function it has the chance to change the palette according to these requirements.

See also: unPolish, QPalette and QApplication::setPalette().

**void QStyle::polish ( QWidget * )** `[virtual]`

Initializes the appearance of a widget.

This function is called for every widget, after it has been fully created just *before* it is shown the

very first time.

Reasonable actions in this function might be to set the

of the widget and the background pixmap, for example. Unreasonable use would be setting the

geometry!

The QWidget::inherits() function may provide enough information to allow class-specific

customizations. But be careful not to hard-code things too much, as new QStyle sub-classes will be

expected to work reasonably with all current *and future* widgets.

See also: unPolish(QWidget*).

**void QStyle::scrollBarMetrics ( const QScrollBar *, int &, int &, int &, int & )** `[virtual]`

Returns the metrics of the passed scrollbar: sliderMin, sliderMax, sliderLength and buttonDim.

Reimplemented in QPlatinumStyle, QWindowsStyle and QMotifStyle.

**int QStyle::sliderLength () const** `[virtual]`

The length of a slider.

Reimplemented in QPlatinumStyle, QMotifStyle and QWindowsStyle.

**int QStyle::splitterWidth () const** `[virtual]`

Returns the width of a splitter handle.

See also: drawSplitter().

Reimplemented in QMotifStyle and QWindowsStyle.

**void QStyle::tabbarMetrics ( const QTabBar * t, int & hframe, int & vframe, int & overlap )** `[virtual]`

TODO.

Reimplemented in QMotifStyle, QWindowsStyle and QCommonStyle.

**void QStyle::unPolish ( QApplication * )** `[virtual]`

Redo the application polish

See also: polish.

**void QStyle::unPolish ( QWidget * )** `[virtual]`

Undos the initializataion of a widget's appearance

This function is the counterpart to polish. Is is called for every polished widget when the style is dynamically changed. The former style has to un-polish its settings before the new style can polish them again.

See also: polish(QWidget*).

#### 204.2.1. URL

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

# 205. QStyleSheet Class Reference

`#include <qstylesheet.h>`

Inherits QObject.

## 205.1. Detailed Description

A collection of styles for rich text rendering and a generator of tags.

By creating QStyleSheetItem objects for a style sheet, you build a definition of a set of tags. This definition will be used by the internal rich text rendering system to parse and display text documents to which the style sheet applies. Rich text is normally visualized in a QTextView or a QTextBrowser. But also QLabel, QWhatsThis and QMessageBox support it for now, with others likely to follow. With QSimpleRichText it is possible to use the rich text renderer for custom widgets as well.

The default QStyleSheet object has the following style bindings:

- `<qt>...</qt>` - A Qt rich text document. It understands the following attributes
- `title` - the caption of the document. This attribute is easily accessible with QTextView::documentTitle()
- `type` - The type of the document. The default type is `page` . It indicates that the document is displayed in a page of its own. Another style is `detail.` It can be used to explain certain expressions more detailed in a few sentences. The QTextBrowser will then keep the current page and display the new document in a small popup similar to QWhatsThis. Note that links will not work in documents with `<qt type="detail" >...</qt>`
- `bgcolor` - The background color, for example `bgcolor="yellow"` or `bgcolor="#0000FF"`
- `background` - The background pixmap, for example `background="granit.xpm".` The pixmap name will be resolved by a QMimeSourceFactory().
- `text` - The default text color, for example `text="red"`
- `link` - The link color, for example `link="green"`

- <a>...</a> - An anchor or link. The reference target is defined in the href attribute of the tag as in \c<a href="target.qml">...</a>. You can also specify an additional anchor within the specified target document, for example <a href="target.qml#123">...</a>. If a is meant to be an anchor, the reference source is given in the name attribute.
- <font>...</font> - Customizes the font size and text color. The tag understands two attributes:
- color - the text color, for example color="red" or color="#FF0000".
- size - the logical size of the font. Logical sizes 1 to 7 are supported. The value may either be absolute, for example size=3, or relative. In the latter case, the sizes are simply added.
- ... - Emphasized. As default, this is the same as ... (Italic)
- ... - Strong. As default, this is the same as <bold>...</bold> (bold)
- <big>...</big> - A larger font size.
- <small>...</small> - A smaller font size.
- <code>...</code> - Indicates Code. As default, this is the same as <tt>...</tt> (typewriter)
- <pre>...</pre> - For larger junks of code. Whitespaces in the contents are preserved.
- <large>...</large> - Large font size.
- ... - style->setFontWeight( QFont::Bold); Bold font style.
- <h1>...</h1> - A top-level heading.
- <h2>...</h2> - A sub-level heading.
- <h3>...</h3> - A sub-sub-level heading.

- ... - A paragraph.
- ... - A centered paragraph.
- <blockquote>...</blockquote> - An indented paragraph, useful for quotes.
- <multicol cols=*n* >...</multicol> - *Multicol display with n columns*
- <twocolumn>...</twocolumn> - Two-column display.
- <ul>...</ul> - An un-ordered list. You can also pass a type argument to define the bullet style. The default is type=disc, other types are circle and square.
- <ol>...</ol> - An ordered list. You can also pass a type argument to define the enumeration label style. The default is type="1", other types are "a" and "A".
- <li>...</li> - A list item.
- <img/> - An image. The image name for the mime source factory is given in the source attribute, for example <img source="qt.xpm"/> The image tag also understands the attributes width and height that determine the size of the image. If the pixmap does not fit to the specified size, it will be scaled automatically ( by using QImage::smoothScale() ).
- <br/> - A line break
- <hr/> - A horizonal line

### 205.2. Member Function Documentation

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

Create a style sheet. Like any QObject, the created object will be deleted when its parent destructs (if the child still exists then).

By default, the style sheet has the tag definitions defined above.

**QStyleSheet::~QStyleSheet ()** [virtual]

Destroys the style sheet. All styles inserted into the style sheet will be deleted.

**QString QStyleSheet::convertFromPlainText ( const QString & plain )** `[static]`

Auxiliary function. Converts the plain text string *plain* to a rich text formatted string while preserving its look.

**QStyleSheet* QStyleSheet::defaultSheet ()** `[static]`

Returns the application-wide default style sheet.This style sheet is used by rich text rendering classes such as QSimpleRichText, QWhatsThis and also QMessageBox to define the rendering style and available tags within rich text documents. It serves also as initial style sheet for the more complex render widgets QTextView and QTextBrowser.

See also: setDefaultSheet().

Examples:

qbrowser/main.cpp

**void QStyleSheet::error ( const QString & msg ) const** `[virtual]`

This virtual function is called when an error occurs when processsing rich text. Reimplement if if you need to catch error messages.

Errors might occur if some rich text strings contain tags that are not understood by the stylesheet, if

some tags are nested wrongly or if tags are not closed properly.

*msg* is the error message.

**void QStyleSheet::insert ( QStyleSheetItem * style )**

For internal use only.

**QStyleSheetItem* QStyleSheet::item ( const QString & name )**

Returns the style with name *name* or 0 if there is no such style.

**const QStyleSheetItem* QStyleSheet::item ( const QString & name ) const**

Returns the style with name *name* or 0 if there is no such style (const version).

**bool QStyleSheet::mightBeRichText ( const QString & text )** `[static]`

Returns whether the string *text* is likely to be rich text formatted.

Note: The function uses a fast and therefore simple heuristic. It mainly checks whether there is something that looks like a tag before the first line break. While the result may be correct for most common cases, there is no guarantee.

**void QStyleSheet::scaleFont ( QFont & font, int logicalSize ) const** `[virtual]`

Scales the font *font* to the appropriate physical point size corresponding to the logical font size *logicalSize.*

When calling this function, *font* has a point size corresponding to the logical font size 3.

Typical logical font sizes range from 1 to 7, with 1 being the smallest.

See also: logicalFontSize(), logicalFontSizeStep() and QFont::setPointSize().

**void QStyleSheet::setDefaultSheet ( QStyleSheet * sheet )** `[static]`

Sets the application-wide default style sheet, deleting any style sheet previously set. The ownership is transferred.

See also: defaultSheet().

**QTextNode* QStyleSheet::tag ( const QString & name, const QMap<QString, QString> & attr, const QString & context, const QMimeSourceFactory & factory, bool emptyTag = FALSE ) const** `[virtual]`

Generates an internal object for tag named *name,* given the attributes *attr,* and using additional information provided by the mime source factory *factory* .

This function should not (yet) be used in application code.

### 205.2.1. URL

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

# 206. QStyleSheetItem Class Reference

`#include <qstylesheet.h>`

Inherits Qt.

## 206.1. Detailed Description

The QStyleSheetItem class encapsulates a text format.

A style consists of a name and a set of font, color, and other display properties. When used in a style sheet, items define the name of a rich text tag, and the display property changes associated with it.

Examples:

qbrowser/main.cpp

## 206.2. Member Type Documentation

### QStyleSheetItem::DisplayMode

This enum type defines the way adjacent elements are displayed. The possible values are:

- `DisplayBlock` - elements are displayed as a rectangular block. (eg. <P> ... </P>)
- `DisplayInline` - elements are displayed in a horizontally flowing sequence. (eg. <EM> ... </EM>)
- `DisplayListItem` - elements are displayed in a vertically sequence. (eg. <EM> ... </EM>)
- `DisplayNone` - elements are not displayed at all.

### QStyleSheetItem::ListStyle

This enum type defines how the items in a list are prefixed when displayed. The currently defined values are:

- `ListDisc` - a filled circle
- `ListCircle` - an unfilled circle
- `ListSquare` - a filled circle

- `ListDecimal` - an integer in base 10: *1, 2, 3, ...*
- `ListLowerAlpha` - a lowercase letter: *a, b, c, ...*
- `ListUpperAlpha` - an uppercase letter: *A, B, C, ...*

### 206.3. Member Function Documentation

**QStyleSheetItem::QStyleSheetItem ( QStyleSheet * parent, const QString & name )**

Constructs a new style named *name* for the stylesheet *parent.*

All properties in QStyleSheetItem are initially in the "do not change" state, except display mode,

which defaults to `DisplayInline.`

**QStyleSheetItem::~QStyleSheetItem ()**

Destroys the style. Note that QStyleSheetItem objects become owned by QStyleSheet when they are created.

**int QStyleSheetItem::alignment () const**

Returns the alignment of this style. Possible values are AlignLeft, AlignRight and AlignCenter.

See also: setAlignment().

**bool QStyleSheetItem::allowedInContext ( const QStyleSheetItem * s ) const**

Returns whether this style can be nested into an element of style *s* .

See also: contxts() and setContexts().

**QColor QStyleSheetItem::color () const**

Returns the text color of this style, or an invalid color if no color has been set yet.

See also: setColor().

**QString QStyleSheetItem::contexts () const**

Returns a space separated list of names of styles that may contain elements of this style. As default, contexs() returns an empty string, which indicates that this style can be nested everywhere.

See also: setContexts().

**bool QStyleSheetItem::definesFontItalic () const**

Returns whether the style defines a font shape. A style does not define any shape until setFontItalic() is called.

See also: setFontItalic() and fontItalic().

**bool QStyleSheetItem::definesFontUnderline () const**

Returns whether the style defines a setting for the underline property of the font. A style does not define this until setFontUnderline() is called.

See also: setFontUnderline() and fontUnderline().

**QStyleSheetItem::DisplayMode QStyleSheetItem::displayMode() const**

Returns the display mode of the style.

See also: setDisplayMode().

**QString QStyleSheetItem::fontFamily () const**

Returns the font family setting of the style. This is either a valid font family or QString::null if no family has been set.

See also: setFontFamily(), QFont::family() and QFont::setFamily().

**bool QStyleSheetItem::fontItalic () const**

Returns whether the styles sets an italic or upright font.

See also: setFontItalic() and definesFontItalic().

**int QStyleSheetItem::fontSize () const**

Returns the font size setting of the style. This is either a valid pointsize or QStyleSheetItem::Undefined.

See also: setFontSize(), QFont::pointSize() and QFont::setPointSize().

**bool QStyleSheetItem::fontUnderline () const**

Returns whether the styles sets an underlined font.

See also: setFontUnderline() and definesFontUnderline().

**int QStyleSheetItem::fontWeight () const**

Returns the font weight setting of the style. This is either a valid QFont::Weight or the value QStyleSheetItem::Undefined.

See also: setFontWeight and QFont.

**bool QStyleSheetItem::isAnchor () const**

Returns whether this style is an anchor.

See also: setAnchor().

**QStyleSheetItem::ListStyle QStyleSheetItem::listStyle() const**

Returns the list style of the style.

See also: setListStyle().

**int QStyleSheetItem::logicalFontSize () const**

Returns the logical font size setting of the style. This is either a valid size between 1 and 7 or QStyleSheetItem::Undefined.

See also: setLogicalFontSize(), setLogicalFontSizeStep(), QFont::pointSize() and

QFont::setPointSize().

**int QStyleSheetItem::logicalFontSizeStep () const**

Returns the logical font size step of this style.

The default is 0. Tags like `big` define `+1`, `small` defines `-1`

See also: setLogicalFontSizeStep().

**int QStyleSheetItem::margin ( Margin m ) const**

Returns the width of margin *m* in pixel.

The margin determinator *m* can be `MarginLeft`, `MarginRight`, `MarginTop`,

`MarginBottom`, `MarginAll`, `MarginVertical` or `MarginHorizontal`.

See also: setMargin().

**QString QStyleSheetItem::name () const**

Returns the name of style.

**int QStyleSheetItem::numberOfColumns () const**

Returns the number of columns for this style.

See also: setNumberOfColumns(), displayMode() and setDisplayMode().

**bool QStyleSheetItem::selfNesting () const**

Returns whether this style has self nesting enabled.

See also: setSelfNesting().

**void QStyleSheetItem::setAlignment ( int f )**

Sets the alignment. This only makes sense for styles with display mode DisplayBlock. Possible values are AlignLeft, AlignRight and AlignCenter.

See also: alignment() and displayMode().

**void QStyleSheetItem::setAnchor ( bool anc )**

Sets whether the style is an anchor (link). Elements in this style have connections to other documents or anchors.

See also: isAnchor().

**void QStyleSheetItem::setColor ( const QColor & c )**

Sets the text color of this style.

See also: color().

**void QStyleSheetItem::setContexts ( const QString & c )**

Sets a space separated list of names of styles that may contain elements of this style. If *c* is empty, the style can be nested everywhere.

See also: contexts().

Examples:

qbrowser/main.cpp

**void QStyleSheetItem::setDisplayMode ( DisplayMode m )**

Sets the display mode of the style to *m.*

See also: displayMode().

Examples:

qbrowser/main.cpp

**void QStyleSheetItem::setFontFamily ( const QString & fam )**

Sets the font family setting of the style.

See also: fontFamily(), QFont::family() and QFont::setFamily().

**void QStyleSheetItem::setFontItalic ( bool italic )**

Sets italic or upright shape for the style.

See also: fontItalic() and definesFontItalic().

**void QStyleSheetItem::setFontSize ( int s )**

Sets the font size setting of the style, in point measures.

See also: fontSize(), QFont::pointSize() and QFont::setPointSize().

**void QStyleSheetItem::setFontUnderline ( bool underline )**

Sets underline for the style.

See also: fontUnderline() and definesFontUnderline().

**void QStyleSheetItem::setFontWeight ( int w )**

Sets the font weight setting of the style. Valid values are those defined by QFont::Weight.

See also: QFont and fontWeight().

**void QStyleSheetItem::setListStyle ( ListStyle s )**

Sets the list style of the style.

This is used by nested elements which have a display mode of DisplayListItem.

See also: listStyle() and DisplayMode.

**void QStyleSheetItem::setLogicalFontSize ( int s )**

Sets the logical font size setting of the style tp *s*. Valid logical sizes are 1 to 7.

See also: logicalFontSize(), QFont::pointSize() and QFont::setPointSize().

**void QStyleSheetItem::setLogicalFontSizeStep ( int s )**

Sets the logical font size step of this style to *s*.

See also: logicalFontSizeStep().

**void QStyleSheetItem::setMargin ( Margin m, int v )**

Sets the width of margin *m* to *v* pixels.

The margin determinator *m* can be `MarginLeft, MarginRight, MarginTop, MarginBottom, MarginAll, MarginVertical` or `MarginHorizontal.` The value *v* must be >= 0.

See also: border().

**void QStyleSheetItem::setNumberOfColumns ( int ncols )**

Sets the number of columns for this style. Elements in the style are divided into columns.

This only makes sense if the style uses a block display mode.

See also: numberOfColumns().

**void QStyleSheetItem::setSelfNesting ( bool nesting )**

Sets the self nesting property for this style to *nesting*.

Usually, all styles are self nesting, i.e. they can legally be nested recursively. A paragraph, for example, might contain other paragraphs as subparagraphs.

You may nevertheles want to disable self nesting for common HTML tags such as <p> or <li>, if you want to parse and display rich text documents based on "dirty" HTML. However, we recommend fixing the documents instead to close all non-empty tags, if that is possible.

**void QStyleSheetItem::setWhiteSpaceMode ( WhiteSpaceMode m )**

Sets the white space mode to *m*. Possible values are

- `WhiteSpaceNormal` - white spaces in the document only serve as seperators. Multiple spaces or indentation therefore is ignored.
- `WhiteSpacePre` - white spaces are preserved. This is particulary useful to display programming code.

**QStyleSheet* QStyleSheetItem::styleSheet ()**

Returns the stylesheet this item is in.

**const QStyleSheet* QStyleSheetItem::styleSheet () const**

Returns the stylesheet this item is in.

### 206.3.1. URL

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

# 207. Qt Class Reference

```
#include <qnamespace.h>
```

Inherited by QBrush, QEvent, QListViewItem, QObject, QPainter, QPen, QPixmap,

QStyleSheetItem, QToolTip and QWhatsThis.

### 207.1. Detailed Description

The Qt class is a namespace for miscellaneous identifiers that need to be global-like.

Normally, you can ignore this class. QObject and a few other classes inherit it, so that all the

identifiers in the Qt namespace are visible to you without qualification.

However, occasionally you may need to say `Qt::black` instead just `black,` particularly in static

utility functions (such as many class factories).

Examples:

menu/menu.cpp

### 207.2. Member Type Documentation

<h3>Qt::Orientation</h3> This type is used to signify whether an object should be *Horizontal* or *Vertical* (for example in QScrollBar).

#### 207.2.1. URL

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

# 208. QTab Class Reference

```
#include <qtabbar.h>
```

### 208.1. Detailed Description

The structures in a QTabBar.

For custom QTabBar tab headings.

See also: QTabBar.

#### 208.1.1. URL

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

# 209. QTabBar Class Reference

`#include <qtabbar.h>`

Inherits QWidget.

## 209.1. 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.

## 209.2. Member Function Documentation

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

Creates a new, empty tab bar.

**QTabBar::~QTabBar ()**

Deletes the tab control and free the memory it used.

**int QTabBar::addTab ( QTab * newTab )** `[virtual]`

Adds *newTab* to the tab control.

Allocates a new id, sets *newTab's* id, locates it just to the right of the existing tabs, inserts an accelerator if the tab's label contains the string "&p" for some value of p, adds it to the bar, and returns 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().

**int QTabBar::insertTab ( QTab * newTab, int index = -1 )** `[virtual]`

Inserts *newTab* to the tab control.

If *index* is not specified, the tab is simply added. Otherwise it's inserted at the specified position.

Allocates a new id, sets *newTab's* id, locates it respectively, inserts an accelerator if the tab's label contains the string "&p" for some value of p, adds it to the bar, and returns the newly allocated id.

**bool QTabBar::isTabEnabled ( int id ) const**

Returns 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 spacebar 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::layoutTabs ()** `[virtual]`

Layout all existing tabs (i.e. setting their r attribute ) according to their label and their iconset.

**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.

**void QTabBar::paintLabel ( QPainter * p, const QRect & br, QTab * t, bool has_focus ) const** `[virtual protected]`

Paints the label of tab *t* centered in rectangle *br* using painter *p* and draws a focus indication if *has_focus* is TRUE.

**void QTabBar::removeTab ( QTab * tab )** `[virtual]`

Removes *tab* from the tab control.

**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 )** `[virtual slot]`

Raises *tab* and emits the selected() signal unless the tab was already current.

See also: currentTab() and selected().

**void QTabBar::setCurrentTab ( int id )** `[virtual slot]`

Raises the tab with ID *id* and emits the selected() signal.

See also: currentTab(), selected() and tab().

**void QTabBar::setShape ( Shape s )** `[virtual]`

Sets the shape of this tab bar to *s* and refreshes the bar.

**void QTabBar::setTabEnabled ( int id, bool enabled )** `[virtual]`

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().

**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.

**QSizePolicy QTabBar::sizePolicy () const** `[virtual]`

Specifies that this widget can use more, but is able to survive on less, horizontal space; and has a fixed height.

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.

**QList<QTab> * QTabBar::tabList ()** `[protected]`

The list of QTab objects added.

**void QTabBar::updateMask ()** `[virtual protected]`

Draws the mask for this tab bar.

Reimplemented from QWidget.

### 209.2.1. URL

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

# 210. QTabDialog Class Reference

```
#include <qtabdialog.h>
```

Inherits QDialog.

## 210.1. 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, Defaults, and Help 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).

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;
- helpButtonPressed() is emitted when the user clicks Help; 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 QTabWidget.

See also: QDialog.

Examples:

tabdialog/tabdialog.cpp

### 210.2. 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 QIconSet & iconset, const QString & label )**

Adds another tab and page to the tab view.

This function is the same as addTab() but with an additional iconset.

Examples:

tabdialog/tabdialog.cpp

**void QTabDialog::addTab ( QWidget * child, const QString & label )**

Adds 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.

**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::changeTab ( QWidget * w, const QString & label )**

Defines a new label for the tab of page \w.

**QWidget * QTabDialog::currentPage () const**

Returns a pointer to the page currently being displayed by the tab dialog. The tab dialog does its best to make sure that this value is never 0, but if you try hard enough it can be.

**void QTabDialog::defaultButtonPressed ()** `[signal]`

This signal is emitted when the Defaults button is pressed. It should reset the dialog (but not the application) to the "factory defaults."

The application's state should not be changed until the user clicks Apply or OK.

See also: applyButtonPressed(), 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::hasHelpButton () const**

Returns TRUE if the tab dialog has a Help button, FALSE if not.

See also: setApplyButton(), applyButtonPressed(), hasCancelButton() and hasHelpButton().

**bool QTabDialog::hasOkButton () const**

Returns TRUE if the tab dialog has an OK button, FALSE if not.

See also: setDefaultsButton(), defaultButtonPressed(), hasOkButton() and hasCancelButton().

**void QTabDialog::helpButtonPressed ()** `[signal]`

This signal is emitted when the Help button is pressed. It should give instructions about how to use the dialog.

See also: applyButtonPressed(), cancelButtonPressed() and setHelpButton().

**void QTabDialog::insertTab ( QWidget * child, QTab * tab, int index = -1 )**

This is a lower-level method for inserting tabs, similar to the other insertTab() method. It is useful if you are using setTabBar() to set a QTabBar subclass with an overridden QTabBar::paint() routine for a subclass of QTab.

**void QTabDialog::insertTab ( QWidget * child, const QIconSet & iconset, const QString & label, int index = -1 )**

Inserts another tab and page to the tab view.

This function is the same as insertTab() but with an additional iconset.

**void QTabDialog::insertTab ( QWidget * child, const QString & label, int index = -1 )**

Inserts 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->insertTab( graphicsPane, "&Graphics" );
td->insertTab( soundPane, "&Sound" );
```

If *index* is not specified, the tab is simply added. Otherwise it's inserted at the specified position.

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 insertTab() after show(), the screen will flicker and the user will be confused.

**bool QTabDialog::isTabEnabled ( QWidget * w ) const**

Returns TRUE if the page *w* is enabled, and false if it is disabled.

See also: setTabEnabled() and QWidget::isEnabled().

**bool QTabDialog::isTabEnabled ( const char * name ) const**

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

Returns TRUE if the page with object name *name* is enabled, and false if it is disabled.

If *name* is 0 or not the name of any of the pages, isTabEnabled() returns FALSE.

See also: setTabEnabled() and QWidget::isEnabled().

**void QTabDialog::paintEvent ( QPaintEvent * )** `[virtual protected]`

Handles paint events for the tabbed dialog.

Reimplemented from QWidget.

**void QTabDialog::removePage ( QWidget * w )**

Removes page *w* from this stack of widgets. Does not delete *w*.

See also: showPage(), QTabWidget::removePage() and QWidgetStack::removeWidget().

**void QTabDialog::resizeEvent ( QResizeEvent * e )** `[virtual protected]`

Handles resize events for the tab dialog.

Reimplemented from QWidget.

**void QTabDialog::selected ( const QString & tabLabel )** `[signal]`

This signal is emitted whenever a tab is selected (raised), including during the first show().

See also: raise().

**void QTabDialog::setApplyButton ()**

Adds an Apply button to the dialog. The button's text is set to a localizable "Apply".

**void QTabDialog::setApplyButton ( const QString & text )**

Add an Apply button to the dialog. The button's text is set to *text*.

The Apply button should apply the current settings in the dialog box to the application, while keeping the dialog visible.

When Apply is clicked, the applyButtonPressed() signal is emitted.

See also: setCancelButton(), setDefaultButton() and applyButtonPressed().

**void QTabDialog::setCancelButton ()**

Adds a Cancel button to the dialog. The button's text is set to a localizable "Cancel".

**void QTabDialog::setCancelButton ( const QString & text )**

Adds a Cancel button to the dialog. The button's text is set to *text*.

The cancel button should always return the application to the state it was in before the tab view popped up, or if the user has clicked Apply, back 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.

If *text* is a null string, no button is shown.

See also: setApplyButton, setDefaultButton() and cancelButtonPressed().

**void QTabDialog::setDefaultButton ()**

Adds a Defaults button to the dialog. The button's text is set to a localizable "Defaults".

**void QTabDialog::setDefaultButton ( const QString & text )**

Adds a Defaults button to the dialog. The button's text is set to *text*.

The Defaults button should set the dialog (but not the application) back to the application defaults.

When Defaults is clicked, the defaultButtonPressed() signal is emitted.

If *text* is a null string, no button is shown.

See also: setApplyButton(), 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::setHelpButton ()**

Adds a Help button to the dialog. The button's text is set to a localizable "Help".

**void QTabDialog::setHelpButton ( const QString & text )**

Adds a Help button to the dialog. The button's text is set to *text.*

When Help is clicked, the helpButtonPressed() signal is emitted.

If *text* is a null string, no button is shown.

See also: setApplyButton(), setCancelButton() and helpButtonPressed().

**void QTabDialog::setOkButton ()**

Adds an OK to the dialog. The button's text is set to a localizable "OK".

**void QTabDialog::setOkButton ( const QString & text )**

Set the OK button's text to *text.*

When the OK button is clicked, the applyButtonPressed() signal is emitted, and the current settings in the dialog box should be applied to the application. Then the dialog closes.

If *text* is a null string, no button is shown.

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 ( QWidget * w, bool enable )**

Enables/disables page *w* according to the value of *enable,* and redraws the page's tab appropriately.

QTabWidget 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, QTabWidget will not hide it, and if all the pages are disabled, QTabWidget will show one of them.

See also: isTabEnabled() and QWidget::setEnabled().

**void QTabDialog::setTabEnabled ( const char * name, bool enable )**

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

Finds the page with object name *name,* enables/disables it according to the value of *enable,* and redraws the page's tab appropriately.

QTabDialog uses QWidget::setEnabled() internally, rather than 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().

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 ( QStyle & 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().

**QString QTabDialog::tabLabel ( QWidget * w )**

Returns the text in the tab for page *w*.

### 210.2.1. URL

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

# 211. QTableView Class Reference

```
#include <qtableview.h>
```

Inherits QFrame.

Inherited by QMultiLineEdit.

### 211.1. Detailed Description

This is an abstract base class for implementing tables

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(); a typical subclass will consist of 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().

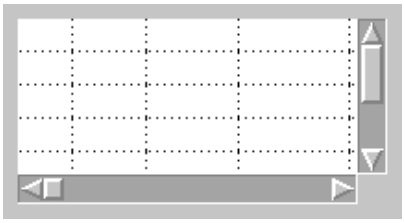
2 The *table* coordinates. (0,0) is the top left corner of the cell at row 0 and column 0. These coordinates are absolute; that is, they are independent of what part of the table is visible at the moment. This is used by functions such as setXOffset() or maxYOffset().

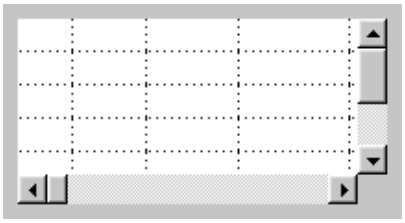
3 The *widget* coordinates. (0,0) is the top left corner of the widget, *including* the frame. This is used by functions such as repaint().

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:** 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.

**Warning:** Experience has shown that use of this widget tends to bring more bugs than expected, and our analysis indicates that widget's very flexibility is the problem. If QScrollView or QListBox can easily be made to do the job you need, we recommend subclassing those widgets rather than QTableView.

See also: QScrollView and GUI Design Handbook: Table

### 211.2. 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::setFrameStyle().

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().

**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().

**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().

**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 QMultiLineEdit.

**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.

**Warning:** You might find that QScrollView offers a higher-level of functionality than using

QTableView and this function.

This function is *not* the same as QWidget::scroll(), in particular, the signs of *xPixels* and *yPixels*

have the reverse semantics.

See also: setXOffset(), setYOffset(), setOffset(), setTopCell(), setLeftCell() and setTopLeftOffset().

**void QTableView::setAutoUpdate ( bool enable )** `[virtual 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.

**void QTableView::setCellHeight ( int cellHeight )** `[virtual 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().

**void QTableView::setCellWidth ( int cellWidth )** `[virtual 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().

**void QTableView::setLeftCell ( int col )** `[virtual 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().

**void QTableView::setNumCols ( int cols )** `[virtual 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().

**void QTableView::setNumRows ( int rows )** `[virtual 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().

**void QTableView::setOffset ( int x, int y, bool updateScrBars = TRUE )** `[virtual 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().

**void QTableView::setTableFlags ( uint f )** `[virtual 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().

**void QTableView::setTopCell ( int row )** `[virtual 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().

**void QTableView::setTopLeftCell ( int row, int col )** `[virtual 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 )** `[virtual 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 )** `[virtual 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.

**void QTableView::show ()** `[virtual]`

Reimplements QWidget::show() in order to do scroll bar tricks.

Examples:

table/main.cpp

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 reimplemented 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 reimplemented 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().

**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.

**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().

### 211.2.1. URL

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

# 212. QTabWidget Class Reference

```
#include <qtabwidget.h>
```

Inherits QWidget.

## 212.1. Detailed Description

The QTabWidget class provides a stack of tabbed widgets.

A tabbed widget 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.

QTabWidget 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.

The normal way to use QTabWidget is to do the following in the constructor:

1 Create a QTabWidget.

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 Connect to the signals and slots.

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.

A signal 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 widgets can be a very good way to split up a complex dialog, it's also very easy to make a royal mess out of it. See QTabDialog for some design hints.

Most of the functionality in QTabWidget is provided by a QTabBar (at the top, providing the tabs) and a QWidgetStack (most of the area, organizing the individual pages).

[Image Omitted]

[Image Omitted]

See also: QTabDialog.

## 212.2. Member Function Documentation

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

Constructs a QTabWidget.

**QTabWidget::~QTabWidget ()**

Destructs the tab widget.

**void QTabWidget::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 QTabWidget::addTab ( QWidget * child, const QIconSet & iconset, const QString & label )**

Adds another tab and page to the tab view.

This function is the same as addTab() but with an additional iconset.

**void QTabWidget::addTab ( QWidget * child, const QString & label )**

Adds 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.

**void QTabWidget::changeTab ( QWidget * w, const QString & label )**

Defines a new label for the tab of page \w.

**QWidget * QTabWidget::currentPage () const**

Returns a pointer to the page currently being displayed by the tab dialog. The tab dialog does its best to make sure that this value is never 0, but if you try hard enough it can be.

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

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QObject.

**void QTabWidget::insertTab ( QWidget * child, QTab * tab, int index = -1 )**

This is a lower-level method for inserting tabs, similar to the other insertTab() method. It is useful if you are using setTabBar() to set a QTabBar subclass with an overridden QTabBar::paint() routine for a subclass of QTab.

**void QTabWidget::insertTab ( QWidget * child, const QIconSet & iconset, const QString & label, int index = -1 )**

Inserts another tab and page to the tab view.

This function is the same as insertTab() but with an additional iconset.

**void QTabWidget::insertTab ( QWidget * child, const QString & label, int index = -1 )**

Inserts 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->insertTab( graphicsPane, "&Graphics" );
td->insertTab( soundPane, "&Sound" );
```

If *index* is not specified, the tab is simply added. Otherwise it's inserted at the specified position.

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 insertTab() after show(), the screen will flicker and the user will be confused.

**bool QTabWidget::isTabEnabled ( QWidget * w ) const**

Returns TRUE if the page *w* is enabled, and false if it is disabled.

See also: setTabEnabled() and QWidget::isEnabled().

**int QTabWidget::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 the pages.

See also: setMargin().

**QSize QTabWidget::minimumSizeHint () const** `[virtual]`

Returns a suitable minimum size for the tab widget.

Reimplemented from QWidget.

**void QTabWidget::removePage ( QWidget * w )**

Removes page *w* from this stack of widgets. Does not delete *w.*

See also: showPage() and QWidgetStack::removeWidget().

**void QTabWidget::selected ( const QString & tabLabel )** `[signal]`

This signal is emitted whenever a tab is selected (raised), including during the first show().

See also: raise().

**void QTabWidget::setMargin ( int w )**

Sets the width of the margin to *w.*

See also: margin().

**void QTabWidget::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 QTabWidget::setTabEnabled ( QWidget * w, bool enable )**

Enables/disables page *w* according to the value of *enable,* and redraws the page's tab appropriately.

QTabWidget 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, QTabWidget will

not hide it, and if all the pages are disabled, QTabWidget will show one of them.

See also: isTabEnabled() and QWidget::setEnabled().

**void QTabWidget::showEvent ( QShowEvent * )** `[virtual protected]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QWidget.

**void QTabWidget::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().

**QSize QTabWidget::sizeHint () const** `[virtual]`

Returns a suitable size for the tab widget.

Reimplemented from QWidget.

**void QTabWidget::styleChange ( QStyle & old )** `[virtual protected]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QWidget.

**QTabBar* QTabWidget::tabBar () const** `[protected]`

Returns the currently set QTabBar.

See also: setTabBar().

**QString QTabWidget::tabLabel ( QWidget * w )**

Returns the text in the tab for page *w*.

**void QTabWidget::updateMask ()** `[virtual protected]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QWidget.

### 212.2.1. URL

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

# 213. QTextBrowser Class Reference

```
#include <qtextbrowser.h>
```

Inherits QTextView.

### 213.1. Detailed Description

A rich text browser with simple navigation.

This class is the same as the QTextView it inherits, with the addition that it provides basic navigation features to follow links in hypertext documents that link to other rich text documents. While QTextView only allows to set its contents with setText(), QTextBrowser has an additional function setSource(), that makes it possible to set documents by name. These names are looked up in the text view's mime source factory. If a document name ends with an anchor, for example "`#anchor`", the text browser will automatically scroll accordingly ( using scrollToAnchor() ). When the user clicks on a hyperlink, the browser will call setSource() itself, with the link's `href` value as argument.

QTextBrowser doesn't provide actual Back and Forward buttons, but it has backward() and forward() slots that implement the functionality. The home() slots brings it back to its very first document displayed.

By using QTextView::setMimeSourceFactory(), you can provide your own subclass of QMimeSourceFactory. This makes it possible to access data from anywhere you need to, may it be the network or a database. See QMimeSourceFactory::data() for details.

If you intend to use the mime factory to read the data directly from the file system, you may have to specify the encoding for the file extension you are using. For example

```
mimeSourceFactory()->setExtensionType("qml", "text/utf8");
```

Otherwise, the factory will not be able to resolve the document names.

For simpler richt text use, see QLabel, QTextView or QSimpleRichText.

### 213.2. Member Function Documentation

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

Constructs an empty QTextBrowser.

**QTextBrowser::~QTextBrowser ()**

Destructs the browser.

**void QTextBrowser::backward ()** `[virtual slot]`

Changes the document displayed to be the previous document in the list of documents build by navigating links.

See also: forward() and backwardAvailable().

**void QTextBrowser::backwardAvailable ( bool available )** `[signal]`

This signal is emitted when the availability of the backward() changes. It becomes available when the user navigates forward, and unavailable when the user is at the home().

**void QTextBrowser::forward ()** `[virtual slot]`

Changes the document displayed to be the next document in the list of documents build by navigating links.

See also: backward() and forwardAvailable().

**void QTextBrowser::forwardAvailable ( bool available )** `[signal]`

This signal is emitted when the availability of the forward() changes. It becomes available after backward() is activated, and unavailable when the user navigates or goes forward() to the last navigated document.

**void QTextBrowser::highlighted ( const QString & href )** `[signal]`

This signal is emitted when the user has selected but not activated a link in the document. *href* is the value of the href tag in the link.

**void QTextBrowser::home ()** `[virtual slot]`

Changes the document displayed to be the first document the browser displayed.

**void QTextBrowser::keyPressEvent ( QKeyEvent * e )** `[virtual protected]`

Add Backward and Forward on ALT-Left and ALT-Right respectively.

Reimplemented from QWidget.

**void QTextBrowser::scrollToAnchor ( const QString & name )**

Scrolls the browser so that the part of the document named *name* is at the top of the view (or as close to the top as the size of the document allows).

**void QTextBrowser::setSource ( const QString & name )** `[virtual]`

Sets the text document with the given *name* to be displayed. The name is looked up in the mimeSourceFactory() of the browser.

In addition to the factory lookup, this functions also checks for optional anchors and scrolls the document accordingly.

If the first tag in the document is `<qt type=detail>`, it is displayed as a popup rather than as new document in the browser window itself. Otherwise, the document is set normally via setText(), with *name* as new context.

If you are using the filesystem access capabilities of the mime source factory, you have to ensure that the factory knows about the encoding of specified text files, otherwise no data will be available.

The default factory handles a couple of common file extensions such as `*.html` and `*.txt` with reasonable defaults. See QMimeSourceFactory::data() for details.

**void QTextBrowser::setText ( const QString & text, const QString & context=QString::null )** `[virtual]`

Sets the contents of the browser to *text,* and emits the textChanged() signal.

Reimplemented from QTextView.

**QString QTextBrowser::source () const**

Returns the source of the currently display document. If no document is displayed or the source is unknown, a null string is returned.

See also: setSource().

**void QTextBrowser::textChanged ()** `[signal]`

This signal is emitted whenever the setText() changes the contents (eg. because the user clicked on a link).

**void QTextBrowser::viewportMouseMoveEvent ( QMouseEvent * e )** `[virtual protected]`

Activate to emit highlighted().

Reimplemented from QScrollView.

**void QTextBrowser::viewportMousePressEvent ( QMouseEvent * e )** `[virtual protected]`

*override* to press anchors.

Reimplemented from QScrollView.

**void QTextBrowser::viewportMouseReleaseEvent ( QMouseEvent * e )** `[virtual protected]`

*override* to activate anchors.

Reimplemented from QScrollView.

### 213.2.1. URL

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

# 214. QTextCodec Class Reference

```
#include <qtextcodec.h>
```

### 214.1. Detailed Description

Provides conversion between text encodings.

By making objects of subclasses of QTextCodec, support for new text encodings can be added to Qt.

The abstract virtual functions describe the encoder to the system and the coder is used as required in the different text file formats supported QTextStream and, under X11 for the locale-specific character input and output (under Windows NT codecs are not needed for GUI I/O since the system works with Unicode already, and Windows 95/98 has built-in convertors for the 8-bit local encoding).

More recently created QTextCodec objects take precedence over earlier ones.

To add support for another 8-bit encoding to Qt, make a subclass or QTextCodec and implement at least the following methods:

`const` char* name() const

> Return the official name for the encoding.

`int` mibEnum() const

> Return the MIB enum for the encoding if it is listed in the IANA character-sets encoding file.

If the encoding is multi-byte then it will have "state"; that is, the interpretation of some bytes will be dependent on some preceeding bytes. For such an encoding, you will need to implement

`QTextDecoder*` makeDecoder() const

> Return a QTextDecoder that remembers incomplete multibyte sequence prefixes or other required state.

If the encoding does *not* require state, you should implement:

`QString` toUnicode(const char* chars, int len) const

Converts *len* characters from *chars* to Unicode.

The base QTextCodec class has default implementations of the above two functions, *but they are mutually recursive*, so you must re-implement at least one of them, or both for improved efficiency.

For conversion from Unicode to 8-bit encodings, it is rarely necessary to maintain state. However, two functions similar to the two above are used for encoding:

`QTextEncoder*` makeEncoder() const

Return a QTextDecoder.

`QCString` fromUnicode(const QString& uc, int& lenInOut ) const;

Converts *lenInOut* characters (of type QChar) from the start of the string *uc,* returning a QCString result, and also returning the length of the result in lenInOut.

Again, these are mutually recursive so only one needs to be implemented, or both if better efficiency is possible.

Finally, you must implement:

`int` heuristicContentMatch(const char* chars, int len) const

Gives a value indicating how likely it is that *len* chracters from *chars* are in the encoding.

A good model for this function is the QWindowsLocalCodec::heuristicContentMatch function found in the Qt sources.

A QTextCodec subclass might have improved performance if you also re-implement:

`bool` canEncode( QChar ) const

Test if a Unicode character can be encoded.

`bool` canEncode( const QString& ) const

Test if a string of Unicode characters can be encoded.

`int` heuristicNameMatch(const char* hint) const

Test if a possibly non-standard name is referring to the codec.

## 214.2. Member Function Documentation

**QTextCodec::QTextCodec () [protected]**

Constructs a QTextCodec, making it of highest precedence. The QTextCodec should always be constructed on the heap (with new), and once constructed it becomes the responsibility of Qt to delete it (which is done at QApplication destruction).

**QTextCodec::~QTextCodec () [virtual]**

Destructs the QTextCodec. Note that you should not delete codecs yourself - once created they become the responsibility of Qt to delete.

**bool QTextCodec::canEncode ( QChar ch ) const [virtual]**

Returns TRUE if the unicode character *ch* can be fully encoded with this codec. The default implementation tests if the result of toUnicode(fromUnicode(ch)) is the original *ch*. Subclasses may be able to improve the efficiency.

**bool QTextCodec::canEncode ( const QString & s ) const [virtual]**

Returns TRUE if the unicode string *s* can be fully encoded with this codec. The default implementation tests if the result of toUnicode(fromUnicode(s)) is the original *s*. Subclasses may be able to improve the efficiency.

**QTextCodec* QTextCodec::codecForContent ( const char * chars, int len ) [static]**

Searches all installed QTextCodec objects, returning the one which most recognizes the given content. May return 0.

See also: heuristicContentMatch().

**QTextCodec* QTextCodec::codecForIndex ( int i ) [static]**

Returns the QTextCodec *i* places from the more recently inserted, or NULL if there is no such QTextCodec. Thus, codecForIndex(0) returns the most recently created QTextCodec.

**QTextCodec* QTextCodec::codecForLocale () [static]**

Returns a pointer to the codec most suitable for this locale.

**QTextCodec* QTextCodec::codecForMib ( int mib ) [static]**

Returns the QTextCodec which matches the MIBenum *mib*.

**QTextCodec* QTextCodec::codecForName ( const char * hint, int accuracy=0 )** `[static]`

Searches all installed QTextCodec objects, returning the one which best matches given name. Returns NULL if no codec has a match closeness above *accuracy.*

See also: heuristicNameMatch().

**void QTextCodec::deleteAllCodecs ()** `[static]`

Deletes all the created codecs.

**Warning:** Do not call this function.

QApplication calls this just before exiting, to delete any QTextCodec objects that may be lying around. Since various other classes hold pointers to QTextCodec objects, it is not safe to call this function earlier.

If you are using the utility classes (like QString) but not using QApplication, calling this function at the very end of your application can be helpful to chasing down memory leaks, as QTextCodec objects will not show up.

**QCString QTextCodec::fromUnicode ( const QString & uc, int & lenInOut ) const** `[virtual]`

Subclasses of QTextCodec must override either this function or makeEncoder(). It converts the first *lenInOut* characters of *uc* from Unicode to the encoding of the subclass. If *lenInOut* is negative or too large, the length of *uc* is used instead.

The value returned is the property of the caller, which is responsible for deleting it with "delete []".

The length of the resulting Unicode character sequence is returned in *lenInOut.*

The default implementation makes an encoder with makeEncoder() and converts the input with that.

Note that the default makeEncoder() implementation makes an encoder that simply calls this function, hence subclasses *must* reimplement one function or the other to avoid infinite recursion.

**QCString QTextCodec::fromUnicode ( const QString & uc ) const**

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

**int QTextCodec::heuristicContentMatch ( const char * chars, int len ) const** `[virtual]`

Subclasses of QTextCodec must override this function. It examines the first *len* bytes of *chars* and returns a value indicating how likely it is that the string is a prefix of text encoded in the encoding of the subclass. Any negative return value indicates that the text is detectably not in the encoding (eg. it contains undefined characters). A return value of 0 indicates that the text should be decoded with this codec rather than as ASCII, but there is no particular evidence. The value should range up to *len.* Thus, most decoders will return -1, 0, or -*len.*

The characters are not null terminated.

See also: codecForContent().

**int QTextCodec::heuristicNameMatch ( const char * hint ) const** `[virtual]`

Returns a value indicating how likely this decoder is for decoding some format that has the given name.

A good match returns a positive number around the length of the string. A bad match is negative.

The default implementation calls simpleHeuristicNameMatch() with the name of the codec.

**QTextCodec* QTextCodec::loadCharmap ( QIODevice * iod )** `[static]`

Reads a POSIX2 charmap definition from *iod.* The parser recognises the following lines:

```
&lt;code_set_name&gt; <i>name</i>
&lt;escape_char&gt; <i>character</i>
% alias <i>alias</i>
CHARMAP
&lt;<i>token</i>&gt; /x<i>hexbyte</i> &lt;U<i>unicode</i>&gt; ...
&lt;<i>token</i>&gt; /d<i>decbyte</i> &lt;U<i>unicode</i>&gt; ...
&lt;<i>token</i>&gt; /<i>octbyte</i> &lt;U<i>unicode</i>&gt; ...
&lt;<i>token</i>&gt; /<i>any</i>/<i>any</i>... &lt;U<i>unicode</i>&gt; ...
END CHARMAP
```

The resulting QTextCodec is returned (and also added to the global list of codecs). The name() of the result is taken from the code_set_name.

Note that a codec constructed in this way uses much more memory and is slower than a hand-written QTextCodec subclass, since tables in code are in memory shared by all applications simultaneously using Qt.

See also: loadCharmapFile().

**QTextCodec* QTextCodec::loadCharmapFile ( QString filename )** `[static]`

A convenience function for QTextCodec::loadCharmap().

**const char* QTextCodec::locale ()** `[static]`

Returns a string representing the current language.

**QTextDecoder* QTextCodec::makeDecoder () const** `[virtual]`

Creates a QTextDecoder which stores enough state to decode chunks of char* data to create chunks of Unicode data. The default implementation creates a stateless decoder, which is sufficient for only the simplest encodings where each byte corresponds to exactly one Unicode character.

The caller is responsible for deleting the returned object.

**QTextEncoder* QTextCodec::makeEncoder () const** `[virtual]`

Creates a QTextEncoder which stores enough state to encode chunks of Unicode data as char* data. The default implementation creates a stateless encoder, which is sufficient for only the simplest encodings where each Unicode character corresponds to exactly one char.

The caller is responsible for deleting the returned object.

**int QTextCodec::mibEnum () const** `[virtual]`

Subclasses of QTextCodec must override this function. It returns the MIBenum (see the IANA character-sets encoding file for more information). It is important that each QTextCodec subclass return the correct unique value for this function.

**const char* QTextCodec::name () const** `[virtual]`

Subclasses of QTextCodec must override this function. It returns the name of the encoding supported by the subclass. When choosing a name for an encoding, consider these points:

- On X11,

`heuristicNameMatch(<em>charset)`
 is used to test if a the QTextCodec can convert between Unicode and the encoding of a font with encoding *charset,* such as "iso8859-1" for Latin-1 fonts, "koi8-r" for Russian KOI8 fonts. The default algorithm of heuristicNameMatch() uses name().

- Some applications may use this function to present encodings to the end user.

**int QTextCodec::simpleHeuristicNameMatch ( const char * name, const char * hint )** `[static protected]`

A simple utility function for heuristicNameMatch() - it does some very minor character-skipping so that almost-exact matches score high.

**QString QTextCodec::toUnicode ( const char * chars, int len ) const** `[virtual]`

Subclasses of QTextCodec must override this function or makeDecoder(). It converts the first *len* characters of *chars* to Unicode.

The default implementation makes an encoder with makeDecoder() and converts the input with that.

Note that the default makeDecoder() implementation makes a decoder that simply calls this

function, hence subclasses *must* reimplement one function or the other to avoid infinite recursion.

**QString QTextCodec::toUnicode ( const QByteArray & a ) const**

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

**QString QTextCodec::toUnicode ( const QByteArray & a, int len ) const**

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

**QString QTextCodec::toUnicode ( const char * chars ) const**

This is an overloaded member function, provided for convenience. It differs from the above function

only in what argument(s) it accepts.

### 214.2.1. URL

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

# 215. QTextDecoder Class Reference

`#include <qtextcodec.h>`

## 215.1. Detailed Description

State-based decoder

A QTextEncoder converts a text format into Unicode, remembering any state that is required between calls.

See also: QTextCodec::makeEncoder().

## 215.2. Member Function Documentation

**QTextDecoder::~QTextDecoder ()** `[virtual]`

Destroys the decoder.

**QString QTextDecoder::toUnicode ( const char * chars, int len )** `[virtual]`

Converts the first *len* bytes at *chars* to Unicode, returning the result.

If not all characters are used (eg. only part of a multi-byte encoding is at the end of the characters), the decoder remembers enough state to continue with the next call to this function.

### 215.2.1. URL

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

# 216. QTextDrag Class Reference

```
#include <qdragobject.h>
```

Inherits QDragObject.

## 216.1. Detailed Description

The QTextDrag provides a drag-and-drop object for transferring plain and Unicode text.

Plain text is single- or multi-line 8-bit text in the local encoding.

Qt provides no built-in mechanism for delivering only single-line.

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.

## 216.2. 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 QString & 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 ( const QMimeSource * e ) [static]

Returns TRUE if the information in *e* can be decoded into a QString.

See also: decode().

**bool QTextDrag::decode ( const QMimeSource * e, QString & str )** `[static]`

Attempts to decode the dropped information in *e* into *str,* returning TRUE if successful.

See also: canDecode().

**bool QTextDrag::decode ( const QMimeSource * e, QString & str, QCString & subtype )** `[static]`

Attempts to decode the dropped information in *e* into *str,* returning TRUE if successful. If *subtype* is null, any text subtype is accepted, otherwise only that specified is accepted. *subtype* is set to the accepted subtype.

See also: canDecode().

**void QTextDrag::setSubtype ( const QCString & st )** `[virtual]`

Sets the MIME subtype of the text begin dragged. The default subtype is "plain", this the default MIME type of the text is "text/plain". You might use this to declare that the text is "text/html" by calling setSubtype("html").

**void QTextDrag::setText ( const QString & text )** `[virtual]`

Sets the text to be dragged. You will need to call this if you did not pass the text during

construction.

### 216.2.1. URL

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

# 217. QTextEncoder Class Reference

`#include <qtextcodec.h>`

## 217.1. Detailed Description

State-based encoder

A QTextEncoder converts Unicode into another format, remembering any state that is required between calls.

See also: QTextCodec::makeEncoder().

## 217.2. Member Function Documentation

**QTextEncoder::~QTextEncoder ()** `[virtual]`

Destroys the encoder.

**QCString QTextEncoder::fromUnicode ( const QString & uc, int & lenInOut )** `[virtual]`

Converts *lenInOut* characters (not bytes) from *uc,* producing a QCString. *lenInOut* will also be set to the length of the result (in bytes).

The encoder is free to record state to use when subsequent calls are made to this function (for example, it might change modes with escape sequences if needed during the encoding of one string, then assume that mode applies when a subsequent call begins).

### 217.2.1. URL

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

# 218. QTextIStream Class Reference

```
#include <qtextstream.h>
```

Inherits QTextStream.

### 218.1. Detailed Description

A convenience class for input streams.

For simple tasks, code should be simple. Hence this class is a shorthand to avoid passing the *mode* argument to the normal QTextStream constructors.

This makes it easy for example, to write things like this:

```
QString data = "123 456";
int a, b;
QTextIStream(&data) >> a >> b;
```

See also: QTextOStream.

### 218.2. Member Function Documentation

**QTextIStream::QTextIStream ( QString * s )**

Constructs a stream to read from string *s*.

**QTextIStream::QTextIStream ( FILE * f )**

Constructs a stream to read from the file *f*.

**QTextIStream::QTextIStream ( QByteArray ba )**

Constructs a stream to read from the array *ba*.

#### 218.2.1. URL

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

# 219. QTextOStream Class Reference

```
#include <qtextstream.h>
```

Inherits QTextStream.

### 219.1. Detailed Description

A convenience class for output streams.

For simple tasks, code should be simple. Hence this class is a shorthand to avoid passing the *mode* argument to the normal QTextStream constructors.

This makes it easy for example, to write things like this:

```
QString result;
QTextOStream(&result) << "pi = " << 3.14;
```

### 219.2. Member Function Documentation

#### QTextOStream::QTextOStream ( QString * s )

Constructs a stream to write to string *s*.

#### QTextOStream::QTextOStream ( FILE * f )

Constructs a stream to write to the file *f*.

#### QTextOStream::QTextOStream ( QByteArray ba )

Constructs a stream to write to the array *ba*.

#### 219.2.1. URL

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

# 220. QTextStream Class Reference

`#include <qtextstream.h>`

Inherited by QTextIStream and QTextOStream.

## 220.1. 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 read/write binary numbers
- `oct` sets the QTextStream to read/write octal numbers
- `dec` sets the QTextStream to read/write decimal numbers
- `hex` sets the QTextStream to read/write 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()).

**Warning:** By default, QTextStream will automatically detect whether integers in the stream are in decimal, octal, hexadecimal or binary format when reading from the stream. In particular, a leading '0' signifies octal, ie. the sequence "0100" will be interpreted as 64.

The QTextStream class reads and writes text and it is not appropriate for dealing with binary data (but QDataStream is).

By default output of Unicode text (ie. QString) is done using the local 8-bit encoding. This can be changed using the setEncoding() method. For input, the QTextStream will auto-detect standard Unicode "byte order marked" text files, but otherwise the local 8-bit encoding is used.

See also: QDataStream.

Examples:

grapher/grapher.cpp

## 220.2. Member Function Documentation

### QTextStream::QTextStream ()

Constructs a data stream that has no IO device.

### QTextStream::QTextStream ( QIODevice * d )

Constructs a text stream that uses the IO device *d*.

### QTextStream::QTextStream ( QString & str, int filemode )

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

This constructor is equivalent to the constructor taking a QString* parameter.

### QTextStream::QTextStream ( QString * str, int filemode )

Constructs a text stream that operates on a Unicode QString through an internal 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.

Note that since QString is Unicode, you should not use readRawBytes() or writeRawBytes() on such a stream.

### QTextStream::QTextStream ( FILE * fh, int mode )

Constructs a text stream that operates on an existing file handle *fh* through an internal QFile device.

Example:

```
QTextStream cout( stdout, IO_WriteOnly );

QTextStream cin ( stdin,  IO_ReadOnly );

QTextStream cerr( stderr, IO_WriteOnly );
```

### QTextStream::QTextStream ( QByteArray a, int mode )

Constructs a text stream that operates on a byte array through an internal QBuffer device.

Example:

```
QByteArray array;

QTextStream ts( array, IO_WriteOnly );

ts << "pi = " << 3.14 << '\0';              // array == "pi = 3.14"
```

Writing data to the text stream will modify the contents of the array. The array will be expanded when data is written beyond the end of the string.

Same example, using a QBuffer:

```
QByteArray array;

QBuffer buf( array );

buf.open( IO_WriteOnly );

QTextStream ts( &buf );

ts << "pi = " << 3.14 << '\0';              // array == "pi = 3.14"

buf.close();
```

### QTextStream::~QTextStream () [virtual]

Destroys the text stream.

The destructor does not affect the current IO device.

**bool QTextStream::atEnd () 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().

**QIODevice * QTextStream::device () const**

Returns the IO device currently set.

See also: setDevice() and unsetDevice().

**bool QTextStream::eof () const**

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

This function has been renamed to atEnd().

See also: QIODevice::atEnd().

Examples:

grapher/grapher.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 QString & s )**

Writes a string 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>> ( QChar & c )

Reads a `char` from the stream and returns a reference to the stream. Note that whitespace is \em not skipped.

### QTextStream & QTextStream::operator>> ( QCString & str )

Reads a word from the stream and returns a reference to the stream.

### 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. Note that whitespace is skipped.

### 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::read ()**

Reads the entire stream and returns a string containing the text.

See also: QIODevice::readLine().

**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().

**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.

\note No Encoding is done by this function.

**Warning:** The behaviour of this function is undefined unless the stream's encoding is set to Unicode or Latin1.

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::setCodec ( QTextCodec * codec )**

Sets the codec for this stream to *codec*. Will not try to autodetect Unicode.

\note This function should be called before any data is read to/written from the stream.

See also: setEncoding.

**void QTextStream::setDevice ( QIODevice * d )**

Sets the IO device to *d*.

See also: device() and unsetDevice().

**void QTextStream::setEncoding ( Encoding e )**

Sets the encoding of this stream to *e,* where *e* is one of:

- `Locale` Using local file format (Latin1 if locale is not set), but autodetecting Unicode(utf16) on input.
- `Unicode` Using Unicode(utf16) for input and output. Output will be written in the order most efficient for the current platform (i.e. the order used internally in QString).
- `Latin1` ISO-8859-1. Will not autodetect utf16.
- `UnicodeNetworkOrder` Using network order Unicode(utf16) for input and output. Useful when reading Unicode data that does not start with the byte order marker.
- `UnicodeReverse` Using reverse network order Unicode(utf16) for input and output. Useful when reading Unicode data that does not start with the byte order marker, or writing data that should be read by buggy Windows applications.
- `RawUnicode` Like Unicode, but does not write the byte order marker, nor does it autodetect the byte order. Only useful when writing to non-persistent storage used by a single process.

`Locale` and all Unicode encodings, except `RawUnicode,` will look at the first two bytes in a

input stream to determine the byte order. The initial byte order marker will be stripped off before

data is read.

\note This function should be called before any data is read to/written from the stream.

See also: setCodec.

**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::skipWhiteSpace ()**

Equivalent to *this << ws.

**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.

\note No Encoding is done by this function.

See also: QIODevice::writeBlock().

### 220.2.1. URL

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

# 221. QTextView Class Reference

```
#include <qtextview.h>
```

Inherits QScrollView.

Inherited by QTextBrowser.

## 221.1. Detailed Description

A sophisticated single-page rich text viewer.

Unlike QSimpleRichText, which merely draws small pieces of rich text, a QTextView is a real widget, with scrollbars when necessary, for showing large text documents.

The rendering style and available tags are defined by a styleSheet(). Currently, a small XML/CSS1 subset including embedded images is supported. See QStyleSheet for details. Possible images within the text document are resolved by using a QMimeSourceFactory. See setMimeSourceFactory() for details.

Using QTextView is quite similar to QLabel. It's mainly a call to setText() to set the contents. Setting the background color is slighty different from other widgets, since a text view is a scrollable widget that naturally provies a scrolling background. You can specify the colorgroup of the displayed text with setPaperColorGroup() or directly define the paper background with setPaper(). QTextView supports both plain color and complex pixmap backgrounds.

Note that we do not intend to add a full-featured web browser widget to Qt (since that would easily double Qt's size and only few applications would benefit from it). In particular, the rich text support in Qt is supposed to provide a fast and sufficient way to add reasonable online help facilities to applications. We will, however, extend it to some degree in future versions of Qt. Most likely some basic table support will be put in and a new class to provide rich text input.

For even more, like hypertext capabilities, see QTextBrowser.

Bugs and limitations:

- No selection possible. No table support (yet).

### 221.2. Member Function Documentation

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

Constructs an empty QTextView with the standard *parent* and *name* optional arguments.

**QTextView::QTextView ( const QString & text, const QString & context = QString::null, QWidget * parent=0, const char * name=0 )**

Constructs a QTextView displaying the contents *text* with context *context,* with the standard *parent* and *name* optional arguments.

**QTextView::~QTextView ()**

Destructs the view.

**QString QTextView::context () const** `[virtual]`

Returns the context of the view.

See also: text() and setText().

**QString QTextView::documentTitle () const**

Returns the document title parsed from the content.

**void QTextView::drawContentsOffset ( QPainter * p, int ox, int oy, int cx, int cy, int cw, int ch )** `[virtual protected]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QScrollView.

**int QTextView::heightForWidth ( int w ) const** `[virtual]`

Returns the height of the view given a width of *w.*

Reimplemented from QWidget.

**void QTextView::keyPressEvent ( QKeyEvent * e )** `[virtual protected]`

Provides scrolling and paging.

Reimplemented from QWidget.

**const QColor& QTextView::linkColor () const**

Returns the current link color.

The color may either have been set with setLinkColor() or stem from the document's body tag.

See also: setLinkColor().

**bool QTextView::linkUnderline () const**

Returns wether or not links should be displayed underlined.

**QMimeSourceFactory* QTextView::mimeSourceFactory () const**

Returns the current mime source factory for the view.

See also: setMimeSourceFactory().

**void QTextView::paletteChange ( const QPalette & p )** `[virtual protected]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QWidget.

**const QBrush& QTextView::paper ()**

Returns the brush used to paint the background.

**const QColorGroup& QTextView::paperColorGroup () const**

Returns the colorgroup of the paper.

See also: setPaperColorGroup() and setPaper().

**void QTextView::resizeEvent ( QResizeEvent * e )** `[virtual protected]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QWidget.

**QRichText& QTextView::richText () const** `[protected]`

Returns the document defining the view as drawable and querable rich text object. This is not currently useful for applications.

**void QTextView::setLinkColor ( const QColor & col )**

Sets the color used to display links in the document to `col`.

See also: linkColor().

**void QTextView::setLinkUnderline ( bool u )**

Defines wether or not links should be displayed underlined.

**void QTextView::setMimeSourceFactory ( QMimeSourceFactory * factory )**

Sets the mime source factory for the view. The factory is used to resolve named references within rich text documents. If no factory has been specified, the text view uses the default factory QMimeSourceFactory::defaultFactory().

Ownership of *factory* is *not* transferred to make it possible for several text view widgets to share the

same mime source.

See also: mimeSourceFactory().

**void QTextView::setPaper ( const QBrush & pap )**

Sets the brush to use as the background to *pap.*

This may be a nice pergament or marble pixmap or simply another plain color.

Technically, setPaper() is just a convenience function to set the base brush of the paperColorGroup().

See also: paper().

**void QTextView::setPaperColorGroup ( const QColorGroup & colgrp )**

Sets the full colorgroup of the paper to *colgrp.* If not specified otherwise in the document itself, any text will use QColorGroup::text(). The background will be painted with QColorGroup::brush(QColorGroup::Base).

See also: paperColorGroup() and setPaper().

**void QTextView::setStyleSheet ( QStyleSheet * styleSheet )**

Sets the style sheet of the view.

See also: styleSheet().

**void QTextView::setText ( const QString & text, const QString & context = QString::null )** `[virtual]`

Changes the contents of the view to the string *text* and the context to *context.*

*text* may be interpreted either as plain text or as rich text, depending on the textFormat(). The default setting is `AutoText`, i.e. the text view autodetects the format from *text.*

The *context* is used to resolve references within the text document, for example image sources. It is passed directly to the mimeSourceFactory() when quering data.

See also: text() and setTextFormat().

Reimplemented in QTextBrowser.

**void QTextView::setTextFormat ( Qt::TextFormat format )**

Sets the text format to *format.* Possible choices are

- `PlainText` - all characters are displayed verbatimely, including all blanks and linebreaks. Word wrap is availbe with the `WordBreak` alignment flag (see setAlignment() for details).
- `RichText` - rich text rendering. The available styles are defined in the default stylesheet QStyleSheet::defaultSheet().
- `AutoText` - this is also the default. The label autodetects which rendering style suits best, `PlainText` or `RichText`. Technically, this is done by using the QStyleSheet::mightBeRichText() heuristic.

**QStyleSheet* QTextView::styleSheet () const**

Returns the current style sheet of the view.

See also: setStyleSheet().

**QString QTextView::text () const** `[virtual]`

Returns the contents of the view.

See also: context() and setText().

**Qt::TextFormat QTextView::textFormat() const**

Returns the current text format.

See also: setTextFormat().

**void QTextView::viewportMouseMoveEvent ( QMouseEvent * )** `[virtual protected]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QScrollView.

**void QTextView::viewportMousePressEvent ( QMouseEvent * )** `[virtual protected]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QScrollView.

**void QTextView::viewportMouseReleaseEvent ( QMouseEvent * )** `[virtual protected]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QScrollView.

**void QTextView::viewportResizeEvent ( QResizeEvent * )** `[virtual protected]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QScrollView.

### 221.2.1. URL

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

# 222. QTime Class Reference

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

### 222.1. 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.

### 222.2. 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.

Note that the counter wraps to zero 24 hours after the last call to start() or restart.

**Warning:** If the system's local time changes, the result is undefined. This can happen e.g. when daylight saving is turned on or off.

See also: start() and restart().

**int QTime::hour () const**

Returns the hour part (0..23) of the time.

**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.

**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.

Note that the counter wraps to zero 24 hours after the last call to start() or restart().

**Warning:** If the system's local time changes, the result is undefined. This can happen e.g. when daylight saving is turned on or off.

See also: start() and elapsed().

**int QTime::second () const**

Returns the second part (0..59) of the time.

**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
qDebug( "%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".

### 222.3. Related Functions

**QDataStream & operator>> (QDataStream & s, QTime & t)**

Reads a time from the stream.

**QDataStream & operator<< (QDataStream & s, const QTime & t)**

Writes a time to the stream.

Serialization format: [Q_UINT32], milliseconds since midnight.

### 222.3.1. URL

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

# 223. QTimer Class Reference

```
#include <qtimer.h>
```

Inherits QObject.

### 223.1. 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:

forever/forever.cpp

## 223.2. 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:

forever/forever.cpp

**void QTimer::stop ()**

Stops the timer.

See also: start().

**void QTimer::timeout () [signal]**

This signal is emitted when the timer is activated.

### 223.2.1. URL

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

# 224. QTimerEvent Class Reference

`#include <qevent.h>`

Inherits QEvent.

### 224.1. 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 interval is 0, then the timer event occurs once every time there are no more window system events to process.

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:

grapher/grapher.cpp
xform/xform.cpp
progress/progress.cpp
qmag/qmag.cpp
forever/forever.cpp
scrollview/scrollview.cpp

### 224.2. 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().

### 224.2.1. URL

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

# 225. QToolBar Class Reference

```
#include <qtoolbar.h>
```

Inherits QWidget.

## 225.1. 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.

## 225.2. 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 QString & 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 QString & label, QMainWindow * parent, QMainWindow::ToolBarDock dock = QMainWindow::Top, bool newLine = FALSE, const char * name = 0 )**

Constructs an empty tool bar which is a child of *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 to the end of the toolbar.

**bool QToolBar::event ( QEvent * e )** `[virtual]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QObject.

**bool QToolBar::eventFilter ( QObject * obj, QEvent * e )** `[virtual]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QObject.

**QString QToolBar::label () const**

Returns the label of this tool bar.

See also: setLabel().

**QMainWindow * QToolBar::mainWindow ()**

Returns a pointer to the QMainWindow which controls this tool bar.

**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::setLabel ( const QString & label )** `[virtual]`

Sets the label of this tool bar to *label*. The label is not currently used; it will be used in a forthcoming tool bar configuration dialog.

See also: label().

**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 )** `[virtual]`

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.

### 225.2.1. URL

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

# 226. QToolButton Class Reference

```
#include <qtoolbutton.h>
```

Inherits QButton.

## 226.1. 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

## 226.2. Member Function Documentation

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

Constructs an empty tool button.

**QToolButton::QToolButton ( const QIconSet & iconSet, const QString & textLabel, const QString & 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 ( const QPixmap & pm, const QString & textLabel, const QString & 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 ( bool on = FALSE ) 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.

If *on* equals TRUE, the special icon set for the on-state of the button is returned.

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, bool on = FALSE )** `[virtual]`

Sets this tool button to display the icons in *set.* (setPixmap() is effectively a wrapper for this function.)

For toggle buttons it is possible to set an extra icon set with *on* equals TRUE, which will be used

exclusively for the on-state.

QToolButton makes a copy of *set,* so you must delete *set* yourself.

See also: iconSet(), QIconSet, setToggleButton() and isOn().

**void QToolButton::setOn ( bool enable )** `[virtual 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 QString & 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 )** `[virtual 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 it

resides. You're strongly urged to use QMainWindow::setUsesBigPixmaps() instead.

**Warning:** If you set some buttons (in a QMainWindow) to have big and others small pixmaps,

QMainWindow may have trouble getting the geometry correct.

**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.

**QSizePolicy QToolButton::sizePolicy () const** `[virtual]`

Specifies that this widget may grow.

Reimplemented from QWidget.

**QString 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().

### 226.2.1. URL

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

# 227. QToolTip Class Reference

```
#include <qtooltip.h>
```

Inherits Qt.

### 227.1. 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-equpped 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

## 227.2. 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 QString & 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 QString & text, QToolTipGroup * group, const QString & 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 QString & 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.

### void QToolTip::add ( QWidget * widget, const QString & text, QToolTipGroup * group, const QString & 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]`

Removes 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 between 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 QString & text ) [protected]**

Pops up a tip saying *text* right now, and removes 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.

**void QToolTip::tip ( const QRect & rect, const QString & text, const QString & groupText )**

`[protected]`

Pops up a tip saying *text* right now, and removes 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.

### 227.2.1. URL

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

# 228. QToolTipGroup Class Reference

`#include <qtooltip.h>`

Inherits QObject.

## 228.1. 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 QString&)),
         myLabel, SLOT(setText(const QString&)) );
connect( g, SIGNAL(removeTip()),
         myLabel, SLOT(clear()) );
QToolTip::add( giraffeButton, "feed giraffe",
               g, "Give the giraffe a meal" );
QToolTip::add( gorillaButton, "feed gorilla",
               g, "Give the gorilla a meal" );
```

This example makes the object myLabel (which you have to supply) display (one assumes, though you can make myLabel do anything, of course) the strings "Give the giraffe a meal" and "Give the gorilla a meal" while the relevant tool tips are being displayed.

Deleting a tool tip group removes the tool tips in it.

## 228.2. 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.

**bool QToolTipGroup::delay () const**

Returns TRUE if the group text is shown delayed (at the same time as the tip) and FALSE if it is shown immediately.

See also: setDelay().

**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::setDelay ( bool enable )** `[slot]`

Sets the group to show its text immediately if *enable* is FALSE, and delayed (at the same time as the tip text) if *enable* is TRUE. The default is TRUE.

See also: delay().

**void QToolTipGroup::showTip ( const QString & longText )** `[signal]`

This signal is emitted when one of the tool tips in the group is displayed. *longText* is the supplementary text for the displayed tool tip.

See also: removeTip().

### 228.2.1. URL

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

# 229. QTranslator Class Reference

```
#include <qtranslator.h>
```

Inherits QObject.

## 229.1. Detailed Description

The QTranslator class provides internationalization support for text output.

The class is conceptually very simple: An object of this class contains a set of translations from the reference language to a target language, and provides functions to add, look up and remove such translations as well as the ability to load and save the object to a file.

The most common use of QTranslator is expected to be loading one from a file, installing it using QApplication::installTranslator(), and using it via QObject::tr().

Slightly more advanced usage of QTranslator includes direct lookup using find(), adding new translations using insert() and removing existing ones using remove() or even clear(), and testing whether the QTranslator contains a translation using contains().

The hash() function mentioned is a variant on the standard ELF hash, modified to work well with Unicode strings in UCS-2 format. Its algorithm is not specified beyond the fact that it will remain unchanged in future versions of Qt.

To examine the contents of a QTranslator, use QTranslatorIterator.

See also: QTranslatorIterator, QApplication::installTranslator, QApplication::removeTranslator(), QObject::tr() and QApplication::translate().

Examples:

i18n/main.cpp

## 229.2. Member Function Documentation

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

Constructs an empty message file, not connected to any file.

**QTranslator::~QTranslator ()**

Destroys the object and frees any allocated resources.

**void QTranslator::clear ()**

Empties this translator of all contents.

**bool QTranslator::contains ( const char * scope, const char * key ) const**

Returns TRUE if this message file contains a message with hash value *h,* and FALSE if it does not.

(This is is a one-liner that calls find().).

**QString QTranslator::find ( const char * scope, const char * message ) const** `[virtual]`

Returns the translation for (*scope, key),* or QString::null in case there is none in this translator.

**void QTranslator::insert ( const char * scope, const char * message, const QString & translation )**

Inserts *translation* of *message* into this message file.

**bool QTranslator::load ( const QString & filename, const QString & directory = QString::null, const QString & search_delimiters = QString::null, const QString & suffix = QString::null )**

Loads *filename,* which may be an absolute file name or relative to *directory.* If the full filename does not exist, other filenames are tried in the following order:

1 Filename with *suffix* appended (".qm" if suffix is QString::null)

2 Filename with text after a character in *search_delimiters* stripped ("_." is the default for *search_delimiters* if it is QString::null)

3 Filename stripped and *suffix* appended.

4 Filename stripped further, etc. For example, load("foo_bar.baz", "/opt/foolib") will search for:

1 /opt/foolib/foo_bar.baz

2 /opt/foolib/foo_bar.baz.qm

3 /opt/foolib/foo_bar

4 /opt/foolib/foo_bar.qm

5 /opt/foolib/foo

6 /opt/foolib/foo.qm

Examples:

i18n/main.cpp

**void QTranslator::remove ( const char * scope, const char * message )**

Removes the string for *h* from this message file. If there is no string for h, this function does nothing.

**void QTranslator::squeeze ()**

Converts this message file to the compact format used to store message files on disk. You should never need to call this directly; save() and other functions call it as necessary.

**void QTranslator::unsqueeze ()**

Converts this message file into an easily modifiable data structure, less compact than the format used in the files.

You should never need to call this function; it is called by insert() etc. as necessary.

See also: squeeze().

### 229.2.1. URL

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

# 230. QUriDrag Class Reference

```
#include <qdragobject.h>
```

Inherits QStoredDrag.

### 230.1. Detailed Description

Provides for drag-and-drop of a list of URI references.

URIs are a useful way to refer to files that may be distributed across multiple machines. Much of the time a URI will refer to a file on a machine local to both the drag source and the drop target, and so the URI will be equivalent to passing a filename, but more extensible.

While presenting URIs to the user, use them in Unicode form so that the user can confortably edit and view them. For use in HTTP or other protocols, use the correctly escaped ASCII form (see.

### 230.2. Member Function Documentation

**QUriDrag::QUriDrag ( QStrList uris, QWidget * dragSource = 0, const char * name = 0 )**

Creates an object to drag the list of URIs in *uris*. The *dragSource* and *name* arguments are passed on to QStoredDrag. Note that URIs are always in escaped UTF8 encoding, as defined by the W3C.

**QUriDrag::QUriDrag ( QWidget * dragSource = 0, const char * name = 0 )**

Creates a object to drag. You will need to call setUris() before you start the drag().

**QUriDrag::~QUriDrag ()**

Destroys the object.

**bool QUriDrag::canDecode ( const QMimeSource * e )** `[static]`

Returns TRUE if decode() would be able to decode *e*.

**bool QUriDrag::decode ( const QMimeSource * e, QStrList & l )** `[static]`

Decodes URIs from *e,* placing the result in *l* (which is first cleared).

Returns TRUE if the event contained a valid list of URIs.

**bool QUriDrag::decodeLocalFiles ( const QMimeSource * e, QStringList & l )** `[static]`

Decodes URIs 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 URIs. The list will be empty if no URIs were

local files.

**bool QUriDrag::decodeToUnicodeUris ( const QMimeSource * e, QStringList & l )** `[static]`

Decodes URIs from *e,* converts them to Unicode URIs (only useful for displaying to humans), placing them in *l* (which is first cleared).

Returns TRUE if the event contained a valid list of URIs.

**QCString QUriDrag::localFileToUri ( const QString & filename )** `[static]`

Returns the URI equivalent to the absolute local file *filename.*

See also: uriToLocalFile().

**void QUriDrag::setFilenames ( QStringList fnames )**

Sets the URIs to be the local-file URIs equivalent to *fnames.*

See also: localFileToUri() and setUris().

**void QUriDrag::setUnicodeUris ( QStringList uuris )**

Sets the URIs to be the Unicode URIs (only useful for displaying to humans) *uuris.*

See also: localFileToUri() and setUris().

**void QUriDrag::setUris ( QStrList uris )** `[virtual]`

Changes the list of *uris* to be dragged.

**QCString QUriDrag::unicodeUriToUri ( const QString & uuri )** `[static]`

Returns the URI equivalent to the Unicode URI (only useful for displaying to humans). *uuri.*

See also: uriToLocalFile().

**QString QUriDrag::uriToLocalFile ( const char * uri )** `[static]`

Returns the name of a local file equivalent to *uri,* or a null string if *uri* is not a local file.

See also: localFileToUri().

**QString QUriDrag::uriToUnicodeUri ( const char * uri )** `[static]`

Returns the Unicode URI (only useful for displaying to humans) equivalent to *uri.*

See also: localFileToUri().

### 230.2.1. URL

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

# 231. QValidator Class Reference

`#include <qvalidator.h>`

Inherits QObject.

Inherited by QDoubleValidator and QIntValidator.

## 231.1. 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).

The three states require some explanation. An `Invalid` string is *clearly* invalid. `Valid` is less obvious - the concept of validity is slippery when the string is incomplete (still being edited). QValidator defines `Valid` as the property of a string that it is not clearly invalid. `Acceptable` means that the string is acceptable as a final result. One might say that any string that is a plausible intermediate state during entry of an `Acceptable` string is `Valid.`

Here are some examples:

1 For a line edit that accepts integers from 0 to 999 inclusive, 42 and 666 are `Acceptable,` the empty string and 1114 are `Valid` and asdf is `Invalid.`

2 For an editable combo box that accepts URLs, any well-formed URL is `Acceptable,` "http://www.troll.no/," is `Valid` (it can be a cut-and-paste job that accidentally took in a comma at the end), the empty string is valid (the user might select and delete all of the text in preparation to entering a new URL), and "http:///./moron" is `Invalid.`

3 For a spin box that accepts lengths, "11cm" and "1in" are `Acceptable,` "11" and the empty string are `Valid,` and "http://www.troll.no" and "hour" are `Invalid.`

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. This allows some `Invalid` strings to be made `Acceptable`, too, spoiling the muddy definition above even more.

QValidator is generally used with QLineEdit, QSpinBox and QComboBox.

## 231.2. 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 ) const** `[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.

### 231.2.1. URL

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

# 232. QValueList Class Reference

```
#include <qvaluelist.h>
```

### 232.1. Detailed Description

The QValueList class is a value based template class that provides doubly linked lists.

Define a template instance QValueList<X> to create a list of values which all have the class X.

Please notice that QValueList does not store pointers to the members of the list. It holds a copy of every member. That is the reason why this kind of classes are called "value based" while QList and QDict are "reference based".

Some classes can not be used within a QValueList. For example everything derived from QObject and as a reason of this that extends to all classes which implement widgets. You can only put stuff in a QValueList which can be easily copied. That means that the class X should implement a fast copy constructor.

Example:

```
#include <qvaluelist.h>
#include <qstring.h>
#include <stdio.h>
class Employee
{
public:
    Employee( const QString& name, int salary ) { n=name; s=salary; }
    Employee( const Employee& _e ) { n = _e.name; s = _e.s; }
    QString     name()   const               { return n; }
    int         salary() const               { return s; }
    void        setSalary( int _s ) const    { s = _s; }
private:
    QString     n;
    int         s;
};
void main()
```

```
{
    typedef QValueList<Employee> EmployeeList;
    EmployeeList list;              // list of Employee
    list.append( Employee("Bill", 50000) );
    list.append( Employee("Steve",80000) );
    list.append( Employee("Ron",  60000) );
    Employee joe( "Joe", 50000 );
    list.append( joe );
    joe.setSalary( 4000 );
    EmployeeList::Iterator it;
    for( it = list.begin(); it != list.end(); ++it )
        printf( "%s earns %d\n", (*it).name(), (*it).salary() );
}
```

Program output:

```
Bill earns 50000
Steve earns 80000
Ron earns 60000
Joe earns 50000
```

As you can see, the latest changes to Joes salary did not affect the value in the list because the list created a copy of Joes entry.

There are three ways of finding items in the list. The first one is by using the at() function. It returns an iterator. The advantages of getting an iterator is that you can now move forward or backward from this position by incrementing/decrementing the iterator. To get the amount of items in the list call count(). Valid indices are 0..count().

The second way of accessing a list is with operator[]. That means you can address it like an array. The return value is a reference to the value stored in the list. There exist two versions of this operator. The first one is const and returns a const reference to the value. The second on is non const and returns a non const reference to the value. It is up to your compiler to choose the correct one.

The third method is to use the functions begin() and end(). With a simple for loop as shown in the example you can iterate over the complete list. It is save to have multiple iterators at the same time. If some member of the list is removed then only iterators pointing to the removed member become invalid. Inserting in the list does not invalidate any iterator. For convenience the function last() returns an iterator for the last and first() for thge first element in the list.

In addition you can search items in the list with the find() function. It exists in a const and a non const version. It starts searching from the beginning of the list, but another flavour of the find() function allows you to specify where searching should start. If you just want to know wether a certain item is at least once in the list, then you can use the contains() function.

Since QValueList is value based there is no need to care about deleting elements in the list. The list holds its own copies and will free them if the corresponding member or the list itself is deleted. You can force the list to free all of its item with clear().

QValueList is implict shared. That means you can just make copies of the list in time O(1). If multiple QValueList instances share the same data and one is doing a modification of the lists data then this modifying instance makes a copy and modifies its private copy. So it does not affect the other instances. From a developers point of view you can think that a QValueList and a copy of this list have nothing to do with each other. Developers may only notice that copying is very fast. People known to a CPUs MMU architecture will know this pattern as "copy on write".

There exist three functions to insert items in the list. append() inserts an item at the end, prepend() inserts at the beginning and insert() inserts in front of the position given by an iterator.

Items can be removed from the list in two ways. The first is to pass an iterator to the remove(). The other possibility is to pass a value to remove() which will delete all members which match this value.

Sorting of the list is not supported by the class itself. That has three reasons: For one Qt does not require that the class of the instances stored in a QValueList do need to implement operator<. Reason number three is that sorting is very often not needed. By adding the sorting algorithm to the

QValueList template the compiler would generate sorting code for every incarnation of the template but it is never used. Reason number three is that you can choose between different sorting algorithms. TODO: Reference to QTL

Example:

```
QValueList l;
l.append( 5 );
l.append( 8 );
l.append( 3 );
l.append( 4 );
qHeapSort( l );
```

See also: QValueListIterator.

## 232.2. Member Function Documentation

### QValueList::QValueList ()

Constructs an empty list.

### QValueList::~QValueList ()

Destroys the list. References to the values in the list and all iterators of this list become invalidated. Since QValueList is highly tuned for performance you wont see warnings if you use invalid iterators, because it is impossible for an iterator to check wether it is valid or not.

### Iterator QValueList::append ( const T & x )

Inserts the value *x* at the end of the list.

Returns an iterator pointing at the inserted item.

See also: insert() and prepend().

### ConstIterator QValueList::at ( uint i ) const

Returns an iterator pointing to the item at position *i* in the list, or end() if the index is out of range.

**Iterator QValueList::at ( uint i )**

Returns an iterator pointing to the item at position *i* in the list, or end() if the index is out of range.

**ConstIterator QValueList::begin () const**

Returns an iterator pointing to the first element in the list. This iterator equals end() if the list is empty;

See also: first() and end().

**Iterator QValueList::begin ()**

Returns an iterator pointing to the first element in the list. This iterator equals end() if the list is empty;

See also: first() and end().

**void QValueList::clear ()**

Removes all items from the list.

See also: remove().

**uint QValueList::contains ( const T & x ) const**

Counts and returns the number of occurrences of the value *x* in the list.

**uint QValueList::count () const**

Returns the number of items in the list.

See also: isEmpty().

**void QValueList::detach ()** `[protected]`

If the list does not share its data with another QValueList instance, then nothing happens, otherwise the function creates a new copy of this data and detaches from the shared one. This function is called whenever the list is modified. The implicit sharing mechanism is implemented this way.

**ConstIterator QValueList::end () const**

Returns an iterator pointing behind the last element in the list. This iterator equals begin() if the list is empty.

See also: last() and begin().

**Iterator QValueList::end ()**

Returns an iterator pointing behind the last element in the list. This iterator equals begin() if the list is empty.

See also: last() and begin().

**ConstIterator QValueList::find ( ConstIterator it, const T & x ) const**

Finds the first occurrence of *x* in the list starting at the position given by *it*.

Returns end() if no item did match.

**ConstIterator QValueList::find ( const T & x ) const**

Finds the first occurrence of *x* in the list.

Returns end() if no item did match.

**Iterator QValueList::find ( Iterator it, const T & x )**

Finds the first occurrence of *x* in the list starting at the position given by *it*.

Returns end() if no item did match.

**Iterator QValueList::find ( const T & x )**

Finds the first occurrence of *x* in the list.

Returns end() if no item did match.

**int QValueList::findIndex ( const T & x ) const**

Returns the first index of the value *x* in the list or -1 if no such value can be found in the list.

**T& QValueList::first ()**

Returns a reference to the first item in the list or the item referenced by end() if no such items exists. Please note that you may not change the value the end() Iterator is pointing to.

See also: begin() and last().

**const T& QValueList::first () const**

Returns a reference to the first item in the list or the item referenced by end() if no such items exists.

See also: begin() and last().

**ConstIterator QValueList::fromLast () const**

Returns an iterator pointing to the last element in the list or end() if no such item exists.

See also: last().

**Iterator QValueList::fromLast ()**

Returns an iterator pointing to the last element in the list or end() if no such item exists.

See also: last().

**Iterator QValueList::insert ( Iterator it, const T & x )**

Inserts the value *x* in front of the iterator *it.*

Returns an iterator pointing at the inserted item.

See also: append() and prepend().

**bool QValueList::isEmpty () const**

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

See also: count().

**T& QValueList::last ()**

Returns a reference to the last item in the list or the item referenced by end() if no such item exists. Please note that you may not change the value the end() Iterator is pointing to.

See also: end(), first() and fromLast().

**const T& QValueList::last () const**

Returns a reference to the last item in the list or the item referenced by end() if no such item exists.

See also: end(), first() and fromLast().

**bool QValueList::operator!= ( const QValueList<T> & l ) const**

Compares both lists.

Returns TRUE if both list are unequal.

**QValueList<T> QValueList::operator+ ( const QValueList<T> & l ) const**

Creates a new list and fills it with the elements of this list. Then the elements of *l* are appended.

Returns the new list.

**QValueList<T>& QValueList::operator+= ( const QValueList<T> & l )**

Adds *list* to this list.

Returns a reference to this list.

**QValueList<T>& QValueList::operator+= ( const T & x )**

Adds the value *x* to the end of the list.

Returns a reference to the list.

**QValueList<T>& QValueList::operator<< ( const T & x )**

Adds the value *x* to the end of the list.

Returns a reference to the list.

**QValueList<T>& QValueList::operator= ( const QValueList<T> & l )**

Assigns *l* to this list and returns a reference to this list.

All iterators of the current list become invalidated by this operation. The cost of such an assignment

is O(1) since QValueList is implict shared.

**bool QValueList::operator== ( const QValueList<T> & l ) const**

Compares both lists.

Returns TRUE if both list are equal.

**T& QValueList::operator[] ( uint i )**

Returns a reference to the item with index *i* in the list. It is up to you to check wether this item really exists. You can do that easily with the count() function. However this operator does not check wether *i* is in range and will deliver undefined results if it does not exist. In contrast to the const operator[] you may manipulate the value returned by this operator.

**const T& QValueList::operator[] ( uint i ) const**

Returns a const reference to the item with index *i* in the list. It is up to you to check wether this item really exists. You can do that easily with the count() function. However this operator does not check wether *i* is in range and will deliver undefined results if it does not exist.

**Iterator QValueList::prepend ( const T & x )**

Inserts the value *x* at the end of the list.

Returns an iterator pointing at the inserted item.

See also: insert() and append().

**Iterator QValueList::remove ( Iterator it )**

Removes the item at position *it* in the list.

Returns an iterator pointing to the item following the removed on or end() if the last item was

deleted.

See also: clear().

**void QValueList::remove ( const T & x )**

Removes all items which have the value *x*.

See also: clear().

### 232.3. Related Functions

**QDataStream& operator<< (QDataStream & s, const QValueList<T> & l)**

Writes a list to the stream. The type *T* stored in the list must implement the streaming operator, too.

**QDataStream& operator>> (QDataStream & s, QValueList<T> & l)**

Reads a list from the stream. The type *T* stored in the list must implement the streaming operator,

too.

#### 232.3.1. URL

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

# 233. QValueListConstIterator Class Reference

```
#include <qvaluelist.h>
```

### 233.1. Detailed Description

The QValueListConstIterator class provides an iterator for QValueList.

In contrast to QValueListIterator this class is used to iterate over a const list. It does not allow to modify the values of the list since this would break the const semantics.

For more informations on QValueList iterators see QValueListIterator.

See also: QValueListIterator, \sa and QValueList.

### 233.2. Member Function Documentation

**QValueListConstIterator::QValueListConstIterator ()**

Creates un unitilaized iterator.

**QValueListConstIterator::QValueListConstIterator ( NodePtr p )**

Internal function.

**QValueListConstIterator::QValueListConstIterator ( const QValueListIterator<T> & it )**

Constructs a copy of the iterator *it*.

**QValueListConstIterator::QValueListConstIterator ( const QValueListConstIterator<T> & it )**

Constructs a copy of the iterator *it*.

**bool QValueListConstIterator::operator!= ( const QValueListConstIterator<T> & it ) const**

Compares both iterators and returns TRUE if they point to different items.

**const T& QValueListConstIterator::operator* () const**

Asterix operator. Returns a reference to the current iterator item.

**QValueListConstIterator<T> QValueListConstIterator::operator++ ( int )**

Postfix ++ makes the succeeding item current and returns an iterator pointing to the new current item. The iterator can not check wether it reached the end of the list. Incrementing the iterator as returned by end() causes undefined results.

**QValueListConstIterator<T>& QValueListConstIterator::operator++ ()**

Prefix ++ makes the succeeding item current and returns an iterator pointing to the new current item. The iterator can not check wether it reached the end of the list. Incrementing the iterator as returned by end() causes undefined results.

**QValueListConstIterator<T> QValueListConstIterator::operator-- ( int )**

Postfix -- makes the previous item current and returns an iterator pointing to the new current item. The iterator can not check wether it reached the beginning of the list. Decrementing the iterator as returned by begin() causes undefined results.

**QValueListConstIterator<T>& QValueListConstIterator::operator-- ()**

Prefix -- makes the previous item current and returns an iterator pointing to the new current item. The iterator can not check wether it reached the beginning of the list. Decrementing the iterator as returned by begin() causes undefined results.

**bool QValueListConstIterator::operator== ( const QValueListConstIterator<T> & it ) const**

Compares both iterators and returns TRUE if they point to the same item.

### 233.2.1. URL

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

# 234. QValueListIterator Class Reference

```
#include <qvaluelist.h>
```

### 234.1. Detailed Description

The QValueListIterator class provides an iterator for QValueList.

You can not create an iterator by yourself. Instead you have to ask a list to give you one. An iterator has only the size of a pointer. On 32 bit machines that means 4 bytes otherwise 8 bytes. That makes them very fast. In fact they resemble the semantics of pointers as good as possible and they are almost as fast as usual pointers.

Example:

```
#include <qvaluelist.h>
#include <qstring.h>
#include <stdio.h>
class Employee
{
public:
    Employee( const QString& name, int salary ) { n=name; s=salary; }
    Employee( const Employee& _e ) { n = _e.name; s = _e.s; }
    QString     name()   const               { return n; }
    int         salary() const               { return s; }
    void        setSalary( int _s ) const    { s = _s; }
private:
    QString     n;
    int         s;
};
void main()
{
    typedef QValueList<Employee> EmployeeList;
    EmployeeList list;              // list of Employee
    list.append( Employee("Bill", 50000) );
    list.append( Employee("Steve",80000) );
```

```
    list.append( Employee("Ron",  60000) );
    Employee joe( "Joe", 50000 );
    list.append( joe );
    joe.setSalary( 4000 );
    EmployeeList::Iterator it;
    for( it = list.begin(); it != list.end(); ++it )
        printf( "%s earns %d\n", (*it).name(), (*it).salary() );
}
```

Program output:

```
Bill earns 50000
Steve earns 80000
Ron earns 60000
Joe earns 50000
```

In contrast to QList there are no built in functions in QValueList to traverse the list. The only way to do this is to use iterators. QValueList is highly optimized for performance and memory usage. On the other hand that means that you have to be a bit more careful by what you are doing. QValueList does not know about all its iterators and the iterators dont even know to which list they belong. That makes things fast and slim but a bit dangerous because it is up to you to make shure that iterators you are using are still valid. QListIterator will be able to give warnings while QValueListIterator may end up in an undefined state.

For every Iterator there is a ConstIterator. When accessing a QValueList in a const environment or if the reference or pointer to the list is itself const, then you have to use the ConstIterator. Its semantics are the same, but it returns only const references to the item it points to.

See also: QValueList, \sa and QValueListConstIterator.

## 234.2. Member Function Documentation

### QValueListIterator::QValueListIterator ()

Creates un unitilaized iterator.

**QValueListIterator::QValueListIterator ( NodePtr p )**

Internal function.

**QValueListIterator::QValueListIterator ( const QValueListIterator<T> & it )**

Constructs a copy of the iterator *it*.

**bool QValueListIterator::operator!= ( const QValueListIterator<T> & it ) const**

Compares both iterators and returns TRUE if they point to different items.

**T& QValueListIterator::operator* ()**

Asterix operator. Returns a reference to the current iterator item.

**const T& QValueListIterator::operator* () const**

Asterix operator. Returns a reference to the current iterator item.

**QValueListIterator<T> QValueListIterator::operator++ ( int )**

Postfix ++ makes the succeeding item current and returns an iterator pointing to the new current item. The iterator can not check wether it reached the end of the list. Incrementing the iterator as returned by end() causes undefined results.

**QValueListIterator<T>& QValueListIterator::operator++ ()**

Prefix ++ makes the succeeding item current and returns an iterator pointing to the new current item. The iterator can not check wether it reached the end of the list. Incrementing the iterator as returned by end() causes undefined results.

**QValueListIterator<T> QValueListIterator::operator-- ( int )**

Postfix -- makes the previous item current and returns an iterator pointing to the new current item. The iterator can not check wether it reached the beginning of the list. Decrementing the iterator as returned by begin() causes undefined results.

**QValueListIterator<T>& QValueListIterator::operator-- ()**

Prefix -- makes the previous item current and returns an iterator pointing to the new current item. The iterator can not check wether it reached the beginning of the list. Decrementing the iterator as returned by begin() causes undefined results.

**bool QValueListIterator::operator== ( const QValueListIterator<T> & it ) const**

Compares both iterators and returns TRUE if they point to the same item.

### 234.2.1. URL

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

# 235. QVBox Class Reference

```
#include <qvbox.h>
```

Inherits QHBox.

### 235.1. Detailed Description

The QVBox widget performs geometry management on its children

All its children will be placed vertically and sized according to their sizeHint()s.

See also: QVBox and QHBox.

Examples:

tabdialog/tabdialog.cpp

### 235.2. Member Function Documentation

**QVBox::QVBox ( QWidget * parent=0, const char * name=0, WFlags f=0, bool allowLines=TRUE )**

Constructs a vbox widget with parent *parent* and name *name*

#### 235.2.1. URL

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

# 236. QVBoxLayout Class Reference

```
#include <qlayout.h>
```

Inherits QBoxLayout.

## 236.1. Detailed Description

The QVBoxLayout class lines up 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 );
```

See also: QHBoxLayout, QGridLayout and Layout overview documentation

Examples:

layout/layout.cpp
scrollview/scrollview.cpp
customlayout/main.cpp

## 236.2. Member Function Documentation

**QVBoxLayout::QVBoxLayout ( QLayout * parentLayout, int space = -1, const char * name=0 )**

Creates a new vertical box and adds it to *parentLayout.*

**QVBoxLayout::QVBoxLayout ( QWidget * parent, int border=0, int space = -1, const char * name=0 )**

Creates a new top-level vertical box.

**QVBoxLayout::QVBoxLayout ( int space = -1, const char * name=0 )**

Creates a new vertical box. You have to add it to another layout.

**QVBoxLayout::~QVBoxLayout ()**

Destroys this box.

### 236.2.1. URL

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

# 237. QVButtonGroup Class Reference

`#include <qvbuttongroup.h>`

Inherits QButtonGroup.

### 237.1. Detailed Description

The QVButtonGroup widget organizes QButton widgets in a group with one vertical column.

QVButtonGroup is a convenience class that offers a thin layer on top of QButtonGroup. Think of it

as a QVBox that offers a frame with a title and is specifically designed for buttons.

### 237.2. Member Function Documentation

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

Constructs a vertical button group with no title.

The *parent* and *name* arguments are passed to the QWidget constructor.

**QVButtonGroup::QVButtonGroup ( const QString & title, QWidget * parent=0, const char * name=0 )**

Constructs a vertical button group with a title.

The *parent* and *name* arguments are passed to the QWidget constructor.

**QVButtonGroup::~QVButtonGroup ()**

Destroys the vertical button group and its child widgets.

#### 237.2.1. URL

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

# 238. QVGroupBox Class Reference

`#include <qvgroupbox.h>`

Inherits QGroupBox.

### 238.1. Detailed Description

The QVGroupBox widget organizes QButton widgets in a group with one vertical column.

QVGroupBox is a convenience class that offers a thin layer on top of QGroupBox. Think of it as a

QVBox that offers a frame with a title.

### 238.2. Member Function Documentation

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

Constructs a vertical group box with no title.

The *parent* and *name* arguments are passed to the QWidget constructor.

**QVGroupBox::QVGroupBox ( const QString & title, QWidget * parent=0, const char * name=0 )**

Constructs a vertical group box with a title.

The *parent* and *name* arguments are passed to the QWidget constructor.

**QVGroupBox::~QVGroupBox ()**

Destroys the vertical group box and its child widgets.

#### 238.2.1. URL

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

# 239. QWhatsThis Class Reference

```
#include <qwhatsthis.h>
```

Inherits Qt.

### 239.1. Detailed Description

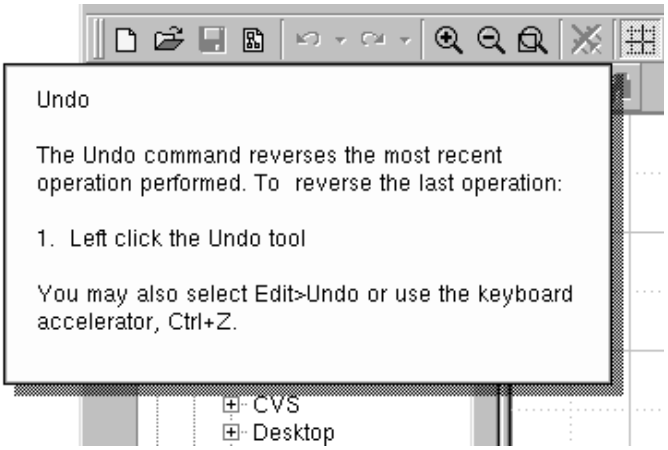
The QWhatsThis class provides a simple description of any widget, e.g. answering the question "what's this?"

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. But you can also connect a "what's this" entry of a help menu (with Shift-F1 as accelerator) to the whatsThis() slot of a QMainWindow, then the focus widget does not have a special meaning.

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.




Futhermore, you can create a dedicated QWhatsThis object for a special widget. By subclassing and reimplementing QWhatsThis::text() it is possible, to have different explanatory texts depending on the position of the mouse click.

If you widget needs even more control, see QWidget::customWhatsThis().

Other Qt classes support What's This help directly. Keyboard accelerators, for example, can also provide explanatory text with access through both they acceleration key itself or an attached menu item. See QAccel::setWhatsThis() for details.

The explanatory text can be both rich text or plain text. If you specify a rich text formatted string, it will be rendered using the default stylesheet. This makes it also possible to embed images. See QStyleSheet::defaultSheet() for details.

See also: QToolTip.

## 239.2. Member Function Documentation

**QWhatsThis::QWhatsThis ( QWidget * widget )**

Constructs a dynamic What's This object for *widget.*

**QWhatsThis::~QWhatsThis ()** `[virtual]`

Destroys the object and frees any allocated resources.

**void QWhatsThis::add ( QWidget * widget, const QString & text )** `[static]`

Adds *text* as What's This help for *widget*. If the text is rich text formatted, it will be rendered with the default stylesheet QStyleSheet::defaultSheet().

See also: remove().

**void QWhatsThis::enterWhatsThisMode ()** `[static]`

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

What's This will install a special cursor and take over mouse input until the user click somewhere, then show any help available and switch out of What's This mode. Finally, What's This removes its cursor and help window and restores ordinary event processing. At this point the left mouse button is not pressed.

See also: inWhatsThisMode() and leaveWhatsThisMode().

**bool QWhatsThis::inWhatsThisMode ()** `[static]`

Returns whether the application is in What's This mode.

See also: enterWhatsThisMode() and leaveWhatsThisMode().

**void QWhatsThis::leaveWhatsThisMode ( const QString & text = QString::null, const QPoint & pos = QCursor::pos ())** `[static]`

Leaves What's This? question mode

This function is used internally by widgets that support QWidget::customWhatsThis(), applications usually never have to call it. An example for such a kind of widget is QPopupMenu: Menus still work normally in What's This mode, but provide help texts for single menu items instead.

If *text* is not a null string, then a What's This help window is displayed at the global position *pos* of the screen.

See also: inWhatsThisMode() and enterWhatsThisMode().

**void QWhatsThis::remove ( QWidget * widget )** `[static]`

Removes the What's This help for *widget.*

See also: add().

**QString QWhatsThis::text ( const QPoint & )** `[virtual]`

This virtual functions returns the text for position *p* in the widget this What's This object documents. If there is no What's This text for a position, QString::null may be returned.

The default implementation returns QString::null.

See also: qml().

**QString QWhatsThis::textFor ( QWidget * widget, const QPoint & pos = QPoint ())** `[static]`

Returns the text for *widget,* or a null string 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.

### 239.2.1. URL

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

# 240. QWheelEvent Class Reference

```
#include <qevent.h>
```

Inherits QEvent.

### 240.1. Detailed Description

The QWheelEvent class contains parameters that describe a wheel event.

Wheel events occur when a mouse wheel is turned while the widget has focus. The rotation distance is provided by delta(). If required, both pos() and globalPos() return the mouse pointer location at the time of the event.

A wheel event contains a special accept flag which tells whether the receiver wants the event. You should call QWheelEvent::accept() if you handle the wheel event, otherwise it will be sent to the parent widget as well.

The QWidget::setEnable() function can be used to enable or disable mouse and keyboard events for a widget.

The event handlers QWidget::wheelEvent() receive wheel events.

See also: QMouseEvent and QWidget::grabMouse().

### 240.2. Member Function Documentation

**QWheelEvent::QWheelEvent ( const QPoint & pos, const QPoint & globalPos, int delta, int state )**

Constructs a wheel event object.

**QWheelEvent::QWheelEvent ( const QPoint & pos, int delta, int state )**

Constructs a wheel event object.

**void QWheelEvent::accept ()**

Sets the accept flag of the wheel event object.

Setting the accept parameter indicates that the receiver of the event wants the wheel event.

Unwanted wheel events are sent to the parent widget.

The accept flag is set by default.

See also: ignore().

**int QWheelEvent::delta () const**

Returns the distance that the wheel is rotated, expressed in multiples or divisions of WHEEL_DELTA, which is set at 120 currently.A positive value indicates that the wheel was rotated forward, away from the user; a negative value indicates that the wheel was rotated backward, toward the user.

The WHEEL_DELTA constant was set to 120 by the wheel mouse vendors to allow building finer-resolution wheels in the future, including perhaps a freely-rotating wheel with no notches. The expectation is that such a device would send more messages per rotation, but with a smaller value in each message.

**const QPoint & QWheelEvent::globalPos () const**

Returns the global position of the mouse pointer *at the time* of the event. This is important on asynchronous window systems like X11: Whenever you move your widgets around in response to mouse evens, globalPos() can differ a lot from the current pointer position QCursor::pos().

See also: globalX() and globalY().

**int QWheelEvent::globalX () const**

Returns the global X position of the mouse pointer at the time of the event

See also: globalY() and globalPos().

**int QWheelEvent::globalY () const**

Returns the global Y position of the mouse pointer at the time of the event

See also: globalX() and globalPos().

**void QWheelEvent::ignore ()**

Clears the accept flag parameter of the wheel event object.

Clearing the accept parameter indicates that the event receiver does not want the wheel event.

Unwanted wheel events are sent to the parent widget.

The accept flag is set by default.

See also: accept().

**bool QWheelEvent::isAccepted () const**

Returns TRUE if the receiver of the event handles the wheel event.

**const QPoint & QWheelEvent::pos () const**

Returns the position of the mouse pointer, relative to the widget that received the event.

See also: x() and y().

**ButtonState QWheelEvent::state () const**

Returns the keyboard modifier flags.

The returned value is `ShiftButton, ControlButton` and `AltButton` OR'ed together.

**int QWheelEvent::x () const**

Returns the X position of the mouse pointer, relative to the widget that received the event.

See also: y() and pos().

**int QWheelEvent::y () const**

Returns the Y position of the mouse pointer, relative to the widget that received the event.

See also: x() and pos().

### 240.2.1. URL

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

# 241. QWidget Class Reference

```
#include <qwidget.h>
```

Inherits QObject and QPaintDevice.

Inherited by QButton, QComboBox, QDialog, QFrame, QGLWidget, QHeader, QLineEdit, QMainWindow, QNPWidget, QScrollBar, QSemiModal, QSizeGrip, QSlider, QStatusBar, QTabBar, QTabWidget, QToolBar and QXtWidget.

### 241.1. 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 that isn't embedded in a parent widget is called a top-level widget. Usually, top-level widgets are windows with a frame and a title bar (though it is also possible to create top level widgets without such decoration by the use of widget flags). In Qt, QMainWindow and the various subclasses of QDialog are the most common top-level windows.

A widget without a parent widget is always a top-level widget.

The opposite of top-level widgets are child widgets. Those are child windows in their parent widgets. You usually cannot distinguish a child widget from its parent visually. Most other widgets in Qt are useful only as child widgets. (You *can* make a e.g. button into a top-level widget, but most people prefer to put their buttons in e.g. dialogs.)

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(), showMinimized().
- Window contents: update(), repaint(), erase(), scroll(), updateMask().
- Geometry: pos(), size(), rect(), x(), y(), width(), height(), sizePolicy(), sizeHint(), updateGeometry(), layout(), move(), resize(), setGeometry(), frameGeometry(), geometry(), childrenRect(), adjustSize(), mapFromGlobal(), mapFromParent() mapToGlobal(), mapToParent(), maximumSize(), minimumSize(), sizeIncrement(), setMaximumSize(), setMinimumSize(), setSizeIncrement(), setBaseSize(), setFixedSize()
- Mode: isVisible(), isVisibleTo(), isVisibleToTLW(), visibleRect(), isMinimized(), 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(), backgroundColor(), 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(), wheelEvent(), enterEvent(), leaveEvent(), paintEvent(), moveEvent(), resizeEvent(), closeEvent(), dragEnterEvent(), dragMoveEvent(), dragLeaveEvent(), dropEvent(), childEvent(),

showEvent(), hideEvent(), customEvent().

- Change handlers: backgroundColorChange(), backgroundPixmapChange(), enabledChange(), fontChange(), paletteChange(), styleChange().
- System functions: parentWidget(), topLevelWidget(), reparentolish((), winId(), find(), metric().
- Internal kernel functions: setFRect(), setCRect(), focusNextPrevChild(), wmapper(), clearWFlags(), getWFlags(), setWFlags(), testWFlags().
- What's this help: customWhatsThis()

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 (Unless you specify `WType_TopLevel` as widget flag).
- `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().
- 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 trigger a different action.)

If your widget only contains child widgets, you probably do not need to implement any event handlers.

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().
- wheelEvent() -- called whenever the user turns the mouse wheel while the widget has the focus.
- 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 almost always useful to reimplement sizePolicy or at least sizeHint(), so users of your class can set up layout management more easily.

sizePolicy() lets you supply good defaults for the layout management handling, so that other widgets can contain yours easily. sizeHint() indicates a "good" size for the widget.

- 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:

tabdialog/tabdialog.cpp
drawlines/connect.cpp
mainlyQt/editor.cpp
xform/xform.cpp
qiconview/main.cpp
layout/layout.cpp
popup/popup.cpp
menu/menu.cpp
progress/progress.cpp
qmag/qmag.cpp
splitter/splitter.cpp
forever/forever.cpp
desktop/desktop.cpp
scrollview/scrollview.cpp
customlayout/main.cpp

## 241.2. Member Type Documentation

### QWidget::BackgroundMode

This enum describes how the background of a widget changes, as the widget's palette changes.

The background is what the widget contains when paintEvent() is called. To minimize flicker, this should be the most common color in the widget.

The following values are valid:

- `PaletteForeground` - use palette() . fillForeground()
- `PaletteBackground` - use palette() . fillBackground()
- `PaletteButton` - use palette() . fillButton()
- `PaletteLight` - use palette() . fillLight()
- `PaletteMidlight` - use palette() . fillMidlight()
- `PaletteDark` - use palette() . fillDark()
- `PaletteMid` - use palette() . fillMid()
- `PaletteText` - use palette() . fillText()
- `PaletteBrightText` - use palette() . fillBrightText()
- `PaletteButtonText` - use palette() . fillButtonText()
- `PaletteBase` - use palette() . fillBase()
- `PaletteShadow` - use palette() . fillShadow()
- `PaletteHighlight` - use palette() . fillHighlight()
- `PaletteHighlightedText` - use palette() . fillHighlightedText()
- `NoBackground` - no color or pixmap is used - the paintEvent() must completely cover the drawing area. This can help avoid flicker.

See also: setBackgroundMode() and backgroundMode().

**QWidget::PropagationMode**

This enum determines how fonts and palette changes are propagated to children of a widget.

If a widget's propagation mode is `NoChildren,` changes to that widget's font or palette do not affect the widget's children. IF it is `AllChildren,` all children are affected. If it is `SameFont` or `SamePalette` (mostly the default mode) then changes affect those child widget for which no separate font/palette has been set, but no other children.

See also: setFont() and setPalette().

### 241.3. 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.* Unless the newly created widget is reparented, it will be deleted when the parent is deleted.

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. A tool window is a small window that lives for a short time and it is typically used for creating popup windows. It there is a parent, the tool window will be kept on top of it. If there isn't a parent, you may consider passing WStyle_StaysOnTop as well. If the window system supports it, a tool window will be decorated with a somewhat lighter frame. It can, however, be combined with `WStyle_NoBorder` as well.
- `WStyle_StaysOnTop` informs the window system that the window should stay on top of all other windows.
- `WStyle_Dialog` indicates that the window is a logical subwindow of its parent, in other words: a dialog. The window will not get its own taskbar entry and be kept on top of its parent by the window system. Usually, it will also be minimized when the parent is minized. If not customized, the window is decorated slightly less. WStyle_Dialog is implied by WType_Modal. It is implicitely defined when using the class QDialog.

Note that the X11 version of Qt may not be able to deliver all combinations of style flags on all systems. This is because on X11, Qt can only ask the window manager, and the window manager can override the application's settings. On Windows, Qt can set whatever flags you want.

Example:

```
QLabel *toolTip = new QLabel( 0, "myToolTip",
                              WStyle_Customize | WStyle_NoBorder |
                              WStyle_Tool );
```

The widget flags are defined in qnamespace.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.

**bool QWidget::autoMask () const**

Returns whether or not a widget has the auto mask feature enabled.

See also: setAutoMask(), updateMask(), setMask() and clearMask().

**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

**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().

**QSize QWidget::baseSize () const**

Returns the widget base size

The base size is used to calculate a proper widget size in case the widget defines sizeIncrement().

See also: setBaseSize() and setSizeIncrement().

**QString QWidget::caption () const**

Returns the widget caption, or a null string 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.

**QRegion QWidget::childrenRegion () const**

Returns the combined region of the widget's children geometry().

**void QWidget::clearFocus ()** `[slot]`

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.

**void QWidget::clearWState ( uint n )** `[protected]`

For internal use only.

**bool QWidget::close ()** `[slot]`

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.

The QApplication::lastWindowClosed() signal is emitted when the last visible top level widget is closed.

See also: close(bool).

Examples:

popup/popup.cpp

**bool QWidget::close ( bool alsoDelete )** `[virtual]`

Closes this widget. Returns TRUE if the widget was closed, otherwise FALSE.

If *alsoDelete* is TRUE or the widget has the `WDestructiveClose` widget flag, the widget is also deleted. Otherwise, the widget can prevent itself from being closed by rejecting the QCloseEvent it gets.

The QApplication::lastWindowClosed() signal is emitted when the last visible top level widget is closed.

Note that closing the QApplication::mainWidget() terminates the application.

See also: closeEvent(), QCloseEvent, hide(), QApplication::quit(), QApplication::setMainWidget() and QApplication::lastWindowClosed().

**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 QPopupMenu, QDialog and QProgressDialog.

**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().

**void QWidget::create ( WId window = 0, bool initializeWindow = TRUE, bool destroyOldWindow = TRUE )** `[virtual 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. If no cursor has been set the parent widget's cursor is returned.

See also: setCursor() and unsetCursor();.

**void QWidget::customEvent ( QCustomEvent * )** `[virtual protected]`

This event handler can be reimplemented in a subclass to receive 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.**

See also: event() and QCustomEvent.

**bool QWidget::customWhatsThis () const** `[virtual]`

Returns whether the widget wants to handle What's This help manually. The default implementation returns FALSE, which means the widget will not receive any events in Whats This mode.

The widget may leave Whats This mode by calling QWhatsThis::leaveWhatsThisMode(), with or without actually displaying any help text.

You may also reimplement customWhatsThis() if your widget is a so-called "passive interactor" that is supposed to work under all circumstances. Simply never call QWhatsThis::leaveWhatsThisMode() in that case.

See also: QWhatsThis::inWhatsThisMode() and QWhatsThis::leaveWhatsThisMode().

Reimplemented in QMenuBar and QPopupMenu.

**void QWidget::destroy ( bool destroyWindow = TRUE, bool destroySubWindows = TRUE )** `[virtual 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::dragEnterEvent ( QDragEnterEvent * )** `[virtual protected]`

This event handler is called when a drag is in progress and the mouse enters this widget.

See the Drag-and-drop documentation for an overview of how to provide drag-and-drop in your application.

See also: QTextDrag, QImageDrag and QDragEnterEvent.

**void QWidget::dragLeaveEvent ( QDragLeaveEvent * )** `[virtual protected]`

This event handler is called when a drag is in progress and the mouse leaves this widget.

See the Drag-and-drop documentation for an overview of how to provide drag-and-drop in your application.

See also: QTextDrag, QImageDrag and QDragLeaveEvent.

Reimplemented in QMultiLineEdit.

**void QWidget::dragMoveEvent ( QDragMoveEvent * )** `[virtual protected]`

This event handler is called when a drag is in progress and the mouse enters this widget, and whenever it moves within the widget.

See the Drag-and-drop documentation for an overview of how to provide drag-and-drop in your application.

See also: QTextDrag, QImageDrag and QDragMoveEvent.

Reimplemented in QMultiLineEdit.

**void QWidget::drawText ( int x, int y, const QString & 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 QString & 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::dropEvent ( QDropEvent * )** `[virtual protected]`

This event handler is called when the drag is dropped on this widget.

See the Drag-and-drop documentation for an overview of how to provide drag-and-drop in your application.

See also: QTextDrag, QImageDrag and QDropEvent.

Reimplemented in QMultiLineEdit.

**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 isEnabled().

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 repaints the visible part of the widget.

See also: setEnabled(), isEnabled(), repaint(), update() and visibleRect().

Reimplemented in QListView 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.

See also: leaveEvent(), mouseMoveEvent() and event().

Reimplemented in QToolButton and QXtWidget.

**void QWidget::erase ( const QRegion & reg )**

Erases the area defined by *reg,* without generating a paint event.

Child widgets are not affected.

Examples:

drawlines/connect.cpp

**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().

**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. It also calls setMicroFocusHint(), hinting any system-specific input tools about the focus of the user's attention.

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 QPushButton, QListBox, QButton, QXtWidget, QMultiLineEdit, QLineEdit, QSlider, QComboBox, QListView and QRadioButton.

**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 QButton and 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( visibleRect() ) since the widget's color group changes from active to normal. You may want to call repaint( visibleRect(), FALSE) to reduce flicker in any reimplementation.

See also: focusInEvent(), setFocusPolicy(), keyPressEvent(), keyReleaseEvent(), event() and QFocusEvent.

Reimplemented in QListBox, QButton, QMenuBar, QMultiLineEdit, QLineEdit, QPushButton, QSlider, QXtWidget and QListView.

**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.

**QFont QWidget::font () const**

Returns the font currently set for the widget.

fontInfo() tells you what font is actually being used.

As long as no special font has been set, this is either a special font for the widget class or the default application font.

See also: setFont(), fontInfo(), fontMetrics() and QApplication::font().

Examples:

grapher/grapher.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 QMenuBar.

**QFontInfo QWidget::fontInfo () const**

Returns the font info for the widget's current font. Equivalent to QFontInto(widget->font()).

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

**QFontMetrics QWidget::fontMetrics () const**

Returns the font metrics for the widget's current font. Equivalent to QFontMetrics(widget->font()).

See also: font(), fontInfo() and setFont().

Examples:

xform/xform.cpp
drawdemo/drawdemo.cpp
qmag/qmag.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().

**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().

**QSize QWidget::frameSize () const**

Returns the size of the window system frame (for top level widgets).

**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.

**uint QWidget::getWState () 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().

**void QWidget::grabMouse ( const QCursor & cursor )**

Grabs the mouse input 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().

Examples:

qmag/qmag.cpp

**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
xform/xform.cpp
drawdemo/drawdemo.cpp
qmag/qmag.cpp

**int QWidget::heightForWidth ( int w ) const** `[virtual]`

Returns the preferred height for this widget, given the width *w*. The default implementation returns 0, indicating that the preferred height does not depend on the width.

**Warning:** Does not look at the widget's layout.

Reimplemented in QMenuBar, QLabel and QTextView.

**void QWidget::hide ()** `[virtual slot]`

Hides the widget.

You almost never have to reimplement this function. If you need to do something after a widget is

hidden, use \link hideEvent() instead.

See also: \hideEvent(), show(), showMinimized(), isVisible() and close().

Examples:

xform/xform.cpp
popup/popup.cpp
progress/progress.cpp

scrollview/scrollview.cpp

Reimplemented in QPopupMenu and QMenuBar.

**void QWidget::hideEvent ( QHideEvent * )** `[virtual protected]`

This event handler can be reimplemented in a subclass to receive widget hide events.

Hide events are sent to widgets right after they have been hidden.

See also: event() and QHideEvent.

**const QPixmap * QWidget::icon () const**

Returns the widget icon pixmap, or null if no icon has been set.

See also: setIcon(), iconText() and caption().

**QString QWidget::iconText () const**

Returns the widget icon text, or a null string if no icon text has been set.

See also: setIconText(), icon() and caption().

**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::isMinimized () const**

Returns TRUE if this widget is a top-level widget that is minimized (iconified), or else FALSE.

See also: showMinimized(), isVisible(), show(), hide() and showNormal().

**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.

A top-level widgets can have a parent widget. It will then be grouped with its parent: deleted when the parent is deleted, minimized when the parent is minimized etc. If supported by the window manager, it will also have a common taskbar entry with its parent.

QDialog and QMainWindow widgets are by default top-level, even if a parent widget is specified in the constructor. This behavior is specified by the `WType_TopLevel` widget flag.

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, as well as having isMinimized() return TRUE.

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(), isVisibleToTLW() and isMinimized().

**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 closes popup widgets if you hit escape. Otherwise the event is ignored.

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 QComboBox, QButton, QTextBrowser, QSlider, QDialog, QMultiLineEdit, QLineEdit, QTextView, QMessageBox, QMenuBar, QListBox, QListView, QPopupMenu, QScrollBar, QFileDialog and QTabBar.

**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().

**QLayout* QWidget::layout () const**

Returns a pointer to the layout engine that manages the geometry of this widget's children.

If the widget does not have a layout, layout() returns a null pointer.

See also: sizePolicy().

**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.

See also: enterEvent(), mouseMoveEvent() and event().

Reimplemented in QToolButton, QMenuBar, QXtWidget, QMultiLineEdit and QLineEdit.

**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().

Examples:

popup/popup.cpp

**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().

**int QWidget::maximumHeight () const**

Returns the widget's maximum height.

See also: maximumSize() and maximumWidth().

**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: maximumWidth(), maximumHeight(), setMaximumSize(), minimumSize() and sizeIncrement().

**int QWidget::maximumWidth () const**

Returns the widget's maximum width.

See also: maximumSize() and maximumHeight().

**int QWidget::metric ( int m ) const** `[virtual protected]`

Internal implementation of the virtual QPaintDevice::metric() function.

Use the QPaintDeviceMetrics class instead.

Reimplemented from QPaintDevice.

**int QWidget::minimumHeight () const**

Returns the widget's minimum height.

See also: minimumSize() and minimumWidth().

**QSize QWidget::minimumSize () const**

Returns the minimum widget size.

The widget cannot be resized to a smaller size than the minimum widget size.

If the returned minimum size equals (0,0) then it means that there are no constraints on the minimum size. However, Qt does nevertheless not allow you to shrink widgets to less than 1 pixel width/height.

See also: maximumWidth(), maximumHeight(), setMinimumSize(), maximumSize() and sizeIncrement().

**QSize QWidget::minimumSizeHint () const** `[virtual]`

Returns a recommended minimum size for the widget, or an invalid size if no minimum size is recommended.

The default implementation returns an invalid size if there is no layout for this widget, the layout's minimum size otherwise.

See also: QSize::isValid(), resize(), setMinimumSize() and sizePolicy().

Reimplemented in QWidgetStack, QSplitter, QLineEdit, QMultiLineEdit, QSlider, QProgressBar, QTabWidget, QListBox, QListView and QScrollView.

**int QWidget::minimumWidth () const**

Returns the widget's minimum width.

See also: minimumSize() and minimumHeight().

**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 QComboBox, QMultiLineEdit, QLineEdit and QListBox.

**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.

See also: setMouseTracking(), mousePressEvent(), mouseReleaseEvent(), mouseDoubleClickEvent(), event() and QMouseEvent.

Reimplemented in QMenuBar, QListBox, QSlider, QScrollBar, QButton, QPopupMenu, QMultiLineEdit, QSizeGrip, QLineEdit and QComboBox.

**void QWidget::mousePressEvent ( QMouseEvent * e )** `[virtual protected]`

This event handler can be reimplemented in a subclass to receive mouse press events for the widget.

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.

The default implementation implements the closing of popup widgets when you click outside the window. For other widget types it does nothing.

See also: mouseReleaseEvent(), mouseDoubleClickEvent(), mouseMoveEvent(), event() and QMouseEvent.

Reimplemented in QListBox, QScrollBar, QLineEdit, QButton, QSlider, QTabBar, QComboBox, QSizeGrip, QPopupMenu, QMenuBar and QMultiLineEdit.

**void QWidget::mouseReleaseEvent ( QMouseEvent * )** `[virtual protected]`

This event handler can be reimplemented in a subclass to receive mouse release events for the widget.

See also: mouseReleaseEvent(), mouseDoubleClickEvent(), mouseMoveEvent(), event() and QMouseEvent.

Reimplemented in QScrollBar, QButton, QPopupMenu, QTabBar, QSlider, QListBox, QMenuBar, QLineEdit, QMultiLineEdit 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:

xform/xform.cpp
drawdemo/drawdemo.cpp
popup/popup.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().

See also: resizeEvent(), event(), move() and QMoveEvent.

Reimplemented in QXtWidget.

**void QWidget::paintEvent ( QPaintEvent * )** `[virtual protected]`

This event handler can be reimplemented in a subclass to receive widget paint events.

When the paint event occurs, the update region QPaintEvent::region() normally has been cleared to the background color or pixmap. An exception is when repaint(FALSE) is called. Inside the paint event handler, QPaintEvent::erased() carries this information.

For many widgets it is sufficient to redraw the entire widget each time, but some need to consider the update rectangle or region of the QPaintEvent to avoid slow update.

During paintEvent(), any QPainter you create on the widget will be clipped to at most the area covered by the update region.

update() and repaint() can be used to force a paint event.

See also: event(), repaint(), update(), QPainter, QPixmap and QPaintEvent.

Reimplemented in QTabDialog, QScrollBar, QSizeGrip, QButton, QToolBar, QLineEdit, QFrame, QGLWidget, QMainWindow, QStatusBar, QTabBar, QPopupMenu, QGroupBox, QSlider, QTableView, QHeader and QComboBox.

**const QPalette & QWidget::palette () const**

Returns the widget palette.

As long as no special palette has been set, this is either a special palette for the widget class, the palette of the parent widget or the default application 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() and palette().

Reimplemented in QTextView.

**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.

**void QWidget::polish ()** `[virtual slot]`

Delayed initialization of a widget.

This function will be called *after* a widget has been fully created just *before* it is shown the very first time.

Polishing is useful for final initialization depending on an instantiated widget. This is something a constructor cannot guarantee since the initialization of the subclasses might not be finished.

The default implementation sets the `WState_Polished` widget state and calls QApplication::polish().

See also: QApplication::polish().

**QPoint QWidget::pos () const**

Returns the position 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().

**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
menu/menu.cpp
desktop/desktop.cpp
picture/picture.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 ( const QRegion & reg, bool erase=TRUE )** `[slot]`

Repaints the widget directly by calling paintEvent() directly, unless updates are disabled or the widget is hidden.

Erases the widget region *reg* if *erase* is TRUE.

Calling repaint() is usually 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().

**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() is usually 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:

xform/xform.cpp
qmag/qmag.cpp
forever/forever.cpp

**void QWidget::repaint ()** `[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 erases and repaints the entire widget.

**void QWidget::repaint ( bool erase )** `[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.

**void QWidget::reparent ( QWidget * parent, WFlags f, const QPoint & p, bool showIt=FALSE )** `[virtual]`

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 reparented.

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, reparent() calls clearFocus() for that widget.

If the new parent widget is in the same top-level widget as the old parent, reparent doesn't change the TAB order or keyboard focus.

**Warning:** Reparenting widgets should be a real exception. In normal applications, you will almost never need it. Dynamic masks can be achieved much easier and cleaner with classes like QWidgetStack or on a higher abstraction level, QWizard.

See also: getWFlags().

**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:

aclock/main.cpp
xform/xform.cpp
drawdemo/drawdemo.cpp
qmag/qmag.cpp
qwerty/main.cpp
rot13/rot13.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:

rangecontrols/main.cpp
themes/main.cpp
listviews/main.cpp
checklists/main.cpp
buttons_groups/main.cpp
dclock/main.cpp
xform/xform.cpp
mainlyXt/editor.cpp
i18n/main.cpp
drawdemo/drawdemo.cpp
lineedits/main.cpp
popup/popup.cpp
scribble/main.cpp
qmag/qmag.cpp
progressbar/main.cpp
mainlyMotif/editor.cpp
richtext/main.cpp
scrollview/scrollview.cpp
qfd/qfd.cpp
listbox_combo/main.cpp
qfileiconview/main.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(), though.

The widget will be erased and receive a paint event immediately after processing the resize event.

No drawing has to (and should) be done inside this handler.

Widgets that have been created with the `WResizeNoErase` flag will not be erased. Nevertheless, they will receive a paint event for their entire area afterwards. Again, no drawing needs to be done inside this handler.

The default implementation calls updateMask() if the widget has automatic masking enabled.

See also: moveEvent(), event(), resize(), QResizeEvent and paintEvent().

Reimplemented in QScrollView, QFileDialog, QScrollBar, QMultiLineEdit, QListView, QSlider, QMenuBar, QMainWindow, QWidgetStack, QTabDialog, QGLWidget, QComboBox, QTableView, QListBox, QGroupBox, QLineEdit, QSpinBox, QFrame, QXtWidget, QProgressDialog, QMessageBox and QTextView.

**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.

**Warning:** You might find that QScrollView offers a higher-level of functionality than using this function.

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::scroll ( int dx, int dy, const QRect & r )**

Scrolls the area *r* of this widget *dx* pixels rightwards and *dy* pixels downwards. If *dx/dy* is negative, the scroll direction is leftwards/upwards. Child widgets are not moved.

**Warning:** You might find that QScrollView offers a higher-level of functionality than using this function.

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 )** `[virtual]`

Announces to the system that this widget *may* be able to accept drop events.

If the widgets is the desktop, this may fail if another application is using the desktop - you can call acceptDrops() to test if this occurs.

See also: acceptDrops().

Examples:

scrollview/scrollview.cpp

**void QWidget::setActiveWindow ()** `[virtual]`

Sets the top-level widget containing this widget to be the active window.

An active window is a visible 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, at least on Windows. On X11, the result depends on the Window Manager. If you want to ensure that the window is stacked on top as well, call raise() in addition. Note that the window has be to visible, otherwise setActiveWindow() has no effect.

See also: isActiveWindow(), topLevelWidget() and show().

**void QWidget::setAutoMask ( bool enable )** `[virtual]`

Transparent widgets use a mask to define their visible region. QWidget has some built-in support to make the task of recalculating the mask easier. When setting auto mask to TRUE, updateMask() will be called whenever the widget is resized or changes its focus state.

Note: When you re-implement resizeEvent(), focusInEvent() or focusOutEvent() in your custom widgets and still want to ensure that the auto mask calculation works, you will have to add

```
if ( autoMask() )
    updateMask();
```

at the end of your event handlers. Same holds for all member functions that change the appearance of the widget in a way that a recalculation of the mask is necessary.

See also: autoMask(), updateMask(), setMask() and clearMask().

Examples:

aclock/main.cpp

**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:

drawlines/connect.cpp
xform/xform.cpp
drawdemo/drawdemo.cpp
splitter/splitter.cpp
desktop/desktop.cpp

hello/main.cpp
customlayout/main.cpp

**void QWidget::setBackgroundMode ( BackgroundMode m )** `[virtual]`

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 fill functions of the colorgroup returns Brushes, which may be either a plain color or a pixmap.

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.

**void QWidget::setBackgroundPixmap ( const QPixmap & pixmap )** `[virtual]`

Sets the background pixmap of the widget to *pixmap*.

This function is deprecated. Use setBackgroundMode() or setPalette(), as they ensure the appropriate clearing pixmap or color is used when the widget is in the Active, Normal, or Disabled

state.

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::setBaseSize ( int basew, int baseh )**

Sets the base size of the widget. The base size is important only in combination with size increments. See setSizeIncrement() for details.

See also: baseSize().

**void QWidget::setBaseSize ( const QSize & )**

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

**void QWidget::setCRect ( const QRect & r )** `[virtual protected]`

For internal use only.

**void QWidget::setCaption ( const QString & caption )** `[virtual slot]`

Sets the window caption (title).

See also: caption(), setIcon() and setIconText().

Examples:

qdir/qdir.cpp
rangecontrols/main.cpp
validator/main.cpp
themes/main.cpp
listviews/main.cpp
checklists/main.cpp
buttons_groups/main.cpp
wizard/main.cpp
application/main.cpp
qiconview/main.cpp
drawdemo/drawdemo.cpp

lineedits/main.cpp
popup/popup.cpp
listbox/main.cpp
progress/progress.cpp
progressbar/main.cpp
richtext/main.cpp
tooltip/main.cpp
forever/forever.cpp
addressbook/main.cpp
movies/main.cpp
listbox_combo/main.cpp
hello/main.cpp
dirview/main.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(), unsetCursor() 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.

**void QWidget::setFRect ( const QRect & r )** `[virtual 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:

layout/layout.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.

**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 ()** `[virtual slot]`

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().

Examples:

tabdialog/tabdialog.cpp
xform/xform.cpp
popup/popup.cpp

**void QWidget::setFocusPolicy ( FocusPolicy policy )** `[virtual]`

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::WheelFocus`, like StrongFocus plus the widget accepts focus by using the mouse wheel.
- `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:

rot13/rot13.cpp

**void QWidget::setFocusProxy ( QWidget * w )** `[virtual]`

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 *SameChildren.*

See also: font(), fontChange(), fontInfo(), fontMetrics() and setFontPropagation().

Examples:

grapher/grapher.cpp

Reimplemented in QWizard, QTabDialog and QPopupMenu.

**void QWidget::setFont ( const QFont & font, bool fixed )**

Like setFont(const QFont&) but has an additional flag to indicate whether the font should be fix for the widget. Fixed means that QApplication::setFont() will not touch the current setting. This Function calls setFont(const QFont&).

Examples:

xform/xform.cpp
hello/main.cpp

**void QWidget::setFontPropagation ( PropagationMode m )** `[virtual]`

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:

xform/xform.cpp

cursor/cursor.cpp
drawdemo/drawdemo.cpp
popup/popup.cpp
qmag/qmag.cpp

Reimplemented in QDialog.

**void QWidget::setGeometry ( const QRect & )** `[virtual slot]`

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

Reimplemented in QDialog.

**void QWidget::setIcon ( const QPixmap & pixmap )** `[virtual slot]`

Sets the window icon pixmap.

See also: icon(), setIconText() and setCaption().

Reimplemented in QMessageBox.

**void QWidget::setIconText ( const QString & iconText )** `[virtual slot]`

Sets the text of the window's icon to *iconText.*

See also: iconText(), setIcon() and setCaption().

**void QWidget::setKeyCompression ( bool compress )** `[virtual protected]`

Enables key event compression. Per default, the compression is turned off. If you enable it, the widget receives compressed key press events whenever the user types too fast for your program. That means you may receive an entire word in the QKeyEvent::text() field of they QKeyEvent, instead of one event for each character. This makes sense for a word processor, for example, since it takes almost as much time to insert a single character as it takes to insert a full word, because the required recalculation of the layout of the paragraph is roughly the same.

If a widgets supports multiple byte unicode input, it is always safe (and also recommended!) to turn

the compression on.

See also: QKeyEvent::text();.

**void QWidget::setMask ( const QBitmap & bitmap )** `[virtual]`

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 )** `[virtual]`

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.

Examples:

splitter/splitter.cpp

**void QWidget::setMaximumSize ( int maxw, int maxh )** `[virtual]`

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().

**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::setMicroFocusHint ( int x, int y, int, int height, bool text=TRUE )** `[virtual protected]`

When a widget gets focus, it should call setMicroFocusHint for some appropriate position and size - *x, y* and *w* by *h.* This has no *visual* effect, it just provides hints to any system-specific input handling tools.

The *text* argument should be TRUE if this is a position for text input.

In the Windows version of Qt, this method sets the system caret, which is used for user Accessibility

focus handling. If *text* is TRUE, it also sets the IME composition window in Far East Asian

language input systems.

In the X11 version of Qt, if *text* is TRUE, this method sets the XIM "spot" point for complex

language input handling.

**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.

**void QWidget::setMinimumSize ( int minw, int minh )** `[virtual]`

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:

menu/menu.cpp
qmag/qmag.cpp

splitter/splitter.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 )** `[virtual 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:

popup/popup.cpp
qmag/qmag.cpp

**void QWidget::setName ( const char * name )** `[virtual]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QObject.

**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 *SameChildren.*

See also: QApplication::setPalette(), palette(), paletteChange(), colorGroup(),

setBackgroundColor() and setPalettePropagation().

**void QWidget::setPalette ( const QPalette & p, bool fixed )**

Like setPalette(const QPalette&) but has an additional flag to indicate whether the palette should be fix for the widget. Fixed means that QApplication::setPalette() will not touch the current setting. This Function calls setPalette(const QPalette&).

**void QWidget::setPalettePropagation ( PropagationMode m )** `[virtual]`

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::setSizeIncrement ( int w, int h )** `[virtual]`

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, with baseSize() as basis. Preferred widget sizes are therefore for non-negative integers *i* and *j:*

```
width = baseSize().width() + i * sizeIncrement().width();
height = baseSize().height() + j * sizeIncrement().height();
```

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.

**void QWidget::setStyle ( QStyle * style )**

Sets the widget's GUI style to *style.* Ownership of the style object is not transfered.

If no style is set, the widget uses the application's style QApplication::style() instead.

Setting a widget's style has no effect on existing or future childwidgets.

**Warning:** This function is particularly useful for demonstration purposes, where you want to show Qt's styling capabilities. Real appliations should stay away from it and use one consistent GUI style instead.

See also: style(), QStyle, QApplication::style() and QApplication::setStyle().

Examples:

grapher/grapher.cpp

**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 )** `[virtual 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 )** `[virtual protected]`

For internal use only.

**void QWidget::setWState ( uint n )** `[virtual 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.

You almost never have to reimplement this function. If you need to change some settings before a widget is shown, use showEvent() instead. If you need to do some delayed initialization use polish().

See also: showEvent, hide(), showMinimized(), isVisible() and polish().

Examples:

showimg/main.cpp
rangecontrols/main.cpp
validator/main.cpp
listviews/main.cpp
aclock/main.cpp
checklists/main.cpp
buttons_groups/main.cpp
drawlines/connect.cpp
dclock/main.cpp
xform/xform.cpp
cursor/cursor.cpp
layout/layout.cpp
tetrix/tetrix.cpp
life/main.cpp
drawdemo/drawdemo.cpp
lineedits/main.cpp
popup/popup.cpp
listbox/main.cpp
menu/menu.cpp
progress/progress.cpp
qmag/qmag.cpp
splitter/splitter.cpp
progressbar/main.cpp
richtext/main.cpp
tooltip/main.cpp
qwerty/main.cpp
forever/forever.cpp
rot13/rot13.cpp
scrollview/scrollview.cpp
movies/main.cpp
picture/picture.cpp
listbox_combo/main.cpp
hello/main.cpp
biff/main.cpp
tictac/main.cpp
customlayout/main.cpp

Reimplemented in QTableView, QListView, QTabBar, QDialog, QScrollView, QMenuBar,

QTabDialog, QWidgetStack, QPopupMenu, QToolBar, QWizard, QSemiModal and QMainWindow.

**void QWidget::showEvent ( QShowEvent * )** `[virtual protected]`

This event handler can be reimplemented in a subclass to receive widget show events.

Non-sponaneous show events are sent to widgets right before they are shown. Spontaneous show events of toplevel widgets are delivered afterwards, naturally.

See also: event() and QShowEvent.

Reimplemented in QListView, QTabWidget and QListBox.

**void QWidget::showMaximized ()** `[virtual slot]`

Shows the widget maximized.

Calling this function has no effect for other than top-level widgets.

See also: showNormal(), showMinimized(), show(), hide() and isVisible().

**void QWidget::showMinimized ()** `[virtual slot]`

Shows the widget minimized, as an icon.

Calling this function has no effect for other than top-level widgets.

See also: showNormal(), showMaximized(), show(), hide(), isVisible() and isMinimized().

**void QWidget::showNormal ()** `[virtual slot]`

Restores the widget after it has been maximized or minimized.

Calling this function has no effect for other than top-level widgets.

See also: showMinimized(), showMaximized(), show(), hide() and isVisible().

**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 if there is no layout for this widget, the layout's preferred size otherwise.

See also: QSize::isValid(), minimumSizeHint(), sizePolicy(), setMinimumSize() and updateGeometry().

Examples:

lineedits/main.cpp

Reimplemented in QHeader, QPopupMenu, QToolButton, QSlider, QListView, QTabBar, QLabel, QCheckBox, QTabWidget, QProgressDialog, QLineEdit, QWidgetStack, QLCDNumber, QPushButton, QMultiLineEdit, QScrollBar, QSplitter, QRadioButton, QListBox, QSizeGrip, QComboBox, QSpinBox and QProgressBar.

**QSize QWidget::sizeIncrement () const**

Returns the widget size increment.

See also: setSizeIncrement(), minimumSize() and maximumSize().

**QSizePolicy QWidget::sizePolicy () const** `[virtual]`

This function can be reimplemented in subclasses to specify the default layout behaviour of that subclass.

If there is a QLayout that manages this widget's children, the size policy specified by that layout is used. If there is no such QLayout, the result of this function is used.

The default implementation returns a value which means that the widget can be freely resized, but prefers to be the size sizeHint() returns.

See also: sizeHint(), QLayout, QSizePolicy and updateGeometry().

Reimplemented in QLabel, QScrollBar, QSlider, QPushButton, QTabBar, QToolButton, QLCDNumber, QFrame, QLineEdit, QHeader, QCheckBox, QSpinBox, QMultiLineEdit, QProgressBar, QScrollView, QRadioButton, QSplitter and QSizeGrip.

**QStyle& QWidget::style () const**

Returns the GUI style for this widget

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

**void QWidget::styleChange ( QStyle & oldStyle )** `[virtual protected]`

This virtual function is called when the style of the widgets. changes.*oldStyle* is the previous GUI 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: QApplication::setStyle() and style().

Reimplemented in QPopupMenu, QComboBox, QSplitter, QListView, QProgressDialog,

QProgressBar, QTabWidget, QMenuBar, QScrollBar, QMainWindow, QSpinBox and QTabDialog.

**bool QWidget::testWFlags ( WFlags n ) const**

Returns TRUE if any of the widget flags in *n* are set. The widget flags are listed in qnamespace.h, and are strictly for internal use.

**bool QWidget::testWState ( uint n ) const**

For internal use only.

**QWidget * QWidget::topLevelWidget () const**

Returns the top-level widget for this widget, i.e. the parent widget that provides the overlapping window.

See also: isTopLevel().

**void QWidget::unsetCursor ()** `[virtual]`

Unset the cursor for this widget. The widget will use the cursor of its parent from now on.

This functions does nothing for toplevel windows.

See also: cursor(), setCursor() and QApplication::setOverrideCursor().

**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:

drawlines/connect.cpp
drawdemo/drawdemo.cpp
desktop/desktop.cpp
picture/picture.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
scrollview/scrollview.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.

**void QWidget::updateGeometry ()**

Notifies the layout system that this widget has changed and may need to change geometry.

Call this function if the sizeHint() or sizePolicy() have changed.

**void QWidget::updateMask ()** `[virtual protected]`

This function can be reimplemented in a subclass to support transparent widgets. It is supposed to be called whenever a widget changes state in a way that the shape mask has to be recalculated.

See also: setAutoMask(), updateMask(), setMask() and clearMask().

Reimplemented in QTabWidget, QSlider, QFrame and QTabBar.

**QRect QWidget::visibleRect () const**

Returns the currently visible rectangle of the widget. This function is in particular useful to optimize immediate repainting of a windget. Typical usage is

```
repaint( w->visibleRect() );
```

or

```
repaint( w->visibleRect(), FALSE );
```

If nothing is visible, the rectangle returned is empty.

**void QWidget::wheelEvent ( QWheelEvent * e )** `[virtual protected]`

This event handler can be reimplemented in a subclass to receive wheel events for the widget.

If you reimplement this handler, it is very important that you ignore() the event if you do not handle

it, so that the widget's parent can interpret it.

The default implementation ignores the event.

See also: QWheelEvent::ignore(), QWheelEvent::accept(), event() and QWheelEvent.

Reimplemented in QScrollBar, QSpinBox, QMultiLineEdit, QSlider and QScrollView.

**int QWidget::width () const**

Returns the width of the widget, excluding the window frame.

See also: geometry(), height() and size().

Examples:

grapher/grapher.cpp
xform/xform.cpp
drawdemo/drawdemo.cpp
menu/menu.cpp
qmag/qmag.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

### 241.3.1. URL

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

# 242. QWidgetItem Class Reference

`#include <qabstractlayout.h>`

Inherits QLayoutItem.

### 242.1. Detailed Description

A QLayoutItem that represents widget.

For custom layouts.

See also: QLayout.

### 242.2. Member Function Documentation

**QWidgetItem::QWidgetItem ( QWidget * w )**

Creates an item containing *w*.

**QRect QWidgetItem::geometry () const** `[virtual]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QLayoutItem.

**bool QWidgetItem::hasHeightForWidth () const** `[virtual]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QLayoutItem.

**int QWidgetItem::heightForWidth ( int w ) const** `[virtual]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QLayoutItem.

**bool QWidgetItem::isEmpty () const** `[virtual]`

Returns TRUE, if the widget has been hidden, FALSE otherwise.

Reimplemented from QLayoutItem.

**QSize QWidgetItem::maximumSize () const** `[virtual]`

Returns the maximum size of this item.

Reimplemented from QLayoutItem.

**QSize QWidgetItem::minimumSize () const** `[virtual]`

Returns the minimum size of this item.

Reimplemented from QLayoutItem.

**void QWidgetItem::setGeometry ( const QRect & r )** `[virtual]`

Sets the geometry of this item's widget to be contained within *r*, taking alignment and maximum size into account.

Reimplemented from QLayoutItem.

**QSize QWidgetItem::sizeHint () const** `[virtual]`

Returns the preferred size of this item.

Reimplemented from QLayoutItem.

**QWidget * QWidgetItem::widget ()** `[virtual]`

Returns the widget managed by this item.

### 242.2.1. URL

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

# 243. QWidgetStack Class Reference

```
#include <qwidgetstack.h>
```

Inherits QFrame.

## 243.1. 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 to 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.

## 243.2. 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 reparent().

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

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QObject.

**void QWidgetStack::frameChanged ()** `[virtual protected]`

Reimplemented for internal reasons; the API is not affected.

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().

**QSize QWidgetStack::minimumSizeHint () const** `[virtual]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QWidget.

**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::resizeEvent ( QResizeEvent * e )** `[virtual protected]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QWidget.

**void QWidgetStack::setChildGeometries ()** `[virtual protected]`

Fix up the children's geometries.

**void QWidgetStack::setFrameRect ( const QRect & r )** `[virtual]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QFrame.

**void QWidgetStack::show ()** `[virtual]`

Reimplemented in order to set the children's geometries appropriately.

Reimplemented from QWidget.

**QSize QWidgetStack::sizeHint () const** `[virtual]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QWidget.

**QWidget * QWidgetStack::visibleWidget () const**

Returns a pointer to the currently visible widget (the one on the top of the stack), or 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 returns 0.

See also: id() and addWidget().

### 243.2.1. URL

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

# 244. QWindowsStyle Class Reference

`#include <qwindowsstyle.h>`

Inherits QCommonStyle.

Inherited by QPlatinumStyle.

### 244.1. Detailed Description

Windows Look and Feel

This class implements the look and feel known from the Windows plattform. Naturally it is also Qt's default GUI style on Windows.

### 244.2. Member Function Documentation

**QWindowsStyle::QWindowsStyle ()**

Constructs a QWindowsStyle.

**QRect QWindowsStyle::comboButtonFocusRect ( int x, int y, int w, int h )** `[virtual]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QStyle.

**QRect QWindowsStyle::comboButtonRect ( int x, int y, int w, int h )** `[virtual]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QStyle.

**void QWindowsStyle::drawArrow ( QPainter * p, ArrowType type, bool down, int x, int y, int w, int h, const QColorGroup & g, bool enabled, const QBrush * fill = 0 )**

Reimplemented for internal reasons; the API is not affected.

**void QWindowsStyle::drawBevelButton ( QPainter * p, int x, int y, int w, int h, const QColorGroup & g, bool sunken = FALSE, const QBrush * fill = 0 )** `[virtual]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QStyle.

**void QWindowsStyle::drawButton ( QPainter * p, int x, int y, int w, int h, const QColorGroup & g, bool sunken = FALSE, const QBrush * fill = 0 )** `[virtual]`

Draws a press-sensitive shape.

Reimplemented from QStyle.

**void QWindowsStyle::drawCheckMark ( QPainter * p, int x, int y, int w, int h, const QColorGroup & g, bool act, bool dis )**

Reimplemented for internal reasons; the API is not affected.

**void QWindowsStyle::drawComboButton ( QPainter * p, int x, int y, int w, int h, const QColorGroup & g, bool sunken = FALSE, bool = FALSE, bool enabled = TRUE, const QBrush * fill = 0 )** `[virtual]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QStyle.

**void QWindowsStyle::drawExclusiveIndicator ( QPainter * p, int x, int y, int w, int h, const QColorGroup & g, bool on, bool down = FALSE, bool = TRUE )** `[virtual]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QStyle.

**void QWindowsStyle::drawExclusiveIndicatorMask ( QPainter * p, int x, int y, int w, int h, bool )** `[virtual]`

Draws the mask of a mark indicating the state of an exclusive choice.

Reimplemented from QStyle.

**void QWindowsStyle::drawFocusRect ( QPainter * p, const QRect & r, const QColorGroup &, const QColor * bg =0, bool = FALSE )** `[virtual]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QStyle.

**void QWindowsStyle::drawIndicator ( QPainter * p, int x, int y, int w, int h, const QColorGroup & g, int s, bool down = FALSE, bool = TRUE )** `[virtual]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QStyle.

**void QWindowsStyle::drawPanel ( QPainter * p, int x, int y, int w, int h, const QColorGroup & g, bool sunken=FALSE, int lineWidth = 1, const QBrush * fill = 0 )** `[virtual]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QStyle.

**void QWindowsStyle::drawPopupMenuItem ( QPainter * p, bool checkable, int maxpmw, int tab, QMenuItem * mi, const QPalette & pal, bool act, bool enabled, int x, int y, int w, int h )**

Reimplemented for internal reasons; the API is not affected.

**void QWindowsStyle::drawPopupPanel ( QPainter * p, int x, int y, int w, int h, const QColorGroup & g, int = 2, const QBrush * fill = 0 )** `[virtual]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QStyle.

**void QWindowsStyle::drawPushButton ( QPushButton * btn, QPainter * p )** `[virtual]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QStyle.

**void QWindowsStyle::drawScrollBarControls ( QPainter * p, const QScrollBar * sb, int sliderStart, uint controls, uint activeControl )** `[virtual]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QStyle.

**void QWindowsStyle::drawSlider ( QPainter * p, int x, int y, int w, int h, const QColorGroup & g, Orientation orient, bool tickAbove, bool tickBelow )** `[virtual]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QStyle.

**void QWindowsStyle::drawSliderGroove ( QPainter * p, int x, int y, int w, int h, const QColorGroup & g, QCOORD c, Orientation orient )** `[virtual]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QStyle.

**void QWindowsStyle::drawSliderMask ( QPainter * p, int x, int y, int w, int h, Orientation orient, bool tickAbove, bool tickBelow )** `[virtual]`

Draws the mask of a slider.

Reimplemented from QStyle.

**void QWindowsStyle::drawSplitter ( QPainter * p, int x, int y, int w, int h, const QColorGroup & g, Orientation )** `[virtual]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QStyle.

**void QWindowsStyle::drawTab ( QPainter * p, const QTabBar * tb, QTab * t, bool selected )** `[virtual]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QStyle.

**void QWindowsStyle::drawTabMask ( QPainter * p, const QTabBar * tb, QTab * t, bool selected )** `[virtual]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QStyle.

**void QWindowsStyle::drawWinShades ( QPainter * p, int x, int y, int w, int h, const QColor & c1, const QColor & c2, const QColor & c3, const QColor & c4, const QBrush * fill )** `[protected]`

This function draws a rectangle with two pixel line width. It is called from qDrawWinButton() and qDrawWinPanel().

c1..c4 and fill are used:

1 1 1 1 1 2 1 3 3 3 4 2 1 3 F F 4 2 1 3 F F 4 2 1 4 4 4 4 2 2 2 2 2 2 2.

**QSize QWindowsStyle::exclusiveIndicatorSize () const** `[virtual]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QStyle.

**int QWindowsStyle::extraPopupMenuItemWidth ( bool checkable, int maxpmw, QMenuItem * mi, const QFontMetrics & )** `[virtual]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QStyle.

**void QWindowsStyle::getButtonShift ( int & x, int & y )** `[virtual]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QStyle.

**QSize QWindowsStyle::indicatorSize () const** `[virtual]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QStyle.

**int QWindowsStyle::maximumSliderDragDistance () const** `[virtual]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QStyle.

**void QWindowsStyle::polishPopupMenu ( QPopupMenu * p )**

Reimplemented for internal reasons; the API is not affected.

**int QWindowsStyle::popupMenuItemHeight ( bool, QMenuItem * mi, const QFontMetrics & fm )**

Reimplemented for internal reasons; the API is not affected.

**void QWindowsStyle::scrollBarMetrics ( const QScrollBar * sb, int & sliderMin, int & sliderMax, int & sliderLength, int & buttonDim )** `[virtual]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QStyle.

**int QWindowsStyle::sliderLength () const** `[virtual]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QStyle.

**int QWindowsStyle::splitterWidth () const** `[virtual]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QStyle.

**void QWindowsStyle::tabbarMetrics ( const QTabBar * t, int & hframe, int & vframe, int & overlap )** `[virtual]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QStyle.

### 244.2.1. URL

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

# 245. QWizard Class Reference

`#include <qwizard.h>`

Inherits QDialog.

## 245.1. Detailed Description

The QWizard class provides a framework for easily writing wizards.

A wizard is a dialog that consists of a sequential number of steps, each consisting of a single page.

QWizard provides a title for each page, and "Next", "Back", "Finish", "Cancel" and "Help" buttons,

as appropriate.

## 245.2. Member Function Documentation

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

Constructs an empty wizard dialog.

**QWizard::~QWizard ()**

Destroys the object and frees any allocated resources, including, of course, all pages and controllers.

**void QWizard::addPage ( QWidget * page, const QString & title )** `[virtual]`

Adds *page* to the end of the wizard, titled *title.*

**bool QWizard::appropriate ( QWidget * w ) const** `[virtual]`

This virtual function returns TRUE if *w* is appropriate for display in the current context of the wizard, and FALSE if QWizard should go on.

It is called when the Next button is clicked.

**Warning:** The last page of a wizard will be displayed if nothing else wants to, and the Next button

was enabled when the user clicked.

The default implementation returns whatever was set using setAppropriate(). The ultimate default is TRUE.

**void QWizard::back ()** `[virtual protected slot]`

Called when the user clicks the Back button, this function shows the page which the user saw prior to the current one.

**QPushButton * QWizard::backButton () const**

Returns the Back button of the dialog.

By default, this button is connected to the back() slot, which is virtual.

**QPushButton * QWizard::cancelButton () const**

Returns the Cancel button of the dialog.

By default, this button is connected to the QDialog::reject() slot, which is virtual so you may override it in a QWizard subclass.

**QWidget * QWizard::currentPage () const**

Returns a pointer to the page currently being displayed by the wizard. The wizard does its best to make sure that this value is never 0, but if you try hard enough it can be.

**bool QWizard::eventFilter ( QObject * o, QEvent * e )** `[virtual]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QObject.

**QPushButton * QWizard::finishButton () const**

Returns the Finish button of the dialog.

By default, this button is connected to the QDialog::accept() slot, which is virtual so you may override it in a QWizard subclass.

**void QWizard::help ()** `[virtual protected slot]`

This slot either makes the wizard help you, if it can. The only way it knows is to emit the helpClicked() signal, and perhaps the QWizardPage::helpClicked() signal too, if a QWizardPage is currently being displayed.

**QPushButton * QWizard::helpButton () const**

Returns the Help button of the dialog.

By default, this button is connected to the help() slot.

**void QWizard::helpClicked ()** `[signal]`

This signal is emitted when the user clicks on the help button.

**void QWizard::layOutButtonRow ( QHBoxLayout * layout )** `[virtual protected]`

This virtual function is responsible for adding the bottom divider and buttons below it.

*layout* is the vertical layout of the entire wizard.

**void QWizard::layOutTitleRow ( QHBoxLayout * layout, const QString & title )** `[virtual protected]`

This virtual function is responsible for laying out the title row and adding the vertical divider between the title and the wizard page. *layout* is the vertical layout for the wizard, *title* is the title for this page, and this function is called every time *title* changes.

**void QWizard::next ()** `[virtual protected slot]`

Called when the user clicks the Next button, this function shows the next appropriate page.

**QPushButton * QWizard::nextButton () const**

Returns the Next button of the dialog.

By default, this button is connected to the next() slot, which is virtual.

**void QWizard::removePage ( QWidget * page )** `[virtual]`

Removes *page* from this wizard. Does not delete *page*. If *page* is currently being displayed, QWizard will display something else.

**void QWizard::setFont ( const QFont & font )** `[virtual]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QWidget.

**void QWizard::show ()** `[virtual]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QWidget.

**void QWizard::showPage ( QWidget * page )** `[virtual]`

Makes *page* be the displayed page.

**QString QWizard::title ( QWidget * page ) const**

Returns the title of *page.*

### 245.2.1. URL

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

# 246. QWMatrix Class Reference

```
#include <qwmatrix.h>
```

### 246.1. 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:

```
double a    = pi/180 * 25;              // convert 25 to radians
double sina = sin(a);
double 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
xform/xform.cpp
drawdemo/drawdemo.cpp
qmag/qmag.cpp
desktop/desktop.cpp
movies/main.cpp

### 246.2. 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 ( double m11, double m12, double m21, double m22, double dx, double dy )**

Constructs a matrix with the specified elements.

**double QWMatrix::dx () const**

Returns the horizontal translation.

**double 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.

**double QWMatrix::m11 () const**

Returns the X scaling factor.

**double QWMatrix::m12 () const**

Returns the vertical shearing factor.

**double QWMatrix::m21 () const**

Returns the horizontal shearing factor.

**double 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.

Examples:

xform/xform.cpp

**QRect QWMatrix::map ( const QRect & r ) const**

Returns the transformed rectangle *r*.

If rotation or shearing has been specified, then the bounding rectangle will be returned.

**void QWMatrix::map ( double x, double y, double * tx, double * 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)
```

**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 ( double a )**

Rotates the coordinate system *a* degrees counterclockwise.

Returns a reference to the matrix.

See also: translate(), scale() and shear().

Examples:

xform/xform.cpp

drawdemo/drawdemo.cpp
desktop/desktop.cpp

**QWMatrix & QWMatrix::scale ( double sx, double 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
xform/xform.cpp
movies/main.cpp

**void QWMatrix::setMatrix ( double m11, double m12, double m21, double m22, double dx, double dy )**

Sets the matrix elements to the specified values.

**QWMatrix & QWMatrix::shear ( double sh, double sv )**

Shears the coordinate system by *sh* horizontally and *sv* vertically.

Returns a reference to the matrix.

See also: translate(), scale() and rotate().

Examples:

xform/xform.cpp
drawdemo/drawdemo.cpp

**QWMatrix & QWMatrix::translate ( double dx, double 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:

xform/xform.cpp
drawdemo/drawdemo.cpp

### 246.3. 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 `double` in the listed order.

**QDataStream & operator>> (QDataStream & s, QWMatrix & m)**

Reads a matrix from the stream and returns a reference to the stream.

#### 246.3.1. URL

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

# 247. QXtApplication Class Reference

`#include <qxt.h>`

Inherits QApplication.

## 247.1. 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:

mainlyQt/editor.cpp
mainlyXt/editor.cpp
mainlyMotif/editor.cpp

## 247.2. 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.

### 247.2.1. URL

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

# 248. QXtWidget Class Reference

```
#include <qxt.h>
```

Inherits QWidget.

## 248.1. 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:

mainlyQt/editor.cpp
mainlyXt/editor.cpp
mainlyMotif/editor.cpp

## 248.2. 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, QWidget * parent=0, 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 Xt requires that the most toplevel Xt widget is a shell. That means, if *parent* is a QXtWidget, the *widget_class* can be of any kind. If there isn't a parent or the parent is just a normal QWidget, *widget_class* should be something like `topLevelShellWidgetClass`.

If the *managed* parameter is TRUE and *parent* in not NULL, XtManageChild it used to manage the child.

**QXtWidget::~QXtWidget ()**

Destructs the QXtWidget.

**void QXtWidget::enterEvent ( QEvent * ev )** `[virtual protected]`

Reimplemented to produce the Xt effect of getting focus when the mouse enters the widget. *This may be changed.*

Reimplemented from QWidget.

**void QXtWidget::focusInEvent ( QFocusEvent * )** `[virtual protected]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QWidget.

**void QXtWidget::focusOutEvent ( QFocusEvent * )** `[virtual protected]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QWidget.

**bool QXtWidget::isActiveWindow () const**

Different from QWidget::isActiveWindow().

**void QXtWidget::leaveEvent ( QEvent * ev )** `[virtual protected]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QWidget.

**void QXtWidget::moveEvent ( QMoveEvent * )** `[virtual protected]`

Reimplemented for internal reasons; the API is not affected.

Reimplemented from QWidget.

**void QXtWidget::resizeEvent ( QResizeEvent * )** `[virtual protected]`

Reimplemented for internal reasons; the API is not affected.

Examples:

mainlyQt/editor.cpp
mainlyXt/editor.cpp

Reimplemented from QWidget.

**Widget QXtWidget::xtWidget () const**

Returns the Xt widget equivalent for the Qt widget.

Examples:

mainlyXt/editor.cpp
mainlyMotif/editor.cpp

### 248.2.1. URL

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