Events, Actions, Layouts and Styles with Qt

Qt 3.0

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

Contents

Events and Event Filters

In Qt, an event is an object that inherits QEvent. Events are delivered to objects that inherit QObject through calling QObject::event(). Event delivery means that an event has occurred, the QEvent indicates precisely what, and the QObject needs to respond. Most events are specific to QWidget and its subclasses, but there are important events that aren't related to graphics, for example, socket activation, which is the event used by QSocketNotifier for its work.

Some events come from the window system, e.g. QMouseEvent, some from other sources, e.g. QTimerEvent, and some come from the application program. Qt is symmetric, as usual, so you can send events in exactly the same ways as Qt's own event loop does.

Most events types have special classes, most commonly QResizeEvent, QPaintEvent, QMouseEvent, QKeyEvent and QCloseEvent. There are many others, perhaps forty or so, but most are rather odd.

Each class subclasses QEvent and adds event-specific functions; see, for example, QResizeEvent. In the case of QResizeEvent, QResizeEvent::size() and QResizeEvent::oldSize() are added.

Some classes support more than one event type. QMouseEvent supports mouse moves, presses, shift-presses, drags, clicks, right-presses, etc.

Since programs need to react in varied and complex ways, Qt's event delivery mechanisms are flexible. The documentation for QApplication::notify() concisely tells the whole story, here we will explain enough for 99% of applications.

The normal way for an event to be delivered is by calling a virtual function. For example, QPaintEvent is delivered by calling QWidget::paintEvent(). This virtual function is responsible for reacting appropriately, normally by repainting the widget.

Occasionally there isn't such an event-specific function, or the event-specific function isn't sufficient. The most common example is tab key presses. Normally, those are interpreted by QWidget to move the keyboard focus, but a few widgets need the tab key for themselves.

These objects can reimplement QObject::event(), the general event handler, and either do their event handling before or after the usual handling, or replace it completely. A very unusual widget that both interprets tab and has an application-specific custom event might contain:

```
bool MyClass:event( QEvent * e ) {
    if ( e->type() == QEvent::KeyPress ) {
        QKeyEvent * ke = (QKeyEvent*) e;
        if ( ke->key() == Key_Tab ) {
            // special tab handling here
            k->accept();
            return TRUE;
        }
    } else if ( e->type() >= QEvent::User ) {
        QCustomEvent * c = (QCustomEvent*) e;
        // custom event handling here
        return TRUE;
    }
    QWidget::event( e );
```

```
}
```

More commonly, an object needs to look at another's events. Qt supports this using QObject::installEventFilter() (and the corresponding remove). For example, dialogs commonly want to filter key presses for some widgets, e.g. to modify Return-key handling.

An event filter gets to process events before the target object does. The filter's QObject::eventFilter() implementation is called, and can accept or reject the filter, and allow or deny further processing of the event. If all the event filters allow further processing of an event, the event is sent to the target object itself. If one of them stops processing, the target and any later event filters don't get to see the event at all.

It's also possible to filter *all* events for the entire application, by installing an event filter on QApplication. This is what QToolTip does in order to see *all* the mouse and keyboard activity. This is very powerful, but it also slows down event delivery of every single event in the entire application, so it's best avoided.

The global event filters are called before the object-specific filters.

Finally, many applications want to create and send their own events.

Creating an event of a built-in type is very simple: create an object of the relevant type, and then call QApplication::sendEvent() or QApplication::postEvent().

sendEvent() processes the event immediately - when sendEvent() returns, (the event filters and) the object have already processed the event. For many event classes there is a function called isAccepted() that tells you whether the event was accepted or rejected by the last handler that was called.

postEvent() posts the event on a queue for later dispatch. The next time Qt's main event loop runs, it dispatches all posted events, with some optimization. For example, if there are several resize events, they are are compacted into one. The same applies to paint events: QWidget::update() calls postEvent(), which minimizes flickering and increases speed by avoiding multiple repaints.

postEvent() is also often used during object initialization, since the posted event will typically be dispatched very soon after the initialization of the object is complete.

To create events of a custom type, you need to define an event number, which must be greater than QEvent::User, and probably you also need to subclass QCustomEvent in order to pass characteristics about your custom event. See the documentation to QCustomEvent for details.

Writing your own layout manager

Here we present an example in detail. The class CardLayout is inspired by the Java layout manager of the same name. It lays out the items (widgets or nested layouts) on top of each other, each item offset by QLayout::spacing().

To write your own layout class, you must define the following:

- A data structure to store the items handled by the layout. Each item is a QLayoutItem. We will use a QPtrList in this example.
- addItem(), how to add items to the layout.
- setGeometry(), how to perform the layout.
- sizeHint(), the preferred size of the layout.
- iterator(), how to iterate over the layout.

In most cases, you will also implement minimumSize().

card.h

```
#ifndef CARD_H
#define CARD_H

#include <qlayout.h>
#include <qptrlist.h>

class CardLayout : public QLayout
{
public:
    CardLayout( QWidget *parent, int dist )
        : QLayout( parent, 0, dist ) { }
    CardLayout( QLayout* parent, int dist)
        : QLayout( parent, dist ) { }
    CardLayout( int dist )
        : QLayout( dist ) { }
    ~CardLayout();

    void addItem(QLayoutItem *item);
    QSize sizeHint() const;
    QSize minimumSize() const;
    QLayoutIterator iterator();
    void setGeometry(const QRect &rect);

private:
    QPtrList list;
};
```

```
#endif
```

card.cpp

```
#include "card.h"
```

First we define an iterator over the layout. Layout iterators are used internally by the layout system to handle deletion of widgets. They are also available for application programmers.

There are two different classes involved: QLayoutIterator is the class that is visible to application programmers, it is explicitly shared. The QLayoutIterator contains a QGLayoutIterator that does all the work. We must create a subclass of QGLayoutIterator that knows how to iterate over our layout class.

In this case, we choose a simple implementation: we store an integer index into the list and a pointer to the list. Every QGLayoutIterator subclass must implement current(), next() and takeCurrent(), as well as a constructor. In our example we do not need a destructor.

```
class CardLayoutIterator : public QGLayoutIterator
{
public:
    CardLayoutIterator( QPtrList *l )
        : idx( 0 ), list( l ) { }

    QLayoutItem *current()
    { return idx count()) ? list->at(idx) : 0;  }

    QLayoutItem *next()
    { idx++; return current(); }

    QLayoutItem *takeCurrent()
    { return list->take( idx ); }

private:
    int idx;
    QPtrList *list;
};
```

We must implement QLayout:iterator() to return a QLayoutIterator over this layout.

```
QLayoutIterator CardLayout::iterator()
{
    return QLayoutIterator( new CardLayoutIterator(&list) );
}
```

addItem() implements the default placement strategy for layout items. It must be implemented. It is used by QLayout::add(), by the QLayout constructor that takes a layout as parent, and it is used to implement the auto-add feature. If your layout has advanced placement options that require parameters, you will must provide extra access functions such as QGridLayout::addMultiCell().

```
void CardLayout::addItem( QLayoutItem *item )
{
    list.append( item );
}
```

The layout takes over responsibility of the items added. Since QLayoutItem does not inherit QObject, we must delete the items manually. The function QLayout::deleteAllItems() uses the iterator we defined above to delete all the items in the layout.

```
CardLayout::~CardLayout()
{
    deleteAllItems();
}
```

The setGeometry() function actually performs the layout. The rectangle supplied as an argument does not include margin(). If relevant, use spacing() as the distance between items.

```
void CardLayout::setGeometry( const QRect &rect )
{
    QLayout::setGeometry( rect );

    QPtrListIterator it( list );
    if (it.count() == 0)
        return;

    QLayoutItem *o;

    int i = 0;

    int w = rect.width() - ( list.count() - 1 ) * spacing();
    int h = rect.height() - ( list.count() - 1 ) * spacing();

    while ( (o = it.current()) != 0 ) {
        ++it;
        QRect geom( rect.x() + i * spacing(), rect.y() + i * spacing(),
                    w, h );
        o->setGeometry( geom );
        ++i;
    }
}
```

sizeHint() and minimumSize() are normally very similar in implementation. The sizes returned by both functions should include spacing(), but not margin().

```
QSize CardLayout::sizeHint() const
{
    QSize s( 0, 0 );
    int n = list.count();
    if ( n > 0 )
        s = QSize( 100, 70 ); // start with a nice default size
    QPtrListIterator it( list );
    QLayoutItem *o;
    while ( (o = it.current()) != 0 ) {
        ++it;
        s = s.expandedTo( o->minimumSize() );
    }
    return s + n * QSize( spacing(), spacing() );
}

QSize CardLayout::minimumSize() const
{
```

```
    QSize s( 0, 0 );
    int n = list.count();
    QPtrListIterator it( list );
    QLayoutItem *o;
    while ( (o = it.current()) != 0 ) {
        ++it;
        s = s.expandedTo( o->minimumSize() );
    }
    return s + n * QSize( spacing(), spacing() );
}
```

Further Notes

This layout does not implement heightForWidth().

We ignore QLayoutItem::isEmpty(), this means that the layout will treat hidden widgets as visible.

For complex layouts, speed can be greatly increased by caching calculated values. In that case, implement QLayoutItem::invalidate() to mark the cached data as dirty.

Calling QLayoutItem::sizeHint(), etc. may be expensive, so you should store the value in a local variable if you need it again later in the same function.

You should not call QLayoutItem::setGeometry() twice on the same item in the same function. That can be very expensive if the item has several child widgets, because it will have to do a complete layout every time. Instead, calculate the geometry and then set it. (This doesn't only apply to layouts, you should do the same if you implement your own resizeEvent().)

Style overview

A style in Qt implements the look and feel found in GUIs on different platforms. For instance the Windows style used in Windows and the Motif style that are common on many Unix platforms.

This is a short guide that describes the steps that are necessary to get started creating and using custom styles with the style API in Qt 3.x. First, we go through the steps necessary to create a style: 1) picking a base style to inherit from and 2) re-implementing the necessary functions in the derived class. Then we show how to use the new style from within your own applications, or as a plugin together with existing Qt applications.

Creating a custom style

1. Pick a base style to inherit from.

The first step is to pick one of the base styles provided with Qt to build your custom style on. Which of the available styles to start from does of course depend on what look & feel you want. Basically you should choose from the QWindowsStyle derived classes or the QMotifStyle derived classes. These are the two base look & feel classes in the Qt style engine. Inheriting directly from QCommonStyle is also an option if you want to start almost from scratch when implementing your style. In this simple example we will inherit from QWindowsStyle.

2. Re-implement the necessary functions in your derived class.

Depending on which parts of the base style you want to change, you have to re-implement the functions that are used to draw those parts of the interface. If you take a look at the QStyle documentation, you will find a list of the different primitives, controls and complex controls. You will also find an illustration that shows where the different primitives, controls and complex controls are used. In this example we will first change the look of the standard arrows that are used in the QWindowsStyle. The arrows are PrimitiveElements that are drawn in the drawPrimitive() function, therefore we need to re-implement that function. We get the following class declaration:

```
#include <qwindowsstyle.h>

class CustomStyle : public QWindowsStyle {
    Q_OBJECT
public:
    CustomStyle();
    ~CustomStyle();

    void drawPrimitive( PrimitiveElement pe,
                        QPainter *p,
                        const QRect & r,
                        const QColorGroup & cg,
                        SFlags flags = Style_Default,
                        const QStyleOption & = QStyleOption::Default ) const;

private:
    // Disabled copy constructor and operator=
    CustomStyle( const CustomStyle & );
```

```
    CustomStyle& operator=( const CustomStyle & );
};
```

Note that we disable the copy constructor and the '=' operator for our style. QObject is the base class for all style classes in Qt, and a QObject inherently cannot be copied; there are some aspects of it that are not copyable.

From the QStyle docs we see that PE_ArrowUp, PE_ArrowDown, PE_ArrowLeft and PE_ArrowRight are the primitives we need to do something with. We get the following in our drawPrimitive() function:

```
CustomStyle::CustomStyle()
{
}

CustomStyle::~CustomStyle()
{
}

void CustomStyle::drawPrimitive( PrimitiveElement pe,
                                 QPainter * p,
                                 const QRect & r,
                                 const QColorGroup & cg,
                                 SFlags flags,
                                 const QStyleOption & opt ) const
{
    // we are only interested in the arrows
    if (pe >= PE_ArrowUp && pe <= PE_ArrowLeft) {
        QPointArray pa( 3 );
        // make the arrow cover half the area it is supposed to be
        // painted on
        int x = r.x();
        int y = r.y();
        int w = r.width() / 2;
        int h = r.height() / 2;
        x += (r.width() - w) / 2;
        y += (r.height() - h) /2;

        switch( pe ) {
        case PE_ArrowDown:
            pa.setPoint( 0, x, y );
            pa.setPoint( 1, x + w, y );
            pa.setPoint( 2, x + w / 2, y + h );
            break;
        case PE_ArrowUp:
            pa.setPoint( 0, x, y + h );
            pa.setPoint( 1, x + w, y + h );
            pa.setPoint( 2, x + w / 2, y );
            break;
        case PE_ArrowLeft:
            pa.setPoint( 0, x + w, y );
            pa.setPoint( 1, x + w, y + h );
            pa.setPoint( 2, x, y + h / 2 );
            break;
        case PE_ArrowRight:
            pa.setPoint( 0, x, y );
            pa.setPoint( 1, x, y + h );
            pa.setPoint( 2, x + w, y + h / 2 );
            break;
        default: break;
```

```
        }

        // use different colors to indicate that the arrow is
        // enabled/disabled
        if ( flags & Style_Enabled ) {
            p->setPen( cg.mid() );
            p->setBrush( cg.brush( QColorGroup::ButtonText ) );
        } else {
            p->setPen( cg.buttonText() );
            p->setBrush( cg.brush( QColorGroup::Mid ) );
        }
        p->drawPolygon( pa );
    } else {
        // let the base style handle the other primitives
        QWindowsStyle::drawPrimitive( pe, p, r, cg, flags, data );
    }
}
```

Using a custom style

There are several ways of using a custom style in a Qt application. The easiest and most simple way is to include the following lines of code in the application's main() function:

```
#include "customstyle.h"

int main( int argc, char ** argv )
{
    QApplication::setStyle( new CustomStyle() );
    // do the usual routine on creating your QApplication object etc.
}
```

Note that you also have to include the `customstyle.h` and `customstyle.cpp` files in your project.

2. Creating and using a pluggable style

You may want to use your custom style in a Qt application that you don't want to, or have the opportunity to recompile. The Qt Plugin system makes it possible to create styles as plugins. Styles created as plugins are loaded as shared objects at runtime by Qt itself. Please refer to the Qt Plugin documentation for more information on how to go about creating a style plugin.

Compile your plugin and put it into $QTDIR/plugins/styles. We now have a pluggable style that Qt can load automatically. To use your new style with existing applications, simply start the application with the following argument:

```
./application -style custom
```

The application should appear with the look & feel from the custom style you implemented.

QAccel Class Reference

The QAccel class handles keyboard accelerator and shortcut keys.

`#include <qaccel.h>`

Inherits QObject [Additional Functionality with Qt].

Public Members

- **QAccel** (QWidget * parent, const char * name = 0)
- **QAccel** (QWidget * watch, QObject * parent, const char * name = 0)
- **~QAccel** ()
- bool **isEnabled** () const
- void **setEnabled** (bool enable)
- uint **count** () const
- int **insertItem** (const QKeySequence & key, int id = -1)
- void **removeItem** (int id)
- void **clear** ()
- QKeySequence **key** (int id)
- int **findKey** (const QKeySequence & key) const
- bool **isItemEnabled** (int id) const
- void **setItemEnabled** (int id, bool enable)
- bool **connectItem** (int id, const QObject * receiver, const char * member)
- bool **disconnectItem** (int id, const QObject * receiver, const char * member)
- void **repairEventFilter** ()
- void **setWhatsThis** (int id, const QString & text)
- QString **whatsThis** (int id) const

Signals

- void **activated** (int id)

Static Public Members

- QKeySequence **shortcutKey** (const QString & str)
- QString keyToString (QKeySequence k) *(obsolete)*
- QKeySequence stringToKey (const QString & s) *(obsolete)*

Protected Members

- virtual bool **eventFilter** (QObject * o, QEvent * e)

Detailed Description

The QAccel class handles keyboard accelerator and shortcut keys.

A keyboard accelerator triggers an action when a certain key combination is pressed. The accelerator handles all keyboard activity for all children of one top-level widget, so it is not affected by the keyboard focus.

In most cases, you will not need to use this class directly. Use the QAction class to create actions with accelerators that can be used in both menus and toolbars. If you're only interested in menus use QMenuData::insertItem() or QMenuData::setAccel() to make accelerators for operations that are also available on menus. Many widgets automatically generate accelerators, such as QButton, QGroupBox, QLabel (with QLabel::setBuddy()), QMenuBar and QTabBar. Example:

```
QPushButton p( "&Exit", parent ); //automatic shortcut ALT+Key_E
QPopupMenu *fileMenu = new fileMenu( parent );
fileMenu->insertItem( "Undo", parent, SLOT(undo()), CTRL+Key_Z );
```

A QAccel contains a list of accelerator items that can be manipulated using insertItem(), removeItem(), clear(), key() and findKey().

Each accelerator item consists of an identifier and a QKeySequence. A single key sequence consists of 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. As an alternative, use UNICODE_ACCEL with the unicode code point of the character. For example, `UNICODE_ACCEL + 'A'` gives the same accelerator as Key_A.

When an accelerator key is pressed, the accelerator 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() and disconnectItem()).

Use setEnabled() to enable/disable all items in the accelerator, or setItemEnabled() to enable/disable individual items. An item is active only when the QAccel is enabled and the item itself is.

The function setWhatsThis() specifies a help text that appears when the user presses an accelerator key in What's This mode.

A QAccel object handles key events to the QWidget::topLevelWidget() containing *parent*, and hence to any child widgets of that window. The accelerator will be deleted when *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 [p. 95], QWidget::keyPressEvent() [Widgets with Qt], QMenuData::setAccel() [Dialogs and Windows with Qt], QButton::accel [Widgets with Qt], QLabel::setBuddy() [Widgets with Qt], GUI Design Handbook: Keyboard Shortcuts and Miscellaneous Classes.

Member Function Documentation

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

Constructs a QAccel object with parent *parent* and name *name*. The accelerator operates on *parent*.

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

Constructs a QAccel object that operates on *watch*, but is a child of *parent*. The object is called *name*.

This constructor is not needed for normal application programming.

QAccel::~QAccel ()

Destroys the accelerator object and frees all allocated resources.

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 the accelerator item *id* to the slot *member* of *receiver*.

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

Of course, you can also send a signal as *member*.

See also disconnectItem() [p. 15].

Example: t14/gamebrd.cpp.

uint QAccel::count () const

Returns the number of accelerator items in this accelerator.

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

Disconnects an accelerator item with id *id* from the function called *member* in the *receiver* object.

See also connectItem() [p. 15].

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

Processes accelerator events intended for the top level widget. *e* is the event that occurred on object *o*.

Reimplemented from QObject [Additional Functionality with Qt].

int QAccel::findKey (const QKeySequence & 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 (const QKeySequence & key, int id = -1)

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

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 negative identifier less than -1.

```
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 a unique negative id insertItem( Key_P + CTRL + SH
```

Example: t14/gamebrd.cpp.

bool QAccel::isEnabled () const

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

See also setEnabled() [p. 17] and isItemEnabled() [p. 16].

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() [p. 17] and isEnabled() [p. 16].

QKeySequence 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 (QKeySequence k) [static]

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

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

The function is superfluous. Cast the QKeySequence *k* to a QString for the same effect.

See also stringToKey() [p. 17].

void QAccel::removeItem (int id)

Removes the accelerator item with the identifier *id*.

void QAccel::repairEventFilter ()

Makes sure that the accelerator is watching the correct event filter. This function is called automatically; you should not need to call it in application code.

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 using setItemEnabled(). To work, a key must be an enabled item in an enabled QAccel.

See also isEnabled() [p. 16] and setItemEnabled() [p. 17].

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

Enables the accelerator item with the identifier *id* if *enable* is TRUE, and disables *id* if *enable* is FALSE.

To work, an item must be enabled and be in an enabled QAccel.

See also isItemEnabled() [p. 16] and isEnabled() [p. 16].

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

Sets a What's This help for the accelerator item *id* to *text*.

The text will be shown when the application is in What's This mode and the user hits the accelerator key.

To set What's This help on a menu item (with or without an accelerator key), use QMenuData::setWhatsThis().

See also whatsThis() [p. 18], QWhatsThis::inWhatsThisMode() [Widgets with Qt], QMenuData::setWhatsThis() [Dialogs and Windows with Qt] and QAction::whatsThis [p. 27].

QKeySequence QAccel::shortcutKey (const QString & str) [static]

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

For example, shortcutKey("E&xit") returns ALT+Key_X, shortcutKey("&Exit") returns ALT+Key_E and shortcutKey("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 this writing, Microsoft and The Open Group do not appear to have issued equivalent recommendations for other languages.

QKeySequence QAccel::stringToKey (const QString & s) [static]

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

Returns an accelerator code for the string *s*. For example "Ctrl+O" gives CTRL+UNICODE_ACCEL+'O'. The strings "Ctrl", "Shift", "Alt" are recognized, as well as their translated equivalents in the "QAccel" scope (using QObject::tr()). Returns 0 if *s* is not recognized.

This function is typically used with tr(), so that accelerator keys can be replaced in translations:

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

Notice the `"File|Open"` translator comment. It is by no means necessary, but it provides some context for the human translator.

The function is superfluous. Construct a QKeySequence from the string *s* for the same effect.

See also QObject::tr() [Additional Functionality with Qt] and Internationalization with Qt [Accessibility and Internationalization with Qt].

Example: i18n/mywidget.cpp.

QString QAccel::whatsThis (int id) const

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

See also setWhatsThis() [p. 17].

QAction Class Reference

The QAction class provides an abstract user interface action that can appear both in menus and tool bars.

`#include <qaction.h>`

Inherits QObject [Additional Functionality with Qt].

Inherited by QActionGroup [p. 28].

Public Members

- **QAction** (QObject * parent, const char * name = 0, bool toggle = FALSE)
- **QAction** (const QString & text, const QIconSet & icon, const QString & menuText, QKeySequence accel, QObject * parent, const char * name = 0, bool toggle = FALSE)
- **QAction** (const QString & text, const QString & menuText, QKeySequence accel, QObject * parent, const char * name = 0, bool toggle = FALSE)
- **~QAction** ()
- virtual void **setIconSet** (const QIconSet &)
- QIconSet **iconSet** () const
- virtual void **setText** (const QString &)
- QString **text** () const
- virtual void **setMenuText** (const QString &)
- QString **menuText** () const
- virtual void **setToolTip** (const QString &)
- QString **toolTip** () const
- virtual void **setStatusTip** (const QString &)
- QString **statusTip** () const
- virtual void **setWhatsThis** (const QString &)
- QString **whatsThis** () const
- virtual void **setAccel** (const QKeySequence & key)
- QKeySequence **accel** () const
- virtual void **setToggleAction** (bool)
- bool **isToggleAction** () const
- bool **isOn** () const
- bool **isEnabled** () const
- virtual bool **addTo** (QWidget * w)
- virtual bool **removeFrom** (QWidget * w)

Public Slots

- void **toggle** ()
- virtual void **setOn** (bool)
- virtual void **setEnabled** (bool)

Signals

- void **activated** ()
- void **toggled** (bool)

Properties

- QKeySequence **accel** — the action's accelerator key
- bool **enabled** — whether the action is enabled
- QIconSet **iconSet** — the action's icon
- QString **menuText** — the action's menu text
- bool **on** — whether a toggle action is on
- QString **statusTip** — the action's status tip
- QString **text** — the action's descriptive text
- bool **toggleAction** — whether the action is a toggle action
- QString **toolTip** — the action's tool tip
- QString **whatsThis** — the action's "What's This?" help text

Protected Members

- virtual void **addedTo** (QWidget * actionWidget, QWidget * container)
- virtual void **addedTo** (int index, QPopupMenu * menu)

Detailed Description

The QAction class provides an abstract user interface action that can appear both in menus and tool bars.

In GUI applications many commands can be invoked via a menu option, a toolbar button and a keyboard accelerator. Since the same action must be performed regardless of how the action was invoked and since the menu and toolbar should be kept in sync it is useful to represent a command as an *action*. An action can be added to a menu and a toolbar and will automatically be kept in sync, for example, if the user presses a Bold toolbar button the Bold menu item will be checked.

A QAction may contain an icon, a menu text, an accelerator, a status text, a whats this text and a tool tip. Most of these can be set in the constructor. They can all be set independently with setIconSet(), setText(), setMenuText(), setToolTip(), setStatusTip(), setWhatsThis() and setAccel().

An action may be a toggle action e.g. a Bold toolbar button, or a command action, e.g. 'Open File' which invokes an open file dialog. Toggle actions emit the toggled() signal when their state changes. Both command and toggle actions emit the activated() signal when they are invoked. Use setToggleAction() to set an action's toggled status. To see if an action is a toggle action use isToggleAction(). A toggle action may be "on", isOn() returns TRUE, or "off", isOn() returns FALSE.

Actions are added to widgets (menus or toolbars) using addTo(), and removed using removeFrom().

Once a QAction has been created it should be added to the relevant menu and toolbar and then connected to the slot which will perform the action. For example:

```
fileSaveAction = new QAction( "Save File", QPixmap( filesave ),
                              "&Save", CTRL+Key_S, this, "save" );
connect( fileSaveAction, SIGNAL( activated() ) , this, SLOT( save() ) );
```

We create a "Save File" action with a menu text of "&Save" and *Ctrl+S* as the keyboard accelerator. We connect the fileSaveAction's activated() signal to our save() slot. Note that at this point there is no menu or toolbar action, we'll add them next:

```
QToolBar * fileTools = new QToolBar( this, "file operations" );

fileSaveAction->addTo( fileTools );

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

fileSaveAction->addTo( file );
```

We create a toolbar and add our fileSaveAction to it. Similarly we create a menu, add a top-level menu item, and add our fileSaveAction.

(See the Simple Application Walkthrough featuring QAction for a detailed example.)

We recommend that actions are created as children of the window that they are used in. In most cases actions will be children of the application's main window.

To prevent recursion don't create an action as a child of a widget that the action is later added to.

See also Main Window and Related Classes and Basic Widgets.

Member Function Documentation

QAction::QAction (QObject * parent, const char * name = 0, bool toggle = FALSE)

Constructs an action with parent *parent* and name *name*.

If *toggle* is TRUE the action will be a toggle action otherwise it will be a command action.

If *parent* is a QActionGroup, the new action inserts itself into *parent*.

Note: for accelerators and status tips to work, *parent* must be a widget.

QAction::QAction (const QString & text, const QIconSet & icon, const QString & menuText, QKeySequence accel, QObject * parent, const char * name = 0, bool toggle = FALSE)

This constructor creates an action with the following properties: the description *text*, the icon or iconset *icon*, the menu text *menuText* and keyboard accelerator *accel*. It is a child of *parent* and named *name*. If *toggle* is TRUE the action will be a toggle action otherwise it will be a command action.

The *parent* should be a widget for accelerators and status tips to work.

If *parent* is a QActionGroup, the action automatically becomes a member of it.

The *text* and *accel* will be used for tool tips and status tips unless you provide specific text for these using setToolTip() and setStatusTip().

QAction::QAction (const QString & text, const QString & menuText, QKeySequence accel, QObject * parent, const char * name = 0, bool toggle = FALSE)

This constructor results in an iconless action with the description *text*, the menu text *menuText* and the keyboard accelerator *accel*. Its parent is *parent* and its name *name*. If *toggle* is TRUE the action will be a toggle action otherwise it will be a command action.

The action automatically becomes a member of *parent* if *parent* is a QActionGroup.

The *parent* should be a widget for accelerators and status tips to work.

The *text* and *accel* will be used for tool tips and status tips unless you provide specific text for these using setToolTip() and setStatusTip().

QAction::~QAction ()

Destroys the object and frees allocated resources.

QKeySequence QAction::accel () const

Returns the action's accelerator key. See the "accel" [p. 25] property for details.

void QAction::activated () [signal]

This signal is emitted when an action is activated by the user, i.e. when the user clicks a menu option or a toolbar button or presses an action's accelerator key combination.

Connect to this signal for command actions. Connect to the toggled() signal for toggle actions.

Example: action/application.cpp.

bool QAction::addTo (QWidget * w) [virtual]

Adds this action to widget *w*.

Currently actions may be added to QToolBar and QPopupMenu widgets.

An action added to a tool bar is automatically displayed as a tool button; an action added to a pop up menu appears as a menu option.

addTo() returns TRUE if the action was added successfully and FALSE otherwise. (If *w* is not a QToolBar or QPopupMenu the action will not be added and FALSE will be returned.)

See also removeFrom() [p. 23].

Examples: action/application.cpp, action/toggleaction/toggleaction.cpp and textedit/textedit.cpp.

Reimplemented in QActionGroup.

void QAction::addedTo (QWidget * actionWidget, QWidget * container) [virtual protected]

This function is called from the addTo() function when it created a widget (*actionWidget*) for the action in the *container*.

void QAction::addedTo (int index, QPopupMenu * menu) [virtual protected]

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

This function is called from the addTo() function when it created a menu item at the index *index* in the popup menu *menu*.

QIconSet QAction::iconSet () const

Returns the action's icon. See the "iconSet" [p. 25] property for details.

bool QAction::isEnabled () const

Returns TRUE if the action is enabled; otherwise returns FALSE. See the "enabled" [p. 25] property for details.

bool QAction::isOn () const

Returns TRUE if a toggle action is on; otherwise returns FALSE. See the "on" [p. 26] property for details.

bool QAction::isToggleAction () const

Returns TRUE if the action is a toggle action; otherwise returns FALSE. See the "toggleAction" [p. 26] property for details.

QString QAction::menuText () const

Returns the action's menu text. See the "menuText" [p. 26] property for details.

bool QAction::removeFrom (QWidget * w) [virtual]

Removes the action from widget *w*.

Returns TRUE if the action was removed successfully, FALSE otherwise.

See also addTo() [p. 22].

void QAction::setAccel (const QKeySequence & key) [virtual]

Sets the action's accelerator key to *key*. See the "accel" [p. 25] property for details.

void QAction::setEnabled (bool) [virtual slot]

Sets whether the action is enabled. See the "enabled" [p. 25] property for details.

void QAction::setIconSet (const QIconSet &) [virtual]

Sets the action's icon. See the "iconSet" [p. 25] property for details.

void QAction::setMenuText (const QString &) [virtual]

Sets the action's menu text. See the "menuText" [p. 26] property for details.

void QAction::setOn (bool) [virtual slot]

Sets whether a toggle action is on. See the "on" [p. 26] property for details.

void QAction::setStatusTip (const QString &) [virtual]

Sets the action's status tip. See the "statusTip" [p. 26] property for details.

void QAction::setText (const QString &) [virtual]

Sets the action's descriptive text. See the "text" [p. 26] property for details.

void QAction::setToggleAction (bool) [virtual]

Sets whether the action is a toggle action. See the "toggleAction" [p. 26] property for details.

void QAction::setToolTip (const QString &) [virtual]

Sets the action's tool tip. See the "toolTip" [p. 27] property for details.

void QAction::setWhatsThis (const QString &) [virtual]

Sets the action's "What's This?" help text. See the "whatsThis" [p. 27] property for details.

QString QAction::statusTip () const

Returns the action's status tip. See the "statusTip" [p. 26] property for details.

QString QAction::text () const

Returns the action's descriptive text. See the "text" [p. 26] property for details.

void QAction::toggle () [slot]

Toggles the state of a toggle action.

See also on [p. 26], toggled() [p. 24] and toggleAction [p. 26].

void QAction::toggled (bool) [signal]

This signal is emitted when a toggle action changes state; command actions and QActionGroups don't emit toggled().

The argument denotes the new state; i.e. TRUE if the toggle action was switched on and FALSE if it was switched off.

To trigger a user command depending on whether a toggle action has been switched on or off connect it to a slot that takes a bool to indicate the state, e.g.

```
QMainWindow * window = new QMainWindow;

QAction * labelonoffaction = new QAction( window, "labelonoff", TRUE );

QObject::connect( labelonoffaction, SIGNAL( toggled( bool ) ),
                  window, SLOT( setUsesTextLabel( bool ) ) );
```

See also activated() [p. 22], toggleAction [p. 26] and on [p. 26].

Example: action/toggleaction/toggleaction.cpp.

QString QAction::toolTip () const

Returns the action's tool tip. See the "toolTip" [p. 27] property for details.

QString QAction::whatsThis () const

Returns the action's "What's This?" help text. See the "whatsThis" [p. 27] property for details.

Property Documentation

QKeySequence accel

This property holds the action's accelerator key.

The keycodes can be found in Qt::Key and Qt::Modifier. There is no default accelerator key.

Set this property's value with setAccel() and get this property's value with accel().

bool enabled

This property holds whether the action is enabled.

Disabled actions can't be chosen by the user. They don't disappear from the menu/tool bar but are displayed in a way which indicates that they are unavailable, e.g. they might be displayed greyed out.

What's this? help on disabled actions is still available provided the QAction::whatsThis property is set.

Set this property's value with setEnabled() and get this property's value with isEnabled().

QIconSet iconSet

This property holds the action's icon.

The icon is used as tool button icon and in the menu to the left of the menu text. There is no default icon.

(See the action/toggleaction/toggleaction.cpp example.)

Set this property's value with setIconSet() and get this property's value with iconSet().

QString menuText

This property holds the action's menu text.

If the action is added to a menu the menu option will consist of the icon (if there is one), the menu text and the accelerator (if there is one). If the menu text is not explicitly set in the constructor or using setMenuText() the action's description text will be used as the menu text. There is no default menu text.

See also text [p. 26].

Set this property's value with setMenuText() and get this property's value with menuText().

bool on

This property holds whether a toggle action is on.

This property is always on for command actions and QActionGroups. setOn() has no effect on them. This property's default is FALSE.

See also toggleAction [p. 26].

Set this property's value with setOn() and get this property's value with isOn().

QString statusTip

This property holds the action's status tip.

The statusTip is displayed on all status bars that the toplevel widget parenting this action provides.

If no status tip is defined, the action uses the tool tip text.

There is no default tooltip text.

See also statusTip [p. 26] and toolTip [p. 27].

Set this property's value with setStatusTip() and get this property's value with statusTip().

QString text

This property holds the action's descriptive text.

If QMainWindow::usesTextLabel is TRUE, the text appears as a label in the relevant toolbutton. It also serves as the default text in menus and tips if these have not been specifically defined. There is no default text.

See also menuText [p. 26], toolTip [p. 27] and statusTip [p. 26].

Set this property's value with setText() and get this property's value with text().

bool toggleAction

This property holds whether the action is a toggle action.

A toggle action is one which has an on/off state. For example a Bold toolbar button is either on or off. An action which is not a toggle action is a command action; a command action is simply executed. For example a file open toolbar button would invoke a file open dialog. This property's default is FALSE.

For exclusive toggling, add toggle actions to a QActionGroup with the QActionGroup::exclusive property set to TRUE.

Set this property's value with setToggleAction() and get this property's value with isToggleAction().

QString toolTip

This property holds the action's tool tip.

This text is used for the tool tip. If no status tip has been set the tool tip will be used for the status tip.

If no tool tip is specified the action's text and accelerator description are used as a default tool tip.

There is no default tool tip text.

See also statusTip [p. 26] and accel [p. 25].

Set this property's value with setToolTip() and get this property's value with toolTip().

QString whatsThis

This property holds the action's "What's This?" help text.

The whats this text is used to provide a brief description of the action. The text may contain rich text (i.e. HTML tags — see QStyleSheet for the list of supported tags). There is no default "What's This" text.

See also QWhatsThis [Widgets with Qt].

Set this property's value with setWhatsThis() and get this property's value with whatsThis().

QActionGroup Class Reference

The QActionGroup class groups actions together.

`#include <qaction.h>`

Inherits QAction [p. 19].

Public Members

- **QActionGroup** (QObject * parent, const char * name = 0, bool exclusive = TRUE)
- **~QActionGroup** ()
- void **setExclusive** (bool)
- bool **isExclusive** () const
- void **add** (QAction * action)
- void **addSeparator** ()
- virtual bool **addTo** (QWidget * w)
- void **setUsesDropDown** (bool enable)
- bool **usesDropDown** () const
- void insert (QAction * a) *(obsolete)*

Signals

- void **selected** (QAction *)

Properties

- bool **exclusive** — whether the action group does exclusive toggling
- bool **usesDropDown** — whether the group's actions are displayed in a subwidget of the widgets the action group is added to

Protected Members

- virtual void **addedTo** (QWidget * actionWidget, QWidget * container, QAction * a)
- virtual void **addedTo** (int index, QPopupMenu * menu, QAction * a)

Detailed Description

The QActionGroup class groups actions together.

In some situations it is useful to group actions together. For example, if you have a left justify action, a right justify action and a center action, only one of these actions should be active at any one time, and one simple way of achieving this is to group the actions together in an action group and setExclusive(TRUE).

An action group can also be added to a menu or a toolbar as a single unit, with all the actions within the action group appearing as separate menu options and toolbar buttons.

Here's an example from examples/textedit:

```
QActionGroup *grp = new QActionGroup( this );
grp->setExclusive( TRUE );
connect( grp, SIGNAL( selected( QAction* ) ), this, SLOT( textAlign( QAction* ) ) );
```

We create a new action group, call setExclusive() to ensure that only one of the actions in the group is ever active at any one time. We then connect the group to our textAlign() slot.

```
actionAlignLeft = new QAction( tr( "Left" ), QPixmap( "textleft.xpm" ), tr( "&Left" ), CTRL + Key_
actionAlignLeft->addTo( tb );
actionAlignLeft->addTo( menu );
actionAlignLeft->setToggleAction( TRUE );
```

We create a left align action, add it to the toolbar and the menu and make it a toggle action. We create center and right align actions in exactly the same way.

The actions in an action group emit their activated() (and for toggle actions, toggled()) signals as usual.

The setExclusive() function is used to ensure that only one action is active at any one time: it should be used with actions which have their toggleAction set to TRUE.

Action group actions appear as individual menu options and toolbar buttons. For exclusive action groups use setUsesDropDown() to display the actions in a subwidget of any widget the action group is added to. For example, the actions would appear in a combobox in a toolbar or as a submenu in a menu.

Actions can be added to an action group using add(), but normally they are added by creating the action with the action group as parent. Actions can have separators dividing them using addSeparator(). Action groups are added to widgets with addTo().

See also Main Window and Related Classes and Basic Widgets.

Member Function Documentation

QActionGroup::QActionGroup (QObject * parent, const char * name = 0, bool exclusive = TRUE)

Constructs an action group with parent *parent* and name *name*.

If *exclusive* is TRUE only one toggle action in the group will ever be active.

QActionGroup::~QActionGroup ()

Destroys the object and frees allocated resources.

void QActionGroup::add (QAction * action)

Adds action *action* to this group.

Normally an action is added to a group by creating it with the group as parent, so this function is not usually used.

See also addTo() [p. 30].

void QActionGroup::addSeparator ()

Adds a separator to the group.

bool QActionGroup::addTo (QWidget * w) [virtual]

Adds this action group to the widget *w*.

If usesDropDown() is TRUE and exclusive is TRUE (see setExclusive()) the actions are presented in a combobox if *w* is a toolbar and as a submenu if *w* is a menu. Otherwise (the default) the actions within the group are added to the widget individually, for example if the widget is a menu the actions will appear as individual menu options and if the widget is a toolbar the actions will appear as toolbar buttons.

It is recommended that actions is action groups, especially where usesDropDown() is TRUE, have their menuText() or text() property set.

All actions should be added to the action group *before* the action group is added to the widget. If actions are added to the action group *after* the action group has been added to the widget these later actions will *not* appear.

See also exclusive [p. 31], usesDropDown [p. 32] and removeFrom() [p. 23].

Example: action/actiongroup/editor.cpp.

Reimplemented from QAction [p. 22].

void QActionGroup::addedTo (QWidget * actionWidget, QWidget * container, QAction * a) [virtual protected]

This function is called from the addTo() function when it created a widget (*actionWidget*) for the child action *a* in the *container*.

void QActionGroup::addedTo (int index, QPopupMenu * menu, QAction * a) [virtual protected]

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

This function is called from the addTo() function when it created a menu item for the child action at the index *index* in the popup menu *menu*.

void QActionGroup::insert (QAction * a)

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

Use add() instead, or better still create the action with the action group as its parent.

bool QActionGroup::isExclusive () const

Returns TRUE if the action group does exclusive toggling; otherwise returns FALSE. See the "exclusive" [p. 31] property for details.

void QActionGroup::selected (QAction *) [signal]

This signal is emitted from exclusive groups when toggle actions change state.

The argument is the action whose state changed to "on".

```
QActionGroup * colors = new QActionGroup( this, "colors", TRUE );

QObject::connect( colors, SIGNAL( selected( QAction * ) ),
                  this, SLOT( setFontColor( QAction * ) ) );
```

In this example we connect the selected() signal to our setFontColor() slot, passing the QAction so that we know which action was chosen by the user.

(See the QActionGroup Walkthrough.)

See also exclusive [p. 31] and on [p. 26].

Examples: action/actiongroup/editor.cpp and textedit/textedit.cpp.

void QActionGroup::setExclusive (bool)

Sets whether the action group does exclusive toggling. See the "exclusive" [p. 31] property for details.

void QActionGroup::setUsesDropDown (bool enable)

Sets whether the group's actions are displayed in a subwidget of the widgets the action group is added to to *enable*. See the "usesDropDown" [p. 32] property for details.

bool QActionGroup::usesDropDown () const

Returns TRUE if the group's actions are displayed in a subwidget of the widgets the action group is added to; otherwise returns FALSE. See the "usesDropDown" [p. 32] property for details.

Property Documentation

bool exclusive

This property holds whether the action group does exclusive toggling.

If exclusive is TRUE only one toggle action in the action group can ever be active at any one time. If the user chooses another toggle action in the group the one they chose becomes active and the one that was active becomes inactive. By default this property is FALSE.

See also QAction::toggleAction [p. 26].

Set this property's value with setExclusive() and get this property's value with isExclusive().

bool usesDropDown

This property holds whether the group's actions are displayed in a subwidget of the widgets the action group is added to.

Exclusive action groups added to a toolbar display their actions in a combobox with the action's QAction::text and QAction::iconSet properties shown. Non-exclusive groups are represented by a tool button showing their QAction::iconSet and — depending on QMainWindow::usesTextLabel() — text() property.

In a popup menu the member actions are displayed in a submenu.

Changing usesDropDown effects subsequent calls to addTo() only.

This property's default is FALSE.

Set this property's value with setUsesDropDown() and get this property's value with usesDropDown().

QAquaStyle Class Reference

The QAquaStyle class implements the aqua 'Look and Feel'.

`#include <qaquastyle.h>`

Inherits QWindowsStyle [p. 179].

Public Members

- **QAquaStyle** ()

Detailed Description

The QAquaStyle class implements the aqua 'Look and Feel'.

This class implements the Aqua look and feel. It's an experimental class that tries to resemble a Macintosh-like GUI style with the QStyle system. The emulation is far from being perfect.

Note that the functions provided by QAquaStyle are reimplementations of QStyle functions; see QStyle for their documentation.

See also Widget Appearance and Style.

Member Function Documentation

QAquaStyle::QAquaStyle ()

Constructs a QAquaStyle object.

QBoxLayout Class Reference

The QBoxLayout class lines up child widgets horizontally or vertically.

`#include <qlayout.h>`

Inherits QLayout [p. 101].

Inherited by QHBoxLayout [p. 86] and QVBoxLayout [p. 174].

Public Members

- enum **Direction** { LeftToRight, RightToLeft, TopToBottom, BottomToTop, Down = TopToBottom, Up = BottomToTop }
- **QBoxLayout** (QWidget * parent, Direction d, int margin = 0, int spacing = -1, const char * name = 0)
- **QBoxLayout** (QLayout * parentLayout, Direction d, int spacing = -1, const char * name = 0)
- **QBoxLayout** (Direction d, int spacing = -1, const char * name = 0)
- **~QBoxLayout** ()
- virtual void **addItem** (QLayoutItem * item)
- Direction **direction** () const
- void **setDirection** (Direction direction)
- void **addSpacing** (int size)
- void **addStretch** (int stretch = 0)
- void **addWidget** (QWidget * widget, int stretch = 0, int alignment = 0)
- void **addLayout** (QLayout * layout, int stretch = 0)
- void **addStrut** (int size)
- void **insertSpacing** (int index, int size)
- void **insertStretch** (int index, int stretch = 0)
- void **insertWidget** (int index, QWidget * widget, int stretch = 0, int alignment = 0)
- void **insertLayout** (int index, QLayout * layout, int stretch = 0)
- bool **setStretchFactor** (QWidget * w, int stretch)
- bool **setStretchFactor** (QLayout * l, int stretch)
- virtual QSize **sizeHint** () const
- virtual QSize **minimumSize** () const
- virtual QSize **maximumSize** () const
- virtual bool **hasHeightForWidth** () const
- virtual int **heightForWidth** (int w) const
- virtual QSizePolicy::ExpandData **expanding** () const
- virtual void **invalidate** ()
- virtual void **setGeometry** (const QRect & r)
- int **findWidget** (QWidget * w)

Protected Members

- void **insertItem** (int index, QLayoutItem * item)

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's orientation is Horizontal the boxes are placed in a row, with suitable sizes. Each widget (or other box) will get at least its minimum size and at most its maximum size. Any excess space is shared according to the stretch factors (more about that below).

If the QBoxLayout's orientation is Vertical, the boxes are placed in a column, again with suitable sizes.

The easiest way to create a QBoxLayout is to use one of the convenience classes, e.g. 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 (i.e. it 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 this, 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.

Use insertWidget(), insertSpacing(), insertStretch() or insertLayout() to insert a box at a specified position in the layout.

QBoxLayout also includes two margin widths:

- setMargin() sets the width of the outer border. This is the width of the reserved space along each of the QBoxLayout's four sides.
- setSpacing() sets the width between neighboring boxes. (You can use addSpacing() to get more space at a peculiar spot.)

The margin defaults to 0; the spacing defaults to the same as the margin width for a top-level layout, or otherwise to the same as the parent layout. Both are parameters to the constructor.

To remove a widget from a layout, either delete it or reparent it with QWidget::reparent(). Hiding a widget with QWidget::hide() also effectively removes the widget from the layout, until QWidget::show() is called.

You will almost always want to use QVBoxLayout and QHBoxLayout rather than QBoxLayout because of their convenient constructors.

See also Layout Overview [Programming with Qt], Widget Appearance and Style and Layout Management.

Member Type Documentation

QBoxLayout::Direction

This type is used to determine the direction of a box layout. The possible values are the following:

- `QBoxLayout::LeftToRight` - Horizontal, from left to right
- `QBoxLayout::RightToLeft` - Horizontal, from right to left
- `QBoxLayout::TopToBottom` - Vertical, from top to bottom
- `QBoxLayout::Down` - The same as TopToBottom
- `QBoxLayout::BottomToTop` - Vertical, from bottom to top
- `QBoxLayout::Up` - The same as BottomToTop

Member Function Documentation

QBoxLayout::QBoxLayout (QWidget * parent, Direction d, int margin = 0, int spacing = -1, const char * name = 0)

Constructs a new QBoxLayout with direction *d* and main widget *parent*. *parent* may not be 0.

The *margin* is the number of pixels between the edge of the widget and its managed children. The *spacing* is the default number of pixels between neighboring children. If *spacing* is -1 the value of *margin* is used for *spacing*.

name is the internal object name.

See also direction() [p. 37].

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

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

The *spacing* is the default number of pixels between neighboring children. If *spacing* is -1, this QBoxLayout will inherit its parent's spacing().

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

Constructs a new QBoxLayout with direction *d* and name *name*.

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

You have to insert this box into another layout.

QBoxLayout::~QBoxLayout ()

Destroys this box layout.

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

Adds *item* to the end of this box layout.

Reimplemented from QLayout [p. 103].

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

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

See also insertLayout() [p. 38], setAutoAdd() [p. 106], addWidget() [p. 37] and addSpacing() [p. 37].

Examples: fonts/simple-qfont-demo/viewer.cpp, listbox/listbox.cpp and tictac/tictac.cpp.

void QBoxLayout::addSpacing (int size)

Adds a non-stretchable space with size *size* to the end of this box layout. QBoxLayout gives default margin and spacing. This function adds additional space.

See also insertSpacing() [p. 38] and addStretch() [p. 37].

Example: listbox/listbox.cpp.

void QBoxLayout::addStretch (int stretch = 0)

Adds a stretchable space with zero minimum size and stretch factor *stretch* to the end of this box layout.

See also addSpacing() [p. 37].

Examples: layout/layout.cpp, listbox/listbox.cpp and t13/gamebrd.cpp.

void QBoxLayout::addStrut (int size)

Limits the perpendicular dimension of the box (e.g. height if the box is LeftToRight) to a minimum of *size*. Other constraints may increase the limit.

void QBoxLayout::addWidget (QWidget * widget, int stretch = 0, int alignment = 0)

Adds *widget* to the end of this box layout, with a stretch factor of *stretch* and alignment *alignment*.

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 *alignment* which is a bitwise OR of Qt::AlignmentFlags values. 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 insertWidget() [p. 39], setAutoAdd() [p. 106], addLayout() [p. 37] and addSpacing() [p. 37].

Examples: checklists/checklists.cpp, fonts/simple-qfont-demo/viewer.cpp, layout/layout.cpp, lineedits/lineedits.cpp, listbox/listbox.cpp, t13/gamebrd.cpp and t13/lcdrange.cpp.

Direction QBoxLayout::direction () const

Returns the direction of the box. addWidget() and addSpacing() work in this direction; the stretch stretches in this direction.

See also QBoxLayout::Direction [p. 36], addWidget() [p. 37] and addSpacing() [p. 37].

QSizePolicy::ExpandData QBoxLayout::expanding () const [virtual]

Returns the expansiveness of this layout.

Reimplemented from QLayout [p. 104].

int QBoxLayout::findWidget (QWidget * w)

Searches for widget *w* in this layout (not including child layouts).

Returns the index of *w*, or -1 if *w* is not found.

bool QBoxLayout::hasHeightForWidth () const [virtual]

Returns TRUE if this layout's preferred height depends on its width; otherwise returns FALSE.

Reimplemented from QLayoutItem [p. 109].

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

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

Reimplemented from QLayoutItem [p. 109].

void QBoxLayout::insertItem (int index, QLayoutItem * item) [protected]

Inserts *item* in this box layout at position *index*. If *index* is negative, the item is added at the end.

Warning: Does not call QLayout::insertChildLayout() if *item* is a QLayout.

See also addItem() [p. 36] and findWidget() [p. 38].

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

Inserts *layout* at position *index*, with stretch factor *stretch*. If *index* is negative, the layout is added at the end.

See also setAutoAdd() [p. 106], insertWidget() [p. 39] and insertSpacing() [p. 38].

void QBoxLayout::insertSpacing (int index, int size)

Inserts a non-stretchable space at position *index*, with size *size*. If *index* is negative the space is added at the end.

The box layout has default margin and spacing. This function adds additional space.

See also insertStretch() [p. 38].

void QBoxLayout::insertStretch (int index, int stretch = 0)

Inserts a stretchable space at position *index*, with zero minimum size and stretch factor *stretch*. If *index* is negative the space is added at the end.

See also insertSpacing() [p. 38].

void QBoxLayout::insertWidget (int index, QWidget * widget, int stretch = 0, int alignment = 0)

Inserts *widget* at position *index*, with stretch factor *stretch* and alignment *alignment*. If *index* is negative, the widget is added at the end.

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 factors 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 *alignment*, which is a bitwise OR of Qt::AlignmentFlags values. 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() [p. 106], insertLayout() [p. 38] and insertSpacing() [p. 38].

void QBoxLayout::invalidate () [virtual]

Resets cached information.

Reimplemented from QLayout [p. 104].

QSize QBoxLayout::maximumSize () const [virtual]

Returns the maximum size needed by this box layout.

Reimplemented from QLayout [p. 105].

QSize QBoxLayout::minimumSize () const [virtual]

Returns the minimum size needed by this box layout.

Reimplemented from QLayout [p. 105].

void QBoxLayout::setDirection (Direction direction)

Sets the direction of this layout to *direction*.

void QBoxLayout::setGeometry (const QRect & r) [virtual]

Resizes managed widgets within the rectangle *r*.

Reimplemented from QLayout [p. 106].

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.

bool QBoxLayout::setStretchFactor (QLayout * l, int stretch)

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

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

Returns FALSE if *l* is not found.

QSize QBoxLayout::sizeHint () const [virtual]

Returns the preferred size of this box layout.

Reimplemented from QLayoutItem [p. 111].

QCDEStyle Class Reference

The QCDEStyle class provides a CDE look and feel.

`#include <qcdestyle.h>`

Inherits QMotifStyle [p. 116].

Public Members

- **QCDEStyle** (bool useHighlightCols = FALSE)
- virtual **~QCDEStyle** ()

Detailed Description

The QCDEStyle class provides a CDE look and feel.

This style provides a slightly improved Motif look similar to some versions of the Common Desktop Environment (CDE). The main differences 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).

Note that the functions provided by QCDEStyle are reimplementations of QStyle functions; see QStyle for their documentation.

See also Widget Appearance and Style.

Member Function Documentation

QCDEStyle::QCDEStyle (bool useHighlightCols = FALSE)

Constructs a QCDEStyle.

If *useHighlightCols* is FALSE (the default), 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.

QCDEStyle::~QCDEStyle () [virtual]

Destroys the style.

QChildEvent Class Reference

The QChildEvent class contains event parameters for child object events.

`#include <qevent.h>`

Inherits QEvent [p. 67].

Public Members

- **QChildEvent** (Type type, QObject * child)
- QObject * **child** () const
- bool **inserted** () const
- bool **removed** () const

Detailed Description

The QChildEvent class contains event parameters for child object events.

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

A ChildRemoved event is sent immediately, but a ChildInserted event is *posted* (with QApplication::postEvent()).

Note that if a child is removed immediately after it is inserted, the ChildInserted event may be suppressed, but the ChildRemoved event will always be sent. In this case there will be a ChildRemoved event without a corresponding ChildInserted event.

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

See also Event Classes.

Member Function Documentation

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

Constructs a child event object. The *child* is the object that is to be removed or inserted.

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

QObject * QChildEvent::child () const

Returns the child widget that was inserted or removed.

bool QChildEvent::inserted () const

Returns TRUE if the widget received a new child; otherwise returns FALSE.

bool QChildEvent::removed () const

Returns TRUE if the object lost a child; otherwise returns FALSE.

QCloseEvent Class Reference

The QCloseEvent class contains parameters that describe a close event.

`#include <qevent.h>`

Inherits QEvent [p. 67].

Public Members

- **QCloseEvent** ()
- bool **isAccepted** () const
- void **accept** ()
- void **ignore** ()

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 programmatically.

Close events contain a flag that indicates whether the receiver wants the widget to be closed or not. When a widget accepts the close event, it is hidden (and destroyed if it was created with the WDestructiveClose flag). If it refuses to accept the close event nothing happens. (Under X11 it is possible that the window manager will forcibly close the window; but at the time of writing we are not aware of any window manager that does this.)

The main widget of the application - QApplication::mainWidget() - is a special case. When it accepts the close event, Qt leaves the main event loop and the application is immediately terminated (i.e., it returns from the call to QApplication::exec() in the main() function).

The event handler QWidget::closeEvent() receives close events. The default implementation of this event handler accepts the close event. If you do not want your widget to be hidden, or want some special handing, you should reimplement the event handler.

The closeEvent() in the Application Walkthrough shows a close event handler that asks whether to save a document before closing.

If you want the widget to be deleted when it is closed, simply create it with the WDestructiveClose widget flag. This is very useful for independent top-level windows in a multi-window application.

QObject emits the destroyed() signal when it is deleted.

If the last top-level window is closed, the QApplication::lastWindowClosed() signal is emitted.

The isAccepted() function returns TRUE if the event's receiver has agreed to close the widget; call accept() to agree to close the widget and call ignore() if the receiver of this event does not want the widget to be closed.

See also QWidget::close() [Widgets with Qt], QWidget::hide() [Widgets with Qt], QObject::destroyed() [Additional Functionality with Qt], QApplication::setMainWidget() [Additional Functionality with Qt], QApplication::lastWindowClosed() [Additional Functionality with Qt], QApplication::exec() [Additional Functionality with Qt], QApplication::quit() [Additional Functionality with Qt] and Event Classes.

Member Function Documentation

QCloseEvent::QCloseEvent ()

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

See also accept() [p. 45].

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's WDestructiveClose flag is set, it will also be destroyed.

See also ignore() [p. 45] and QWidget::hide() [Widgets with Qt].

Examples: action/application.cpp, application/application.cpp, popup/popup.cpp and qwerty/qwerty.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 close event is constructed with the accept flag cleared.

See also accept() [p. 45].

Examples: action/application.cpp, application/application.cpp and qwerty/qwerty.cpp.

bool QCloseEvent::isAccepted () const

Returns TRUE if the receiver of the event has agreed to close the widget; otherwise returns FALSE.

See also accept() [p. 45] and ignore() [p. 45].

QColorDrag Class Reference

The QColorDrag class provides a drag and drop object for transferring colors.

`#include <qdragobject.h>`

Inherits QStoredDrag [p. 132].

Public Members

- **QColorDrag** (const QColor & col, QWidget * dragsource = 0, const char * name = 0)
- **QColorDrag** (QWidget * dragsource = 0, const char * name = 0)
- void **setColor** (const QColor & col)

Static Public Members

- bool **canDecode** (QMimeSource * e)
- bool **decode** (QMimeSource * e, QColor & col)

Detailed Description

The QColorDrag class provides a drag and drop object for transferring colors.

This class provides a drag object which can be used to transfer data about colors for drag and drop and over the clipboard. For example, it is used in the QColorDialog.

The color is set in the constructor but can be changed with setColor().

For more information about drag and drop, see the QDragObject class and the drag and drop documentation.

See also Drag And Drop Classes.

Member Function Documentation

QColorDrag::QColorDrag (const QColor & col, QWidget * dragsource = 0, const char * name = 0)

Constructs a color drag object with the color *col*. Passes *dragsource* and *name* to the QStoredDrag constructor.

QColorDrag::QColorDrag (QWidget * dragsource = 0, const char * name = 0)

Constructs a color drag object with a white color Passes *dragsource* and *name* to the QStoredDrag constructor.

bool QColorDrag::canDecode (QMimeSource * e) [static]

Returns TRUE if the color drag object can decode the mime source *e*.

bool QColorDrag::decode (QMimeSource * e, QColor & col) [static]

Decodes the mime source *e* and sets the decoded values to *col*.

void QColorDrag::setColor (const QColor & col)

Sets the color of the color drag to *col*.

QCommonStyle Class Reference

The QCommonStyle class encapsulates the common Look and Feel of a GUI.

`#include <qcommonstyle.h>`

Inherits QStyle [p. 134].

Inherited by QWindowsStyle [p. 179] and QMotifStyle [p. 116].

Public Members

- **QCommonStyle** ()

Detailed Description

The QCommonStyle class encapsulates the 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.

All the functions are documented in QStyle.

See also Widget Appearance and Style.

Member Function Documentation

QCommonStyle::QCommonStyle ()

Constructs a QCommonStyle.

QContextMenuEvent Class Reference

The QContextMenuEvent class contains parameters that describe a context menu event.

`#include <qevent.h>`

Inherits QEvent [p. 67].

Public Members

- enum **Reason** { Mouse, Keyboard, Other }
- **QContextMenuEvent** (Reason reason, const QPoint & pos, const QPoint & globalPos, int state)
- **QContextMenuEvent** (Reason reason, const QPoint & pos, int state)
- int **x** () const
- int **y** () const
- int **globalX** () const
- int **globalY** () const
- const QPoint & **pos** () const
- const QPoint & **globalPos** () const
- ButtonState **state** () const
- bool **isAccepted** () const
- bool **isConsumed** () const
- void **consume** ()
- void **accept** ()
- void **ignore** ()
- Reason **reason** () const

Detailed Description

The QContextMenuEvent class contains parameters that describe a context menu event.

Context events are sent to widgets when a user triggers a menu. What triggers this is platform dependent. On windows, for example, pressing the menu button or releasing the right mouse button will cause this event to be sent. It is customary to use this to show a QPopupMenu when this event is triggered if you have a relevant context menu.

ContextMenu events contain a special accept flag that indicates whether the receiver accepted the contextMenu. If the event handler does not accept the event, then whatever triggered the event will be handled as a regular input event if possible.

See also QPopupMenu [Dialogs and Windows with Qt] and Event Classes.

Member Type Documentation

QContextMenuEvent::Reason

This enum describes the reason the ContextMenuEvent was sent. The values are:

- `QContextMenuEvent::Mouse` - The mouse caused the event to be sent. Normally this means the right mouse button was clicked, but this is platform specific.
- `QContextMenuEvent::Keyboard` - The keyboard caused this event to be sent. On windows this means the menu button was pressed.
- `QContextMenuEvent::Other` - The event was sent by some other means (i.e. not by the mouse or keyboard).

Member Function Documentation

QContextMenuEvent::QContextMenuEvent (Reason reason, const QPoint & pos, const QPoint & globalPos, int state)

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

The *reason* parameter must be QContextMenuEvent::Mouse or QContextMenuEvent::Keyboard.

The *pos* parameter specifies the mouse position relative to the receiving widget. *globalPos* is the mouse position in absolute coordinates. *state* is the ButtonState at the time of the event.

QContextMenuEvent::QContextMenuEvent (Reason reason, const QPoint & pos, int state)

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

The *reason* parameter must be QContextMenuEvent::Mouse or QContextMenuEvent::Keyboard.

The *pos* parameter specifies the mouse position relative to the receiving widget. *state* is the ButtonState at the time of the event.

The globalPos() is initialized to QCursor::pos(), which may not be appropriate. Use the other constructor to specify the global position explicitly.

void QContextMenuEvent::accept ()

Sets the accept flag of the context event object.

Setting the accept flag indicates that the receiver of this event has processed the event. Processing the event means you did something with it and it will be implicitly consume().

The accept flag is not set by default.

See also ignore() [p. 51] and consume() [p. 50].

void QContextMenuEvent::consume ()

Sets the consume flag of the context event object.

Setting the consume flag indicates that the receiver of this event asked that the event not be propagated further (to parent classes).

The consumed flag is not set by default.

See also ignore() [p. 51] and accept() [p. 50].

const QPoint & QContextMenuEvent::globalPos () const

Returns the global position of the mouse pointer at the time of the event.

See also x() [p. 52], y() [p. 52] and pos() [p. 51].

int QContextMenuEvent::globalX () const

Returns the global X position of the mouse pointer at the time of the event.

See also globalY() [p. 51] and globalPos() [p. 51].

int QContextMenuEvent::globalY () const

Returns the global Y position of the mouse pointer at the time of the event.

See also globalX() [p. 51] and globalPos() [p. 51].

void QContextMenuEvent::ignore ()

Clears the accept flag of the context event object.

Clearing the accept flag indicates that the receiver of this event does not need to show a context menu. This will implicitly remove the consumed flag as well.

The accept flag is not set by default.

See also accept() [p. 50] and consume() [p. 50].

bool QContextMenuEvent::isAccepted () const

Returns TRUE if the receiver has processed the event; otherwise returns FALSE.

See also accept() [p. 50], ignore() [p. 51] and consume() [p. 50].

bool QContextMenuEvent::isConsumed () const

Returns TRUE (which stops propagation of the event) if the receiver has blocked the event; otherwise returns FALSE.

See also accept() [p. 50], ignore() [p. 51] and consume() [p. 50].

const QPoint & QContextMenuEvent::pos () const

Returns the position of the mouse pointer relative to the widget that received the event.

See also x() [p. 52], y() [p. 52] and globalPos() [p. 51].

Reason QContextMenuEvent::reason () const

Returns the reason for this context event.

ButtonState QContextMenuEvent::state () const

Returns the button state (a combination of mouse buttons and keyboard modifiers), i.e., what buttons and keys were being pressed immediately before the event was generated.

The returned value is LeftButton, RightButton, MidButton, ShiftButton, ControlButton and AltButton OR'ed together.

int QContextMenuEvent::x () const

Returns the X position of the mouse pointer, relative to the widget that received the event.

See also y() [p. 52] and pos() [p. 51].

int QContextMenuEvent::y () const

Returns the Y position of the mouse pointer, relative to the widget that received the event.

See also x() [p. 52] and pos() [p. 51].

QCustomEvent Class Reference

The QCustomEvent class provides support for custom events.

`#include <qevent.h>`

Inherits QEvent [p. 67].

Public Members

- **QCustomEvent** (int type)
- **QCustomEvent** (Type type, void * data)
- void * **data** () const
- void **setData** (void * data)

Detailed Description

The QCustomEvent class provides support for custom events.

QCustomEvent is a generic event class for user-defined events. User defined events can be sent to widgets or other QObject instances using QApplication::postEvent() or QApplication::sendEvent(). Subclasses of QWidget can easily receive custom events by implementing the QWidget::customEvent() event handler function.

QCustomEvent objects should be created with a type id that uniquely identifies the event type. To avoid clashes with the Qt-defined events types, the value should be at least as large as the value of the "User" entry in the QEvent::Type enum.

QCustomEvent contains a generic void* data member that may be used for transferring event-specific data to the receiver. Note that since events are normally delivered asynchronously, the data pointer, if used, must remain valid until the event has been received and processed.

QCustomEvent can be used as-is for simple user-defined event types, but normally you will want to make a subclass of it for your event types. In a subclass, you can add data members that are suitable for your event type.

Example:

```
class ColorChangeEvent : public QCustomEvent
{
public:
  ColorChangeEvent( QColor color )
      : QCustomEvent( 346798 ), c( color ) {};
  QColor color() const { return c; };
private:
  QColor c;
};

// To send an event of this custom event type:
```

```
ColorChangeEvent* ce = new ColorChangeEvent( blue );
QApplication::postEvent( receiver, ce );  // Qt will delete it when done

// To receive an event of this custom event type:

void MyWidget::customEvent( QCustomEvent * e )
{
  if ( e->type() == 346798 ) {  // It must be a ColorChangeEvent
    ColorChangeEvent* ce = (ColorChangeEvent*)e;
    newColor = ce->color();
  }
}
```

See also QWidget::customEvent() [Additional Functionality with Qt], QApplication::notify() [Additional Functionality with Qt] and Event Classes.

Member Function Documentation

QCustomEvent::QCustomEvent (int type)

Constructs a custom event object with event type *type*. The value of *type* must be at least as large as QEvent::User. The data pointer is set to 0.

QCustomEvent::QCustomEvent (Type type, void * data)

Constructs a custom event object with the event type *type* and a pointer to *data*. (Note that any int value may safely be cast to QEvent::Type).

void * QCustomEvent::data () const

Returns a pointer to the generic event data.

See also setData() [p. 54].

void QCustomEvent::setData (void * data)

Sets the generic data pointer to *data*.

See also data() [p. 54].

QDragEnterEvent Class Reference

The QDragEnterEvent class provides an event which is sent to the widget when a drag and drop first drags onto the widget.

`#include <qevent.h>`

Inherits QDragMoveEvent [p. 57].

Public Members

- **QDragEnterEvent** (const QPoint & pos)

Detailed Description

The QDragEnterEvent class provides an event which is sent to the widget when a drag and drop first drags onto the widget.

This event is always immediately followed by a QDragMoveEvent, so you only need to respond to one or the other event. This class inherits most of its functionality from QDragMoveEvent, which in turn inherits most of its functionality from QDropEvent.

See also QDragLeaveEvent [p. 56], QDragMoveEvent [p. 57], QDropEvent [p. 63], Drag And Drop Classes and Event Classes.

Member Function Documentation

QDragEnterEvent::QDragEnterEvent (const QPoint & pos)

Constructs a QDragEnterEvent entering at the given point, *pos*.

Do not create a QDragEnterEvent yourself since these objects rely on Qt's internal state.

QDragLeaveEvent Class Reference

The QDragLeaveEvent class provides an event which is sent to the widget when a drag and drop leaves the widget.

`#include <qevent.h>`

Inherits QEvent [p. 67].

Public Members

- **QDragLeaveEvent** ()

Detailed Description

The QDragLeaveEvent class provides an event which is sent to the widget when a drag and drop leaves the widget.

This event is always preceded by a QDragEnterEvent and a series of QDragMoveEvents. It is not sent if a QDropEvent is sent instead.

See also QDragEnterEvent [p. 55], QDragMoveEvent [p. 57], QDropEvent [p. 63], Drag And Drop Classes and Event Classes.

Member Function Documentation

QDragLeaveEvent::QDragLeaveEvent ()

Constructs a QDragLeaveEvent.

Do not create a QDragLeaveEvent yourself since these objects rely on Qt's internal state.

QDragMoveEvent Class Reference

The QDragMoveEvent class provides an event which is sent while a drag-and-drop is in progress.

`#include <qevent.h>`

Inherits QDropEvent [p. 63].

Inherited by QDragEnterEvent [p. 55].

Public Members

- **QDragMoveEvent** (const QPoint & pos, Type type = DragMove)
- QRect **answerRect** () const
- void **accept** (const QRect & r)
- void **ignore** (const QRect & r)

Detailed Description

The QDragMoveEvent class provides an event which is sent while a drag-and-drop is in progress.

When a widget accepts drop events, it will receive this event repeatedly while the drag is within the widget's boundaries. The widget should examine the event to see what data it provides, and accept() the drop if appropriate.

Note that this class inherits most of its functionality from QDropEvent.

See also Drag And Drop Classes and Event Classes.

Member Function Documentation

QDragMoveEvent::QDragMoveEvent (const QPoint & pos, Type type = DragMove)

Creates a QDragMoveEvent for which the mouse is at point *pos*, and the event is of type *type*.

Do not create a QDragMoveEvent yourself since these objects rely on Qt's internal state.

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

Returns the rectangle for which the acceptance of the move event applies.

void QDragMoveEvent::ignore (const QRect & r)

The opposite of accept(const QRect&), i.e. says that moves within rectangle *r* are not acceptable (will be ignored). Example: dirview/dirview.cpp.

QDragObject Class Reference

The QDragObject class encapsulates MIME-based data transfer.

`#include <qdragobject.h>`

Inherits QObject [Additional Functionality with Qt] and QMimeSource [Input/Output and Networking with Qt].

Inherited by QStoredDrag [p. 132], QTextDrag [p. 168], QImageDrag [p. 93] and QIconDrag [p. 89].

Public Members

- **QDragObject** (QWidget * dragSource = 0, const char * name = 0)
- virtual **~QDragObject** ()
- bool **drag** ()
- bool **dragMove** ()
- void **dragCopy** ()
- void **dragLink** ()
- virtual void **setPixmap** (QPixmap pm)
- virtual void **setPixmap** (QPixmap pm, const QPoint & hotspot)
- QPixmap **pixmap** () const
- QPoint **pixmapHotSpot** () const
- QWidget * **source** ()
- enum **DragMode** { DragDefault, DragCopy, DragMove, DragLink, DragCopyOrMove }

Static Public Members

- QWidget * **target** ()

Protected Members

- virtual bool **drag** (DragMode mode)

Detailed Description

The QDragObject class encapsulates MIME-based data 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 [Programming with Qt] for an overview of how to provide drag and drop in your application.

See the QClipboard [Input/Output and Networking with Qt] documentation for an overview of how to provide cut-and-paste in your application.

The drag() function is used to start a drag operation. You can specify the DragMode in the call or use one of the convenience functions dragCopy(), dragMove() or dragLink(). The drag source where the data originated is retrieved with source(). If the data was dropped on a widget within the application target() will return a pointer to that widget. Specify the pixmap to display during the drag with setPixmap().

See also Drag And Drop Classes.

Member Type Documentation

QDragObject::DragMode

This enum describes the possible drag modes.

- `QDragObject::DragDefault` - The mode is determined heuristically.
- `QDragObject::DragCopy` - The data is copied, never moved.
- `QDragObject::DragMove` - The data is moved, if dragged at all.
- `QDragObject::DragLink` - The data is linked, if dragged at all.
- `QDragObject::DragCopyOrMove` - The user chooses the mode by using a control key to switch from the default.

Member Function Documentation

QDragObject::QDragObject (QWidget * dragSource = 0, const char * name = 0)

Constructs a drag object called *name*, which is a child of *dragSource*.

Note that the drag object will be deleted when *dragSource* is deleted.

QDragObject::~QDragObject () [virtual]

Destroys the drag object, canceling any drag and drop operation in which it is involved, 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 see target()).

If the drag contains *references* to information (eg. file names in 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!)

Example: dirview/dirview.cpp.

bool QDragObject::drag (DragMode mode) [virtual protected]

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

Starts a drag operation using the contents of this object.

At this point, the object becomes owned by Qt, not the application. You should not delete the drag object nor anything it references. The actual transfer of data to the target application will be done during future event processing - after that time the drag object will be deleted.

Returns TRUE if the dragged data was dragged as a *move*, indicating that the caller should remove the original source of the data (the drag object must continue to have a copy).

The *mode* specifies the drag mode (see QDragObject::DragMode.) Normally one of the simpler drag(), dragMove(), or dragCopy() functions would be used instead.

Warning: in Qt 1.x, drag operations all return FALSE.

void QDragObject::dragCopy ()

Starts a drag operation using the contents of this object, using DragCopy mode. Be sure to read the constraints described in drag().

See also drag() [p. 60], dragMove() [p. 61] and dragLink() [p. 61].

void QDragObject::dragLink ()

Starts a drag operation using the contents of this object, using DragLink mode. Be sure to read the constraints described in drag().

See also drag() [p. 60], dragCopy() [p. 61] and dragMove() [p. 61].

bool QDragObject::dragMove ()

Starts a drag operation using the contents of this object, using DragMove mode. Be sure to read the constraints described in drag().

See also drag() [p. 60], dragCopy() [p. 61] and dragLink() [p. 61].

QPixmap QDragObject::pixmap () const

Returns the currently set pixmap (which isNull() if none is set).

QPoint QDragObject::pixmapHotSpot () const

Returns the currently set pixmap hotspot.

void QDragObject::setPixmap (QPixmap pm, const QPoint & hotspot) [virtual]

Set the pixmap *pm* to display while dragging the object. The platform-specific implementation will use this where it can - so provide a small masked pixmap, and do not assume that the user will actually see it. For example, 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.

Example: fileiconview/qfileiconview.cpp.

void QDragObject::setPixmap (QPixmap pm) [virtual]

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

Uses a hotspot that positions the pixmap below and to the right of the mouse pointer. This allows the user to clearly see the point on the window which they are dragging the data onto.

QWidget * QDragObject::source ()

Returns a pointer to the drag source where this object originated.

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 another application.

This can be useful for detecting the case where drag and drop is to and from the same widget.

QDropEvent Class Reference

The QDropEvent class provides an event which is sent when a drag and drop is completed.

`#include <qevent.h>`

Inherits QEvent [p. 67] and QMimeSource [Input/Output and Networking with Qt].

Inherited by QDragMoveEvent [p. 57].

Public Members

- **QDropEvent** (const QPoint & pos, Type typ = Drop)
- const QPoint & **pos** () const
- bool **isAccepted** () const
- void **ignore** ()
- bool **isActionAccepted** () const
- void **acceptAction** (bool y = TRUE)
- enum **Action** { Copy, Link, Move, Private, UserAction = 100 }
- void **setAction** (Action a)
- Action **action** () const
- QWidget * **source** () const
- virtual const char * **format** (int n = 0) const
- virtual QByteArray **encodedData** (const char * format) const
- virtual bool **provides** (const char * mimeType) const
- QByteArray data (const char * f) const *(obsolete)*
- void **setPoint** (const QPoint & np)

Detailed Description

The QDropEvent class provides an event which is 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.

See also Drag And Drop Classes and Event Classes.

Member Type Documentation

QDropEvent::Action

This enum describes the action which a source requests that a target perform with dropped data.

- `QDropEvent::Copy` - The default action. The source simply uses the data provided in the operation.
- `QDropEvent::Link` - The source should somehow create a link to the location specified by the data.
- `QDropEvent::Move` - The source should somehow move the object from the location specified by the data to a new location.
- `QDropEvent::Private` - The target has special knowledge of the MIME type, which the source should respond to in a similar way to a Copy.
- `QDropEvent::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, for example, text/uri-list file lists (see QUriDrag).

Member Function Documentation

QDropEvent::QDropEvent (const QPoint & pos, Type typ = Drop)

Constructs a drop event that drops a drop of type *typ* on point *pos*.

void QDropEvent::acceptAction (bool y = TRUE)

Call this to indicate that the action described by action() is accepted (i.e. if *y* is TRUE which is the default), not merely the default copy action. If you call acceptAction(TRUE), there is no need to also call accept(TRUE).

Examples: dirview/dirview.cpp and fileiconview/qfileiconview.cpp.

Action QDropEvent::action () const

Returns the Action which the target is requesting to 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().

Examples: dirview/dirview.cpp and fileiconview/qfileiconview.cpp.

QByteArray QDropEvent::data (const char * f) const

This function is obsolete. It is provided to keep old source 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() [p. 65] and QByteArray::size() [Datastructures and String Handling with Qt].

Reimplemented from QMimeSource [Input/Output and Networking with Qt].

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() [p. 64] and provides() [p. 65].

Example: iconview/main.cpp.

Reimplemented from QMimeSource [Input/Output and Networking with Qt].

void QDropEvent::ignore ()

The opposite of accept(), i.e. you have ignored the drop event.

Example: fileiconview/qfileiconview.cpp.

bool QDropEvent::isAccepted () const

Returns TRUE if the drop target accepts the event; otherwise returns FALSE.

bool QDropEvent::isActionAccepted () const

Returns TRUE if the drop action was accepted by the drop site; otherwise returns FALSE.

const QPoint & QDropEvent::pos () const

Returns the position where the drop was made.

Example: dirview/dirview.cpp.

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() [p. 64].

Example: fileiconview/qfileiconview.cpp.

Reimplemented from QMimeSource [Input/Output and Networking with Qt].

void QDropEvent::setAction (Action a)

Sets the action to *a*. This is used internally, you should not need to call this in your code - the *source* decides the action, not the target.

void QDropEvent::setPoint (const QPoint & np)

Sets the drop to happen at point *np*. You do not normally need to use this as it will be set internally before your widget receives the drop event.

QWidget * QDropEvent::source () const

If the source of the drag operation is a widget in this application, this function returns that source, otherwise it returns 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 itself, etc.

See QDragObject::QDragObject() and subclasses.

QEvent Class Reference

The QEvent class is the base class of all event classes. Event objects contain event parameters.

`#include <qevent.h>`

Inherits Qt [Additional Functionality with Qt].

Inherited by QTimerEvent [p. 170], QMouseEvent [p. 118], QWheelEvent [p. 176], QTabletEvent, QKeyEvent [p. 95], QFocusEvent [p. 71], QPaintEvent [p. 125], QMoveEvent [p. 122], QResizeEvent [p. 129], QCloseEvent [p. 44], QShowEvent [p. 131], QHideEvent [p. 88], QContextMenuEvent [p. 49], QIMEvent [Accessibility and Internationalization with Qt], QDropEvent [p. 63], QDragLeaveEvent [p. 56], QChildEvent [p. 42] and QCustomEvent [p. 53].

Public Members

- enum Type { None = 0, Timer = 1, MouseButtonPress = 2, MouseButtonRelease = 3, MouseButtonDblClick = 4, MouseMove = 5, KeyPress = 6, KeyRelease = 7, FocusIn = 8, FocusOut = 9, Enter = 10, Leave = 11, Paint = 12, Move = 13, Resize = 14, Create = 15, Destroy = 16, Show = 17, Hide = 18, Close = 19, Quit = 20, Reparent = 21, ShowMinimized = 22, ShowNormal = 23, WindowActivate = 24, WindowDeactivate = 25, ShowToParent = 26, HideToParent = 27, ShowMaximized = 28, ShowFullScreen = 29, Accel = 30, Wheel = 31, AccelAvailable = 32, CaptionChange = 33, IconChange = 34, ParentFontChange = 35, ApplicationFontChange = 36, ParentPaletteChange = 37, ApplicationPaletteChange = 38, PaletteChange = 39, Clipboard = 40, Speech = 42, SockAct = 50, AccelOverride = 51, DeferredDelete = 52, DragEnter = 60, DragMove = 61, DragLeave = 62, Drop = 63, DragResponse = 64, ChildInserted = 70, ChildRemoved = 71, LayoutHint = 72, ShowWindowRequest = 73, ActivateControl = 80, DeactivateControl = 81, ContextMenu = 82, IMStart = 83, IMCompose = 84, IMEnd = 85, Accessibility = 86, Tablet = 87, User = 1000, MaxUser = 65535 } *(obsolete)*
- **QEvent** (Type type)
- Type **type** () const
- bool **spontaneous** () const

Detailed Description

The QEvent class is the base class of all event classes. Event objects contain event parameters.

The main event loop of Qt (QApplication::exec()) fetches native window system events from the event queue, translates them into QEvents and sends the translated events to QObjects.

Generally, events come from the underlying window system (spontaneous() returns TRUE) but it is also possible to manually send events through the QApplication class using QApplication::sendEvent() and QApplication::postEvent() (spontaneous() returns FALSE).

QObjects receive events by having their QObject::event() function called. The function can be reimplemented in subclasses to customize event handling and add additional event types. QWidget::event() is a notable example.

By default, events are dispatched to event handlers like QObject::timerEvent() and QWidget::mouseMoveEvent(). QObject::installEventFilter() allows an object to intercept events to another object.

The basic QEvent contains only an event type parameter. Subclasses of QEvent contain additional parameters that describe the particular event.

See also QObject::event() [Additional Functionality with Qt], QObject::installEventFilter() [Additional Functionality with Qt], QWidget::event() [Widgets with Qt], QApplication::sendEvent() [Additional Functionality with Qt], QApplication::postEvent() [Additional Functionality with Qt], QApplication::processEvents() [Additional Functionality with Qt], Environment Classes and Event Classes.

Member Type Documentation

QEvent::Type

This enum type defines the valid event types in Qt. The currently defined event types and the specialized classes for each type are:

- `QEvent::None` - Not an event.
- `QEvent::Accessibility` - Accessibility information is requested
- `QEvent::Timer` - Regular timer events, QTimerEvent.
- `QEvent::MouseButtonPress` - Mouse press, QMouseEvent.
- `QEvent::MouseButtonRelease` - Mouse release, QMouseEvent.
- `QEvent::MouseButtonDblClick` - Mouse press again, QMouseEvent.
- `QEvent::MouseMove` - Mouse move, QMouseEvent.
- `QEvent::KeyPress` - Key press (including Shift, for example), QKeyEvent.
- `QEvent::KeyRelease` - Key release, QKeyEvent.
- `QEvent::IMStart` - The start of input method composition.
- `QEvent::IMCompose` - Input method composition is taking place.
- `QEvent::IMEnd` - The end of input method composition.
- `QEvent::FocusIn` - Widget gains keyboard focus, QFocusEvent.
- `QEvent::FocusOut` - Widget loses keyboard focus, QFocusEvent.
- `QEvent::Enter` - Mouse enters widget's boundaries.
- `QEvent::Leave` - Mouse leaves widget's boundaries.
- `QEvent::Paint` - Screen update necessary, QPaintEvent.
- `QEvent::Move` - Widget's position changed, QMoveEvent.
- `QEvent::Resize` - Widget's size changed, QResizeEvent.
- `QEvent::Show` - Widget was shown on screen, QShowEvent.
- `QEvent::Hide` - Widget was hidden, QHideEvent.
- `QEvent::ShowToParent` - A child widget has been shown.
- `QEvent::HideToParent` - A child widget has been hidden.
- `QEvent::Close` - Widget was closed (permanently), QCloseEvent.
- `QEvent::ShowNormal` - Widget should be shown normally.
- `QEvent::ShowMaximized` - Widget should be shown maximized.
- `QEvent::ShowMinimized` - Widget should be shown minimized.
- `QEvent::ShowFullScreen` - Widget should be shown full-screen.

- `QEvent::ShowWindowRequest` - Widget's window should be shown. **This type is obsolete.** It is provided to keep old source working. We strongly advise against using it in new code.
- `QEvent::DeferredDelete` - The object will be deleted after it has cleaned up.
- `QEvent::Accel` - Key press in child for shortcut key handling, QKeyEvent.
- `QEvent::Wheel` - Mouse wheel rolled, QWheelEvent.
- `QEvent::ContextMenu` - context popup menu, QContextMenuEvent
- `QEvent::AccelAvailable` - Internal event used by Qt on some platforms.
- `QEvent::AccelOverride` - Key press in child, for overriding shortcut key handling, QKeyEvent.
- `QEvent::WindowActivate` - Window was activated.
- `QEvent::WindowDeactivate` - Window was deactivated.
- `QEvent::CaptionChange` - Widget's caption changed.
- `QEvent::IconChange` - Widget's icon changed.
- `QEvent::ParentFontChange` - Font of the parent widget changed.
- `QEvent::ApplicationFontChange` - Default application font changed.
- `QEvent::PaletteChange` - Palette of the widget changed.
- `QEvent::ParentPaletteChange` - Palette of the parent widget changed.
- `QEvent::ApplicationPaletteChange` - Default application palette changed.
- `QEvent::Clipboard` - Clipboard contents have changed.
- `QEvent::SockAct` - Socket activated, used to implement QSocketNotifier.
- `QEvent::DragEnter` - A drag-and-drop enters widget, QDragEnterEvent.
- `QEvent::DragMove` - A drag-and-drop is in progress, QDragMoveEvent.
- `QEvent::DragLeave` - A drag-and-drop leaves widget, QDragLeaveEvent.
- `QEvent::Drop` - A drag-and-drop is completed, QDropEvent.
- `QEvent::DragResponse` - Internal event used by Qt on some platforms.
- `QEvent::ChildInserted` - Object gets a child, QChildEvent.
- `QEvent::ChildRemoved` - Object loses a child, QChildEvent.
- `QEvent::LayoutHint` - Widget child has changed layout properties.
- `QEvent::ActivateControl` - Internal event used by Qt on some platforms.
- `QEvent::DeactivateControl` - Internal event used by Qt on some platforms.
- `QEvent::Quit` - Reserved.
- `QEvent::Create` - Reserved.
- `QEvent::Destroy` - Reserved.
- `QEvent::Reparent` - Reserved.
- `QEvent::Speech` - Reserved for speech input.
- `QEvent::Tablet` - Wacom Tablet event.
- `QEvent::User` - User defined event.
- `QEvent::MaxUser` - Last user event id.

User events should have values between User and MaxUser inclusive.

Member Function Documentation

QEvent::QEvent (Type type)

Contructs an event object of type *type*.

bool QEvent::spontaneous () const

Returns TRUE if the event originated outside the application, i.e. it is a system event; otherwise returns FALSE.

Type QEvent::type () const

Returns the event type.

QFocusEvent Class Reference

The QFocusEvent class contains event parameters for widget focus events.

`#include <qevent.h>`

Inherits QEvent [p. 67].

Public Members

- **QFocusEvent** (Type type)
- bool **gotFocus** () const
- bool **lostFocus** () const
- enum **Reason** { Mouse, Tab, Backtab, ActiveWindow, Popup, Shortcut, Other }

Static Public Members

- Reason **reason** ()
- void **setReason** (Reason reason)
- void **resetReason** ()

Detailed Description

The QFocusEvent class contains event parameters for widget focus events.

Focus events are sent to widgets when the keyboard input focus changes. Focus events occur due to mouse actions, keypresses (e.g. Tab or Backtab), the window system, popup menus, keyboard shortcuts or other application specific reasons. The reason for a particular focus event is returned by reason() in the appropriate event handler.

The event handlers QWidget::focusInEvent() and QWidget::focusOutEvent() receive focus events.

Use setReason() to set the reason for all focus events, and resetReason() to set the reason for all focus events to the reason in force before the last setReason() call.

See also QWidget::setFocus() [Widgets with Qt], QWidget::focusPolicy [Widgets with Qt] and Event Classes.

Member Type Documentation

QFocusEvent::Reason

This enum specifies why the focus changed:

- `QFocusEvent::Mouse` - because of a mouse action.
- `QFocusEvent::Tab` - because of a Tab press
- `QFocusEvent::Backtab` - because of a Backtab press (possibly including Shift/Control, e.g. Shift+Tab).
- `QFocusEvent::ActiveWindow` - because the window system made this window (in)active.
- `QFocusEvent::Popup` - because the application opened/closed a popup that grabbed/released focus.
- `QFocusEvent::Shortcut` - because of a keyboard shortcut.
- `QFocusEvent::Other` - any other reason, usually application-specific.

See the keyboard focus [Programming with Qt] overview for more about focus.

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; otherwise returns FALSE.

bool QFocusEvent::lostFocus () const

Returns TRUE if the widget lost the text input focus; otherwise returns FALSE.

Reason QFocusEvent::reason () [static]

Returns the reason for this focus event.

See also setReason() [p. 72].

void QFocusEvent::resetReason () [static]

Resets the reason for all future focus events to the value before the last setReason() call.

See also reason() [p. 72] and setReason() [p. 72].

void QFocusEvent::setReason (Reason reason) [static]

Sets the reason for all future focus events to *reason*.

See also reason() [p. 72] and resetReason() [p. 72].

QGLayoutIterator Class Reference

The QGLayoutIterator class is an abstract base class of internal layout iterators.

`#include <qlayout.h>`

Public Members

- virtual **~QGLayoutIterator** ()
- virtual QLayoutItem * **next** ()
- virtual QLayoutItem * **current** ()
- virtual QLayoutItem * **takeCurrent** ()

Detailed Description

The QGLayoutIterator class is an abstract base class of internal layout iterators.

(This class is *not* OpenGL related, it just happens to start with the letters QGL...)

Subclass this class to create a custom layout. The functions that must be implemented are next(), current(), and takeCurrent().

The QGLayoutIterator implements the functionality of QLayoutIterator. Each subclass of QLayout needs a QGLayoutIterator subclass.

See also Widget Appearance and Style and Layout Management.

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.

Examples: customlayout/border.cpp, customlayout/card.cpp and customlayout/flow.cpp.

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.

Examples: customlayout/border.cpp, customlayout/card.cpp and customlayout/flow.cpp.

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.

Examples: customlayout/border.cpp, customlayout/card.cpp and customlayout/flow.cpp.

QGrid Class Reference

The QGrid widget provides simple geometry management of its children.

`#include <qgrid.h>`

Inherits QFrame [Widgets with Qt].

Public Members

- **QGrid** (int n, QWidget * parent = 0, const char * name = 0, WFlags f = 0)
- **QGrid** (int n, Orientation orient, QWidget * parent = 0, const char * name = 0, WFlags f = 0)
- void **setSpacing** (int space)

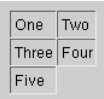
Detailed Description

The QGrid widget provides simple geometry management of its children.

The grid places its widgets either in a single column or in a single row. If you want a multi-column, multi-row grid use QGridLayout.

The number of rows *or* columns is defined in the constructor. All the grid's children will be placed and sized in accordance with their sizeHint() and sizePolicy().

Use setMargin() to add space around the grid itself, and setSpacing() to add space between the widgets.



See also QVBox [Widgets with Qt], QHBox [p. 84], Widget Appearance and Style and Layout Management.

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. The widget flags *f* are passed to the QFrame constructor.

QGrid::QGrid (int n, Orientation orient, QWidget * parent = 0, const char * name = 0, WFlags f = 0)

Constructs a grid widget with parent *parent* and name *name*. If *orient* is Horizontal, *n* specifies the number of columns. If *orient* is Vertical, *n* specifies the number of rows. The widget flags *f* are passed to the QFrame constructor.

void QGrid::setSpacing (int space)

Sets the spacing between the child widgets to *space*.

QGridLayout Class Reference

The QGridLayout class lays out widgets in a grid.

`#include <qlayout.h>`

Inherits QLayout [p. 101].

Public Members

- **QGridLayout** (QWidget * parent, int nRows = 1, int nCols = 1, int margin = 0, int space = -1, const char * name = 0)
- **QGridLayout** (int nRows = 1, int nCols = 1, int spacing = -1, const char * name = 0)
- **QGridLayout** (QLayout * parentLayout, int nRows = 1, int nCols = 1, int spacing = -1, const char * name = 0)
- **~QGridLayout** ()
- virtual QSize **sizeHint** () const
- virtual QSize **minimumSize** () const
- virtual QSize **maximumSize** () const
- virtual void **setRowStretch** (int row, int stretch)
- virtual void **setColStretch** (int col, int stretch)
- int **rowStretch** (int row) const
- int **colStretch** (int col) const
- int **numRows** () const
- int **numCols** () const
- QRect **cellGeometry** (int row, int col) const
- virtual bool **hasHeightForWidth** () const
- virtual int **heightForWidth** (int w) const
- virtual QSizePolicy::ExpandData **expanding** () const
- virtual void **invalidate** ()
- virtual void **addItem** (QLayoutItem * item)
- void **addItem** (QLayoutItem * item, int row, int col)
- void **addMultiCell** (QLayoutItem * item, int fromRow, int toRow, int fromCol, int toCol, int alignment = 0)
- void **addWidget** (QWidget * w, int row, int col, int alignment = 0)
- void **addMultiCellWidget** (QWidget * w, int fromRow, int toRow, int fromCol, int toCol, int alignment = 0)
- void **addLayout** (QLayout * layout, int row, int col)
- void **addMultiCellLayout** (QLayout * layout, int fromRow, int toRow, int fromCol, int toCol, int alignment = 0)
- void **addRowSpacing** (int row, int minsize)
- void **addColSpacing** (int col, int minsize)
- void **expand** (int nRows, int nCols)
- enum **Corner** { TopLeft, TopRight, BottomLeft, BottomRight }

- void **setOrigin** (Corner c)
- Corner **origin** () const
- virtual void **setGeometry** (const QRect & r)

Protected Members

- bool **findWidget** (QWidget * w, int * row, int * col)
- void **add** (QLayoutItem * item, int row, int col)

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 widget it manages into the correct cell.

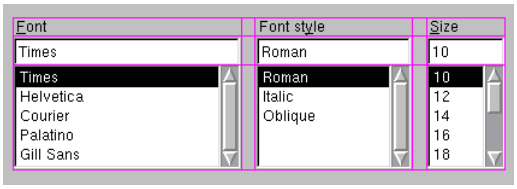
Columns and rows behave identically; we will discuss columns, but there are equivalent functions for rows.

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 widgets into multiple cells using addMultiCellWidget(). If you do this, QGridLayout will guess how to distribute the size over the columns/rows (based on the stretch factors).

To remove a widget from a layout, either delete it or reparent it with QWidget::reparent(). Hiding a widget with QWidget::hide() also effectively removes the widget from the layout, until QWidget::show() is called.

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 3 are placeholders made with addColSpacing(). Row 0 consists of three QLabel objects, row 1 of three QLineEdit objects and row 2 of three QListBox objects. We used placeholder columns (1 and 3) to get the right amount of space between the columns.

Note that the columns and rows are not equally wide or tall. If you want two columns to have the same width, you must set their minimum widths and stretch factors to be the same yourself. You do this using addColSpacing() and setColStretch().

If the QGridLayout is not the top-level layout (i.e. does not manage all of the widget's area and children), you must add it to its parent layout when you create it, but before you do anything with it. The normal way to add a layout is by calling parentLayout->addLayout().

Once you have added your layout you can start putting widgets and other layouts into the cells of your grid layout using addWidget(), addLayout() and addMultiCellWidget().

QGridLayout also includes two margin widths: the border and the spacing. The border is the width of the reserved space along each of the QGridLayout's four sides. The spacing is the width of the automatically allocated spacing between neighboring boxes.

Both the border and the spacing are parameters of the constructor and default to 0.

See also Layout Overview [Programming with Qt], Widget Appearance and Style and Layout Management.

Member Type Documentation

QGridLayout::Corner

This enum identifies which corner is the origin (0, 0) of the layout.

- `QGridLayout::TopLeft` - the top-left corner
- `QGridLayout::TopRight` - the top-right corner
- `QGridLayout::BottomLeft` - the bottom-left corner
- `QGridLayout::BottomRight` - the bottom-right corner

Member Function Documentation

QGridLayout::QGridLayout (QWidget * parent, int nRows = 1, int nCols = 1, int margin = 0, int space = -1, const char * name = 0)

Constructs a new QGridLayout with *nRows* rows, *nCols* columns with parent widget, *parent*. *parent* may not be 0. The grid layout is called *name*.

margin is the number of pixels between the edge of the widget and its managed children. *space* is the default number of pixels between cells. If *space* is -1, the value of *margin* is used.

QGridLayout::QGridLayout (int nRows = 1, int nCols = 1, int spacing = -1, const char * name = 0)

Constructs a new grid with *nRows* rows and *nCols* columns. If *spacing* is -1, this QGridLayout inherits its parent's spacing(); otherwise *spacing* is used. The grid layout is called *name*.

You must insert this grid into another layout. You can insert widgets and layouts into this layout at any time, but layout will not be performed before this is inserted into another layout.

QGridLayout::QGridLayout (QLayout * parentLayout, int nRows = 1, int nCols = 1, int spacing = -1, const char * name = 0)

Constructs a new grid that is placed inside *parentLayout* with *nRows* rows and *nCols* columns. If *spacing* is -1, this QGridLayout inherits its parent's spacing(); otherwise *spacing* is used. The grid layout is called *name*.

This grid is placed according to *parentLayout*'s default placement rules.

QGridLayout::~QGridLayout ()

Destroys the grid layout. 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 ownership of the *item*.

void QGridLayout::addColSpacing (int col, int minsize)

Sets the minimum width of column *col* to *minsize* pixels.

void QGridLayout::addItem (QLayoutItem * item, int row, int col)

Adds *item* at position *row*, *col*. The layout takes ownership of the *item*.

void QGridLayout::addItem (QLayoutItem * item) [virtual]

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

Adds *item* to the next free position of this layout.

Reimplemented from QLayout [p. 103].

void QGridLayout::addLayout (QLayout * layout, int row, int col)

Places the *layout* at position (*row*, *col*) in the grid. The top-left position is (0, 0).

Examples: listbox/listbox.cpp, progressbar/progressbar.cpp, t10/main.cpp and t13/gamebrd.cpp.

void QGridLayout::addMultiCell (QLayoutItem * item, int fromRow, int toRow, int fromCol, int toCol, int alignment = 0)

Adds the *item* to the cell grid, spanning multiple rows/columns.

The cell will span from *fromRow*, *fromCol* to *toRow*, *toCol*. Alignment is specified by *alignment*, which is a bitwise OR of Qt::AlignmentFlags values. The default alignment is 0, which means that the widget fills the entire cell.

void QGridLayout::addMultiCellLayout (QLayout * layout, int fromRow, int toRow, int fromCol, int toCol, int alignment = 0)

Adds the layout *layout* to the cell grid, spanning multiple rows/columns. The cell will span from *fromRow*, *fromCol* to *toRow*, *toCol*.

Alignment is specified by *alignment*, which is a bitwise OR of Qt::AlignmentFlags values. The default alignment is 0, which means that the widget fills the entire cell.

A non-zero alignment indicates that the layout should not grow to fill the available space but should be sized according to sizeHint().

void QGridLayout::addMultiCellWidget (QWidget * w, int fromRow, int toRow, int fromCol, int toCol, int alignment = 0)

Adds the widget *w* to the cell grid, spanning multiple rows/columns. The cell will span from *fromRow*, *fromCol* to *toRow*, *toCol*.

Alignment is specified by *alignment*, which is a bitwise OR of Qt::AlignmentFlags values. 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: cursor/cursor.cpp, layout/layout.cpp and progressbar/progressbar.cpp.

void QGridLayout::addRowSpacing (int row, int minsize)

Sets the minimum height of row *row* to *minsize* pixels.

void QGridLayout::addWidget (QWidget * w, int row, int col, int alignment = 0)

Adds the widget *w* to the cell grid at *row*, *col*. The top-left position is (0, 0) by default.

Alignment is specified by *alignment*, which is a bitwise OR of Qt::AlignmentFlags values. The default alignment is 0, which means that the widget fills the entire cell.

- You should not call this if you have enabled the auto-add facility of the layout.
- 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: addressbook/centralwidget.cpp, layout/layout.cpp, rot13/rot13.cpp, sql/overview/form1/main.cpp, sql/overview/form2/main.cpp, t14/gamebrd.cpp and t8/main.cpp.

QRect QGridLayout::cellGeometry (int row, int col) const

Returns the geometry of the cell with row *row* and column *col* in the grid. Returns an invalid rectangle if *row* or *col* is outside the grid.

Warning: in the current version of Qt this function does not return valid results until setGeometry() has been called, i.e. after the mainWidget() is visible.

int QGridLayout::colStretch (int col) const

Returns the stretch factor for column *col*.

See also setColStretch() [p. 83].

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 because QGridLayout expands automatically as new items are inserted.

QSizePolicy::ExpandData QGridLayout::expanding () const [virtual]

Returns the expansiveness of this layout.

Reimplemented from QLayout [p. 104].

bool QGridLayout::findWidget (QWidget * w, int * row, int * col) [protected]

Searches for widget *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.

Note: if a widget spans multiple rows/columns, the top-left cell is returned.

bool QGridLayout::hasHeightForWidth () const [virtual]

Returns TRUE if this layout's preferred height depends on its width; otherwise returns FALSE.

Reimplemented from QLayoutItem [p. 109].

int QGridLayout::heightForWidth (int w) const [virtual]

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

Reimplemented from QLayoutItem [p. 109].

void QGridLayout::invalidate () [virtual]

Resets cached information.

Reimplemented from QLayout [p. 104].

QSize QGridLayout::maximumSize () const [virtual]

Returns the maximum size needed by this grid.

Reimplemented from QLayout [p. 105].

QSize QGridLayout::minimumSize () const [virtual]

Returns the minimum size needed by this grid.

Reimplemented from QLayout [p. 105].

int QGridLayout::numCols () const

Returns the number of columns in this grid.

int QGridLayout::numRows () const

Returns the number of rows in this grid.

Corner QGridLayout::origin () const

Returns which of the four corners of the grid corresponds to (0, 0).

int QGridLayout::rowStretch (int row) const

Returns the stretch factor for row *row*.

See also setRowStretch() [p. 83].

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 a 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.

See also colStretch() [p. 81], addColSpacing() [p. 80] and setRowStretch() [p. 83].

Examples: layout/layout.cpp, t14/gamebrd.cpp and t8/main.cpp.

void QGridLayout::setGeometry (const QRect & r) [virtual]

Resizes managed widgets within the rectangle *r*.

Reimplemented from QLayout [p. 106].

void QGridLayout::setOrigin (Corner c)

Sets which of the four corners of the grid corresponds to (0, 0) to *c*.

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 a 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.

See also rowStretch() [p. 82], addRowSpacing() [p. 81] and setColStretch() [p. 83].

Example: addressbook/centralwidget.cpp.

QSize QGridLayout::sizeHint () const [virtual]

Returns the preferred size of this grid.

Reimplemented from QLayoutItem [p. 111].

QHBox Class Reference

The QHBox widget provides horizontal geometry management for its children.

`#include <qhbox.h>`

Inherits QFrame [Widgets with Qt].

Inherited by QVBox [Widgets with Qt].

Public Members

- **QHBox** (QWidget * parent = 0, const char * name = 0, WFlags f = 0)
- void **setSpacing** (int space)
- bool **setStretchFactor** (QWidget * w, int stretch)

Protected Members

- **QHBox** (bool horizontal, QWidget * parent = 0, const char * name = 0, WFlags f = 0)

Detailed Description

The QHBox widget provides horizontal geometry management for its children.

All the horizontal box's children will be placed alongside 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. Use setStretchFactor() if you want the widgets to be different sizes in proportion to one another.

See also QHBoxLayout [p. 86], QVBox [Widgets with Qt], QGrid [p. 75], Widget Appearance and Style, Layout Management and Organizers.

Member Function Documentation

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

Constructs an hbox widget with parent *parent* and name *name*. The parent, name and widget flags, *f*, are passed to the QFrame constructor.

QHBox::QHBox (bool horizontal, QWidget * parent = 0, const char * name = 0, WFlags f = 0) [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.

The *parent*, *name* and widget flags, *f*, are passed to the QFrame constructor.

void QHBox::setSpacing (int space)

Sets the spacing between the child widgets to *space*.

Examples: i18n/mywidget.cpp, listboxcombo/listboxcombo.cpp, network/ftpclient/ftpmainwindow.cpp, qdir/qdir.cpp, tabdialog/tabdialog.cpp, wizard/wizard.cpp and xform/xform.cpp.

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

Sets the stretch factor of widget *w* to *stretch*.

See also QBoxLayout::setStretchFactor() [p. 39].

QHBoxLayout Class Reference

The QHBoxLayout class lines up widgets horizontally.

`#include <qlayout.h>`

Inherits QBoxLayout [p. 34].

Public Members

- **QHBoxLayout** (QWidget * parent, int margin = 0, int spacing = -1, const char * name = 0)
- **QHBoxLayout** (QLayout * parentLayout, int spacing = -1, const char * name = 0)
- **QHBoxLayout** (int spacing = -1, const char * name = 0)
- **~QHBoxLayout** ()

Detailed Description

The QHBoxLayout class lines up widgets horizontally.

This class is used to construct horizontal box layout objects. See QBoxLayout for more details.

The simplest way to use this class is like this:

```
QBoxLayout * l = new QHBoxLayout( widget );
l->setAutoAdd( TRUE );
new QSomeWidget( widget );
new QSomeOtherWidget( widget );
new QAnotherWidget( widget );
```

or like this:

```
QBoxLayout * l = new QHBoxLayout( widget );
l->addWidget( existingChildOfWidget );
l->addWidget( anotherChildOfWidget );
```

See also QVBoxLayout [p. 174], QGridLayout [p. 77], the Layout overview [Programming with Qt], Widget Appearance and Style and Layout Management.

Member Function Documentation

QHBoxLayout::QHBoxLayout (QWidget * parent, int margin = 0, int spacing = -1, const char * name = 0)

Constructs a new top-level horizontal box with parent *parent* and name *name*.

The *margin* is the number of pixels between the edge of the widget and its managed children. The *spacing* is the default number of pixels between neighboring children. If *spacing* is -1 the value of *margin* is used for *spacing*.

QHBoxLayout::QHBoxLayout (QLayout * parentLayout, int spacing = -1, const char * name = 0)

Constructs a new horizontal box with the name *name* and adds it to *parentLayout*.

The *spacing* is the default number of pixels between neighboring children. If *spacing* is -1, this QHBoxLayout will inherit its parent's spacing().

QHBoxLayout::QHBoxLayout (int spacing = -1, const char * name = 0)

Constructs a new horizontal box with the name *name*. You must add it to another layout.

The *spacing* is the default number of pixels between neighboring children. If *spacing* is -1, this QHBoxLayout will inherit its parent's spacing().

QHBoxLayout::~QHBoxLayout ()

Destroys this box layout.

QHideEvent Class Reference

The QHideEvent class provides an event which is sent after a widget is hidden.

`#include <qevent.h>`

Inherits QEvent [p. 67].

Public Members

- **QHideEvent** ()

Detailed Description

The QHideEvent class provides an event which is 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.

If spontaneous() is TRUE the event originated outside the application, i.e. the user hid the window using 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() returns TRUE.

See also QShowEvent [p. 131] and Event Classes.

Member Function Documentation

QHideEvent::QHideEvent ()

Constructs a QHideEvent.

QIconDrag Class Reference

The QIconDrag class supports drag and drop operations within a QIconView.

This class is part of the **iconview** module.

`#include <qiconview.h>`

Inherits QDragObject [p. 59].

Public Members

- virtual **~QIconDrag** ()
- void **append** (const QIconDragItem & i, const QRect & pr, const QRect & tr)
- virtual QByteArray **encodedData** (const char * mime) const

Static Public Members

- bool **canDecode** (QMimeSource * e)

Detailed Description

The QIconDrag class supports drag and drop operations within a QIconView.

A QIconDrag object is used to maintain information about the positions of dragged items and data associated with the dragged items. QIconViews are able to use this information to paint the dragged items in the correct positions. Internally QIconDrag stores the data associated with drag items in QIconDragItem objects.

If you want to use the extended drag-and-drop functionality of QIconView create a QIconDrag object in a reimplementation of QIconView::dragObject(). Then create a QIconDragItem for each item which should be dragged, set the data it represents with QIconDragItem::setData(), and add each QIconDragItem to the drag object using append().

The data in QIconDragItems is stored in a QByteArray and is mime-typed (see QMimeSource and the Drag and Drop overview). If you want to use your own mime-types derive a class from QIconDrag and reimplement format(), encodedData() and canDecode().

The fileiconview example program demonstrates the use of the QIconDrag class including subclassing and reimplementing dragObject(), format(), encodedData() and canDecode(). See the files `qt/examples/fileiconview/qfileiconview.h` and `qt/examples/fileiconview/qfileiconview.cpp`.

See also QMimeSource::format() [Input/Output and Networking with Qt] and Drag And Drop Classes.

Member Function Documentation

QIconDrag::~QIconDrag () [virtual]

Destructor.

void QIconDrag::append (const QIconDragItem & i, const QRect & pr, const QRect & tr)

Append the QIconDragItem, *i*, to the QIconDrag object's list of items. You must also supply the geometry of the pixmap, *pr*, and of the textual caption, *tr*.

See also QIconDragItem [p. 91].

Example: fileiconview/qfileiconview.cpp.

bool QIconDrag::canDecode (QMimeSource * e) [static]

Returns TRUE if *e* can be decoded by the QIconDrag, otherwise return FALSE.

Example: fileiconview/qfileiconview.cpp.

QByteArray QIconDrag::encodedData (const char * mime) const [virtual]

Returns the encoded data of the drag object if *mime* is application/x-qiconlist.

Example: fileiconview/qfileiconview.cpp.

Reimplemented from QMimeSource [Input/Output and Networking with Qt].

QIconDragItem Class Reference

The QIconDragItem class encapsulates a drag item.

This class is part of the **iconview** module.

`#include <qiconview.h>`

Public Members

- **QIconDragItem** ()
- virtual **~QIconDragItem** ()
- virtual QByteArray **data** () const
- virtual void **setData** (const QByteArray & d)

Detailed Description

The QIconDragItem class encapsulates a drag item.

The QIconDrag class uses a list of QIconDragItems to support drag and drop operations.

In practice a QIconDragItem object (or an object of a class derived from QIconDragItem) is created for each icon view item which is dragged. Each of these QIconDragItems is stored in a QIconDrag object.

See QIconView::dragObject() for more information.

See the fileiconview/qfileiconview.cpp and iconview/simple_dd/main.cpp examples.

See also Drag And Drop Classes.

Member Function Documentation

QIconDragItem::QIconDragItem ()

Constructs a QIconDragItem with no data.

QIconDragItem::~QIconDragItem () [virtual]

Destructor.

QByteArray QIconDragItem::data () const [virtual]

Returns the data contained in the QIconDragItem.

void QIconDragItem::setData (const QByteArray & d) [virtual]

Sets the data for the QIconDragItem to the data stored in the QByteArray *d*.

Example: fileiconview/qfileiconview.cpp.

QImageDrag Class Reference

The QImageDrag class provides a drag and drop object for transferring images.

`#include <qdragobject.h>`

Inherits QDragObject [p. 59].

Public Members

- **QImageDrag** (QImage image, QWidget * dragSource = 0, const char * name = 0)
- **QImageDrag** (QWidget * dragSource = 0, const char * name = 0)
- **~QImageDrag** ()
- virtual void **setImage** (QImage image)

Static Public Members

- bool **canDecode** (const QMimeSource * e)
- bool **decode** (const QMimeSource * e, QImage & img)
- bool **decode** (const QMimeSource * e, QPixmap & pm)

Detailed Description

The QImageDrag class 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 more information about drag and drop, see the QDragObject class and the drag and drop documentation.

See also Drag And Drop Classes.

Member Function Documentation

QImageDrag::QImageDrag (QImage image, QWidget * dragSource = 0, const char * name = 0)

Constructs 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)

Constructs 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 up all allocated resources.

bool QImageDrag::canDecode (const QMimeSource * e) [static]

Returns TRUE if the information in mime source *e* can be decoded into an image.

See also decode() [p. 94].

Example: desktop/desktop.cpp.

bool QImageDrag::decode (const QMimeSource * e, QImage & img) [static]

Attempts to decode the dropped information in mime source *e* into *img*. Returns TRUE if successful; otherwise returns FALSE.

See also canDecode() [p. 94].

Example: desktop/desktop.cpp.

bool QImageDrag::decode (const QMimeSource * e, QPixmap & pm) [static]

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

Attempts to decode the dropped information in mime source *e* into pixmap *pm*. Returns TRUE if successful; otherwise returns FALSE.

This is a convenience function that converts to *pm* via a QImage.

See also canDecode() [p. 94].

void QImageDrag::setImage (QImage image) [virtual]

Sets the image to be dragged to *image*. You will need to call this if you did not pass the image during construction.

QKeyEvent Class Reference

The QKeyEvent class contains describes a key event.

`#include <qevent.h>`

Inherits QEvent [p. 67].

Public Members

- **QKeyEvent** (Type type, int key, int ascii, int state, const QString & text = QString::null, bool autorep = FALSE, ushort count = 1)
- int **key** () const
- int **ascii** () const
- ButtonState **state** () const
- ButtonState **stateAfter** () const
- bool **isAccepted** () const
- QString **text** () const
- bool **isAutoRepeat** () const
- int **count** () const
- void **accept** ()
- void **ignore** ()

Detailed Description

The QKeyEvent class contains describes 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 that indicates whether the receiver wants the key event. You should call QKeyEvent::ignore() if the key press or release event is not handled by your widget. A key event is propagated up the parent widget chain until a widget accepts it with QKeyEvent::accept() or an event filter consumes it.

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 [p. 71], QWidget::grabKeyboard() [Widgets with Qt] and Event Classes.

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). *ascii* is the ASCII code of the key that was pressed or released. *state* holds the keyboard modifiers. *text* is the Unicode text that the key generated. If *autorep* is TRUE, 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() [p. 96].

int QKeyEvent::ascii () const

Returns the ASCII code of the key that was pressed or released. We recommend using text() instead.

See also text() [p. 97].

Example: picture/picture.cpp.

int QKeyEvent::count () const

Returns the number of single keys for this event. If text() is not empty, this is simply the length of the string.

However, Qt also compresses invisible keycodes such as BackSpace. For those, count() returns the number of key presses/repeats this event represents.

See also QWidget::setKeyCompression() [Widgets with Qt].

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() [p. 96].

bool QKeyEvent::isAccepted () const

Returns TRUE if the receiver of the event wants to keep the key; otherwise returns FALSE

bool QKeyEvent::isAutoRepeat () const

Returns TRUE if this event comes from an auto-repeating key and FALSE if it comes from an initial key press.

Note that if the event is a multiple-key compressed event that is partly due to auto-repeat, this function could return either TRUE or FALSE indeterminately.

int QKeyEvent::key () const

Returns the code of the key that was pressed or released.

See Qt::Key for the list of 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).

Example: fileiconview/qfileiconview.cpp.

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() [p. 97].

Example: fileiconview/qfileiconview.cpp.

ButtonState QKeyEvent::stateAfter () const

Returns the keyboard modifier flags that existed immediately after the event occurred.

Warning: This function cannot be trusted.

See also state() [p. 97].

QString QKeyEvent::text () const

Returns the Unicode text that this key generated.

See also QWidget::setKeyCompression() [Widgets with Qt].

QKeySequence Class Reference

The QKeySequence class encapsulates a key sequence as used by accelerators.

`#include <qkeysequence.h>`

Inherits Qt [Additional Functionality with Qt].

Public Members

- **QKeySequence** ()
- **QKeySequence** (const QString & key)
- **QKeySequence** (int key)
- **operator QString** () const
- **operator int** () const
- **QKeySequence** (const QKeySequence & keysequence)
- QKeySequence & **operator=** (const QKeySequence & keysequence)
- **~QKeySequence** ()
- bool **operator==** (const QKeySequence & keysequence) const
- bool **operator!=** (const QKeySequence & keysequence) const

Related Functions

- QDataStream & **operator<<** (QDataStream & s, const QKeySequence & keysequence)
- QDataStream & **operator>>** (QDataStream & s, QKeySequence & keysequence)

Detailed Description

The QKeySequence class encapsulates a key sequence as used by accelerators.

A key sequence consists of a keyboard code, optionally combined with modifiers, e.g. SHIFT, CTRL, ALT or UNICODE_ACCEL. For example, `CTRL + Key_P` might be a sequence used as a shortcut for printing a document. The key codes are listed in qnamespace.h. As an alternative, use UNICODE_ACCEL with the unicode code point of the character. For example, `UNICODE_ACCEL + 'A'` gives the same key sequence as Key_A.

Key sequences can be constructed either from an integer key code, or from a human readable translatable string. A key sequence can be cast to a QString to obtain a human readable translated version of the sequence. Translations are done in the "QAccel" scope.

See also QAccel [p. 13] and Miscellaneous Classes.

Member Function Documentation

QKeySequence::QKeySequence ()

Constructs an empty key sequence.

QKeySequence::QKeySequence (const QString & key)

Creates a key sequence from the string *key*. For example "Ctrl+O" gives CTRL+UNICODE_ACCEL+'O'. The strings "Ctrl", "Shift" and "Alt" are recognized, as well as their translated equivalents in the "QAccel" scope (using QObject::tr()).

This contructor is typically used with tr(), so that accelerator keys can be replaced in translations:

```
QPopupMenu *file = new QPopupMenu( this );
file->insertItem( tr("&Open..."), this, SLOT(open()),
                  QKeySequence( tr("Ctrl+O", "File|Open") ) );
```

Note the `"File|Open"` translator comment. It is by no means necessary, but it provides some context for the human translator.

QKeySequence::QKeySequence (int key)

Constructs a key sequence from the keycode *key*.

The key codes are listed in qnamespace.h and can be combined with modifiers, e.g. with SHIFT, CTRL, ALT or UNICODE_ACCEL.

QKeySequence::QKeySequence (const QKeySequence & keysequence)

Copy constructor. Makes a copy of *keysequence*.

QKeySequence::~QKeySequence ()

Destroys the key sequence.

QKeySequence::operator QString () const

Creates an accelerator string for the key sequence. For instance CTRL+Key_O gives "Ctrl+O". The strings, "Ctrl", "Shift", etc. are translated (using QObject::tr()) in the "QAccel" scope.

QKeySequence::operator int () const

For backward compatibility: returns the keycode as integer.

If QKeySequence ever supports more than one keycode, this function will return the first one.

bool QKeySequence::operator!= (const QKeySequence & keysequence) const

Returns TRUE if *keysequence* is not equal to this key sequence; otherwise returns FALSE.

QKeySequence & QKeySequence::operator= (const QKeySequence & keysequence)

Assignment operator. Assigns *keysequence* to this object.

bool QKeySequence::operator== (const QKeySequence & keysequence) const

Returns TRUE if *keysequence* is equal to this key sequence; otherwise returns FALSE.

Related Functions

QDataStream & operator<< (QDataStream & s, const QKeySequence & keysequence)

Writes the key sequence *keysequence* to the stream *s*.

Format of the QDataStream operators

QDataStream & operator>> (QDataStream & s, QKeySequence & keysequence)

Reads a key sequence from the stream *s* into the key sequence *keysequence*.

QLayout Class Reference

The QLayout class is the base class of geometry managers.

`#include <qlayout.h>`

Inherits QObject [Additional Functionality with Qt] and QLayoutItem [p. 108].

Inherited by QGridLayout [p. 77] and QBoxLayout [p. 34].

Public Members

- **QLayout** (QWidget * parent, int margin = 0, int spacing = -1, const char * name = 0)
- **QLayout** (QLayout * parentLayout, int spacing = -1, const char * name = 0)
- **QLayout** (int spacing = -1, const char * name = 0)
- int **margin** () const
- int **spacing** () const
- virtual void **setMargin** (int)
- virtual void **setSpacing** (int)
- enum **ResizeMode** { FreeResize, Minimum, Fixed }
- void **setResizeMode** (ResizeMode)
- ResizeMode **resizeMode** () const
- virtual void **setMenuBar** (QMenuBar * w)
- QMenuBar * **menuBar** () const
- QWidget * **mainWidget** ()
- bool **isTopLevel** () const
- virtual void **setAutoAdd** (bool b)
- bool **autoAdd** () const
- virtual void **invalidate** ()
- bool **activate** ()
- void **add** (QWidget * w)
- virtual void **addItem** (QLayoutItem * item)
- virtual QSizePolicy::ExpandData **expanding** () const
- virtual QSize **minimumSize** () const
- virtual QSize **maximumSize** () const
- virtual void **setGeometry** (const QRect & r)
- virtual QLayoutIterator **iterator** ()
- virtual bool **isEmpty** () const
- bool **supportsMargin** () const
- void **setEnabled** (bool enable)
- bool **isEnabled** () const

Properties

- int **margin** — the width of the outside border of the layout
- ResizeMode **resizeMode** — the resize mode of the layout
- int **spacing** — the spacing between widgets inside the layout

Protected Members

- void **addChildLayout** (QLayout * l)
- void **deleteAllItems** ()
- void **setSupportsMargin** (bool b)
- QRect **alignmentRect** (const QRect & r) const

Detailed Description

The QLayout class is the base class of geometry managers.

This is an abstract base class inherited by the concrete classes, QBoxLayout and QGridLayout.

For users of QLayout subclasses or of QMainWindow there is seldom any need to use the basic functions provided by QLayout, such as resizeMode or setMenuBar(). See the layout overview page for more information.

To make your own layout manager, subclass QGLayoutIterator and implement the functions addItem(), sizeHint(), setGeometry(), and iterator(). You should also implement minimumSize() to ensure your layout isn't resized to zero size if there is too little space. To support children whose height depend on their widths, implement hasHeightForWidth() and heightForWidth(). See the custom layout page [p. 6] for an in-depth description.

Geometry management stops when the layout manager is deleted.

See also Widget Appearance and Style and Layout Management.

Member Type Documentation

QLayout::ResizeMode

The possible values are:

- `QLayout::Fixed` - The main widget's size is set to sizeHint(); it cannot be resized at all.
- `QLayout::Minimum` - The main widget's minimum size is set to minimumSize(); it cannot be smaller.
- `QLayout::FreeResize` - The widget is not constrained.

Member Function Documentation

QLayout::QLayout (QWidget * parent, int margin = 0, int spacing = -1, const char * name = 0)

Constructs a new top-level QLayout with main widget *parent*, and name *name*. *parent* may not be 0.

The *margin* is the number of pixels between the edge of the widget and the managed children. The *spacing* sets the value of spacing(), which gives the spacing between the managed widgets. If *spacing* is -1 (the default), spacing is set to the value of *margin*.

There can be only one top-level layout for a widget. It is returned by QWidget::layout()

QLayout::QLayout (QLayout * parentLayout, int spacing = -1, const char * name = 0)

Constructs a new child QLayout called *name*, and places it inside *parentLayout* by using the default placement defined by addItem().

If *spacing* is -1, this QLayout inherits *parentLayout*'s spacing(), otherwise the value of *spacing* is used.

QLayout::QLayout (int spacing = -1, const char * name = 0)

Constructs a new child QLayout called *name*. If *spacing* is -1, this QLayout inherits its parent's spacing(); otherwise the value of *spacing* is used.

This layout has to be inserted into another layout before geometry management will work.

bool QLayout::activate ()

Redoes the layout for mainWidget(). You should generally not need to call this because it is automatically called at the most appropriate times.

However, if you set up a QLayout for a visible widget without resizing that widget, you will need to call this function in order to lay it out.

See also QWidget::updateGeometry() [Widgets with Qt].

Examples: layout/layout.cpp, popup/popup.cpp, scrollview/scrollview.cpp and sql/overview/form1/main.cpp.

void QLayout::add (QWidget * w)

Adds widget *w* to this layout in a manner specific to the layout. This function uses addItem.

Examples: customlayout/border.cpp and customlayout/main.cpp.

void QLayout::addChildLayout (QLayout * l) [protected]

This function is called from addLayout() functions in subclasses to add layout *l* as a sublayout.

void QLayout::addItem (QLayoutItem * item) [virtual]

Implemented in subclasses to add an *item*. How it is added is specific to each subclass.

The ownership of *item* is transferred to the layout, and it's the layout's responsibility to delete it.

Examples: customlayout/border.cpp, customlayout/card.cpp and customlayout/flow.cpp.

Reimplemented in QGridLayout and QBoxLayout.

QRect QLayout::alignmentRect (const QRect & r) const [protected]

Returns the rectangle that should be covered when the geometry of this layout is set to *r*, provided that this layout supports setAlignment().

The result is derived from sizeHint() and expanding(). It is never larger than *r*.

bool QLayout::autoAdd () const

Returns TRUE if this layout automatically grabs all new mainWidget()'s new children and adds them as defined by addItem(); otherwise returns FALSE. This has effect only for top-level layouts, i.e. layouts that are direct children of their mainWidget().

autoAdd() is disabled by default.

See also setAutoAdd() [p. 106].

void QLayout::deleteAllItems () [protected]

Removes and deletes all items in this layout.

QSizePolicy::ExpandData QLayout::expanding () const [virtual]

Returns whether this layout can make use of more space than sizeHint(). A value of Vertical or Horizontal means that it wants to grow in only one dimension, whereas BothDirections means that it wants to grow in both dimensions.

The default implementation returns BothDirections.

Examples: customlayout/border.cpp and customlayout/flow.cpp.

Reimplemented from QLayoutItem [p. 109].

Reimplemented in QGridLayout and QBoxLayout.

void QLayout::invalidate () [virtual]

Invalidates cached information. Reimplementations must call this.

Reimplemented from QLayoutItem [p. 110].

Reimplemented in QGridLayout and QBoxLayout.

bool QLayout::isEmpty () const [virtual]

Returns TRUE if this layout is empty. The default implementation returns FALSE.

Reimplemented from QLayoutItem [p. 110].

bool QLayout::isEnabled () const

Returns TRUE if the layout is enabled; otherwise returns FALSE.

See also setEnabled() [p. 106].

bool QLayout::isTopLevel () const

Returns TRUE if this layout is a top-level layout, i.e., not a child of another layout; otherwise returns FALSE.

QLayoutIterator QLayout::iterator () [virtual]

Implemented in subclasses to return an iterator that iterates over the children of this layout.

A typical implementation will be:

```
QLayoutIterator MyLayout::iterator()
{
    QGLayoutIterator *i = new MyLayoutIterator( internal_data );
    return QLayoutIterator( i );
}
```

where MyLayoutIterator is a subclass of QGLayoutIterator.

Examples: customlayout/border.cpp, customlayout/card.cpp and customlayout/flow.cpp.

Reimplemented from QLayoutItem [p. 110].

QWidget * QLayout::mainWidget ()

Returns the main widget (parent widget) of this layout, or 0 if this layout is a sub-layout that is not yet inserted.

int QLayout::margin () const

Returns the width of the outside border of the layout. See the "margin" [p. 107] property for details.

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 [p. 110].

Reimplemented in QGridLayout and QBoxLayout.

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.

Examples: customlayout/border.cpp, customlayout/card.cpp and customlayout/flow.cpp.

Reimplemented from QLayoutItem [p. 110].

Reimplemented in QGridLayout and QBoxLayout.

ResizeMode QLayout::resizeMode () const

Returns the resize mode of the layout. See the "resizeMode" [p. 107] property for details.

void QLayout::setAutoAdd (bool b) [virtual]

If *b* is TRUE auto-add is enabled; otherwise auto-add is disabled.

See also autoAdd() [p. 104].

Example: i18n/main.cpp.

void QLayout::setEnabled (bool enable)

Enables this layout if *enable* is TRUE, otherwise disables it.

An enabled layout adjusts dynamically to changes; a disabled layout acts as if it did not exist.

By default all layouts are enabled.

See also isEnabled() [p. 104].

void QLayout::setGeometry (const QRect & r) [virtual]

This function is reimplemented in subclasses to perform layout.

The default implementation maintains the geometry() information given by rect *r*. Reimplementors must call this function.

Examples: customlayout/border.cpp, customlayout/card.cpp and customlayout/flow.cpp.

Reimplemented from QLayoutItem [p. 111].

Reimplemented in QGridLayout and QBoxLayout.

void QLayout::setMargin (int) [virtual]

Sets the width of the outside border of the layout. See the "margin" [p. 107] property for details.

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 management: never do addWidget() on a QMenuBar.

Examples: layout/layout.cpp and scrollview/scrollview.cpp.

void QLayout::setResizeMode (ResizeMode)

Sets the resize mode of the layout. See the "resizeMode" [p. 107] property for details.

void QLayout::setSpacing (int) [virtual]

Sets the spacing between widgets inside the layout. See the "spacing" [p. 107] property for details.

void QLayout::setSupportsMargin (bool b) [protected]

Sets the value returned by supportsMargin(). If *b* is TRUE, margin() handling is implemented by the subclass. If *b* is FALSE (the default), QLayout will add margin() around top-level layouts.

If *b* is TRUE, margin handling needs to be implemented in setGeometry(), maximumSize(), minimumSize(), sizeHint() and heightForWidth().

See also supportsMargin() [p. 107].

int QLayout::spacing () const

Returns the spacing between widgets inside the layout. See the "spacing" [p. 107] property for details.

bool QLayout::supportsMargin () const

Returns TRUE if this layout supports QLayout::margin on non-top-level layouts; otherwise returns FALSE.

See also margin [p. 107].

Property Documentation

int margin

This property holds the width of the outside border of the layout.

For some layout classes this property has an effect only on top-level layouts; QBoxLayout and QGridLayout support margins for child layouts. The default value is 0.

See also spacing [p. 107].

Set this property's value with setMargin() and get this property's value with margin().

ResizeMode resizeMode

This property holds the resize mode of the layout.

The default mode is Minimum for top-level widgets and FreeResize for all others.

See also QLayout::ResizeMode [p. 102].

Set this property's value with setResizeMode() and get this property's value with resizeMode().

int spacing

This property holds the spacing between widgets inside the layout.

The default value is -1, which signifies that the layout's spacing should not override the widget's spacing.

See also margin [p. 107].

Set this property's value with setSpacing() and get this property's value with spacing().

QLayoutItem Class Reference

The QLayoutItem class provides an abstract item that a QLayout manipulates.

`#include <qlayout.h>`

Inherited by QLayout [p. 101], QSpacerItem [Widgets with Qt] and QWidgetItem [Widgets with Qt].

Public Members

- **QLayoutItem** (int alignment = 0)
- virtual **~QLayoutItem** ()
- virtual QSize **sizeHint** () const
- virtual QSize **minimumSize** () const
- virtual QSize **maximumSize** () const
- virtual QSizePolicy::ExpandData **expanding** () const
- virtual void **setGeometry** (const QRect & r)
- virtual QRect **geometry** () const
- virtual bool **isEmpty** () const
- virtual bool **hasHeightForWidth** () const
- virtual int **heightForWidth** (int w) const
- virtual void **invalidate** ()
- virtual QWidget * **widget** ()
- virtual QLayoutIterator **iterator** ()
- virtual QLayout * **layout** ()
- virtual QSpacerItem * **spacerItem** ()
- int **alignment** () const
- virtual void **setAlignment** (int a)

Detailed Description

The QLayoutItem class provides an abstract item that a QLayout manipulates.

This is used by custom layouts.

See also QLayout [p. 101], Widget Appearance and Style and Layout Management.

Member Function Documentation

QLayoutItem::QLayoutItem (int alignment = 0)

Constructs a layout item with an *alignment* that is a bitwise OR of the Qt::AlignmentFlags. Not all subclasses support alignment.

QLayoutItem::~QLayoutItem () [virtual]

Destroys 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 QLayout, QSpacerItem and QWidgetItem.

QRect QLayoutItem::geometry () const [virtual]

Returns the rectangle covered by this layout item.

Example: customlayout/border.cpp.

bool QLayoutItem::hasHeightForWidth () const [virtual]

Returns TRUE if this layout's preferred height depends on its width; otherwise returns FALSE. The default implementation returns FALSE.

Reimplement this function in layout managers that support height for width.

See also heightForWidth() [p. 109] and QWidget::heightForWidth() [Widgets with Qt].

Examples: customlayout/border.cpp and customlayout/flow.cpp.

Reimplemented in QGridLayout and QBoxLayout.

int QLayoutItem::heightForWidth (int w) 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"
            MyLayout *that = (MyLayout*)this;
            int h = calculateHeightForWidth( w );
            that->cached_hfw = h;
            return h;
        }
        return cached_hfw;
    }
```

Caching is strongly recommended; without it layout will take exponential time.

See also hasHeightForWidth() [p. 109].

Example: customlayout/flow.cpp.

Reimplemented in QGridLayout and QBoxLayout.

void QLayoutItem::invalidate () [virtual]

Invalidates any cached information in this layout item.

Reimplemented in QLayout.

bool QLayoutItem::isEmpty () const [virtual]

Implemented in subclasses to return whether this item is empty, i.e. whether it contains any widgets.

Reimplemented in QLayout, QSpacerItem and QWidgetItem.

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 QLayout.

QLayout * QLayoutItem::layout () [virtual]

If this item is a QLayout, return it as a QLayout; otherwise return 0. This function provides type-safe casting.

QSize QLayoutItem::maximumSize () const [virtual]

Implemented in subclasses to return the maximum size of this item.

Reimplemented in QLayout, QSpacerItem and QWidgetItem.

QSize QLayoutItem::minimumSize () const [virtual]

Implemented in subclasses to return the minimum size of this item.

Examples: customlayout/border.cpp, customlayout/card.cpp and customlayout/flow.cpp.

Reimplemented in QLayout, QSpacerItem and QWidgetItem.

void QLayoutItem::setAlignment (int a) [virtual]

Sets the alignment of this item to *a*, which is a bitwise OR of the Qt::AlignmentFlags. Not all subclasses support alignment.

void QLayoutItem::setGeometry (const QRect & r) [virtual]

Implemented in subclasses to set this item's geometry to *r*.

Examples: customlayout/border.cpp, customlayout/card.cpp and customlayout/flow.cpp.

Reimplemented in QLayout, QSpacerItem and QWidgetItem.

QSize QLayoutItem::sizeHint () const [virtual]

Implemented in subclasses to return the preferred size of this item.

Examples: customlayout/border.cpp, customlayout/card.cpp and customlayout/flow.cpp.

Reimplemented in QSpacerItem, QWidgetItem, QGridLayout and QBoxLayout.

QSpacerItem * QLayoutItem::spacerItem () [virtual]

If this item is a QSpacerItem, return it as a QSpacerItem; otherwise return 0. This function provides type-safe casting.

QWidget * QLayoutItem::widget () [virtual]

If this item is a QWidgetItem, the managed widget is returned. The default implementation returns 0.

Reimplemented in QWidgetItem.

QLayoutIterator Class Reference

The QLayoutIterator class provides iterators over QLayoutItem.

`#include <qlayout.h>`

Public Members

- **QLayoutIterator** (QGLayoutIterator * gi)
- **QLayoutIterator** (const QLayoutIterator & i)
- **~QLayoutIterator** ()
- QLayoutIterator & **operator=** (const QLayoutIterator & i)
- QLayoutItem * **operator++** ()
- QLayoutItem * **current** ()
- QLayoutItem * **takeCurrent** ()
- void **deleteCurrent** ()

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 possible 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()) != 0 ) {
        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. QLayoutIterator itself is not designed to be subclassed.

See also Widget Appearance and Style and Layout Management.

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*, i.e. 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.

QMotifPlusStyle Class Reference

The QMotifPlusStyle class provides a more sophisticated Motif-ish look and feel.

`#include <qmotifplusstyle.h>`

Inherits QMotifStyle [p. 116].

Public Members

- **QMotifPlusStyle** (bool hoveringHighlight = FALSE)

Detailed Description

The QMotifPlusStyle class provides a more sophisticated Motif-ish look and feel.

This class implements a Motif-ish look and feel with more sophisticated bevelling as used by the GIMP Toolkit (GTK+) for Unix/X11.

See also Widget Appearance and Style.

Member Function Documentation

QMotifPlusStyle::QMotifPlusStyle (bool hoveringHighlight = FALSE)

Constructs a QMotifPlusStyle

If *hoveringHighlight* is FALSE (the default), then the style will not highlight push buttons, checkboxes, radiobuttons, comboboxes, scrollbars or sliders.

QMotifStyle Class Reference

The QMotifStyle class provides Motif look and feel.

`#include <qmotifstyle.h>`

Inherits QCommonStyle [p. 48].

Inherited by QCDEStyle [p. 41], QMotifPlusStyle [p. 115] and QSGIStyle [p. 130].

Public Members

- **QMotifStyle** (bool useHighlightCols = FALSE)
- void **setUseHighlightColors** (bool arg)
- bool **useHighlightColors** () const

Detailed Description

The QMotifStyle class provides Motif look and feel.

This class implements the Motif look and feel. It closely resembles the original Motif look as defined by the Open Group, with the addition of some minor improvements. The Motif style is Qt's default GUI style on UNIX platforms.

See also Widget Appearance and Style.

Member Function Documentation

QMotifStyle::QMotifStyle (bool useHighlightCols = FALSE)

Constructs a QMotifStyle.

If *useHighlightCols* is FALSE (the default), 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::setUseHighlightColors (bool arg)

If *arg* is FALSE, 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 an application palette is set via QApplication::setPalette(). The current color palette of the application remains unchanged.

See also QStyle::polish() [p. 149].

bool QMotifStyle::useHighlightColors () const

Returns TRUE if the style treats the highlight colors of the palette in a Motif-like manner, which is a simple inversion between the base and the text color; otherwise returns FALSE. The default is FALSE.

QMouseEvent Class Reference

The QMouseEvent class contains parameters that describe a mouse event.

`#include <qevent.h>`

Inherits QEvent [p. 67].

Public Members

- **QMouseEvent** (Type type, const QPoint & pos, int button, int state)
- **QMouseEvent** (Type type, const QPoint & pos, const QPoint & globalPos, int button, int state)
- const QPoint & **pos** () const
- const QPoint & **globalPos** () const
- int **x** () const
- int **y** () const
- int **globalX** () const
- int **globalY** () const
- ButtonState **button** () const
- ButtonState **state** () const
- ButtonState **stateAfter** () const
- bool **isAccepted** () const
- void **accept** ()
- void **ignore** ()

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 occur only when a mouse button is pressed down, unless mouse tracking has been enabled with QWidget::setMouseTracking().

Qt automatically grabs the mouse when a mouse button is pressed inside a widget; the widget will continue to receive mouse events until the last mouse button is released.

A mouse event contains a special accept flag that indicates whether the receiver wants the event. You should call QMouseEvent::ignore() if the mouse event is not handled by your widget. A mouse event is propagated up the parent widget chain until a widget accepts it with QMousEvent::accept() or an event filter consumes it.

The functions pos(), x() and y() give the cursor position relative to the widget that receives the mouse event. If you move the widget as a result of the mouse event, use the global position returned by globalPos() to avoid a shaking motion.

The QWidget::setEnable() function can be used to enable or disable mouse and keyboard events for a widget.

The event handlers QWidget::mousePressEvent(), QWidget::mouseReleaseEvent(), QWidget::mouseDoubleClickEvent() and QWidget::mouseMoveEvent() receive mouse events.

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

Member Function Documentation

QMouseEvent::QMouseEvent (Type type, const QPoint & pos, int button, int state)

Constructs a mouse event object.

The *type* parameter must be one of QEvent::MouseButtonPress, QEvent::MouseButtonRelease, QEvent::MouseButtonDblClick or QEvent::MouseMove.

The *pos* parameter specifies the position relative to the receiving widget. *button* specifies the ButtonState of the button that caused the event, which should be 0 if *type* is MouseMove. *state* is the ButtonState at the time of the event.

The globalPos() is initialized to QCursor::pos(), which may not be appropriate. Use the other constructor to specify the global position explicitly.

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.

The *pos* parameter specifies the position relative to the receiving widget. *globalPos* is the position in absolute coordinates. *button* specifies the ButtonState of the button that caused the event, which should be 0 if *type* is MouseMove. *state* is the ButtonState at the time of the event.

void QMouseEvent::accept ()

Sets the accept flag of the mouse event object.

Setting the accept parameter indicates that the receiver of the event wants the mouse event. Unwanted mouse events are sent to the parent widget.

The accept flag is set by default.

See also ignore() [p. 120].

Example: dirview/dirview.cpp.

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 for mouse move events.

See also state() [p. 121].

Examples: dclock/dclock.cpp, life/life.cpp and t14/cannon.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 events, globalPos() may differ a lot from the current pointer position QCursor::pos(), and from QWidget::mapToGlobal(pos()).

See also globalX() [p. 120] and globalY() [p. 120].

Example: aclock/aclock.cpp.

int QMouseEvent::globalX () const

Returns the global X position of the mouse pointer at the time of the event.

See also globalY() [p. 120] and globalPos() [p. 120].

int QMouseEvent::globalY () const

Returns the global Y position of the mouse pointer at the time of the event.

See also globalX() [p. 120] and globalPos() [p. 120].

void QMouseEvent::ignore ()

Clears the accept flag parameter of the mouse event object.

Clearing the accept parameter indicates that the event receiver does not want the mouse event. Unwanted mouse events are sent to the parent widget.

The accept flag is set by default.

See also accept() [p. 119].

bool QMouseEvent::isAccepted () const

Returns TRUE if the receiver of the event wants to keep the key; otherwise returns FALSE.

const QPoint & QMouseEvent::pos () const

Returns the position of the mouse pointer relative to the widget that received the event.

If you move the widget as a result of the mouse event, use the global position returned by globalPos() to avoid a shaking motion.

See also x() [p. 121], y() [p. 121] and globalPos() [p. 120].

Examples: drawlines/connect.cpp, life/life.cpp, popup/popup.cpp, qmag/qmag.cpp, scribble/scribble.cpp, t14/cannon.cpp and tooltip/tooltip.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 pressed 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, whereas for QEvent::MouseButtonRelease it will.

This value is mainly interesting for QEvent::MouseMove; for the other cases, button() is more useful.

The returned value is LeftButton, RightButton, MidButton, ShiftButton, ControlButton and AltButton OR'ed together.

See also button() [p. 119] and stateAfter() [p. 121].

Examples: popup/popup.cpp and showimg/showimg.cpp.

ButtonState QMouseEvent::stateAfter () const

Returns the state of buttons after the event.

See also state() [p. 121].

int QMouseEvent::x () const

Returns the X position of the mouse pointer, relative to the widget that received the event.

See also y() [p. 121] and pos() [p. 120].

Example: showimg/showimg.cpp.

int QMouseEvent::y () const

Returns the Y position of the mouse pointer, relative to the widget that received the event.

See also x() [p. 121] and pos() [p. 120].

Example: showimg/showimg.cpp.

QMoveEvent Class Reference

The QMoveEvent class contains event parameters for move events.

`#include <qevent.h>`

Inherits QEvent [p. 67].

Public Members

- **QMoveEvent** (const QPoint & pos, const QPoint & oldPos)
- const QPoint & **pos** () const
- const QPoint & **oldPos** () const

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::pos [Widgets with Qt], QWidget::geometry [Widgets with Qt] and Event Classes.

Member Function Documentation

QMoveEvent::QMoveEvent (const QPoint & pos, const QPoint & oldPos)

Constructs a move event with the new and old widget positions, *pos* and *oldPos* respectively.

const QPoint & QMoveEvent::oldPos () const

Returns the old position of the widget.

const QPoint & QMoveEvent::pos () const

Returns the new position of the widget, excluding window frame for top level widgets.

QObjectCleanupHandler Class Reference

The QObjectCleanupHandler class watches the lifetime of multiple QObjects.

`#include <qobjectcleanuphandler.h>`

Inherits QObject [Additional Functionality with Qt].

Public Members

- **QObjectCleanupHandler** ()
- **~QObjectCleanupHandler** ()
- QObject * **add** (QObject * object)
- void **remove** (QObject * object)
- bool **isEmpty** () const
- void **clear** ()

Detailed Description

The QObjectCleanupHandler class watches the lifetime of multiple QObjects.

A QObjectCleanupHandler is useful whenever you need to know when a number of QObjects that are owned by someone else has been deleted. This is e.g. important when referencing memory in an application that has been allocated in a shared library.

Example:

```
class FactoryComponent : public FactoryInterface, public QLibraryInterface
{
public:
   ...

   QObject *createObject();

   bool init();
   void cleanup();
   bool canUnload() const;

private:
   QObjectCleanupHandler objects;
};

// allocate a new object, and add it to the cleanup handler
QObject *FactoryComponent::createObject()
{
```

```
    return objects.add( new QObject() );
}

// QLibraryInterface implementation
bool FactoryComponent::init()
{
    return TRUE;
}

void FactoryComponent::cleanup()
{
}

// it is only safe to unload the library when all QObject's have been destroyed
bool FactoryComponent::canUnload() const
{
    return objects.isEmpty();
}
```

See also Object Model.

Member Function Documentation

QObjectCleanupHandler::QObjectCleanupHandler ()

Constructs an empty QObjectCleanupHandler.

QObjectCleanupHandler::~QObjectCleanupHandler ()

Destroys the cleanup handler. All objects in this cleanup handler will be deleted.

QObject * QObjectCleanupHandler::add (QObject * object)

Adds *object* to this cleanup handler and returns the pointer to the object.

void QObjectCleanupHandler::clear ()

Deletes all objects in this cleanup handler. The cleanup handler becomes empty.

bool QObjectCleanupHandler::isEmpty () const

Returns TRUE if this cleanup handler is empty or all objects in this cleanup handler have been destroyed, otherwise return FALSE.

void QObjectCleanupHandler::remove (QObject * object)

Removes the *object* from this cleanup handler. The object will not be destroyed.

QPaintEvent Class Reference

The QPaintEvent class contains event parameters for paint events.

`#include <qevent.h>`

Inherits QEvent [p. 67].

Public Members

- **QPaintEvent** (const QRegion & paintRegion, bool erased = TRUE)
- **QPaintEvent** (const QRect & paintRect, bool erased = TRUE)
- const QRect & **rect** () const
- const QRegion & **region** () const
- bool **erased** () const

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 part of a widget is exposed because an overlying widget is moved.

The event contains a region() that needs to be updated, and a rect() that is the bounding rectangle of that region. Both are provided because many widgets can't make much use of region(), and rect() can be much faster than region().boundingRect(). Painting is clipped to region() during processing of a paint event.

The erased() function returns TRUE if the region() has been cleared to the widget's background (see QWidget::backgroundMode()), and FALSE if the region's contents are arbitrary.

See also QPainter [Graphics with Qt], QWidget::update() [Widgets with Qt], QWidget::repaint() [Widgets with Qt], QWidget::paintEvent() [Widgets with Qt], QWidget::backgroundMode [Widgets with Qt], QRegion [Graphics with Qt] and Event Classes.

Member Function Documentation

QPaintEvent::QPaintEvent (const QRegion & paintRegion, bool erased = TRUE)

Constructs a paint event object with the region that should be updated. The region is given by *paintRegion*. If *erased* is TRUE the region will be cleared before repainting.

QPaintEvent::QPaintEvent (const QRect & paintRect, bool erased = TRUE)

Constructs a paint event object with the rectangle that should be updated. The region is given by *paintRect*. If *erased* is TRUE the region will be cleared before repainting.

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() [p. 126] and QPainter::setClipRect() [Graphics with Qt].

Examples: life/life.cpp, qfd/fontdisplayer.cpp, showimg/showimg.cpp, t10/cannon.cpp, t11/cannon.cpp, t13/cannon.cpp and tooltip/tooltip.cpp.

const QRegion & QPaintEvent::region () const

Returns the region that should be updated.

See also rect() [p. 126] and QPainter::setClipRegion() [Graphics with Qt].

Examples: qfd/fontdisplayer.cpp and scribble/scribble.cpp.

QPlatinumStyle Class Reference

The QPlatinumStyle class provides Mac/Platinum look and feel.

`#include <qplatinumstyle.h>`

Inherits QWindowsStyle [p. 179].

Public Members

- **QPlatinumStyle** ()

Protected Members

- QColor **mixedColor** (const QColor & c1, const QColor & c2) const
- void **drawRiffles** (QPainter * p, int x, int y, int w, int h, const QColorGroup & g, bool horizontal) const

Detailed Description

The QPlatinumStyle class provides Mac/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.

See also Widget Appearance and Style.

Member Function Documentation

QPlatinumStyle::QPlatinumStyle ()

Constructs a QPlatinumStyle

void QPlatinumStyle::drawRiffles (QPainter * p, int x, int y, int w, int h, const QColorGroup & g, bool horizontal) const [protected]

Draws the nifty Macintosh decoration used on sliders using painter *p* and colorgroup *g*. *x*, *y*, *w*, *h* and *horizontal* specify the geometry and orientation of the riffles.

QColor QPlatinumStyle::mixedColor (const QColor & c1, const QColor & c2) const [protected]

Mixes two colors *c1* and *c2* to a new color.

QResizeEvent Class Reference

The QResizeEvent class contains event parameters for resize events.

`#include <qevent.h>`

Inherits QEvent [p. 67].

Public Members

- **QResizeEvent** (const QSize & size, const QSize & oldSize)
- const QSize & **size** () const
- const QSize & **oldSize** () const

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::size [Widgets with Qt], QWidget::geometry [Widgets with Qt] and Event Classes.

Member Function Documentation

QResizeEvent::QResizeEvent (const QSize & size, const QSize & oldSize)

Constructs a resize event with the new and old widget sizes, *size* and *oldSize* respectively.

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().

Example: life/life.cpp.

QSGIStyle Class Reference

The QSGIStyle class provides SGI/Irix look and feel.

`#include <qsgistyle.h>`

Inherits QMotifStyle [p. 116].

Public Members

- **QSGIStyle** (bool useHighlightCols = FALSE)
- virtual **~QSGIStyle** ()

Detailed Description

The QSGIStyle class provides SGI/Irix look and feel.

This class implements the SGI look and feel. It resembles the SGI/Irix Motif GUI style as closely as QStyle allows.

See also Widget Appearance and Style.

Member Function Documentation

QSGIStyle::QSGIStyle (bool useHighlightCols = FALSE)

Constructs a QSGIStyle.

If *useHighlightCols* is FALSE (default value), 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.

See also QMotifStyle::useHighlightColors() [p. 117].

QSGIStyle::~QSGIStyle () [virtual]

Destroys the style.

QShowEvent Class Reference

The QShowEvent class provides an event which is sent when a widget is shown.

`#include <qevent.h>`

Inherits QEvent [p. 67].

Public Members

- **QShowEvent** ()

Detailed Description

The QShowEvent class provides an event which is sent when a widget is shown.

There are two kinds 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 [p. 88] and Event Classes.

Member Function Documentation

QShowEvent::QShowEvent ()

Constructs a QShowEvent.

QStoredDrag Class Reference

The QStoredDrag class provides a simple stored-value drag object for arbitrary MIME data.

`#include <qdragobject.h>`

Inherits QDragObject [p. 59].

Inherited by QUriDrag [p. 171] and QColorDrag [p. 46].

Public Members

- **QStoredDrag** (const char * mimeType, QWidget * dragSource = 0, const char * name = 0)
- **~QStoredDrag** ()
- virtual void **setEncodedData** (const QByteArray & encodedData)
- virtual QByteArray **encodedData** (const char * m) const

Detailed Description

The QStoredDrag class provides a simple stored-value drag object for arbitrary MIME data.

When a block of data has only one representation, you can use a QStoredDrag to hold it.

For more information about drag and drop, see the QDragObject class and the drag and drop documentation.

See also Drag And Drop Classes.

Member Function Documentation

QStoredDrag::QStoredDrag (const char * mimeType, QWidget * dragSource = 0, const char * name = 0)

Constructs a QStoredDrag. The *dragSource* and *name* 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 up all allocated resources.

QByteArray QStoredDrag::encodedData (const char * m) const [virtual]

Returns the stored data. *m* contains the data's format.

See also setEncodedData() [p. 133].

Reimplemented from QMimeSource [Input/Output and Networking with Qt].

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. It 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.

QStyle Class Reference

The QStyle class specifies the look and feel of a GUI.

`#include <qstyle.h>`

Inherits QObject [Additional Functionality with Qt].

Inherited by QCommonStyle [p. 48].

Public Members

- **QStyle** ()
- virtual **~QStyle** ()
- virtual void **polish** (QWidget *)
- virtual void **unPolish** (QWidget *)
- virtual void **polish** (QApplication *)
- virtual void **unPolish** (QApplication *)
- virtual void **polish** (QPalette &)
- virtual void **polishPopupMenu** (QPopupMenu *)
- virtual QRect **itemRect** (QPainter * p, const QRect & r, int flags, bool enabled, const QPixmap * pixmap, const QString & text, int len = -1) const
- virtual void **drawItem** (QPainter * p, const QRect & r, int flags, const QColorGroup & g, bool enabled, const QPixmap * pixmap, const QString & text, int len = -1, const QColor * penColor = 0) const
- enum **PrimitiveElement** { PE_ButtonCommand, PE_ButtonDefault, PE_ButtonBevel, PE_ButtonTool, PE_ButtonDropDown, PE_FocusRect, PE_ArrowUp, PE_ArrowDown, PE_ArrowRight, PE_ArrowLeft, PE_SpinWidgetUp, PE_SpinWidgetDown, PE_SpinWidgetPlus, PE_SpinWidgetMinus, PE_Indicator, PE_IndicatorMask, PE_ExclusiveIndicator, PE_ExclusiveIndicatorMask, PE_DockWindowHandle, PE_DockWindowSeparator, PE_DockWindowResizeHandle, PE_Splitter, PE_Panel, PE_PanelPopup, PE_PanelMenuBar, PE_PanelDockWindow, PE_TabBarBase, PE_HeaderSection, PE_HeaderArrow, PE_StatusBarSection, PE_GroupBoxFrame, PE_Separator, PE_SizeGrip, PE_CheckMark, PE_ScrollBarAddLine, PE_ScrollBarSubLine, PE_ScrollBarAddPage, PE_ScrollBarSubPage, PE_ScrollBarSlider, PE_ScrollBarFirst, PE_ScrollBarLast, PE_ProgressBarChunk, PE_CustomBase = 0xf000000 }
- enum **StyleFlags** { Style_Default = 0x00000000, Style_Enabled = 0x00000001, Style_Raised = 0x00000002, Style_Sunken = 0x00000004, Style_Off = 0x00000008, Style_NoChange = 0x00000010, Style_On = 0x00000020, Style_Down = 0x00000040, Style_Horizontal = 0x00000080, Style_HasFocus = 0x00000100, Style_Top = 0x00000200, Style_Bottom = 0x00000400, Style_FocusAtBorder = 0x00000800, Style_AutoRaise = 0x00001000, Style_MouseOver = 0x00002000, Style_Up = 0x00004000, Style_Selected = 0x00008000, Style_Active = 0x00010000, Style_ButtonDefault = 0x00020000 }
- virtual void **drawPrimitive** (PrimitiveElement pe, QPainter * p, const QRect & r, const QColorGroup & cg, SFlags flags = Style_Default, const QStyleOption & opt = QStyleOption::Default) const
- enum **ControlElement** { CE_PushButton, CE_PushButtonLabel, CE_CheckBox, CE_CheckBoxLabel, CE_RadioButton, CE_RadioButtonLabel, CE_TabBarTab, CE_TabBarLabel, CE_ProgressBarGroove, CE_ProgressBarContents, CE_ProgressBarLabel, CE_PopupMenuItem, CE_MenuBarItem, CE_ToolButtonLabel, CE_CustomBase = 0xf0000000 }

- virtual void **drawControl** (ControlElement element, QPainter * p, const QWidget * widget, const QRect & r, const QColorGroup & cg, SFlags how = Style_Default, const QStyleOption & opt = QStyleOption::Default) const
- virtual void **drawControlMask** (ControlElement element, QPainter * p, const QWidget * widget, const QRect & r, const QStyleOption & opt = QStyleOption::Default) const
- enum **SubRect** { SR_PushButtonContents, SR_PushButtonFocusRect, SR_CheckBoxIndicator, SR_CheckBoxContents, SR_CheckBoxFocusRect, SR_RadioButtonIndicator, SR_RadioButtonContents, SR_RadioButtonFocusRect, SR_ComboBoxFocusRect, SR_SliderFocusRect, SR_DockWindowHandleRect, SR_ProgressBarGroove, SR_ProgressBarContents, SR_ProgressBarLabel, SR_ToolButtonContents, SR_CustomBase = 0xf0000000 }
- virtual QRect **subRect** (SubRect subrect, const QWidget * widget) const
- enum **ComplexControl** { CC_SpinWidget, CC_ComboBox, CC_ScrollBar, CC_Slider, CC_ToolButton, CC_TitleBar, CC_ListView, CC_CustomBase = 0xf0000000 }
- enum **SubControl** { SC_None = 0x00000000, SC_ScrollBarAddLine = 0x00000001, SC_ScrollBarSubLine = 0x00000002, SC_ScrollBarAddPage = 0x00000004, SC_ScrollBarSubPage = 0x00000008, SC_ScrollBarFirst = 0x00000010, SC_ScrollBarLast = 0x00000020, SC_ScrollBarSlider = 0x00000040, SC_ScrollBarGroove = 0x00000080, SC_SpinWidgetUp = 0x00000001, SC_SpinWidgetDown = 0x00000002, SC_SpinWidgetFrame = 0x00000004, SC_SpinWidgetEditField = 0x00000008, SC_SpinWidgetButtonField = 0x00000010, SC_ComboBoxFrame = 0x00000001, SC_ComboBoxEditField = 0x00000002, SC_ComboBoxArrow = 0x00000004, SC_SliderGroove = 0x00000001, SC_SliderHandle = 0x00000002, SC_SliderTickmarks = 0x00000004, SC_ToolButton = 0x00000001, SC_ToolButtonMenu = 0x00000002, SC_TitleBarLabel = 0x00000001, SC_TitleBarSysMenu = 0x00000002, SC_TitleBarMinButton = 0x00000004, SC_TitleBarMaxButton = 0x00000008, SC_TitleBarCloseButton = 0x00000010, SC_TitleBarNormalButton = 0x00000020, SC_TitleBarShadeButton = 0x00000040, SC_TitleBarUnshadeButton = 0x00000080, SC_ListView = 0x00000001, SC_ListViewBranch = 0x00000002, SC_ListViewExpand = 0x00000004, SC_All = 0xffffffff }
- virtual void **drawComplexControl** (ComplexControl control, QPainter * p, const QWidget * widget, const QRect & r, const QColorGroup & cg, SFlags how = Style_Default, SCFlags sub = SC_All, SCFlags subActive = SC_None, const QStyleOption & opt = QStyleOption::Default) const
- virtual void **drawComplexControlMask** (ComplexControl control, QPainter * p, const QWidget * widget, const QRect & r, const QStyleOption & opt = QStyleOption::Default) const
- virtual QRect **querySubControlMetrics** (ComplexControl control, const QWidget * widget, SubControl subcontrol, const QStyleOption & opt = QStyleOption::Default) const
- virtual SubControl **querySubControl** (ComplexControl control, const QWidget * widget, const QPoint & pos, const QStyleOption & opt = QStyleOption::Default) const
- enum **PixelMetric** { PM_ButtonMargin, PM_ButtonDefaultIndicator, PM_MenuButtonIndicator, PM_ButtonShiftHorizontal, PM_ButtonShiftVertical, PM_DefaultFrameWidth, PM_SpinBoxFrameWidth, PM_MaximumDragDistance, PM_ScrollBarExtent, PM_ScrollBarSliderMin, PM_SliderThickness, PM_SliderControlThickness, PM_SliderLength, PM_SliderTickmarkOffset, PM_SliderSpaceAvailable, PM_DockWindowSeparatorExtent, PM_DockWindowHandleExtent, PM_DockWindowFrameWidth, PM_MenuBarFrameWidth, PM_TabBarTabOverlap, PM_TabBarTabHSpace, PM_TabBarTabVSpace, PM_TabBarBaseHeight, PM_TabBarBaseOverlap, PM_ProgressBarChunkWidth, PM_SplitterWidth, PM_TitleBarHeight, PM_IndicatorWidth, PM_IndicatorHeight, PM_ExclusiveIndicatorWidth, PM_ExclusiveIndicatorHeight, PM_CustomBase = 0xf0000000 }
- virtual int **pixelMetric** (PixelMetric metric, const QWidget * widget = 0) const
- enum **ContentsType** { CT_PushButton, CT_CheckBox, CT_RadioButton, CT_ToolButton, CT_ComboBox, CT_Splitter, CT_DockWindow, CT_ProgressBar, CT_PopupMenuItem, CT_CustomBase = 0xf0000000 }
- virtual QSize **sizeFromContents** (ContentsType contents, const QWidget * widget, const QSize & contentsSize, const QStyleOption & opt = QStyleOption::Default) const
- enum **StyleHint** { SH_EtchDisabledText, SH_GUIStyle, SH_ScrollBar_BackgroundMode, SH_ScrollBar_MiddleClickAbsolutePosition, SH_ScrollBar_ScrollWhenPointerLeavesControl, SH_TabBar_SelectMouseType, SH_TabBar_Alignment, SH_Header_ArrowAlignment, SH_Slider_SnapToValue, SH_Slider_SloppyKeyEvents, SH_ProgressDialog_CenterCancelButton, SH_ProgressDialog_TextLabelAlignment, SH_PrintDialog_RightAlignButtons, SH_MainWindow_SpaceBelowMenuBar, SH_FontDialog_SelectAssociatedText,

SH_PopupMenu_AllowActiveAndDisabled, SH_PopupMenu_SpaceActivatesItem, SH_PopupMenu_SubMenuPopupDelay, SH_ScrollView_FrameOnlyAroundContents, SH_MenuBar_AltKeyNavigation, SH_ComboBox_ListMouseTracking, SH_PopupMenu_MouseTracking, SH_MenuBar_MouseTracking, SH_ItemView_ChangeHighlightOnFocus, SH_Widget_ShareActivation, SH_Workspace_FillSpaceOnMaximize, SH_ComboBox_Popup, SH_CustomBase = 0xf0000000 }

- virtual int **styleHint** (StyleHint stylehint, const QWidget * widget = 0, const QStyleOption & opt = QStyleOption::Default, QStyleHintReturn * returnData = 0) const
- enum **StylePixmap** { SP_TitleBarMinButton, SP_TitleBarMaxButton, SP_TitleBarCloseButton, SP_TitleBarNormalButton, SP_TitleBarShadeButton, SP_TitleBarUnshadeButton, SP_DockWindowCloseButton, SP_MessageBoxInformation, SP_MessageBoxWarning, SP_MessageBoxCritical, SP_CustomBase = 0xf0000000 }
- virtual QPixmap **stylePixmap** (StylePixmap stylepixmap, const QWidget * widget = 0, const QStyleOption & opt = QStyleOption::Default) const
- int defaultFrameWidth () const *(obsolete)*
- void tabbarMetrics (const QWidget * t, int & hf, int & vf, int & ov) const *(obsolete)*
- QSize scrollBarExtent () const *(obsolete)*

Static Public Members

- QRect **visualRect** (const QRect & logical, const QWidget * w)
- QRect **visualRect** (const QRect & logical, const QRect & bounding)

Detailed Description

The QStyle class specifies the look and feel of a GUI.

A large number of GUI 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 functions. It also provides two feel options: Motif and Windows.

Although it is not possible to fully enumerate the look of graphic elements and the feel of widgets in a GUI, QStyle provides a considerable amount of control and customisability.

In Qt 1.x 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 reimplement some or all of the drawing functions to modify the look of all widgets that use those functions.

Languages written from right to left (such as Arabic and Hebrew) usually also mirror the whole layout of widgets. If you design a style, you should take special care when drawing asymmetric elements to make sure that they also look correct in a mirrored layout. You can start your application with -reverse to check the mirrored layout. Also notice, that for a reversed layout, the light usually comes from top right instead of top left.

The actual reverse layout is performed automatically when possible. However, for the sake of flexibility, the translation cannot be performed everywhere. The documentation for each function in the QStyle API states whether the function expects/returns logical or screen coordinates. Using logical coordinates (in ComplexControls, for example) provides great flexibility in controlling the look of a widget. Use visualRect() when necessary to translate logical coordinates into screen coordinates for drawing.

In Qt versions prior to 3.0 if you wanted a low level route into changing the appearance of a widget you would reimplement polish(). With the new 3.0 style engine the recommended approach is to reimplement the draw functions, for example drawItem(), drawPrimitive(), drawControl(), drawControlMask(), drawComplexControl() and drawComplexControlMask(). Each of these functions is called with a range of parameters that provide information that you can use to determine how to draw them, e.g. style flags, rectangle, color group, etc.

For information on changing elements of an existing style or creating your own style see the Style overview.

Styles can also be created as plugins.

See also Widget Appearance and Style.

Member Type Documentation

QStyle::ComplexControl

This enum represents a ComplexControl. ComplexControls have different behaviour depending upon where the user clicks on them or which keys are pressed.

- `QStyle::CC_SpinWidget`
- `QStyle::CC_ComboBox`
- `QStyle::CC_ScrollBar`
- `QStyle::CC_Slider`
- `QStyle::CC_ToolButton`
- `QStyle::CC_TitleBar`
- `QStyle::CC_ListView`

- `QStyle::CC_CustomBase` - base value for custom ControlElements. All values above this are reserved for custom use. Therefore, custom values must be greater than this value.

See also SubControl [p. 143] and drawComplexControl() [p. 145].

QStyle::ContentsType

This enum represents a ContentsType. It is used to calculate sizes for the contents of various widgets.

- `QStyle::CT_PushButton`
- `QStyle::CT_CheckBox`
- `QStyle::CT_RadioButton`
- `QStyle::CT_ToolButton`
- `QStyle::CT_ComboBox`
- `QStyle::CT_Splitter`
- `QStyle::CT_DockWindow`
- `QStyle::CT_ProgressBar`
- `QStyle::CT_PopupMenuItem`

- `QStyle::CT_CustomBase` - base value for custom ControlElements. All values above this are reserved for custom use. Therefore, custom values must be greater than this value.

See also sizeFromContents() [p. 151].

QStyle::ControlElement

This enum represents a ControlElement. A ControlElement is part of a widget that performs some action or display information to the user.

- `QStyle::CE_PushButton` - the bevel and default indicator of a QPushButton.
- `QStyle::CE_PushButtonLabel` - the label (iconset with text or pixmap) of a QPushButton.

- `QStyle::CE_CheckBox` - the indicator of a QCheckBox.
- `QStyle::CE_CheckBoxLabel` - the label (text or pixmap) of a QCheckBox.

- `QStyle::CE_RadioButton` - the indicator of a QRadioButton.
- `QStyle::CE_RadioButtonLabel` - the label (text or pixmap) of a QRadioButton.

- `QStyle::CE_TabBarTab` - the tab within a QTabBar (a QTab).
- `QStyle::CE_TabBarLabel` - the label within a QTab.

- `QStyle::CE_ProgressBarGroove` - the groove where the progress indicator is drawn in a QProgressBar.
- `QStyle::CE_ProgressBarContents` - the progress indicator of a QProgressBar.
- `QStyle::CE_ProgressBarLabel` - the text label of a QProgressBar.

- `QStyle::CE_PopupMenuItem` - a menu item in a QPopupMenu.

- `QStyle::CE_MenuBarItem` - a menu item in a QMenuBar.
- `QStyle::CE_ToolButtonLabel` - a tool button's label.

- `QStyle::CE_CustomBase` - base value for custom ControlElements. All values above this are reserved for custom use. Therefore, custom values must be greater than this value.

See also drawControl() [p. 146].

QStyle::PixelMetric

This enum represents a PixelMetric. A PixelMetric is a style dependent size represented as a single pixel value.

- `QStyle::PM_ButtonMargin` - amount of whitespace between pushbutton labels and the frame.
- `QStyle::PM_ButtonDefaultIndicator` - width of the default-button indicator frame.
- `QStyle::PM_MenuButtonIndicator` - width of the menu button indicator proportional to the widget height.
- `QStyle::PM_ButtonShiftHorizontal` - horizontal contents shift of a button when the button is down.
- `QStyle::PM_ButtonShiftVertical` - vertical contents shift of a button when the button is down.

- `QStyle::PM_DefaultFrameWidth` - default frame width, usually 2.
- `QStyle::PM_SpinBoxFrameWidth` - frame width of a spin box.

- `QStyle::PM_MaximumDragDistance` - 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. A value of -1 disables this behavior.

- `QStyle::PM_ScrollBarExtent` - width of a vertical scrollbar and the height of a horizontal scrollbar.

- `QStyle::PM_ScrollBarSliderMin` - the minimum height of a vertical scrollbar's slider and the minimum width of a horiztonal scrollbar slider.

- `QStyle::PM_SliderThickness` - total slider thickness.
- `QStyle::PM_SliderControlThickness` - thickness of the slider handle.
- `QStyle::PM_SliderLength` - length of the slider.
- `QStyle::PM_SliderTickmarkOffset` - the offset between the tickmarks and the slider.
- `QStyle::PM_SliderSpaceAvailable` - the available space for the slider to move.

- `QStyle::PM_DockWindowSeparatorExtent` - width of a separator in a horiztonal dock window and the height of a separator in a vertical dock window.
- `QStyle::PM_DockWindowHandleExtent` - width of the handle in a horizontal dock window and the height of the handle in a vertical dock window.
- `QStyle::PM_DockWindowFrameWidth` - frame width of a dock window.

- `QStyle::PM_MenuBarFrameWidth` - frame width of a menubar.

- `QStyle::PM_TabBarTabOverlap` - number of pixels the tabs should overlap.
- `QStyle::PM_TabBarTabHSpace` - extra space added to the tab width.
- `QStyle::PM_TabBarTabVSpace` - extra space added to the tab height.
- `QStyle::PM_TabBarBaseHeight` - height of the area between the tab bar and the tab pages.
- `QStyle::PM_TabBarBaseOverlap` - number of pixels the tab bar overlaps the tab bar base.

- `QStyle::PM_ProgressBarChunkWidth` - width of a chunk in a progress bar indicator.

- `QStyle::PM_SplitterWidth` - width of a splitter.
- `QStyle::PM_TitleBarHeight` - height of the title bar.
- `QStyle::PM_IndicatorWidth` - width of a check box indicator.
- `QStyle::PM_IndicatorHeight` - height of a checkbox indicator.
- `QStyle::PM_ExclusiveIndicatorWidth` - width of a radio button indicator.
- `QStyle::PM_ExclusiveIndicatorHeight` - height of a radio button indicator.

- `QStyle::PM_CustomBase` - base value for custom ControlElements. All values above this are reserved for custom use. Therefore, custom values must be greater than this value.

See also pixelMetric() [p. 149].

QStyle::PrimitiveElement

This enum represents the PrimitiveElements of a style. A PrimitiveElement is a common GUI element, such as a checkbox indicator or pushbutton bevel.

- `QStyle::PE_ButtonCommand` - button used to initiate an action, for example, a QPushButton.
- `QStyle::PE_ButtonDefault` - this button is the default button, e.g. in a dialog.
- `QStyle::PE_ButtonBevel` - generic button bevel.
- `QStyle::PE_ButtonTool` - tool button, for example, a QToolButton.
- `QStyle::PE_ButtonDropDown` - drop down button, for example, a tool button that displays a popup menu, for example, QPopupMenu.

- `QStyle::PE_FocusRect` - generic focus indicator.

- `QStyle::PE_ArrowUp` - up arrow.
- `QStyle::PE_ArrowDown` - down arrow.
- `QStyle::PE_ArrowRight` - right arrow.
- `QStyle::PE_ArrowLeft` - left arrow.

- `QStyle::PE_SpinWidgetUp` - up symbol for a spin widget, for example a QSpinBox.
- `QStyle::PE_SpinWidgetDown` - down symbol for a spin widget.
- `QStyle::PE_SpinWidgetPlus` - increase symbol for a spin widget.
- `QStyle::PE_SpinWidgetMinus` - decrease symbol for a spin widget.

- `QStyle::PE_Indicator` - on/off indicator, for example, a QCheckBox.
- `QStyle::PE_IndicatorMask` - bitmap mask for an indicator.
- `QStyle::PE_ExclusiveIndicator` - exclusive on/off indicator, for example, a QRadioButton.
- `QStyle::PE_ExclusiveIndicatorMask` - bitmap mask for an exclusive indicator.

- `QStyle::PE_DockWindowHandle` - tear off handle for dock windows and toolbars, for example QDockWindows and QToolBars.
- `QStyle::PE_DockWindowSeparator` - item separator for dock window and toolbar contents.
- `QStyle::PE_DockWindowResizeHandle` - resize handle for dock windows.
- `QStyle::PE_Splitter` - splitter handle; see also QSplitter.

- `QStyle::PE_Panel` - generic panel frame; see also QFrame.
- `QStyle::PE_PanelPopup` - panel frame for popup windows/menus; see also QPopupMenu.
- `QStyle::PE_PanelMenuBar` - panel frame for menu bars.
- `QStyle::PE_PanelDockWindow` - panel frame for dock windows and toolbars.
- `QStyle::PE_TabBarBase` - area below tabs in a tab widget, for example, QTab.

- `QStyle::PE_HeaderSection` - section of a list or table header; see also QHeader.
- `QStyle::PE_HeaderArrow` - arrow used to indicate sorting on a list or table header
- `QStyle::PE_StatusBarSection` - section of a status bar; see also QStatusBar.

- `QStyle::PE_GroupBoxFrame` - frame around a group box; see also QGroupBox.

- `QStyle::PE_Separator` - generic separator.

- `QStyle::PE_SizeGrip` - window resize handle; see also QSizeGrip.

- `QStyle::PE_CheckMark` - generic check mark; see also QCheckBox.

- `QStyle::PE_ScrollBarAddLine` - scrollbar line increase indicator (i.e. scroll down); see also QScrollBar.
- `QStyle::PE_ScrollBarSubLine` - scrollbar line decrease indicator (i.e. scroll up).
- `QStyle::PE_ScrollBarAddPage` - scolllbar page increase indicator (i.e. page down).
- `QStyle::PE_ScrollBarSubPage` - scrollbar page decrease indicator (i.e. page up).
- `QStyle::PE_ScrollBarSlider` - scrollbar slider
- `QStyle::PE_ScrollBarFirst` - scrollbar first line indicator (i.e. home).

- `QStyle::PE_ScrollBarLast` - scrollbar last line indicator (i.e. end).
- `QStyle::PE_ProgressBarChunk` - section of a progress bar indicator; see also QProgressBar.
- `QStyle::PE_CustomBase` - base value for custom ControlElements. All values above this are reserved for custom use. Therefore, custom values must be greater than this value.

See also drawPrimitive() [p. 148].

QStyle::StyleFlags

This enum represents flags for drawing PrimitiveElements. Not all primitives use all of these flags. Note that these flags may mean different things to different primitives. For an explanation of the relationship between primitives and their flags, as well as the different meanings of the flags, see the Style overview.

- `QStyle::Style_Default`
- `QStyle::Style_Enabled`
- `QStyle::Style_Raised`
- `QStyle::Style_Sunken`
- `QStyle::Style_Off`
- `QStyle::Style_NoChange`
- `QStyle::Style_On`
- `QStyle::Style_Down`
- `QStyle::Style_Horizontal`
- `QStyle::Style_HasFocus`
- `QStyle::Style_Top`
- `QStyle::Style_Bottom`
- `QStyle::Style_FocusAtBorder`
- `QStyle::Style_AutoRaise`
- `QStyle::Style_MouseOver`
- `QStyle::Style_Up`
- `QStyle::Style_Selected`
- `QStyle::Style_HasFocus`
- `QStyle::Style_Active`
- `QStyle::Style_ButtonDefault`

See also drawPrimitive() [p. 148].

QStyle::StyleHint

This enum represents a StyleHint. A StyleHint is a general look and/or feel hint.

- `QStyle::SH_EtchDisabledText` - disabled text is "etched" like Windows.
- `QStyle::SH_GUIStyle` - the GUI style to use.
- `QStyle::SH_ScrollBar_BackgroundMode` - the background mode for a QScrollBar. Possible values are any of those in the BackgroundMode enum.

- `QStyle::SH_ScrollBar_MiddleClickAbsolutePosition` - a boolean value. If TRUE, middle clicking on a scrollbar causes the slider to jump to that position. If FALSE, the middle clicking is ignored.
- `QStyle::SH_ScrollBar_ScrollWhenPointerLeavesControl` - a boolean value. If TRUE, when clicking a scrollbar SubControl, holding the mouse button down and moving the pointer outside the SubControl, the scrollbar continues to scroll. If FALSE, the scollbar stops scrolling when the pointer leaves the SubControl.

- `QStyle::SH_TabBar_Alignment` - the alignment for tabs in a QTabWidget. Possible values are Qt::AlignLeft, Qt::AlignCenter and Qt::AlignRight.
- `QStyle::SH_Header_ArrowAlignment` - the placement of the sorting indicator may appear in list or table headers. Possible values are Qt::Left or Qt::Right.
- `QStyle::SH_Slider_SnapToValue` - sliders snap to values while moving, like Windows
- `QStyle::SH_Slider_SloppyKeyEvents` - key presses handled in a sloppy manner, i.e. left on a vertical slider subtracts a line.
- `QStyle::SH_ProgressDialog_CenterCancelButton` - center button on progress dialogs, like Motif, otherwise right aligned.
- `QStyle::SH_ProgressDialog_TextLabelAlignment` - Qt::AlignmentFlags — text label alignment in progress dialogs; Center on windows, Auto|VCenter otherwise.
- `QStyle::SH_PrintDialog_RightAlignButtons` - right align buttons in the print dialog, like Windows.
- `QStyle::SH_MainWindow_SpaceBelowMenuBar` - 1 or 2 pixel space between the menubar and the dockarea, like Windows.
- `QStyle::SH_FontDialog_SelectAssociatedText` - select the text in the line edit, or when selecting an item from the listbox, or when the line edit receives focus, like Windows.
- `QStyle::SH_PopupMenu_AllowActiveAndDisabled` - allows disabled menu items to be active.
- `QStyle::SH_PopupMenu_SpaceActivatesItem` - pressing Space activates the item, like Motif.
- `QStyle::SH_PopupMenu_SubMenuPopupDelay` - the number of milliseconds to wait before opening a submenu; 256 on windows, 96 on Motif.
- `QStyle::SH_ScrollView_FrameOnlyAroundContents` - whether scrollviews draw their frame only around contents (like Motif), or around contents, scrollbars and corner widgets (like Windows).
- `QStyle::SH_MenuBar_AltKeyNavigation` - menubars items are navigable by pressing Alt, followed by using the arrow keys to select the desired item.
- `QStyle::SH_ComboBox_ListMouseTracking` - mouse tracking in combobox dropdown lists.
- `QStyle::SH_PopupMenu_MouseTracking` - mouse tracking in popup menus.
- `QStyle::SH_MenuBar_MouseTracking` - mouse tracking in menubars.
- `QStyle::SH_ItemView_ChangeHighlightOnFocus` - gray out selected items when losing focus.
- `QStyle::SH_Widget_ShareActivation` - turn on sharing activation with floating modeless dialogs.
- `QStyle::SH_TabBar_SelectMouseType` - which type of mouse event should cause a tab to be selected.
- `QStyle::SH_ComboBox_Popup` - allows popups as a combobox dropdown menu.
- `QStyle::SH_Workspace_FillSpaceOnMaximize` - the workspace should maximize the client area.

- `QStyle::SH_CustomBase` - base value for custom ControlElements. All values above this are reserved for custom use. Therefore, custom values must be greater than this value.

See also styleHint() [p. 151].

QStyle::StylePixmap

This enum represents a StylePixmap. A StylePixmap is a pixmap that can follow some existing GUI style or guideline.

- `QStyle::SP_TitleBarMinButton` - minimize button on titlebars. For example, in a QWorkspace.
- `QStyle::SP_TitleBarMaxButton` - maximize button on titlebars.
- `QStyle::SP_TitleBarCloseButton` - close button on titlebars.
- `QStyle::SP_TitleBarNormalButton` - normal (restore) button on titlebars.
- `QStyle::SP_TitleBarShadeButton` - shade button on titlebars.
- `QStyle::SP_TitleBarUnshadeButton` - unshade button on titlebars.
- `QStyle::SP_MessageBoxInformation` - the 'information' icon.
- `QStyle::SP_MessageBoxWarning` - the 'warning' icon.
- `QStyle::SP_MessageBoxCritical` - the 'critical' icon.

- `QStyle::SP_DockWindowCloseButton` - close button on dock windows; see also QDockWindow.

- `QStyle::SP_CustomBase` - base value for custom ControlElements. All values above this are reserved for custom use. Therefore, custom values must be greater than this value.

See also stylePixmap() [p. 151].

QStyle::SubControl

This enum represents a SubControl within a ComplexControl.

- `QStyle::SC_None` - special value that matches no other SubControl.

- `QStyle::SC_ScrollBarAddLine` - scrollbar add line (i.e. down/right arrow); see also QScrollbar.
- `QStyle::SC_ScrollBarSubLine` - scrollbar sub line (i.e. up/left arrow).
- `QStyle::SC_ScrollBarAddPage` - scrollbar add page (i.e. page down).
- `QStyle::SC_ScrollBarSubPage` - scrollbar sub page (i.e. page up).
- `QStyle::SC_ScrollBarFirst` - scrollbar first line (i.e. home).
- `QStyle::SC_ScrollBarLast` - scrollbar last line (i.e. end).
- `QStyle::SC_ScrollBarSlider` - scrollbar slider handle.
- `QStyle::SC_ScrollBarGroove` - special subcontrol which contains the area in which the slider handle may move.

- `QStyle::SC_SpinWidgetUp` - spinwidget up/increase; see also QSpinBox.
- `QStyle::SC_SpinWidgetDown` - spinwidget down/decrease.
- `QStyle::SC_SpinWidgetFrame` - spinwidget frame.
- `QStyle::SC_SpinWidgetEditField` - spinwidget edit field.
- `QStyle::SC_SpinWidgetButtonField` - spinwidget button field.

- `QStyle::SC_ComboBoxEditField` - combobox edit field; see also QComboBox.
- `QStyle::SC_ComboBoxArrow` - combobox arrow
- `QStyle::SC_ComboBoxFrame` - combobox frame

- `QStyle::SC_SliderGroove` - special subcontrol which contains the area in which the slider handle may move.
- `QStyle::SC_SliderHandle` - slider handle.
- `QStyle::SC_SliderTickmarks` - slider tickmarks.

- `QStyle::SC_ToolButton` - tool button; see also QToolbutton.
- `QStyle::SC_ToolButtonMenu` - subcontrol for opening a popup menu in a tool button; see also QPopupMenu.

- `QStyle::SC_TitleBarSysMenu` - system menu button (i.e. restore, close, etc.).
- `QStyle::SC_TitleBarMinButton` - minimize button.
- `QStyle::SC_TitleBarMaxButton` - maximize button.
- `QStyle::SC_TitleBarCloseButton` - close button.
- `QStyle::SC_TitleBarLabel` - window title label.
- `QStyle::SC_TitleBarNormalButton` - normal (restore) button.
- `QStyle::SC_TitleBarShadeButton` - shade button.
- `QStyle::SC_TitleBarUnshadeButton` - unshade button.

- `QStyle::SC_ListView` - (internal)
- `QStyle::SC_ListViewBranch` - (internal)
- `QStyle::SC_ListViewExpand` - expand item (i.e. show/hide child items).

- `QStyle::SC_All` - special value that matches all SubControls.

See also ComplexControl [p. 137].

QStyle::SubRect

This enum represents a sub-area of a widget. Style implementations would use these areas to draw the different parts of a widget.

- `QStyle::SR_PushButtonContents` - area containing the label (iconset with text or pixmap).
- `QStyle::SR_PushButtonFocusRect` - area for the focus rect (usually larger than the contents rect).
- `QStyle::SR_CheckBoxIndicator` - area for the state indicator (e.g. check mark).
- `QStyle::SR_CheckBoxContents` - area for the label (text or pixmap).
- `QStyle::SR_CheckBoxFocusRect` - area for the focus indicator.

- `QStyle::SR_RadioButtonIndicator` - area for the state indicator.
- `QStyle::SR_RadioButtonContents` - area for the label.
- `QStyle::SR_RadioButtonFocusRect` - area for the focus indicator.

- `QStyle::SR_ComboBoxFocusRect` - area for the focus indicator.

- `QStyle::SR_SliderFocusRect` - area for the focus indicator.

- `QStyle::SR_DockWindowHandleRect` - area for the tear-off handle.

- `QStyle::SR_ProgressBarGroove` - area for the groove.
- `QStyle::SR_ProgressBarContents` - area for the progress indicator.
- `QStyle::SR_ProgressBarLabel` - area for the text label.

- `QStyle::SR_ToolButtonContents` - area for the tool button's label.

- `QStyle::SR_CustomBase` - base value for custom ControlElements. All values above this are reserved for custom use. Therefore, custom values must be greater than this value.

See also subRect() [p. 152].

Member Function Documentation

QStyle::QStyle ()

Constructs a QStyle.

QStyle::~QStyle () [virtual]

Destroys the style and frees all allocated resources.

int QStyle::defaultFrameWidth () const

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

void QStyle::drawComplexControl (ComplexControl control, QPainter * p, const QWidget * widget, const QRect & r, const QColorGroup & cg, SFlags how = Style_Default, SCFlags sub = SC_All, SCFlags subActive = SC_None, const QStyleOption & opt = QStyleOption::Default) const [virtual]

Draws the ComplexControl *control* using the painter *p* in the area *r*. Colors are used from the color group *cg*. The *sub* argument specifies which SubControls to draw. Multiple SubControls can be OR'ed together. The *subActive* argument specifies which SubControl is active.

The rect *r* should be in logical coordinates. Reimplementations of this function should use visualRect() to change the logical coordinates into screen coordinates when using drawPrimitive() and drawControl().

The *how* argument is used to control how the ComplexControl is drawn. Multiple flags can OR'ed together. See the table below for an explanation of which flags are used with the various ComplexControls.

The *widget* argument is a pointer to a QWidget or one of its subclasses. The widget can be cast to the appropriate type based on the value of *control*. The *opt* argument can be used to pass extra information required when drawing the ComplexControl. Note that *opt* may be the default value even for ComplexControls that can make use of the extra options. See the table below for the appropriate *widget* and *opt* usage:

CC_ToolButton(const QToolButton *)	Style_HasFocus Set if the toolbutton has input focus. Style_Down Set if the toolbutton is down (ie. mouse button or space pressed). Style_On Set if the toolbutton is a toggle button and is toggled on. Style_AutoRaise Set if the toolbutton has auto-raise enabled. Style_Raised Set if the button is not down, not on and doesn't contain the mouse when auto-raise is enabled.	When the QToolButton only contains the tool button
CC_TitleBar(const QWidget *)	Style_Enabled Set if the titlebar is enabled. Unused.	
CC_ListView(const QListView *)	Style_Enabled Set if the titlebar is enabled.	QStyleOption (QListViewItem *item) • opt.listViewItem() `

item is the item that needs branches drawn | an arrow, *t* is the arrow's type

See also ComplexControl [p. 137] and SubControl [p. 143].

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

void QStyle::drawComplexControlMask (ComplexControl control, QPainter * p, const QWidget * widget, const QRect & r, const QStyleOption & opt = QStyleOption::Default) const [virtual]

Draw a bitmask for the ComplexControl *control* using the painter *p* in the area *r*. See drawComplexControl() for an explanation of the use of the *widget* and *opt* arguments.

The rect *r* should be in logical coordinates. Reimplementations of this function should use visualRect() to change the logical corrdinates into screen coordinates when using drawPrimitive() and drawControl().

See also drawComplexControl() [p. 145] and ComplexControl [p. 137].

Example: themes/wood.cpp.

void QStyle::drawControl (ControlElement element, QPainter * p, const QWidget * widget, const QRect & r, const QColorGroup & cg, SFlags how = Style_Default, const QStyleOption & opt = QStyleOption::Default) const [virtual]

Draws the ControlElement *element* using the painter *p* in the area *r*. Colors are used from the color group *cg*.

The rect *r* should be in screen coordinates.

The *how* argument is used to control how the ControlElement is drawn. Multiple flags can be OR'ed together. See the table below for an explanation of which flags are used with the various ControlElements.

The *widget* argument is a pointer to a QWidget or one of its subclasses. The widget can be cast to the appropriate type based on the value of *element*. The *opt* argument can be used to pass extra information required when drawing the ControlElement. Note that *opt* may be the default value even for ControlElements that can make use of the extra options. See the table below for the appropriate *widget* and *opt* usage:

	Style_Selected Set if this tab is the current tab.	QTab being drawn.
CE_ProgressBarContents(const QProgressBar *)		and
CE_ProgressBarLabel(const QProgressBar *)	Style_Enabled Set if the progressbar is enabled.	Unused.
	Style_HasFocus Set if the progressbar has input focus.	
CE_PopupMenuItem(const QPopupMenu *)	Style_Enabled Set if the menuitem is enabled.	opt.menuItem() opt.tabWidth() opt.maxIconWidth()

mi is the menu item being drawn. QMenuItem is currently an internal class.

Style_Active Set if the menuitem is the current item. *tabwidth* is the width of the tab column where key accelerators are drawn.

Style_Down Set if the menuitem is down (i.e., the mouse button or space bar is pressed). *maxpmwidth* is the maximum width of the check column where checkmarks and iconsets are drawn.

CE_MenuBarItem(const QMenuBar *)	Style_Enabled Set if the menuitem is enabled	opt.menuItem()

mi is the menu item being drawn.

Style_Active Set if the menuitem is the current item.

Style_Down Set if the menuitem is down (i.e., a mouse button or the space bar is pressed).

Style_HasFocus Set if the menubar has input focus.

CE_ToolButtonLabel(const QToolButton *)

Style_Enabled Set if the toolbutton is enabled.

opt.arrowType()

When the tool button only contains an arrow, *r* is the arrow's type.

Style_HasFocus Set if the toolbutton has input focus.

Style_Down Set if the toolbutton is down (i.e., a mouse button or the space is pressed).

Style_On Set if the toolbutton is a toggle button and is toggled on.

Style_AutoRaise Set if the toolbutton has auto-raise enabled.

Style_MouseOver Set if the mouse pointer is over the toolbutton.

Style_Raised Set if the button is not down, not on and doesn't contain the mouse when auto-raise is enabled.

`

See also ControlElement [p. 138] and StyleFlags [p. 141].

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

void QStyle::drawControlMask (ControlElement element, QPainter * p, const QWidget * widget, const QRect & r, const QStyleOption & opt = QStyleOption::Default) const [virtual]

Draw a bitmask for the ControlElement *element* using the painter *p* in the area \r. See drawControl() for an explanation of the use of the *widget* and *opt* arguments.

The rect *r* should be in screen coordinates.

See also drawControl() [p. 146] and ControlElement [p. 138].

Example: themes/wood.cpp.

void QStyle::drawItem (QPainter * p, const QRect & r, int flags, const QColorGroup & g, bool enabled, const QPixmap * pixmap, const QString & text, int len = -1, const QColor * penColor = 0) const [virtual]

Draws the *text* or *pixmap* in rectangle *r* using painter *p* and color group *g*. The pen color is specified with *penColor*. The *enabled* bool indicates whether or not the item is enabled; when reimplementing this bool should influence how the item is drawn. If *len* is -1 (the default) all the *text* is drawn; otherwise only the first *len* characters of *text* are drawn. The text is aligned and wrapped according to the alignment *flags* (see Qt::AlignmentFlags).

By default, if both the text and the pixmap are not null, the pixmap is drawn and the text is ignored.

void QStyle::drawPrimitive (PrimitiveElement pe, QPainter * p, const QRect & r, const QColorGroup & cg, SFlags flags = Style_Default, const QStyleOption & opt = QStyleOption::Default) const [virtual]

Draws the style PrimitiveElement *pe* using the painter *p* in the area *r*. Colors are used from the color group *cg*.

The rect *r* should be in screen coordinates.

The *flags* argument is used to control how the PrimitiveElement is drawn. Multiple flags can be OR'ed together.

For example, a pressed button would be drawn with the flags Style_Enabled and Style_Down.

The *opt* argument can be used to control how various PrimitiveElements are drawn. Note that *opt* may be the default value even for PrimitiveElements that make use of extra options. When *opt* is non-default, it is used as follows:

	linewidth is the line width for drawing the panel. *midlinewidth* is the mid-line width for drawing the panel.
PE_PanelPopup	opt.lineWidth() opt.midLineWidth()
	linewidth is the line width for drawing the panel.
	midlinewidth is the mid-line width for drawing the panel.
PE_PanelMenuBar	opt.lineWidth() opt.midLineWidth()
	linewidth is the line width for drawing the panel.
	midlinewidth is the mid-line width for drawing the panel.
PE_PanelDockWindow	opt.lineWidth() opt.midLineWidth()
	linewidth is the line width for drawing the panel.
	midlinewidth is the mid-line width for drawing the panel.
PE_GroupBoxFrame	opt.lineWidth() opt.midLineWidth() opt.frameShape() opt.frameShadow()
	linewidth is the line width for the group box.
	midlinewidth is the mid-line width for the group box.

shape is the frame shape for the group box.

---

shadow is the frame shadow for the group box.

`

For all other PrimitiveElements, *opt* is unused.

See also StyleFlags [p. 141].

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

QRect QStyle::itemRect (QPainter * p, const QRect & r, int flags, bool enabled, const QPixmap * pixmap, const QString & text, int len = -1) const [virtual]

Returns the appropriate area (see below) within rectangle *r* in which to draw the *text* or *pixmap* using painter *p*. If *len* is -1 (the default) all the *text* is drawn; otherwise only the first *len* characters of *text* are drawn. The text is aligned in accordance with the alignment *flags* (see Qt::AlignmentFlags). The *enabled* bool indicates whether or not the item is enabled.

If *r* is larger than the area needed to render the *text* the rectangle that is returned will be offset within *r* in accordance with the alignment *flags*. For example if *flags* is AlignCenter the returned rectangle will be centered within *r*. If *r* is smaller than the area needed the rectangle that is returned will be *larger* than *r* (the smallest rectangle large enough to render the *text* or *pixmap*).

By default, if both the text and the pixmap are not null, the the text is ignored.

int QStyle::pixelMetric (PixelMetric metric, const QWidget * widget = 0) const [virtual]

Returns the pixel metric for *metric*. The *widget* argument is a pointer to a QWidget or one of its subclasses. The widget can be cast to the appropriate type based on the value of *metric*. Note that *widget* may be zero even for PixelMetrics that can make use of *widget*. See the table below for the appropriate *widget* casts:

PixelMetric	Widget Cast
PM_SliderControlThickness	(const QSlider *)
PM_SliderLength	(const QSlider *)
PM_SliderTickmarkOffset	(const QSlider *)
PM_SliderSpaceAvailable	(const QSlider *)
PM_TabBarTabOverlap	(const QTabBar *)
PM_TabBarTabHSpace	(const QTabBar *)
PM_TabBarTabVSpace	(const QTabBar *)
PM_TabBarBaseHeight	(const QTabBar *)
PM_TabBarBaseOverlap	(const QTabBar *)

Example: themes/metal.cpp.

void QStyle::polish (QWidget *) [virtual]

Initializes the appearance of a widget.

This function is called for every widget at some point after it has been fully created but just *before* it is shown the very first time.

Reasonable actions in this function might be to call QWidget::setBackgroundMode for the widget. An example of highly unreasonable use would be setting the geometry! Reimplementing this function gives you a backdoor

through which you can change the appearance of a widget. With Qt 3.0's style engine you will rarely need to write your own polish(); instead reimplement drawItem(), drawPrimitive(), etc.

The QWidget::inherits() function may provide enough information to allow class-specific customizations. But be careful not to hard-code things too much because new QStyle subclasses will be expected to work reasonably with all current and *future* widgets.

See also unPolish() [p. 153].

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

void QStyle::polish (QApplication *) [virtual]

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

Late initialization of the QApplication object.

See also unPolish() [p. 153].

void QStyle::polish (QPalette &) [virtual]

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

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 QPalette [Graphics with Qt] and QApplication::setPalette() [Additional Functionality with Qt].

void QStyle::polishPopupMenu (QPopupMenu *) [virtual]

Polishes the popup menu according to the GUI style. This usually means setting the mouse tracking (QPopupMenu::setMouseTracking()) and whether the menu is checkable by default (QPopupMenu::setCheckable()).

SubControl QStyle::querySubControl (ComplexControl control, const QWidget * widget, const QPoint & pos, const QStyleOption & opt = QStyleOption::Default) const [virtual]

Returns the SubControl for *widget* at the point *pos*. The *widget* argument is a pointer to a QWidget or one of its subclasses. The widget can be casted to the appropriate type based on the value of *control*. The *opt* argument can be used to pass extra information required when drawing the ComplexControl. Note that *opt* may be the default value even for ComplexControls that can make use of the extra options. See drawComplexControl() for an explanation of the *widget* and *opt* arguments.

Note that *pos* is passed in screen coordinates. When using querySubControlMetrics() to check for hits and misses, use visualRect() to change the logical coordinates into screen coordinates.

See also drawComplexControl() [p. 145], ComplexControl [p. 137], SubControl [p. 143] and querySubControlMetrics() [p. 150].

QRect QStyle::querySubControlMetrics (ComplexControl control, const QWidget * widget, SubControl subcontrol, const QStyleOption & opt = QStyleOption::Default) const [virtual]

Returns the rect for the SubControl *subcontrol* for *widget* in logical coordinates.

The *widget* argument is a pointer to a QWidget or one of its subclasses. The widget can be cast to the appropriate type based on the value of *control*. The *opt* argument can be used to pass extra information required when drawing the ComplexControl. Note that *opt* may be the default value even for ComplexControls that can make use of the extra options. See drawComplexControl() for an explanation of the *widget* and *opt* arguments.

See also drawComplexControl() [p. 145], ComplexControl [p. 137] and SubControl [p. 143].

Example: themes/wood.cpp.

QSize QStyle::scrollBarExtent () const

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

QSize QStyle::sizeFromContents (ContentsType contents, const QWidget * widget, const QSize & contentsSize, const QStyleOption & opt = QStyleOption::Default) const [virtual]

Returns the size of *widget* based on the contents size *contentsSize*.

The *widget* argument is a pointer to a QWidget or one of its subclasses. The widget can be cast to the appropriate type based on the value of *contents*. The *opt* argument can be used to pass extra information required when calculating the size. Note that *opt* may be the default value even for ContentsTypes that can make use of the extra options. See the table below for the appropriate *widget* and *opt* usage:

ContentsType	Widget Cast	Options	Notes
CT_PushButton	(const QPushButton *)	Unused.	
CT_CheckBox	(const QCheckBox *)	Unused.	
CT_RadioButton	(const QRadioButton *)	Unused.	
CT_ToolButton	(const QToolButton *)	Unused.	
CT_ComboBox	(const QComboBox *)	Unused.	
CT_Splitter	(const QSplitter *)	Unused.	
CT_DockWindow	(const QDockWindow *)	Unused.	
CT_ProgressBar	(const QProgressBar *)	Unused.	
CT_PopupMenuItem	(const QPopupMenu *)	QStyleOption (QMenuItem *mi) • opt.menuItem()	*mi* is the menu item to use when calculating the size. QMenuItem is currently an internal class.

int QStyle::styleHint (StyleHint stylehint, const QWidget * widget = 0, const QStyleOption & opt = QStyleOption::Default, QStyleHintReturn * returnData = 0) const [virtual]

Returns the style hint *stylehint* for *widget*. Currently, *widget*, *opt*, and *returnData* are unused, and are provided only for future development considerations.

For an explanation of the return value see StyleHint.

QPixmap QStyle::stylePixmap (StylePixmap stylepixmap, const QWidget * widget = 0, const QStyleOption & opt = QStyleOption::Default) const [virtual]

Returns a pixmap for *stylepixmap*.

The *opt* argument can be used to pass extra information required when drawing the ControlElement. Note that *opt* may be the default value even for StylePixmaps that can make use of the extra options. Currently, the *opt* argument is unused.

The *widget* argument is a pointer to a QWidget or one of its subclasses. The widget can be cast to the appropriate type based on the value of *stylepixmap*. See the table below for the appropriate *widget* casts:

StylePixmap	Widget Cast
SP_TitleBarMinButton	(const QWidget *)
SP_TitleBarMaxButton	(const QWidget *)
SP_TitleBarCloseButton	(const QWidget *)
SP_TitleBarNormalButton	(const QWidget *)
SP_TitleBarShadeButton	(const QWidget *)
SP_TitleBarUnshadeButton	(const QWidget *)
SP_DockWindowCloseButton	(const QDockWindow *)

See also StylePixmap [p. 142].

QRect QStyle::subRect (SubRect subrect, const QWidget * widget) const [virtual]

Returns the sub-area *subrect* for the *widget* in logical coordinates.

The *widget* argument is a pointer to a QWidget or one of its subclasses. The widget can be cast to the appropriate type based on the value of *subrect*. See the table below for the appropriate *widget* casts:

SubRect	Widget Cast
SR_PushButtonContents	(const QPushButton *)
SR_PushButtonFocusRect	(const QPushButton *)
SR_CheckBoxIndicator	(const QCheckBox *)
SR_CheckBoxContents	(const QCheckBox *)
SR_CheckBoxFocusRect	(const QCheckBox *)
SR_RadioButtonIndicator	(const QRadioButton *)
SR_RadioButtonContents	(const QRadioButton *)
SR_RadioButtonFocusRect	(const QRadioButton *)
SR_ComboBoxFocusRect	(const QComboBox *)
SR_DockWindowHandleRect	(const QWidget *)
SR_ProgressBarGroove	(const QProgressBar *)
SR_ProgressBarContents	(const QProgressBar *)
SR_ProgressBarLabel	(const QProgressBar *)

The tear-off handle (SR_DockWindowHandleRect) for QDockWindow is a private class. Use QWidget::parentWidget() to access the QDockWindow:

```
if ( !widget->parentWidget() )
    return;
const QDockWindow *dw = (const QDockWindow *) widget->parentWidget();
```

See also SubRect [p. 144].

Example: themes/wood.cpp.

void QStyle::tabbarMetrics (const QWidget * t, int & hf, int & vf, int & ov) const

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

void QStyle::unPolish (QWidget *) [virtual]

Undoes the initialization of a widget's appearance.

This function is the counterpart to polish. It is called for every polished widget when the style is dynamically changed. The former style has to unpolish its settings before the new style can polish them again.

See also polish() [p. 149].

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

void QStyle::unPolish (QApplication *) [virtual]

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

Undoes the application polish.

See also polish() [p. 149].

QRect QStyle::visualRect (const QRect & logical, const QWidget * w) [static]

Returns the rect *logical* in screen coordinates. The bounding rect for widget *w* is used to perform the translation. This function is provided to aid style implementors in supporting right-to-left mode.

See also QApplication::reverseLayout() [Additional Functionality with Qt].

QRect QStyle::visualRect (const QRect & logical, const QRect & bounding) [static]

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

Returns the rect *logical* in screen coordinates. The rect *bounding* is used to perform the translation. This function is provided to aid style implementors in supporting right-to-left mode.

See also QApplication::reverseLayout() [Additional Functionality with Qt].

QStyleSheet Class Reference

The QStyleSheet class is a collection of styles for rich text rendering and a generator of tags.

`#include <qstylesheet.h>`

Inherits QObject [Additional Functionality with Qt].

Public Members

- **QStyleSheet** (QObject * parent = 0, const char * name = 0)
- virtual **~QStyleSheet** ()
- QStyleSheetItem * **item** (const QString & name)
- const QStyleSheetItem * **item** (const QString & name) const
- virtual QTextCustomItem * **tag** (const QString & name, const QMap<QString, QString> & attr, const QString & context, const QMimeSourceFactory & factory, bool emptyTag, QTextDocument * doc) const
- virtual void **scaleFont** (QFont & font, int logicalSize) const
- virtual void **error** (const QString & msg) const

Static Public Members

- QStyleSheet * **defaultSheet** ()
- void **setDefaultSheet** (QStyleSheet * sheet)
- QString **escape** (const QString & plain)
- QString **convertFromPlainText** (const QString & plain, QStyleSheetItem::WhiteSpaceMode mode = QStyleSheetItem::WhiteSpacePre)
- bool **mightBeRichText** (const QString & text)

Detailed Description

The QStyleSheet class is 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. However, QLabel, QWhatsThis and QMessageBox also support it, and other classes are 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, sorted by structuring bindings, anchors, character style bindings (i.e. inline styles), special elements such as horizontal lines or images, and other tags. In addition, rich text supports simple HTML tables.

The structuring tags are

- <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, which can be used to explain certain expressions in more detail 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".
- <h1>...</h1> - A top-level heading.
- <h2>...</h2> - A sublevel heading.
- <h3>...</h3> - A sub-sublevel heading.
- ... - A left-aligned paragraph. Adjust the alignment with the align attribute. Possible values are left, right and center.
- ... - A centered paragraph.
- <blockquote>...</blockquote> - An indented paragraph that is useful for quotes.
- ... - An unordered list. You can also pass a type argument to define the bullet style. The default is type=disc; other types are circle and square.
- ... - 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".
- ... - A list item. This tag can be used only within the context of ol or ul.
- <pre>...</pre> - For larger junks of code. Whitespaces in the contents are preserved. For small bits of code use the inline-style code.

Anchors and links are done with a single tag:

- <a>... - An anchor or link. The reference target is defined in the href attribute of the tag as in .... You can also specify an additional anchor within the specified target document, for example .... If a is meant to be an anchor, the reference source is given in the name attribute.

The default character style bindings are

- ... - Emphasized. By default this is the same as ... (italic).
- ... - Strong. By default this is the same as ... (bold).
- ... - Italic font style.
- ... - Bold font style.
- <u>...</u> - Underlined font style.
- <big>...</big> - A larger font size.
- <small>...</small> - A smaller font size.
- <code>...</code> - Indicates code. By default this is the same as <tt>...</tt> (typewriter). For larger junks of code use the block-tag pre.
- <tt>...</tt> - Typewriter font style.
- ... - Customizes the font size, family and text color. The tag understands the following 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 (`size=-2`). In the latter case the sizes are simply added.
  - face - The family of the font, for example `face=times`.

Special elements are:

- `<img>` - An image. The image name for the mime source factory is given in the source attribute, for example `<img src="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 the specified size it will be scaled automatically (by using QImage::smoothScale()).

  The `align` attribute determines where the image is placed. By default, an image is placed inline just like a normal character. Specify `left` or `right` to place the image at the respective side.
- `<hr>` - A horizonal line.
- `<br>` - A line break.

Another tag not in any of the above cathegories is

- `<nobr>...</nobr>` - No break. Prevents word wrap.

In addition, rich text supports simple HTML tables. A table consists of one or more rows each of which contains one or more cells. Cells are either data cells or header cells, depending on their content. Cells which span rows and columns are supported.

- `<table>...</table>` - A table. Tables support the following attributes:
  - bgcolor - The background color.
  - width - The table width. This is either an absolute pixel width or a relative percentage of the table's width, for example `width=80%`.
  - border - The width of the table border. The default is 0 (= no border).
  - cellspacing - Additional space around the table cells. The default is 2.
  - cellpadding - Additional space around the contents of table cells. The default is 1.
- `<tr>...</tr>` - A table row. This is only valid within a `table`. Rows support the following attribute:
  - bgcolor - The background color.
- `<th>...</th>` - A table header cell. Similar to `td`, but defaults to center alignment and a bold font.
- `<td>...</td>` - A table data cell. This is only valid within a `tr`. Cells support the following attributes:
  - bgcolor - The background color.
  - width - The cell width. This is either an absolute pixel width or a relative percentage of table's width, for example `width=50%`.
  - colspan - Specifies how many columns this cell spans. The default is 1.
  - rowspan - Specifies how many rows this cell spans. The default is 1.
  - align - Alignment; possible values are `left`, `right`, and `center`. The default is left.

See also Graphics Classes, Help System and Text Related Classes.

Member Function Documentation

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

Creates a style sheet with parent *parent* and name *name*. Like any QObject it will be deleted when its parent is destroyed (if the child still exists).

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, QStyleSheetItem::WhiteSpaceMode mode = QStyleSheetItem::WhiteSpacePre) [static]

Auxiliary function. Converts the plain text string *plain* to a rich text formatted paragraph while preserving its look.

mode defines the whitespace mode. Possible values are QStyleSheetItem::WhiteSpacePre (no wrapping, all whitespaces preserved) and QStyleSheetItem::WhiteSpaceNormal (wrapping, simplified whitespaces).

See also escape() [p. 157].

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 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 QTextEdit and QTextBrowser.

See also setDefaultSheet() [p. 158].

void QStyleSheet::error (const QString & msg) const [virtual]

This virtual function is called when an error occurs when processing rich text. Reimplement it 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 incorrectly, or if tags are not closed properly.

msg is the error message.

QString QStyleSheet::escape (const QString & plain) [static]

Auxiliary function. Converts the plain text string *plain* to a rich text formatted string with any HTML meta-characters escaped.

See also convertFromPlainText() [p. 157].

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

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

Returns the style with name *name* or 0 if there is no such style (const version)

bool QStyleSheet::mightBeRichText (const QString & text) [static]

Returns TRUE if the string *text* is likely to be rich text; otherwise returns FALSE.

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. Although 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.

Logical font sizes range from 1 to 7, with 1 being the smallest.

See also QStyleSheetItem::logicalFontSize() [p. 164], QStyleSheetItem::logicalFontSizeStep() [p. 164] and QFont::setPointSize() [Additional Functionality with Qt].

void QStyleSheet::setDefaultSheet (QStyleSheet * sheet) [static]

Sets the application-wide default style sheet to *sheet*, deleting any style sheet previously set. The ownership is transferred to QStyleSheet.

See also defaultSheet() [p. 157].

QTextCustomItem * QStyleSheet::tag (const QString & name, const QMap<QString, QString> & attr, const QString & context, const QMimeSourceFactory & factory, bool emptyTag, QTextDocument * doc) const [virtual]

This function is under development and is subject to change.

Generates an internal object for the tag called *name*, given the attributes *attr*, and using additional information provided by the mime source factory *factory*.

context is the optional context of the document, i.e. the path to look for relative links. This becomes important if the text contains relative references, for example within image tags. QSimpleRichText always uses the default mime source factory (see QMimeSourceFactory::defaultFactory()) to resolve these references. The context will then be used to calculate the absolute path. See QMimeSourceFactory::makeAbsolute() for details.

emptyTag and *doc* are for internal use only.

This function should not (yet) be used in application code.

QStyleSheetItem Class Reference

The QStyleSheetItem class provides an encapsulation of a set of text styles.

`#include <qstylesheet.h>`

Inherits Qt [Additional Functionality with Qt].

Public Members

- **QStyleSheetItem** (QStyleSheet * parent, const QString & name)
- **QStyleSheetItem** (const QStyleSheetItem & other)
- **~QStyleSheetItem** ()
- QString **name** () const
- QStyleSheet * **styleSheet** ()
- const QStyleSheet * **styleSheet** () const
- enum **DisplayMode** { DisplayBlock, DisplayInline, DisplayListItem, DisplayNone }
- DisplayMode **displayMode** () const
- void **setDisplayMode** (DisplayMode m)
- int **alignment** () const
- void **setAlignment** (int f)
- enum **VerticalAlignment** { VAlignBaseline, VAlignSub, VAlignSuper }
- VerticalAlignment **verticalAlignment** () const
- void **setVerticalAlignment** (VerticalAlignment valign)
- int **fontWeight** () const
- void **setFontWeight** (int w)
- int **logicalFontSize** () const
- void **setLogicalFontSize** (int s)
- int **logicalFontSizeStep** () const
- void **setLogicalFontSizeStep** (int s)
- int **fontSize** () const
- void **setFontSize** (int s)
- QString **fontFamily** () const
- void **setFontFamily** (const QString & fam)
- int numberOfColumns () const *(obsolete)*
- void setNumberOfColumns (int ncols) *(obsolete)*
- QColor **color** () const
- void **setColor** (const QColor & c)
- bool **fontItalic** () const
- void **setFontItalic** (bool italic)
- bool **definesFontItalic** () const
- bool **fontUnderline** () const

- void **setFontUnderline** (bool underline)
- bool **definesFontUnderline** () const
- bool **isAnchor** () const
- void **setAnchor** (bool anc)
- enum **WhiteSpaceMode** { WhiteSpaceNormal, WhiteSpacePre, WhiteSpaceNoWrap }
- WhiteSpaceMode **whiteSpaceMode** () const
- void **setWhiteSpaceMode** (WhiteSpaceMode m)
- enum **Margin** { MarginLeft, MarginRight, MarginTop, MarginBottom, MarginFirstLine, MarginAll, MarginVertical, MarginHorizontal }
- int **margin** (Margin m) const
- void **setMargin** (Margin m, int v)
- enum **ListStyle** { ListDisc, ListCircle, ListSquare, ListDecimal, ListLowerAlpha, ListUpperAlpha }
- ListStyle **listStyle** () const
- void **setListStyle** (ListStyle s)
- QString **contexts** () const
- void **setContexts** (const QString & c)
- bool **allowedInContext** (const QStyleSheetItem * s) const
- bool **selfNesting** () const
- void **setSelfNesting** (bool nesting)
- void **setLineSpacing** (int ls)
- int **lineSpacing** () const

Detailed Description

The QStyleSheetItem class provides an encapsulation of a set of text styles.

A style sheet item consists of a name and a set of attributes that specifiy its font, color, etc. When used in a style sheet (see styleSheet()), items define the name() of a rich text tag and the display property changes associated with it.

The display mode attribute indicates whether the item is a block, an inline element or a list element; see setDisplayMode(). The treatment of whitespace is controlled by the white space mode; see setWhiteSpaceMode(). An item's margins are set with setMargin(), and line spacing is set with setLineSpacing(). In the case of list items, the list style is set with setListStyle(). An item may be a hypertext link anchor; see setAnchor(). Other attributes are set with setAlignment(), setVerticalAlignment(), setFontFamily(), setFontSize(), setFontWeight(), setFontItalic(), setFontUnderline() and setColor().

See also Text Related Classes.

Member Type Documentation

QStyleSheetItem::DisplayMode

This enum type defines the way adjacent elements are displayed. The possible values are:

- `QStyleSheetItem::DisplayBlock` - elements are displayed as a rectangular block (e.g. `<p>...</p>`).
- `QStyleSheetItem::DisplayInline` - elements are displayed in a horizontally flowing sequence (e.g. `<em>...</em>`).
- `QStyleSheetItem::DisplayListItem` - elements are displayed in a vertical sequence (e.g. `<li>...</li>`).
- `QStyleSheetItem::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:

- `QStyleSheetItem::ListDisc` - a filled circle (i.e. a bullet)
- `QStyleSheetItem::ListCircle` - an unfilled circle
- `QStyleSheetItem::ListSquare` - a filled square
- `QStyleSheetItem::ListDecimal` - an integer in base 10: *1*, *2*, *3*, ...
- `QStyleSheetItem::ListLowerAlpha` - a lowercase letter: *a*, *b*, *c*, ...
- `QStyleSheetItem::ListUpperAlpha` - an uppercase letter: *A*, *B*, *C*, ...

QStyleSheetItem::Margin

- `QStyleSheetItem::MarginLeft` - left margin
- `QStyleSheetItem::MarginRight` - right margin
- `QStyleSheetItem::MarginTop` - top margin
- `QStyleSheetItem::MarginBottom` - bottom margin
- `QStyleSheetItem::MarginAll` - all margins (left, right, top and bottom)
- `QStyleSheetItem::MarginVertical` - top and bottom margins
- `QStyleSheetItem::MarginHorizontal` - left and right margins
- `QStyleSheetItem::MarginFirstLine` - margin (indentation) of the first line of a paragarph (in addition to the MarginLeft of the paragraph)

QStyleSheetItem::VerticalAlignment

This enum type defines the way elements are aligned vertically. This is supported for text elements only. The possible values are:

- `QStyleSheetItem::VAlignBaseline` - align the baseline of the element (or the bottom, if the element doesn't have a baseline) with the baseline of the parent
- `QStyleSheetItem::VAlignSub` - subscript the element
- `QStyleSheetItem::VAlignSuper` - superscript the element

QStyleSheetItem::WhiteSpaceMode

This enum defines the ways in which QStyleSheet can treat whitespace. There are three values at present:

- `QStyleSheetItem::WhiteSpaceNormal` - any sequence of whitespace (including line-breaks) is equivalent to a single space.
- `QStyleSheetItem::WhiteSpacePre` - whitespace must be output exactly as given in the input.
- `QStyleSheetItem::WhiteSpaceNoWrap` - multiple spaces are collapsed as with WhiteSpaceNormal, but no automatic line-breaks occur. To break lines manually, use the `<br>` tag.

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 (const QStyleSheetItem & other)

Copy constructor. Constructs a copy of *other* that is not bound to any style sheet.

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 AlignAuto, AlignLeft, AlignRight, AlignCenter and AlignJustify.

See also setAlignment() [p. 165] and Qt::AlignmentFlags [Additional Functionality with Qt].

bool QStyleSheetItem::allowedInContext (const QStyleSheetItem * s) const

Returns TRUE if this style can be nested into an element of style *s*; otherwise returns FALSE.

See also contexts() [p. 162] and setContexts() [p. 165].

QColor QStyleSheetItem::color () const

Returns the text color of this style or an invalid color if no color has been set.

See also setColor() [p. 165] and QColor::isValid() [Graphics with Qt].

QString QStyleSheetItem::contexts () const

Returns a space-separated list of names of styles that may contain elements of this style. If nothing has been set, contexts() returns an empty string, which indicates that this style can be nested everywhere.

See also setContexts() [p. 165].

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() [p. 165] and fontItalic() [p. 163].

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() [p. 166] and fontUnderline() [p. 163].

DisplayMode QStyleSheetItem::displayMode () const

Returns the display mode of the style.

See also setDisplayMode() [p. 165].

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() [p. 165], QFont::family() [Additional Functionality with Qt] and QFont::setFamily() [Additional Functionality with Qt].

bool QStyleSheetItem::fontItalic () const

Returns TRUE if the style sets an italic font; otherwise returns FALSE.

See also setFontItalic() [p. 165] and definesFontItalic() [p. 162].

int QStyleSheetItem::fontSize () const

Returns the font size setting of the style. This is either a valid point size or QStyleSheetItem::Undefined.

See also setFontSize() [p. 165], QFont::pointSize() [Additional Functionality with Qt] and QFont::setPointSize() [Additional Functionality with Qt].

bool QStyleSheetItem::fontUnderline () const

Returns TRUE if the style sets an underlined font; otherwise returns FALSE.

See also setFontUnderline() [p. 166] and definesFontUnderline() [p. 163].

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() [p. 166] and QFont [Additional Functionality with Qt].

bool QStyleSheetItem::isAnchor () const

Returns whether this style is an anchor.

See also setAnchor() [p. 165].

int QStyleSheetItem::lineSpacing () const

Returns the linespacing

ListStyle QStyleSheetItem::listStyle () const

Returns the list style of the style.

See also setListStyle() [p. 166] and ListStyle [p. 161].

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() [p. 166], setLogicalFontSizeStep() [p. 166], QFont::pointSize() [Additional Functionality with Qt] and QFont::setPointSize() [Additional Functionality with Qt].

int QStyleSheetItem::logicalFontSizeStep () const

Returns the logical font size step of this style.

The default is 0. Tags such as `big` define `+1`; `small` defines `-1`.

See also setLogicalFontSizeStep() [p. 166].

int QStyleSheetItem::margin (Margin m) const

Returns the width of margin *m* in pixels.

The margin, *m*, can be MarginLeft, MarginRight, MarginTop, MarginBottom, MarginAll, MarginVertical or MarginHorizontal.

See also setMargin() [p. 166] and Margin [p. 161].

QString QStyleSheetItem::name () const

Returns the name of the style item.

int QStyleSheetItem::numberOfColumns () const

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

Returns the number of columns for this style.

See also setNumberOfColumns() [p. 166], displayMode() [p. 163] and setDisplayMode() [p. 165].

bool QStyleSheetItem::selfNesting () const

Returns TRUE if this style has self-nesting enabled; otherwise returns FALSE.

See also setSelfNesting() [p. 167].

void QStyleSheetItem::setAlignment (int f)

Sets the alignment to *f*. This only makes sense for styles with a display mode of DisplayBlock. Possible values are AlignAuto, AlignLeft, AlignRight, AlignCenter and AlignJustify.

See also alignment() [p. 162], displayMode() [p. 163] and Qt::AlignmentFlags [Additional Functionality with Qt].

void QStyleSheetItem::setAnchor (bool anc)

If *anc* is TRUE sets this style to be an anchor (hypertext link); otherwise sets it to not be an anchor. Elements in this style have connections to other documents or anchors.

See also isAnchor() [p. 163].

void QStyleSheetItem::setColor (const QColor & c)

Sets the text color of this style to *c*.

See also color() [p. 162].

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() [p. 162].

void QStyleSheetItem::setDisplayMode (DisplayMode m)

Sets the display mode of the style to *m*.

See also displayMode() [p. 163].

void QStyleSheetItem::setFontFamily (const QString & fam)

Sets the font family setting of the style to *fam*.

See also fontFamily() [p. 163], QFont::family() [Additional Functionality with Qt] and QFont::setFamily() [Additional Functionality with Qt].

void QStyleSheetItem::setFontItalic (bool italic)

If *italic* is TRUE sets italic for the style; otherwise sets upright.

See also fontItalic() [p. 163] and definesFontItalic() [p. 162].

void QStyleSheetItem::setFontSize (int s)

Sets the font size setting of the style to *s* points.

See also fontSize() [p. 163], QFont::pointSize() [Additional Functionality with Qt] and QFont::setPointSize() [Additional Functionality with Qt].

void QStyleSheetItem::setFontUnderline (bool underline)

If *underline* is TRUE sets underline for the style; otherwise sets no underline.

See also fontUnderline() [p. 163] and definesFontUnderline() [p. 163].

void QStyleSheetItem::setFontWeight (int w)

Sets the font weight setting of the style to *w*. Valid values are those defined by QFont::Weight.

See also QFont [Additional Functionality with Qt] and fontWeight() [p. 163].

void QStyleSheetItem::setLineSpacing (int ls)

Sets the linespacing to be *ls* pixels

void QStyleSheetItem::setListStyle (ListStyle s)

Sets the list style of the style to *s*.

This is used by nested elements that have a display mode of DisplayListItem.

See also listStyle() [p. 164], DisplayMode [p. 160] and ListStyle [p. 161].

void QStyleSheetItem::setLogicalFontSize (int s)

Sets the logical font size setting of the style to *s*. Valid logical sizes are 1 to 7.

See also logicalFontSize() [p. 164], QFont::pointSize() [Additional Functionality with Qt] and QFont::setPointSize() [Additional Functionality with Qt].

void QStyleSheetItem::setLogicalFontSizeStep (int s)

Sets the logical font size step of this style to *s*.

See also logicalFontSizeStep() [p. 164].

void QStyleSheetItem::setMargin (Margin m, int v)

Sets the width of margin *m* to *v* pixels.

The margin, *m*, can be MarginLeft, MarginRight, MarginTop, MarginBottom, MarginAll, MarginVertical or MarginHorizontal. The value *v* must be >= 0.

See also margin() [p. 164].

void QStyleSheetItem::setNumberOfColumns (int ncols)

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

Sets the number of columns for this style. Elements in the style are divided into columns.

This makes sense only if the style uses a block display mode (see QStyleSheetItem::DisplayMode).

See also numberOfColumns() [p. 164].

void QStyleSheetItem::setSelfNesting (bool nesting)

Sets the self-nesting property for this style to *nesting*.

In order to support "dirty" HTML, paragraphs `<p>` and list items `<li>` are not self-nesting. This means that starting a new paragraph or list item automatically closes the previous one.

See also selfNesting() [p. 164].

void QStyleSheetItem::setVerticalAlignment (VerticalAlignment valign)

Sets the vertical alignment to *valign*. Possible values are VAlignBaseline, VAlignSub and VAlignSuper.

The vertical alignment property is not inherited.

See also verticalAlignment() [p. 167].

void QStyleSheetItem::setWhiteSpaceMode (WhiteSpaceMode m)

Sets the whitespace mode to *m*.

See also WhiteSpaceMode [p. 161].

QStyleSheet * QStyleSheetItem::styleSheet ()

Returns the style sheet this item is in.

const QStyleSheet * QStyleSheetItem::styleSheet () const

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

Returns the style sheet this item is in.

VerticalAlignment QStyleSheetItem::verticalAlignment () const

Returns the vertical alignment of the style. Possible values are VAlignBaseline, VAlignSub and VAlignSuper.

psa setVerticalAlignment()

WhiteSpaceMode QStyleSheetItem::whiteSpaceMode () const

Returns the whitespace mode.

See also setWhiteSpaceMode() [p. 167] and WhiteSpaceMode [p. 161].

QTextDrag Class Reference

The QTextDrag class is a drag and drop object for transferring plain and Unicode text.

`#include <qdragobject.h>`

Inherits QDragObject [p. 59].

Public Members

- **QTextDrag** (const QString & text, QWidget * dragSource = 0, const char * name = 0)
- **QTextDrag** (QWidget * dragSource = 0, const char * name = 0)
- **~QTextDrag** ()
- virtual void **setText** (const QString & text)
- virtual void **setSubtype** (const QCString & st)

Static Public Members

- bool **canDecode** (const QMimeSource * e)
- bool **decode** (const QMimeSource * e, QString & str)
- bool **decode** (const QMimeSource * e, QString & str, QCString & subtype)

Detailed Description

The QTextDrag class is a drag and drop object for transferring plain and Unicode text.

Plain text is passed in a QString which may contain multiple lines (i.e. may contain newline characters).

Qt provides no built-in mechanism for delivering only single-line.

For more information about drag and drop, see the QDragObject class and the drag and drop documentation.

See also Drag And Drop Classes.

Member Function Documentation

QTextDrag::QTextDrag (const QString & text, QWidget * dragSource = 0, const char * name = 0)

Constructs a text drag object and sets it to *text*. *dragSource* must be the drag source; *name* is the object name.

QTextDrag::QTextDrag (QWidget * dragSource = 0, const char * name = 0)

Constructs a default text drag object. *dragSource* must be the drag source; *name* is the object name.

QTextDrag::~QTextDrag ()

Destroys the text drag object and frees up all allocated resources.

bool QTextDrag::canDecode (const QMimeSource * e) [static]

Returns TRUE if the information in *e* can be decoded into a QString; otherwise returns FALSE.

See also decode() [p. 169].

bool QTextDrag::decode (const QMimeSource * e, QString & str) [static]

Attempts to decode the dropped information in *e* into *str*. Returns TRUE if successful; otherwise returns FALSE.

See also canDecode() [p. 169].

bool QTextDrag::decode (const QMimeSource * e, QString & str, QCString & subtype) [static]

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

Attempts to decode the dropped information in *e* into *str*. Returns TRUE if successful; otherwise returns FALSE. If *subtype* is null, any text subtype is accepted; otherwise only the specified *subtype* is accepted.

See also canDecode() [p. 169].

void QTextDrag::setSubtype (const QCString & st) [virtual]

Sets the MIME subtype of the text being dragged to *st*. The default subtype is "plain", so 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 to *text*. You will need to call this if you did not pass the text during construction.

QTimerEvent Class Reference

The QTimerEvent class contains parameters that describe a timer event.

`#include <qevent.h>`

Inherits QEvent [p. 67].

Public Members

- **QTimerEvent** (int timerId)
- int **timerId** () const

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. A timer is started with QObject::startTimer().

The QTimer class provides a high-level programming interface that uses signals instead of events. It also provides one-shot timers.

The event handler QObject::timerEvent() receives timer events.

See also QTimer [Additional Functionality with Qt], QObject::timerEvent() [Additional Functionality with Qt], QObject::startTimer() [Additional Functionality with Qt], QObject::killTimer() [Additional Functionality with Qt], QObject::killTimers() [Additional Functionality with Qt] and Event Classes.

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().

Example: dclock/dclock.cpp.

QUriDrag Class Reference

The QUriDrag class provides a drag object for a list of URI references.

`#include <qdragobject.h>`

Inherits QStoredDrag [p. 132].

Public Members

- **QUriDrag** (QStrList uris, QWidget * dragSource = 0, const char * name = 0)
- **QUriDrag** (QWidget * dragSource = 0, const char * name = 0)
- **~QUriDrag** ()
- void setFilenames (const QStringList & fnames) *(obsolete)*
- void **setFileNames** (const QStringList & fnames)
- void **setUnicodeUris** (const QStringList & uuris)
- virtual void **setUris** (QStrList uris)

Static Public Members

- QString **uriToLocalFile** (const char * uri)
- QCString **localFileToUri** (const QString & filename)
- QString **uriToUnicodeUri** (const char * uri)
- QCString **unicodeUriToUri** (const QString & uuri)
- bool **canDecode** (const QMimeSource * e)
- bool **decode** (const QMimeSource * e, QStrList & l)
- bool **decodeToUnicodeUris** (const QMimeSource * e, QStringList & l)
- bool **decodeLocalFiles** (const QMimeSource * e, QStringList & l)

Detailed Description

The QUriDrag class provides a drag object for a list of URI references.

URIs are a useful way to refer to files that may be distributed across multiple machines. A URI will often refer to a file on a machine local to both the drag source and the drop target, so the URI will be equivalent to passing a file name but will be more extensible.

Use URIs in Unicode form so that the user can comfortably edit and view them. For use in HTTP or other protocols, use the correctly escaped ASCII form.

You can convert a list of file names to file URIs using setFileNames(), or into human-readble for with setUnicodeUris().

Static functions are provided to convert between filenames and URIs, e.g. uriToLocalFile() and localFileToUri(), and to and from human-readable form, e.g. uriToUnicodeUri(), unicodeUriToUri(). You can also decode URIs from a mimesource into a list with decodeLocalFiles() and decodeToUnicodeUris().

See also Drag And Drop Classes.

Member Function Documentation

QUriDrag::QUriDrag (QStrList uris, QWidget * dragSource = 0, const char * name = 0)

Constructs 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)

Constructs a object to drag. You will need to call setUris() before you start the drag(). Passes *dragSource* and *name* to the QStoredDrag constructor.

QUriDrag::~QUriDrag ()

Destroys the object.

bool QUriDrag::canDecode (const QMimeSource * e) [static]

Returns TRUE if decode() would be able to decode *e*; otherwise returns FALSE.

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; otherwise returns FALSE.

Examples: dirview/dirview.cpp and fileiconview/qfileiconview.cpp.

bool QUriDrag::decodeLocalFiles (const QMimeSource * e, QStringList & l) [static]

Decodes URIs from the mime source event *e*, converts them to local files if they refer to local files, and places them in *l* (which is first cleared).

Returns TRUE if *contained* a valid list of URIs; otherwise returns FALSE. The list will be empty if no URIs were local files.

bool QUriDrag::decodeToUnicodeUris (const QMimeSource * e, QStringList & l) [static]

Decodes URIs from the mime source event *e*, converts them to Unicode URIs (only useful for displaying to humans), placing them in *l* (which is first cleared).

Returns TRUE if *contained* a valid list of URIs; otherwise returns FALSE.

QCString QUriDrag::localFileToUri (const QString & filename) [static]

Returns the URI equivalent to the absolute local file *filename*.

See also uriToLocalFile() [p. 173].

void QUriDrag::setFileNames (const QStringList & fnames)

Sets the URIs to be the local-file URIs equivalent to *fnames*.

See also localFileToUri() [p. 173] and setUris() [p. 173].

void QUriDrag::setFilenames (const QStringList & fnames)

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

Use setFileNames() instead (notice the N).

void QUriDrag::setUnicodeUris (const QStringList & uuris)

Sets the URIs to be the Unicode URIs (only useful for displaying to humans) *uuris*.

See also localFileToUri() [p. 173] and setUris() [p. 173].

Example: dirview/dirview.cpp.

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() [p. 173].

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() [p. 173].

QString QUriDrag::uriToUnicodeUri (const char * uri) [static]

Returns the Unicode URI (only useful for displaying to humans) equivalent to *uri*.

See also localFileToUri() [p. 173].

QVBoxLayout Class Reference

The QVBoxLayout class lines up widgets vertically.

`#include <qlayout.h>`

Inherits QBoxLayout [p. 34].

Public Members

- **QVBoxLayout** (QWidget * parent, int margin = 0, int spacing = -1, const char * name = 0)
- **QVBoxLayout** (QLayout * parentLayout, int spacing = -1, const char * name = 0)
- **QVBoxLayout** (int spacing = -1, const char * name = 0)
- **~QVBoxLayout** ()

Detailed Description

The QVBoxLayout class lines up widgets vertically.

This class is used to construct vertical box layout objects. See QBoxLayout for more details.

The simplest way to use this class is like this:

```
QBoxLayout * l = new QVBoxLayout( widget );
l->addWidget( aWidget );
l->addWidget( anotherWidget );
```

See also QHBoxLayout [p. 86], QGridLayout [p. 77], the Layout overview [Programming with Qt], Widget Appearance and Style and Layout Management.

Member Function Documentation

QVBoxLayout::QVBoxLayout (QWidget * parent, int margin = 0, int spacing = -1, const char * name = 0)

Constructs a new top-level vertical box with parent *parent* and name *name*.

The *margin* is the number of pixels between the edge of the widget and its managed children. The *spacing* is the default number of pixels between neighboring children. If *spacing* is -1 the value of *margin* is used for *spacing*.

QVBoxLayout::QVBoxLayout (QLayout * parentLayout, int spacing = -1, const char * name = 0)

Constructs a new vertical box with the name *name* and adds it to *parentLayout*.

The *spacing* is the default number of pixels between neighboring children. If *spacing* is -1, this QVBoxLayout will inherit its parent's spacing().

QVBoxLayout::QVBoxLayout (int spacing = -1, const char * name = 0)

Constructs a new vertical box with the name *name*. You must add it to another layout.

The *spacing* is the default number of pixels between neighboring children. If *spacing* is -1, this QVBoxLayout will inherit its parent's spacing().

QVBoxLayout::~QVBoxLayout ()

Destroys this box layout.

QWheelEvent Class Reference

The QWheelEvent class contains parameters that describe a wheel event.

`#include <qevent.h>`

Inherits QEvent [p. 67].

Public Members

- **QWheelEvent** (const QPoint & pos, int delta, int state, Orientation orient = Vertical)
- **QWheelEvent** (const QPoint & pos, const QPoint & globalPos, int delta, int state, Orientation orient = Vertical)
- int **delta** () const
- const QPoint & **pos** () const
- const QPoint & **globalPos** () const
- int **x** () const
- int **y** () const
- int **globalX** () const
- int **globalY** () const
- ButtonState **state** () const
- Orientation **orientation** () const
- bool **isAccepted** () const
- void **accept** ()
- void **ignore** ()

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(). The functions pos() and globalPos() return the mouse pointer location at the time of the event.

A wheel event contains a special accept flag that indicates 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.

The QWidget::setEnable() function can be used to enable or disable mouse and keyboard events for a widget.

The event handler QWidget::wheelEvent() receives wheel events.

See also QMouseEvent [p. 118], QWidget::grabMouse() [Widgets with Qt] and Event Classes.

Member Function Documentation

QWheelEvent::QWheelEvent (const QPoint & pos, int delta, int state, Orientation orient = Vertical)

Constructs a wheel event object.

The globalPos() is initialized to QCursor::pos(), i.e. *pos*, which is usually (but not always) right. Use the other constructor if you need to specify the global position explicitly. *delta* contains the rotation distance, *state* holds the keyboard modifier flags at the time of the event and *orient* holds the wheel's orientation.

See also pos() [p. 178], delta() [p. 177] and state() [p. 178].

QWheelEvent::QWheelEvent (const QPoint & pos, const QPoint & globalPos, int delta, int state, Orientation orient = Vertical)

Constructs a wheel event object. The position when the event occurred is given in *pos* and *globalPos*. *delta* contains the rotation distance, *state* holds the keyboard modifier flags at the time of the event and *orient* holds the wheel's orientation.

See also pos() [p. 178], globalPos() [p. 177], delta() [p. 177] and state() [p. 178].

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() [p. 178].

int QWheelEvent::delta () const

Returns the distance that the wheel is rotated expressed in multiples or divisions of WHEEL_DELTA, which is currently set at 120. A positive value indicates that the wheel was rotated forwards away from the user; a negative value indicates that the wheel was rotated backwards 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 such as X11; whenever you move your widgets around in response to mouse events, globalPos() can differ a lot from the current pointer position QCursor::pos().

See also globalX() [p. 177] and globalY() [p. 178].

int QWheelEvent::globalX () const

Returns the global X position of the mouse pointer at the time of the event.

See also globalY() [p. 178] and globalPos() [p. 177].

int QWheelEvent::globalY () const

Returns the global Y position of the mouse pointer at the time of the event.

See also globalX() [p. 177] and globalPos() [p. 177].

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() [p. 177].

bool QWheelEvent::isAccepted () const

Returns TRUE if the receiver of the event handles the wheel event; otherwise returns FALSE.

Orientation QWheelEvent::orientation () const

Returns the wheel's orientation.

const QPoint & QWheelEvent::pos () const

Returns the position of the mouse pointer, relative to the widget that received the event.

If you move your widgets around in response to mouse events, use globalPos() instead of this function.

See also x() [p. 178], y() [p. 178] and globalPos() [p. 177].

ButtonState QWheelEvent::state () const

Returns the keyboard modifier flags of the event.

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() [p. 178] and pos() [p. 178].

int QWheelEvent::y () const

Returns the Y position of the mouse pointer, relative to the widget that received the event.

See also x() [p. 178] and pos() [p. 178].

QWindowsStyle Class Reference

The QWindowsStyle class provides a Microsoft Windows-like look and feel.

`#include <qwindowsstyle.h>`

Inherits QCommonStyle [p. 48].

Inherited by QAquaStyle [p. 33] and QPlatinumStyle [p. 127].

Public Members

- **QWindowsStyle** ()

Detailed Description

The QWindowsStyle class provides a Microsoft Windows-like look and feel.

This style is Qt's default GUI style on Windows.

See also Widget Appearance and Style.

Member Function Documentation

QWindowsStyle::QWindowsStyle ()

Constructs a QWindowsStyle

Index