Widgets 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

QButton Class Reference

The QButton class is the abstract base class of button widgets, providing functionality common to buttons.

`#include <qbutton.h>`

Inherits QWidget [p. 412].

Inherited by QCheckBox [p. 19], QPushButton [p. 230], QRadioButton [p. 240] and QToolButton [Dialogs and Windows with Qt].

Public Members

- **QButton** (QWidget * parent = 0, const char * name = 0, WFlags f = 0)
- **~QButton** ()
- QString **text** () const
- virtual void **setText** (const QString &)
- const QPixmap * **pixmap** () const
- virtual void **setPixmap** (const QPixmap &)
- QKeySequence **accel** () const
- virtual void **setAccel** (const QKeySequence &)
- bool **isToggleButton** () const
- enum **ToggleType** { SingleShot, Toggle, Tristate }
- ToggleType **toggleType** () const
- virtual void **setDown** (bool)
- bool **isDown** () const
- bool **isOn** () const
- enum **ToggleState** { Off, NoChange, On }
- ToggleState **state** () const
- bool autoResize () const *(obsolete)*
- void setAutoResize (bool) *(obsolete)*
- bool **autoRepeat** () const
- virtual void **setAutoRepeat** (bool)
- bool **isExclusiveToggle** () const
- QButtonGroup * **group** () const

Public Slots

- void **animateClick** ()
- void **toggle** ()

Signals

- void **pressed** ()
- void **released** ()
- void **clicked** ()
- void **toggled** (bool on)
- void **stateChanged** (int state)

Properties

- QKeySequence **accel** — the accelerator associated with the button
- bool **autoRepeat** — whether autoRepeat is enabled
- bool autoResize — whether autoResize is enabled *(obsolete)*
- bool **down** — whether the button is pressed
- bool **exclusiveToggle** — whether the button is an exclusive toggle *(read only)*
- bool **on** — whether the button is toggled *(read only)*
- QPixmap **pixmap** — the pixmap shown on the button
- QString **text** — the text shown on the button
- bool **toggleButton** — whether the button is a toggle button *(read only)*
- ToggleState **toggleState** — whether the button is toggled *(read only)*
- ToggleType **toggleType** — the type of toggle on the button *(read only)*

Protected Members

- void **setToggleButton** (bool b)
- virtual void **setToggleType** (ToggleType type)
- void **setOn** (bool on)
- virtual void **setState** (ToggleState s)
- virtual bool **hitButton** (const QPoint & pos) const
- virtual void **drawButton** (QPainter *)
- virtual void **drawButtonLabel** (QPainter *)
- virtual void **paintEvent** (QPaintEvent *)

Detailed Description

The QButton class is the abstract base class of button widgets, providing functionality common to buttons.

If you want to create a button use QPushButton.

The QButton class implements an *abstract* button, and lets subclasses specify how to reply to user actions and how to draw the button.

QButton provides both push and toggle buttons. The QRadioButton and QCheckBox classes provide only toggle buttons; QPushButton and QToolButton provide both toggle and push buttons.

Any button can have either a text or pixmap label. setText() sets the button to be a text button and setPixmap() sets it to be a pixmap button. The text/pixmap is manipulated as necessary to create the "disabled" appearance when the button is disabled.

QButton provides most of the states used for buttons:

- isDown() determines whether the button is *pressed* down.
- isOn() determines whether the button is *on*. Only toggle buttons can be switched on and off (see below).
- isEnabled() determines whether the button can be pressed by the user.
- setAutoRepeat() determines whether the button will auto-repeat if the user holds it down.
- setToggleButton() determines whether the button is a toggle button or not.

The difference between isDown() and isOn() is as follows: When the user clicks a toggle button to toggle it on, the button is first *pressed* and then released into *on* state. When the user clicks it again (to toggle it off), the button moves first to the *pressed* state, then to the *off* state (isOn() and isDown() are both FALSE).

Default buttons (as used in many dialogs) are provided by QPushButton::setDefault() and QPushButton::setAutoDefault().

QButton provides five signals:

1. pressed() is emitted when the left mouse button is pressed while the mouse cursor is inside the button.
2. released() is emitted when the left mouse button is released.
3. clicked() is emitted when the button is first pressed and then released when the accelerator key is typed, or when animateClick() is called.
4. toggled(bool) is emitted when the state of a toggle button changes.
5. stateChanged(int) is emitted when the state of a tristate toggle button changes.

If the button is a text button with "&" in its text, QButton creates an automatic accelerator key. This code creates a push button labelled "Rock & Roll" (where the c is underscored). The button gets an automatic accelerator key, Alt+C:

```
QPushButton *p = new QPushButton( "Ro&ck && Roll", this );
```

In this example, when the user presses Alt+C the button will call animateClick().

You can also set a custom accelerator using the setAccel() function. This is useful mostly for pixmap buttons because they have no automatic accelerator.

```
QPushButton *p;
p->setPixmap( QPixmap("print.png") );
p->setAccel( ALT+Key_F7 );
```

All of the buttons provided by Qt (QPushButton, QToolButton, QCheckBox and QRadioButton) can display both text and pixmaps.

To subclass QButton, you have to reimplement at least drawButton() (to draw the button's outline) and drawButtonLabel() (to draw its text or pixmap). It is generally advisable to reimplement sizeHint() as well, and sometimes hitButton() (to determine whether a button press is within the button).

To reduce flickering, QButton::paintEvent() sets up a pixmap that the drawButton() function draws in. You should not reimplement paintEvent() for a subclass of QButton unless you want to take over all drawing.

See also QButtonGroup [p. 14] and Abstract Widget Classes.

Member Type Documentation

QButton::ToggleState

This enum defines the state of a toggle button at any moment. The possible values are as follows:

- `QButton::Off` - the button is in the "off" state
- `QButton::NoChange` - the button is in the default/unchanged state
- `QButton::On` - the button is in the "on" state

QButton::ToggleType

This enum type defines what a button can do in response to a mouse/keyboard press:

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

Member Function Documentation

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

Constructs a standard button with the parent *parent* and the name *name* using the widget flags *f*.

If *parent* is a QButtonGroup, this constructor calls QButtonGroup::insert().

QButton::~QButton ()

Destroys the button.

QKeySequence QButton::accel () const

Returns the accelerator associated with the button. See the "accel" [p. 11] property for details.

void QButton::animateClick () [slot]

Performs an animated click: the button is pressed and released a short while later.

The pressed(), released(), clicked(), toggled(), and stateChanged() signals are emitted as appropriate.

This function does nothing if the button is disabled.

See also accel [p. 11].

bool QButton::autoRepeat () const

Returns TRUE if autoRepeat is enabled; otherwise returns FALSE. See the "autoRepeat" [p. 11] property for details.

bool QButton::autoResize () const

Returns TRUE if autoResize is enabled; otherwise returns FALSE. See the "autoResize" [p. 11] property for details.

void QButton::clicked () [signal]

This signal is emitted when the button is activated (i.e. first pressed down and then released when the mouse cursor is inside the button), when the accelerator key is typed or when animateClick() is called.

The QButtonGroup::clicked() signal does the same job, if you want to connect several buttons to the same slot.

See also pressed() [p. 9], released() [p. 9] and toggled() [p. 11].

Examples: fonts/simple-qfont-demo/viewer.cpp, listbox/listbox.cpp, network/clientserver/client/client.cpp, network/ftpclient/ftpmainwindow.cpp, richtext/richtext.cpp, t2/main.cpp and t4/main.cpp.

void QButton::drawButton (QPainter *) [virtual protected]

Draws the button. The default implementation does nothing.

This virtual function is reimplemented by subclasses to draw real buttons. At some point in time, these reimplementations are supposed to call drawButtonLabel().

See also drawButtonLabel() [p. 8] and paintEvent() [p. 9].

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

Draws the button text or pixmap.

This virtual function is reimplemented by subclasses to draw real buttons. It's invoked by drawButton().

See also drawButton() [p. 8] and paintEvent() [p. 9].

Example: tictac/tictac.cpp.

QButtonGroup * QButton::group () const

Returns a pointer to the group of which this button is a member.

If the button is not a member of any QButtonGroup, this function returns 0.

See also QButtonGroup [p. 14].

bool QButton::hitButton (const QPoint & pos) const [virtual protected]

Returns TRUE if *pos* is inside the clickable button rectangle, or FALSE if it is outside.

By default, the clickable area is the entire widget. Subclasses may reimplement it, though.

bool QButton::isDown () const

Returns TRUE if the button is pressed; otherwise returns FALSE. See the "down" [p. 11] property for details.

bool QButton::isExclusiveToggle () const

Returns TRUE if the button is an exclusive toggle; otherwise returns FALSE. See the "exclusiveToggle" [p. 12] property for details.

bool QButton::isOn () const

Returns TRUE if the button is toggled; otherwise returns FALSE. See the "on" [p. 12] property for details.

bool QButton::isToggleButton () const

Returns TRUE if the button is a toggle button; otherwise returns FALSE. See the "toggleButton" [p. 12] property for details.

void QButton::paintEvent (QPaintEvent *) [virtual protected]

Handles paint events for buttons. Small and typically complex buttons are painted double-buffered to reduce flicker. The actually drawing is done in the virtual functions drawButton() and drawButtonLabel().

See also drawButton() [p. 8] and drawButtonLabel() [p. 8].

Reimplemented from QWidget [p. 441].

const QPixmap * QButton::pixmap () const

Returns the pixmap shown on the button. See the "pixmap" [p. 12] property for details.

void QButton::pressed () [signal]

This signal is emitted when the button is pressed down.

See also released() [p. 9] and clicked() [p. 8].

Examples: network/httpd/httpd.cpp and popup/popup.cpp.

void QButton::released () [signal]

This signal is emitted when the button is released.

See also pressed() [p. 9], clicked() [p. 8] and toggled() [p. 11].

void QButton::setAccel (const QKeySequence &) [virtual]

Sets the accelerator associated with the button. See the "accel" [p. 11] property for details.

void QButton::setAutoRepeat (bool) [virtual]

Sets whether autoRepeat is enabled. See the "autoRepeat" [p. 11] property for details.

void QButton::setAutoResize (bool)

Sets whether autoResize is enabled. See the "autoResize" [p. 11] property for details.

void QButton::setDown (bool) [virtual]

Sets whether the button is pressed. See the "down" [p. 11] property for details.

void QButton::setOn (bool on) [protected]

Sets the state of this button to On when *on* is TRUE, otherwise to Off.

See also toggleState [p. 12].

void QButton::setPixmap (const QPixmap &) [virtual]

Sets the pixmap shown on the button. See the "pixmap" [p. 12] property for details.

void QButton::setState (ToggleState s) [virtual protected]

Sets the toggle state of the button to *s*. *s* can be Off, NoChange or On.

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

Sets the text shown on the button. See the "text" [p. 12] property for details.

void QButton::setToggleButton (bool b) [protected]

Makes this button a toggle button when *b* is TRUE, otherwise it becomes a normal button.

See also toggleButton [p. 12].

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

Sets the toggle type of the button to *type*.

type can be set to SingleShot, Toggle and `TriState`.

ToggleState QButton::state () const

Returns TRUE if the button is toggled; otherwise returns FALSE. See the "toggleState" [p. 12] property for details.

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

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

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

See also clicked() [p. 8].

QString QButton::text () const

Returns the text shown on the button. See the "text" [p. 12] property for details.

void QButton::toggle () [slot]

Toggles the state of a toggle button.

See also on [p. 12], setOn() [p. 10], toggled() [p. 11] and toggleButton [p. 12].

ToggleType QButton::toggleType () const

Returns the type of toggle on the button. See the "toggleType" [p. 13] property for details.

void QButton::toggled (bool on) [signal]

This signal is emitted whenever a toggle button changes status. *on* is TRUE if the button is on, or FALSE if the button is off.

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

See also clicked() [p. 8].

Example: listbox/listbox.cpp.

Property Documentation

QKeySequence accel

This property holds the accelerator associated with the button.

This property is 0 if there is no accelerator set. If you set this property to 0 then any current accelerator is removed.

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

bool autoRepeat

This property holds whether autoRepeat is enabled.

If autoRepeat is enabled then the clicked() signal is emitted at regular intervals if the button is down. This property has no effect on toggle buttons. autoRepeat is off by default.

Set this property's value with setAutoRepeat() and get this property's value with autoRepeat().

bool autoResize

This property holds whether autoResize is enabled.

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

If autoResize is enabled then the button will resize itself whenever the contents are changed.

Set this property's value with setAutoResize() and get this property's value with autoResize().

bool down

This property holds whether the button is pressed.

If this property is TRUE, the button is set to be pressed down. The signals pressed() and clicked() are not emitted if you set this property to TRUE. The default is FALSE.

Set this property's value with setDown() and get this property's value with isDown().

bool exclusiveToggle

This property holds whether the button is an exclusive toggle.

If this property is TRUE and the button is in a QButtonGroup, the button can only be toggled off by another one being toggled on. The default is FALSE.

Get this property's value with isExclusiveToggle().

bool on

This property holds whether the button is toggled.

This property should only be set for toggle buttons.

Get this property's value with isOn().

QPixmap pixmap

This property holds the pixmap shown on the button.

If the pixmap is monochrome (i.e., it is a QBitmap or its depth is 1) and it does not have a mask, this property will set the pixmap to be its own mask. The purpose of this is to draw transparent bitmaps which are important for toggle buttons, for example.

pixmap() returns 0 if no pixmap was set.

Set this property's value with setPixmap() and get this property's value with pixmap().

QString text

This property holds the text shown on the button.

This property will return a null string if the button has no text. If the text has an ampersand ('&') in it, then an accelerator is automatically created for it using the character after the '&' as the accelerator key.

There is no default text.

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

bool toggleButton

This property holds whether the button is a toggle button.

The default value is FALSE.

Get this property's value with isToggleButton().

ToggleState toggleState

This property holds whether the button is toggled.

If this is property is changed then it does not cause the button to be repainted.

Get this property's value with state().

ToggleType toggleType

This property holds the type of toggle on the button.

The default toggle type is SingleShot.

Get this property's value with toggleType().

QButtonGroup Class Reference

The QButtonGroup widget organizes QButton widgets in a group.

`#include <qbuttongroup.h>`

Inherits QGroupBox [p. 78].

Inherited by QHButtonGroup [p. 83] and QVButtonGroup [p. 406].

Public Members

- **QButtonGroup** (QWidget * parent = 0, const char * name = 0)
- **QButtonGroup** (const QString & title, QWidget * parent = 0, const char * name = 0)
- **QButtonGroup** (int strips, Orientation orientation, QWidget * parent = 0, const char * name = 0)
- **QButtonGroup** (int strips, Orientation orientation, const QString & title, QWidget * parent = 0, const char * name = 0)
- bool **isExclusive** () const
- bool **isRadioButtonExclusive** () const
- virtual void **setExclusive** (bool)
- virtual void **setRadioButtonExclusive** (bool)
- int **insert** (QButton * button, int id = -1)
- void **remove** (QButton * button)
- QButton * **find** (int id) const
- int **id** (QButton * button) const
- int **count** () const
- virtual void **setButton** (int id)
- virtual void **moveFocus** (int key)
- QButton * **selected** () const

Signals

- void **pressed** (int id)
- void **released** (int id)
- void **clicked** (int id)

Properties

- bool **exclusive** — whether the button group is exclusive
- bool **radioButtonExclusive** — whether the radiobuttons in the group are exclusive

Detailed Description

The QButtonGroup widget organizes QButton widgets in a group.

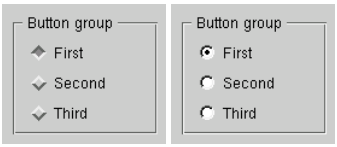
A button group widget makes it easier to deal with groups of buttons. Each button in a button group has a unique identifier. The button group emits a clicked() signal with this identifier when the button is clicked. This makes a button group particularly useful when you have several similar buttons and want to connect all their clicked() signals to one slot.

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

There are two ways of using a button group:

- The button group is a parent widget of a number of buttons, i.e., the button group is the parent argument in the button constructor. The buttons are assigned identifiers 0, 1, 2, etc. in the order they are created. A QButtonGroup can display a frame and a title because it inherits QGroupBox.
- The button group is an invisible widget and the contained buttons have some other parent widget. A button must then be manually inserted using the insert() function with an identifier.

A button can be removed from the group with remove(). A pointer to a button with a given id can be obtained using find(). The id of a button is available using id(). A button can be set on with setButton(). The number of buttons in the group is returned by count().



See also QButton [p. 4], QPushButton [p. 230], QCheckBox [p. 19], QRadioButton [p. 240], Widget Appearance and Style, Layout Management and Organizers.

Member Function Documentation

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

Constructs a button group with no title.

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

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

Constructs a button group with the title *title*.

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

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

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

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

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

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

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

void QButtonGroup::clicked (int id) [signal]

This signal is emitted when a button in the group is clicked. The *id* argument is the button's identifier.

See also QButton::clicked() [p. 8] and insert() [p. 16].

Examples: drawdemo/drawdemo.cpp and xform/xform.cpp.

int QButtonGroup::count () const

Returns the number of buttons in the group.

QButton * QButtonGroup::find (int id) const

Finds and returns a pointer to the button with the specified identifier *id*.

Returns null if the button was not found.

int QButtonGroup::id (QButton * button) const

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

int QButtonGroup::insert (QButton * button, int id = -1)

Inserts the *button* with the identifier *id* into the button group. Returns the button identifier.

Buttons are normally inserted into a button group automatically by giving the button group as the parent when the button is constructed. So it is not necessary to manually insert buttons that have this button group as their parent widget. An exception is when you want custom identifiers instead of the default 0, 1, 2, etc.

The button is assigned the identifier *id* or an automatically generated identifier. It works as follows: If *id* >= 0, this identifier is assigned. If *id* == -1 (default), the identifier is equal to the number of buttons in the group. If *id* is any other negative integer, for instance -2, a unique identifier (negative integer <= -2) is generated. No button has an id of -1.

See also find() [p. 16], remove() [p. 17] and exclusive [p. 18].

Examples: listbox/listbox.cpp and xform/xform.cpp.

bool QButtonGroup::isExclusive () const

Returns TRUE if the button group is exclusive; otherwise returns FALSE. See the "exclusive" [p. 18] property for details.

bool QButtonGroup::isRadioButtonExclusive () const

Returns TRUE if the radiobuttons in the group are exclusive; otherwise returns FALSE. See the "radioButtonExclusive" [p. 18] property for details.

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

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

This function does nothing unless the keyboard focus points to one of the button group members and *key* is one of Key_Up, Key_Down, Key_Left and Key_Right.

void QButtonGroup::pressed (int id) [signal]

This signal is emitted when a button in the group is pressed. The *id* argument is the button's identifier.

void QButtonGroup::released (int id) [signal]

This signal is emitted when a button in the group is released. The *id* argument is the button's identifier.

void QButtonGroup::remove (QButton * button)

Removes the *button* from the button group.

See also insert() [p. 16].

QButton * QButtonGroup::selected () const

Returns a pointer to the selected toggle button if exactly one is selected; returns 0 otherwise.

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

Sets the button with id *id* to be on; if this is an exclusive group, all other buttons in the group will be set to off.

void QButtonGroup::setExclusive (bool) [virtual]

Sets whether the button group is exclusive. See the "exclusive" [p. 18] property for details.

void QButtonGroup::setRadioButtonExclusive (bool) [virtual]

Sets whether the radiobuttons in the group are exclusive. See the "radioButtonExclusive" [p. 18] property for details.

Property Documentation

bool exclusive

This property holds whether the button group is exclusive.

If this property is TRUE, then the buttons in the group are toggled, and to untoggle a button you must click on another button in the group. The default value is FALSE.

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

bool radioButtonExclusive

This property holds whether the radiobuttons in the group are exclusive.

If this property is TRUE (the default), the radiobuttons in the group are treated exclusively.

Set this property's value with setRadioButtonExclusive() and get this property's value with isRadioButtonExclusive().

QCheckBox Class Reference

The QCheckBox widget provides a checkbox with a text label.

`#include <qcheckbox.h>`

Inherits QButton [p. 4].

Public Members

- **QCheckBox** (QWidget * parent, const char * name = 0)
- **QCheckBox** (const QString & text, QWidget * parent, const char * name = 0)
- bool **isChecked** () const
- void **setNoChange** ()
- void **setTristate** (bool y = TRUE)
- bool **isTristate** () const

Public Slots

- void **setChecked** (bool check)

Important Inherited Members

- QString **text** () const
- virtual void **setText** (const QString &)
- const QPixmap * **pixmap** () const
- virtual void **setPixmap** (const QPixmap &)
- QKeySequence **accel** () const
- virtual void **setAccel** (const QKeySequence &)
- bool **isToggleButton** () const
- virtual void **setDown** (bool)
- bool **isDown** () const
- bool **isOn** () const
- ToggleState **state** () const
- bool **autoRepeat** () const
- virtual void **setAutoRepeat** (bool)
- bool **isExclusiveToggle** () const
- QButtonGroup * **group** () const
- void **toggle** ()
- void **pressed** ()

- void **released** ()
- void **clicked** ()
- void **toggled** (bool on)
- void **stateChanged** (int state)

Properties

- bool **autoMask** — whether the checkbox is automatically masked *(read only)*
- bool **checked** — whether the checkbox is checked
- bool **tristate** — whether the checkbox is a tri-state checkbox

Detailed Description

The QCheckBox widget provides a checkbox with a text label.

QCheckBox and QRadioButton are both option buttons. That is, they can be switched on (checked) or off (unchecked). The classes differ in how the choices for the user are restricted. Radio buttons define a "one of many" choice, whereas checkboxes provide "many of many" choices.

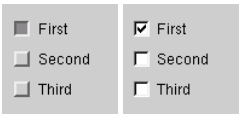
Although it is technically possible to implement radio behavior with checkboxes and vice versa, we strongly recommended sticking with the well-known semantics.

A QButtonGroup can be used to group check buttons visually.

Whenever a checkbox is checked or cleared it emits the signal toggled(). Connect to this signal if you want to trigger an action each time the checkbox changes state. You can use isChecked() to query whether or not a checkbox is checked.

In addition to the usual checked and unchecked states, QCheckBox optionally provides a third state to indicate "no change". This is useful whenever you need to give the user the option of neither checking nor unchecking a checkbox. If you need this third state, enable it with setTristate() and use state() to query the current toggle state. When a tristate checkbox changes state, it emits the stateChanged() signal.

Just like QPushButton, a checkbox can display text or a pixmap. The text can be set in the constructor or with setText(); the pixmap is set with setPixmap().



See also QButton [p. 4], QRadioButton [p. 240], Fowler: Check Box and Basic Widgets.

Member Function Documentation

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

Constructs a checkbox with no text.

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

QCheckBox::QCheckBox (const QString & text, QWidget * parent, const char * name = 0)

Constructs a checkbox with text *text*.

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

QKeySequence QButton::accel () const

Returns the accelerator associated with the button. See the "accel" [p. 11] property for details.

bool QButton::autoRepeat () const

Returns TRUE if autoRepeat is enabled; otherwise returns FALSE. See the "autoRepeat" [p. 11] property for details.

void QButton::clicked () [signal]

This signal is emitted when the button is activated (i.e. first pressed down and then released when the mouse cursor is inside the button), when the accelerator key is typed or when animateClick() is called.

The QButtonGroup::clicked() signal does the same job, if you want to connect several buttons to the same slot.

See also pressed() [p. 9], released() [p. 9] and toggled() [p. 11].

Examples: fonts/simple-qfont-demo/viewer.cpp, listbox/listbox.cpp, network/clientserver/client/client.cpp, network/ftpclient/ftpmainwindow.cpp, richtext/richtext.cpp, t2/main.cpp and t4/main.cpp.

QButtonGroup * QButton::group () const

Returns a pointer to the group of which this button is a member.

If the button is not a member of any QButtonGroup, this function returns 0.

See also QButtonGroup [p. 14].

bool QCheckBox::isChecked () const

Returns TRUE if the checkbox is checked; otherwise returns FALSE. See the "checked" [p. 24] property for details.

bool QButton::isDown () const

Returns TRUE if the button is pressed; otherwise returns FALSE. See the "down" [p. 11] property for details.

bool QButton::isExclusiveToggle () const

Returns TRUE if the button is an exclusive toggle; otherwise returns FALSE. See the "exclusiveToggle" [p. 12] property for details.

bool QButton::isOn () const

Returns TRUE if the button is toggled; otherwise returns FALSE. See the "on" [p. 12] property for details.

bool QButton::isToggleButton () const

Returns TRUE if the button is a toggle button; otherwise returns FALSE. See the "toggleButton" [p. 12] property for details.

bool QCheckBox::isTristate () const

Returns TRUE if the checkbox is a tri-state checkbox; otherwise returns FALSE. See the "tristate" [p. 25] property for details.

const QPixmap * QButton::pixmap () const

Returns the pixmap shown on the button. See the "pixmap" [p. 12] property for details.

void QButton::pressed () [signal]

This signal is emitted when the button is pressed down.

See also released() [p. 9] and clicked() [p. 8].

Examples: network/httpd/httpd.cpp and popup/popup.cpp.

void QButton::released () [signal]

This signal is emitted when the button is released.

See also pressed() [p. 9], clicked() [p. 8] and toggled() [p. 11].

void QButton::setAccel (const QKeySequence &) [virtual]

Sets the accelerator associated with the button. See the "accel" [p. 11] property for details.

void QButton::setAutoRepeat (bool) [virtual]

Sets whether autoRepeat is enabled. See the "autoRepeat" [p. 11] property for details.

void QCheckBox::setChecked (bool check) [slot]

Sets whether the checkbox is checked to *check*. See the "checked" [p. 24] property for details.

void QButton::setDown (bool) [virtual]

Sets whether the button is pressed. See the "down" [p. 11] property for details.

void QCheckBox::setNoChange ()

Sets the checkbox to the "no change" state.

See also tristate [p. 25].

void QButton::setPixmap (const QPixmap &) [virtual]

Sets the pixmap shown on the button. See the "pixmap" [p. 12] property for details.

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

Sets the text shown on the button. See the "text" [p. 12] property for details.

void QCheckBox::setTristate (bool y = TRUE)

Sets whether the checkbox is a tri-state checkbox to *y*. See the "tristate" [p. 25] property for details.

ToggleState QButton::state () const

Returns TRUE if the button is toggled; otherwise returns FALSE. See the "toggleState" [p. 12] property for details.

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

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

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

See also clicked() [p. 8].

QString QButton::text () const

Returns the text shown on the button. See the "text" [p. 12] property for details.

void QButton::toggle () [slot]

Toggles the state of a toggle button.

See also on [p. 12], setOn() [p. 10], toggled() [p. 11] and toggleButton [p. 12].

void QButton::toggled (bool on) [signal]

This signal is emitted whenever a toggle button changes status. *on* is TRUE if the button is on, or FALSE if the button is off.

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

See also clicked() [p. 8].

Example: listbox/listbox.cpp.

Property Documentation

QKeySequence accel

This property holds the accelerator associated with the button.

This property is 0 if there is no accelerator set. If you set this property to 0 then any current accelerator is removed.

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

bool autoMask

This property holds whether the checkbox is automatically masked.

See also QWidget::autoMask [p. 459].

bool autoRepeat

This property holds whether autoRepeat is enabled.

If autoRepeat is enabled then the clicked() signal is emitted at regular intervals if the button is down. This property has no effect on toggle buttons. autoRepeat is off by default.

Set this property's value with setAutoRepeat() and get this property's value with autoRepeat().

bool checked

This property holds whether the checkbox is checked.

The default is unchecked, i.e. FALSE.

Set this property's value with setChecked() and get this property's value with isChecked().

QPixmap pixmap

This property holds the pixmap shown on the button.

If the pixmap is monochrome (i.e., it is a QBitmap or its depth is 1) and it does not have a mask, this property will set the pixmap to be its own mask. The purpose of this is to draw transparent bitmaps which are important for toggle buttons, for example.

pixmap() returns 0 if no pixmap was set.

Set this property's value with setPixmap() and get this property's value with pixmap().

QString text

This property holds the text shown on the button.

This property will return a null string if the button has no text. If the text has an ampersand ('&') in it, then an accelerator is automatically created for it using the character after the '&' as the accelerator key.

There is no default text.

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

bool tristate

This property holds whether the checkbox is a tri-state checkbox.

The default is two-state, i.e. tri-state is FALSE.

Set this property's value with setTristate() and get this property's value with isTristate().

QCheckListItem Class Reference

The QCheckListItem class provides checkable list view items.

`#include <qlistview.h>`

Inherits QListViewItem [p. 199].

Public Members

- enum **Type** { RadioButton, CheckBox, Controller }
- **QCheckListItem** (QCheckListItem * parent, const QString & text, Type tt = Controller)
- **QCheckListItem** (QListViewItem * parent, const QString & text, Type tt = Controller)
- **QCheckListItem** (QListView * parent, const QString & text, Type tt = Controller)
- **QCheckListItem** (QListViewItem * parent, const QString & text, const QPixmap & p)
- **QCheckListItem** (QListView * parent, const QString & text, const QPixmap & p)
- **~QCheckListItem** ()
- virtual void **paintCell** (QPainter * p, const QColorGroup & cg, int column, int width, int align)
- virtual void **paintFocus** (QPainter * p, const QColorGroup & cg, const QRect & r)
- virtual void **setOn** (bool b)
- bool **isOn** () const
- Type **type** () const
- QString **text** () const
- virtual int **rtti** () const

Protected Members

- virtual void **activate** ()
- void **turnOffChild** ()
- virtual void **stateChange** (bool)

Detailed Description

The QCheckListItem class provides checkable list view items.

There are three types of check list items: checkboxes, radio buttons and controllers.

Checkboxes may be inserted at the top level in the list view. A radio button must be the child of a controller.

The item can be checked or unchecked with setOn(). Its type can be retrieved with type() and its text retrieved with text().

See also Advanced Widgets.

Member Type Documentation

QCheckListItem::Type

This enum type specifies a QCheckListItem's type:

- `QCheckListItem::RadioButton`
- `QCheckListItem::CheckBox`
- `QCheckListItem::Controller`

Member Function Documentation

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

Constructs a checkable item with parent *parent*, text *text* and type *tt*. Note that a RadioButton must be the child of a Controller, otherwise it will not toggle.

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

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

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

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

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

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

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

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

QCheckListItem::~QCheckListItem ()

Destroys the item, deleting all its children, freeing up all allocated resources.

void QCheckListItem::activate () [virtual protected]

Toggle check box or set radio button to on.

Reimplemented from QListViewItem [p. 203].

bool QCheckListItem::isOn () const

Returns TRUE if the item is toggled on; otherwise returns FALSE.

void QCheckListItem::paintCell (QPainter * p, const QColorGroup & cg, int column, int width, int align) [virtual]

Paints the item using the painter *p* and the color group *cg*. The item is in column *column*, has width *width* and is aligned *align*. (See Qt::AlignmentFlags for valid alignments.)

Reimplemented from QListViewItem [p. 208].

void QCheckListItem::paintFocus (QPainter * p, const QColorGroup & cg, const QRect & r) [virtual]

Draws the focus rectangle *r* using the color group *cg* on the painter *p*.

Reimplemented from QListViewItem [p. 208].

int QCheckListItem::rtti () const [virtual]

Returns 1.

Make your derived classes return their own values for rtti(), and you can distinguish between listview items. You should use values greater than 1000 preferably a large random number, to allow for extensions to this class.

Reimplemented from QListViewItem [p. 209].

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

Sets the button on if *b* is TRUE, otherwise sets it off. Maintains radio button exclusivity.

void QCheckListItem::stateChange (bool) [virtual protected]

This virtual function is called when the item changes its on/off state.

QString QCheckListItem::text () const

Returns the text of the item.

void QCheckListItem::turnOffChild () [protected]

If this is a Controller that has RadioButton children, turn off the child that is on.

Type QCheckListItem::type () const

Returns the type of this item.

QCheckTableItem Class Reference

The QCheckTableItem class provides checkboxes in QTables.

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

`#include <qtable.h>`

Inherits QTableItem [p. 351].

Public Members

- **QCheckTableItem** (QTable * table, const QString & txt)
- virtual void **setChecked** (bool b)
- bool **isChecked** () const
- virtual int **rtti** () const

Detailed Description

The QCheckTableItem class provides checkboxes in QTables.

A QCheckTableItem is a table item which looks and behaves like a checkbox. The advantage of using QCheckTableItems rather than real checkboxes is that a QCheckTableItem uses far less resources than a real checkbox. When the cell has the focus it displays a real checkbox which the user can interact with. When the cell does not have the focus the cell *looks* like a checkbox. Pixmaps may not be used in QCheckTableItems.

QCheckTableItem items have the edit type WhenCurrent (see EditType).

To change the checkbox's label use setText(). The checkbox can be checked and unchecked with setChecked() and its state retrieved using isChecked().

To populate a table cell with a QCheckTableItem use QTable::setItem().

QCheckTableItems can be distinguished from QTableItems and QComboTableItems using their Run Time Type Identification (rtti) value.

See also rtti() [p. 31], EditType [p. 352] and Advanced Widgets.

Member Function Documentation

QCheckTableItem::QCheckTableItem (QTable * table, const QString & txt)

Creates a QCheckTableItem with an EditType of WhenCurrent as a child of *table*. The checkbox is initially unchecked and its label is set to the string *txt*.

bool QCheckTableItem::isChecked () const

Returns TRUE if the checkbox table item is checked; otherwise returns FALSE.

See also setChecked() [p. 31].

int QCheckTableItem::rtti () const [virtual]

Returns 2.

Make your derived classes return their own values for rtti()to distinguish between different table item subclasses. You should use values greater than 1000, preferably a large random number, to allow for extensions to this class.

See also QTableItem::rtti() [p. 355].

Reimplemented from QTableItem [p. 355].

void QCheckTableItem::setChecked (bool b) [virtual]

If *b* is TRUE the checkbox is checked; if *b* is FALSE the checkbox is unchecked.

See also isChecked() [p. 31].

QComboBox Class Reference

The QComboBox widget is a combined button and popup list.

`#include <qcombobox.h>`

Inherits QWidget [p. 412].

Public Members

- **QComboBox** (QWidget * parent = 0, const char * name = 0)
- **QComboBox** (bool rw, QWidget * parent = 0, const char * name = 0)
- **~QComboBox** ()
- int **count** () const
- void **insertStringList** (const QStringList & list, int index = -1)
- void **insertStrList** (const QStrList & list, int index = -1)
- void **insertStrList** (const QStrList * list, int index = -1)
- void **insertStrList** (const char ** strings, int numStrings = -1, int index = -1)
- void **insertItem** (const QString & t, int index = -1)
- void **insertItem** (const QPixmap & pixmap, int index = -1)
- void **insertItem** (const QPixmap & pixmap, const QString & text, int index = -1)
- void **removeItem** (int index)
- int **currentItem** () const
- virtual void **setCurrentItem** (int index)
- QString **currentText** () const
- virtual void **setCurrentText** (const QString &)
- QString **text** (int index) const
- const QPixmap * **pixmap** (int index) const
- void **changeItem** (const QString & t, int index)
- void **changeItem** (const QPixmap & im, int index)
- void **changeItem** (const QPixmap & im, const QString & t, int index)
- bool autoResize () const *(obsolete)*
- virtual void setAutoResize (bool) *(obsolete)*
- virtual void **setPalette** (const QPalette & palette)
- virtual void **setFont** (const QFont & font)
- virtual void **setSizeLimit** (int)
- int **sizeLimit** () const
- virtual void **setMaxCount** (int)
- int **maxCount** () const
- enum **Policy** { NoInsertion, AtTop, AtCurrent, AtBottom, AfterCurrent, BeforeCurrent }
- virtual void **setInsertionPolicy** (Policy policy)
- Policy **insertionPolicy** () const

- virtual void **setValidator** (const QValidator * v)
- const QValidator * **validator** () const
- virtual void **setListBox** (QListBox * newListBox)
- QListBox * **listBox** () const
- virtual void **setLineEdit** (QLineEdit * edit)
- QLineEdit * **lineEdit** () const
- virtual void **setAutoCompletion** (bool)
- bool **autoCompletion** () const
- void **setDuplicatesEnabled** (bool enable)
- bool **duplicatesEnabled** () const
- bool **editable** () const
- void **setEditable** (bool)
- virtual void **popup** ()

Public Slots

- void **clear** ()
- void **clearValidator** ()
- void **clearEdit** ()
- virtual void **setEditText** (const QString & newText)

Signals

- void **activated** (int index)
- void **highlighted** (int index)
- void **activated** (const QString & string)
- void **highlighted** (const QString & string)
- void **textChanged** (const QString & string)

Properties

- bool **autoCompletion** — whether auto-completion is enabled
- bool **autoMask** — whether the combobox is automatically masked *(read only)*
- bool autoResize — whether auto resize is enabled *(obsolete)*
- int **count** — the number of items in the combobox *(read only)*
- int **currentItem** — the index of the current item in the combobox
- QString **currentText** — the text of the combobox's current item
- bool **duplicatesEnabled** — whether duplicates are allowed
- bool **editable** — whether the combobox is editable
- Policy **insertionPolicy** — the position of the items inserted by the user
- int **maxCount** — the maximum number of items allowed in the combobox
- int **sizeLimit** — the maximum on-screen size of the combobox

Detailed Description

The QComboBox widget is a combined button and popup list.

A combobox is a selection widget which displays the current item and can pop up a list of items. A combobox may be editable in which case the user can enter arbitrary strings.

Since comboboxes occupy little screen space and always display the current item, they are well suited to displaying items that the user will want to see, such as font family or size. Using a combobox the user can always see which item they've selected with the minimum amount of screen space being used.

QComboBox supports three different display styles: Aqua/Motif 1.x, Motif 2.0 and Windows 95. In Motif 1.x, a combobox was called XmOptionMenu. In Motif 2.0, OSF introduced an improved combobox and named that XmComboBox. QComboBox provides both.

QComboBox provides two different constructors. The simplest constructor creates an old-style combobox in Motif (or Aqua) style:

```
QComboBox *c = new QComboBox( this, "read-only combobox" );
```

The other constructor creates a new-style combobox in Motif style, and can create both read-only and read-write comboboxes:

```
QComboBox *c1 = new QComboBox( FALSE, this, "read-only combobox" );
QComboBox *c2 = new QComboBox( TRUE, this, "read-write combobox" );
```

New-style comboboxes use a list box in both Motif and Windows styles, and both the content size and the on-screen size of the list box can be limited with sizeLimit() and setMaxCount() respectively. Old-style comboboxes use a popup in Aqua and Motif style, and that popup will happily grow larger than the desktop if you put enough data into it.

The two constructors create identical-looking comboboxes in Windows style.

Comboboxes can contain pixmaps as well as strings; the insertItem() and changeItem() functions are suitably overloaded. For read-write comboboxes, the function clearEdit() is provided, to clear the displayed string without changing the combobox's contents.

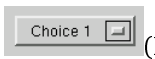
A combobox emits two signals, activated() and highlighted(), when a new item has been activated (selected) or highlighted (made current). Both signals exist in two versions, one with a QString argument and one with an `int` argument. If the user highlights or activates a pixmap, only the `int` signals are emitted. Whenever the text of an editable combobox is changed the textChanged() signal is emitted.

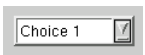
When the user enters a new string in a read-write combobox, the widget may or may not insert it, and it can insert it in several locations. The default policy is is AtBottom but you can change this using setInsertionPolicy().

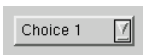
It is possible to constrain the input to an editable combobox using QValidator; see setValidator(). By default, all input is accepted.

If the combo box is not editable then it has a default focusPolicy() of TabFocus, i.e. it will not grab focus if clicked. This differs from both Windows and Motif. If the combo box is editable then it has a default focusPolicy() of StrongFocus, i.e. it will grab focus if clicked.

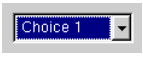
A combobox can be populated using the insert functions, insertStringList() and insertItem() for example. Items can be changed with changeItem(). An item can be removed with removeItem() and all items can be removed with clear(). The text of the current item is returned by currentText(), and the text of a numbered item is returned with text(). The current item can be set with setCurrentItem() or setCurrentText(). The number of items in the combobox is returned by count(); the maximum number of items can be set with setMaxCount(). You can allow editing using setEditable(). For editable comboboxes you can set auto-completion using setAutoCompletion() and whether or not the user can add duplicates is set with setDuplicatesEnabled().



(Motif 1, read-only) 

(Motif 2, read-write) 

(Motif 2, read-only)



(Windows style)

See also QLineEdit [p. 132], QListBox [p. 146], QSpinBox [p. 295], QRadioButton [p. 240], QButtonGroup [p. 14], GUI Design Handbook: Combo Box, GUI Design Handbook: Drop-Down List Box and Basic Widgets.

Member Type Documentation

QComboBox::Policy

This enum specifies what the QComboBox should do when a new string is entered by the user. The following policies are defined:

- `QComboBox::NoInsertion` - the string will not be inserted into the combobox.
- `QComboBox::AtTop` - insert the string as the first item in the combobox.
- `QComboBox::AtCurrent` - replace the previously selected item with the string the user has entered.
- `QComboBox::AtBottom` - insert the string as the last item in the combobox.
- `QComboBox::AfterCurrent` - insert the string after the previously selected item.
- `QComboBox::BeforeCurrent` - insert the string before the previously selected item.

activated() is always emitted when the string is entered.

If inserting the new string would cause the combobox to breach its content size limit, the item at the other end of the list is deleted. The definition of "other end" is implementation-dependent.

Member Function Documentation

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

Constructs a combobox widget with parent *parent* and name *name*.

This constructor creates a popup list if the program uses Motif (or Aqua) look and feel; this is compatible with Motif 1.x and Aqua.

QComboBox::QComboBox (bool rw, QWidget * parent = 0, const char * name = 0)

Constructs a combobox with a maximum size and either Motif 2.0 or Windows look and feel.

The input field can be edited if *rw* is TRUE, otherwise the user may only choose one of the items in the combobox.

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

QComboBox::~QComboBox ()

Destroys the combobox.

void QComboBox::activated (int index) [signal]

This signal is emitted when a new item has been activated (selected). The *index* is the position of the item in the combobox.

Examples: fileiconview/mainwindow.cpp, helpviewer/helpwindow.cpp, lineedits/lineedits.cpp, listboxcombo/listboxcombo.cpp, network/ftpclient/ftpmainwindow.cpp and qmag/qmag.cpp.

void QComboBox::activated (const QString & string) [signal]

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

This signal is emitted when a new item has been activated (selected). *string* is the selected string.

You can also use the activated(int) signal, but be aware that its argument is meaningful only for selected strings, not for user entered strings.

bool QComboBox::autoCompletion () const

Returns TRUE if auto-completion is enabled; otherwise returns FALSE. See the "autoCompletion" [p. 41] property for details.

bool QComboBox::autoResize () const

Returns TRUE if auto resize is enabled; otherwise returns FALSE. See the "autoResize" [p. 42] property for details.

void QComboBox::changeItem (const QString & t, int index)

Replaces the item at position *index* with the text *t*.

void QComboBox::changeItem (const QPixmap & im, int index)

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

Replaces the item at position *index* with the pixmap *im*, unless the combobox is editable.

See also insertItem() [p. 37].

void QComboBox::changeItem (const QPixmap & im, const QString & t, int index)

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

Replaces the item at position *index* with the pixmap *im* and the text *t*.

See also insertItem() [p. 37].

void QComboBox::clear () [slot]

Removes all combobox items.

void QComboBox::clearEdit () [slot]

Clears the line edit without changing the combobox's contents. Does nothing if the combobox isn't editable.

This is particularly handy when using a combobox as a line edit with history. For example you can connect the combobox's activated() signal to clearEdit() in order to present the user with a new, empty line as soon as Return is pressed.

See also setEditText() [p. 40].

void QComboBox::clearValidator () [slot]

This slot is equivalent to setValidator(0).

int QComboBox::count () const

Returns the number of items in the combobox. See the "count" [p. 42] property for details.

int QComboBox::currentItem () const

Returns the index of the current item in the combobox. See the "currentItem" [p. 42] property for details.

QString QComboBox::currentText () const

Returns the text of the combobox's current item. See the "currentText" [p. 42] property for details.

bool QComboBox::duplicatesEnabled () const

Returns TRUE if duplicates are allowed; otherwise returns FALSE. See the "duplicatesEnabled" [p. 42] property for details.

bool QComboBox::editable () const

Returns TRUE if the combobox is editable; otherwise returns FALSE. See the "editable" [p. 43] property for details.

void QComboBox::highlighted (int index) [signal]

This signal is emitted when a new item has been set to current. The *index* is the position of the item in the combobox.

void QComboBox::highlighted (const QString & string) [signal]

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

This signal is emitted when a new item has been highlighted. *string* is the highlighted string.

You can also use highlighted(int) signal.

void QComboBox::insertItem (const QString & t, int index = -1)

Inserts a text item with text *t*, at position *index*. The item will be appended if *index* is negative.

Examples: fileiconview/mainwindow.cpp, helpviewer/helpwindow.cpp, lineedits/lineedits.cpp, listboxcombo/listboxcombo.cpp, network/ftpclient/ftpmainwindow.cpp and tictac/tictac.cpp.

void QComboBox::insertItem (const QPixmap & pixmap, int index = -1)

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

Inserts a *pixmap* item at position *index*. The item will be appended if *index* is negative.

void QComboBox::insertItem (const QPixmap & pixmap, const QString & text, int index = -1)

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

Inserts a *pixmap* item with additional text *text* at position *index*. The item will be appended if *index* is negative.

void QComboBox::insertStrList (const char ** strings, int numStrings = -1, int index = -1)

Inserts the array of char * *strings* at position *index* in the combobox.

The *numStrings* argument is the number of strings. If *numStrings* is -1 (default), the *strings* array must be terminated with 0.

Example:

```
static const char* items[] = { "red", "green", "blue", 0 };
combo->insertStrList( items );
```

Example: qmag/qmag.cpp.

void QComboBox::insertStrList (const QStrList & list, int index = -1)

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

Inserts the *list* of strings at position *index* in the combobox.

This is only for compatibility, as it does not support Unicode strings. See insertStringList().

void QComboBox::insertStrList (const QStrList * list, int index = -1)

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

Inserts the *list* of strings at position *index* in the combobox.

This is only for compatibility, as it does not support Unicode strings. See insertStringList().

void QComboBox::insertStringList (const QStringList & list, int index = -1)

Inserts the *list* of strings at position *index* in the combobox.

Policy QComboBox::insertionPolicy () const

Returns the position of the items inserted by the user. See the "insertionPolicy" [p. 43] property for details.

QLineEdit * QComboBox::lineEdit () const

Returns the line editor, or 0 if there is no line editor.

Only editable listboxes have a line editor.

QListBox * QComboBox::listBox () const

Returns the current list box, or 0 if there is no list box. (QComboBox can use QPopupMenu instead of QListBox.) Provided to match setListBox().

See also setListBox() [p. 40].

Example: listboxcombo/listboxcombo.cpp.

int QComboBox::maxCount () const

Returns the maximum number of items allowed in the combobox. See the "maxCount" [p. 43] property for details.

const QPixmap * QComboBox::pixmap (int index) const

Returns the pixmap item at position *index*, or 0 if the item is not a pixmap.

void QComboBox::popup () [virtual]

Pops up the combobox popup list.

If the list is empty, no items appear.

void QComboBox::removeItem (int index)

Removes the item at position *index*.

void QComboBox::setAutoCompletion (bool) [virtual]

Sets whether auto-completion is enabled. See the "autoCompletion" [p. 41] property for details.

void QComboBox::setAutoResize (bool) [virtual]

Sets whether auto resize is enabled. See the "autoResize" [p. 42] property for details.

void QComboBox::setCurrentItem (int index) [virtual]

Sets the index of the current item in the combobox to *index*. See the "currentItem" [p. 42] property for details.

void QComboBox::setCurrentText (const QString &) [virtual]

Sets the text of the combobox's current item. See the "currentText" [p. 42] property for details.

void QComboBox::setDuplicatesEnabled (bool enable)

Sets whether duplicates are allowed to *enable*. See the "duplicatesEnabled" [p. 42] property for details.

void QComboBox::setEditText (const QString & newText) [virtual slot]

Sets the text in the line edit to *newText* without changing the combobox's contents. Does nothing if the combobox isn't editable.

This is useful e.g. for providing a good starting point for the user's editing and entering the change in the combobox only when the user presses Enter.

See also clearEdit() [p. 36] and insertItem() [p. 37].

Example: network/ftpclient/ftpmainwindow.cpp.

void QComboBox::setEditable (bool)

Sets whether the combobox is editable. See the "editable" [p. 43] property for details.

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

Reimplements QWidget::setFont().

Sets the font for both the combobox button and the combobox popup list to *font*.

Reimplemented from QWidget [p. 449].

void QComboBox::setInsertionPolicy (Policy policy) [virtual]

Sets the position of the items inserted by the user to *policy*. See the "insertionPolicy" [p. 43] property for details.

void QComboBox::setLineEdit (QLineEdit * edit) [virtual]

Sets the lineedit to use *edit* instead of the current lineedit.

void QComboBox::setListBox (QListBox * newListBox) [virtual]

Sets the combobox to use *newListBox* instead of the current list box or popup. As a side effect, it clears the combobox of its current contents.

Warning: QComboBox assumes that newListBox->text(n) returns non-null for 0 <= n < newListbox->count(). This assumption is necessary because of the line edit in QComboBox.

void QComboBox::setMaxCount (int) [virtual]

Sets the maximum number of items allowed in the combobox. See the "maxCount" [p. 43] property for details.

void QComboBox::setPalette (const QPalette & palette) [virtual]

Reimplements QWidget::setPalette().

Sets the palette for both the combobox button and the combobox popup list to *palette*.

Reimplemented from QWidget [p. 451].

void QComboBox::setSizeLimit (int) [virtual]

Sets the maximum on-screen size of the combobox. See the "sizeLimit" [p. 43] property for details.

void QComboBox::setValidator (const QValidator * v) [virtual]

Applies the validator *v* to the combobox so that only text which is valid according to *v* is accepted.

This function does nothing if the combo is not editable.

See also validator() [p. 41], clearValidator() [p. 37] and QValidator [Additional Functionality with Qt].

int QComboBox::sizeLimit () const

Returns the maximum on-screen size of the combobox. See the "sizeLimit" [p. 43] property for details.

QString QComboBox::text (int index) const

Returns the text item at position *index*, or null string if the item is not a string.

See also currentText [p. 42].

Examples: fileiconview/mainwindow.cpp and helpviewer/helpwindow.cpp.

void QComboBox::textChanged (const QString & string) [signal]

This signal is used for editable comboboxes. It is emitted whenever the contents of the text entry field changes. *string* contains the new text.

const QValidator * QComboBox::validator () const

Returns the validator which constrains editing for this combobox if there is one, otherwise returns 0.

See also setValidator() [p. 41], clearValidator() [p. 37] and QValidator [Additional Functionality with Qt].

Property Documentation

bool autoCompletion

This property holds whether auto-completion is enabled.

This property can only be set for editable comboboxes, for non-editable comboboxes it has no effect. It is FALSE by default.

Set this property's value with setAutoCompletion() and get this property's value with autoCompletion().

bool autoMask

This property holds whether the combobox is automatically masked.

See also QWidget::autoMask [p. 459].

bool autoResize

This property holds whether auto resize is enabled.

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

If this property is set to TRUE then the combobox will resize itself whenever its contents change. The default is FALSE.

Set this property's value with setAutoResize() and get this property's value with autoResize().

int count

This property holds the number of items in the combobox.

Get this property's value with count().

int currentItem

This property holds the index of the current item in the combobox.

Set this property's value with setCurrentItem() and get this property's value with currentItem().

QString currentText

This property holds the text of the combobox's current item.

Set this property's value with setCurrentText() and get this property's value with currentText().

bool duplicatesEnabled

This property holds whether duplicates are allowed.

If the combobox is editable and the user enters some text in the lineedit of the combobox and presses Enter (and the insertionPolicy() is different from NoInsertion), then what happens is as follows:

- If the text is not already in the list, the text is inserted.
- If the text is in the list and this property is TRUE (the default), the text is inserted.
- If the text is in the list and this property is FALSE, the text is *not* inserted; instead the item which has matching text becomes the current item.

This property only affects user-interaction. You can use insertItem() to insert duplicates if you wish regardless of this setting.

Set this property's value with setDuplicatesEnabled() and get this property's value with duplicatesEnabled().

bool editable

This property holds whether the combobox is editable.

This property's default is FALSE.

Set this property's value with setEditable() and get this property's value with editable().

Policy insertionPolicy

This property holds the position of the items inserted by the user.

The default insertion policy is AtBottom.

Set this property's value with setInsertionPolicy() and get this property's value with insertionPolicy().

int maxCount

This property holds the maximum number of items allowed in the combobox.

Set this property's value with setMaxCount() and get this property's value with maxCount().

int sizeLimit

This property holds the maximum on-screen size of the combobox.

This is disregarded in Motif 1.x style. The default limit is ten lines. If the number of items in the combobox is or grows larger than lines, a scrollbar is added.

Set this property's value with setSizeLimit() and get this property's value with sizeLimit().

QComboTableItem Class Reference

The QComboTableItem class provides a means of using comboboxes in QTables.

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

`#include <qtable.h>`

Inherits QTableItem [p. 351].

Public Members

- **QComboTableItem** (QTable * table, const QStringList & list, bool editable = FALSE)
- virtual void **setCurrentItem** (int i)
- virtual void **setCurrentItem** (const QString & s)
- int **currentItem** () const
- QString **currentText** () const
- int **count** () const
- QString **text** (int i) const
- virtual void **setEditable** (bool b)
- bool **isEditable** () const
- virtual void **setStringList** (const QStringList & l)
- virtual int **rtti** () const

Detailed Description

The QComboTableItem class provides a means of using comboboxes in QTables.

A QComboTableItem is a table item which looks and behaves like a combobox. The advantage of using QComboTableItems rather than real comboboxes is that a QComboTableItem uses far less resources than a real combobox. When the cell has the focus it displays a real combobox which the user can interact with. When the cell does not have the focus the cell *looks* like a combobox. Only text items (i.e. no pixmaps) may be used in QComboTableItems.

QComboTableItem items have the edit type WhenCurrent (see EditType). The QComboTableItem's list of items is provided by a QStringList passed to the constructor.

The list of items may be changed using setStringList(). The current item can be set with setCurrentItem() and retrieved with currentItem(). The text of the current item can be obtained with currentText(), and the text of a particular item can be retrieved with text().

If isEditable() is TRUE the QComboTableItem will permit the user to either choose an existing list item, or create a new list item by entering their own text; otherwise the user may only choose one of the existing list items.

To populate a table cell with a QComboTableItem use QTable::setItem().

QComboTableItems may be deleted with QTable::clearCell().

QComboTableItems can be distinguished from QTableItems and QCheckTableItems using their Run Time Type Identification number (see rtti()).

See also Advanced Widgets.

Member Function Documentation

QComboTableItem::QComboTableItem (QTable * table, const QStringList & list, bool editable = FALSE)

Creates a combo table item for the table *table*. The combobox's list of items is passed in the *list* argument. If *editable* is TRUE the user may type in new list items; if *editable* is FALSE the user may only select from the list of items provided.

By default QComboTableItems cannot be replaced by other table items since isReplaceable() returns FALSE by default.

See also QTable::clearCell() [p. 333] and EditType [p. 352].

int QComboTableItem::count () const

Returns the total number of list items in the combo table item.

int QComboTableItem::currentItem () const

Returns the index of the combo table item's current list item.

See also setCurrentItem() [p. 45].

QString QComboTableItem::currentText () const

Returns the text of the combo table item's current list item.

See also currentItem() [p. 45] and text() [p. 46].

bool QComboTableItem::isEditable () const

Returns whether the user may add their own list items to the combo's list of items.

See also setEditable() [p. 46].

int QComboTableItem::rtti () const [virtual]

For QComboTableItems this function returns a Run Time Identification value of 1.

See also QTableItem::rtti() [p. 355].

Reimplemented from QTableItem [p. 355].

void QComboTableItem::setCurrentItem (int i) [virtual]

Sets the list item *i* to be the combo table item's current list item.

See also currentItem() [p. 45].

void QComboTableItem::setCurrentItem (const QString & s) [virtual]

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

Sets the list item whose text is *s* to be the combo table item's current list item. Does nothing if no list item has the text *s*.

See also currentItem() [p. 45].

void QComboTableItem::setEditable (bool b) [virtual]

If *b* is TRUE the combo table item can be edited, i.e. the user may enter a new text item themselves. If *b* is FALSE the user may may only choose one of the existing items.

See also isEditable() [p. 45].

void QComboTableItem::setStringList (const QStringList & l) [virtual]

Sets the list items of this QComboTableItem to the strings in the string list *l*.

QString QComboTableItem::text (int i) const

Returns the text of the combo's list item at index *i*.

See also currentText() [p. 45].

QDateEdit Class Reference

The QDateEdit class provides a date editor.

`#include <qdatetimeedit.h>`

Public Members

- **QDateEdit** (QWidget * parent = 0, const char * name = 0)
- **QDateEdit** (const QDate & date, QWidget * parent = 0, const char * name = 0)
- **~QDateEdit** ()
- enum **Order** { DMY, MDY, YMD, YDM }
- virtual void **setDate** (const QDate & date)
- QDate **date** () const
- virtual void **setOrder** (Order order)
- Order **order** () const
- virtual void **setAutoAdvance** (bool advance)
- bool **autoAdvance** () const
- virtual void **setMinValue** (const QDate & d)
- QDate **minValue** () const
- virtual void **setMaxValue** (const QDate & d)
- QDate **maxValue** () const
- virtual void **setRange** (const QDate & min, const QDate & max)
- QString **separator** () const
- virtual void **setSeparator** (const QString & s)

Signals

- void **valueChanged** (const QDate & date)

Properties

- bool **autoAdvance** — whether the editor automatically advances to the next section
- QDate **date** — the date value of the editor
- QDate **maxValue** — the maximum editor value
- QDate **minValue** — the minimum editor value
- Order **order** — the order in which the year, month and day appear

Protected Members

- virtual QString **sectionFormattedText** (int sec)
- virtual void **setYear** (int year)
- virtual void **setMonth** (int month)
- virtual void **setDay** (int day)
- virtual void **fix** ()

Protected Slots

- void **updateButtons** ()

Detailed Description

The QDateEdit class provides a date editor.

QDateEdit allows the user to edit dates by using the keyboard or the arrow keys to increase/decrease date values. The arrow keys can be used to move from section to section within the QDateEdit box. Dates appear according the local date/time settings or in year, month, day order if the system doesn't provide this information. It is recommended that the QDateEdit be initialised with a date, e.g.

```
QDateEdit *dateEdit = new QDateEdit( QDate::currentDate(), this );
dateEdit->setRange( QDate::currentDate().addDays( -365 ),
                    QDate::currentDate().addDays(  365 ) );
dateEdit->setOrder( QDateEdit::MDY );
dateEdit->setAutoAdvance( TRUE );
```

Here we've created a new QDateEdit object initialised with today's date and restricted the valid date range to today plus or minus 365 days. We've set the order to month, day, year. If the auto advance property is TRUE (as we've set it here) when the user completes a section of the date, e.g. enters two digits for the month, they are automatically taken to the next section.

The maximum and minimum values for a date value in the date editor default to the maximum and minimum values for a QDate. You can change this by calling setMinValue(), setMaxValue() or setRange().

Terminology: A QDateEdit widget comprises three 'sections', one each for the year, month and day. You can change the separator character using QDateTimeEditor::setSeparator(), by default the separator will be taken from the systems settings. If that is impossible, it defaults to "-".

See also QDate [Additional Functionality with Qt], QTimeEdit [p. 400], QDateTimeEdit [p. 53], Advanced Widgets and Time and Date.

Member Type Documentation

QDateEdit::Order

This enum defines the order in which the sections that comprise a date appear.

- `QDateEdit::MDY` - month-day-year
- `QDateEdit::DMY` - day-month-year
- `QDateEdit::YMD` - year-month-day (the default)
- `QDateEdit::YDM` - year-day-month (a very bad idea)

Member Function Documentation

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

Constructs an empty date editor which is a child of *parent* and the name *name*.

QDateEdit::QDateEdit (const QDate & date, QWidget * parent = 0, const char * name = 0)

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

Constructs a date editor with the initial value *date*, parent *parent* and name *name*.

The date editor is initialized with *date*.

QDateEdit::~QDateEdit ()

Destroys the object and frees any allocated resources.

bool QDateEdit::autoAdvance () const

Returns TRUE if the editor automatically advances to the next section; otherwise returns FALSE. See the "autoAdvance" [p. 51] property for details.

QDate QDateEdit::date () const

Returns the date value of the editor. See the "date" [p. 51] property for details.

void QDateEdit::fix () [virtual protected]

Attempts to fix any invalid date entries.

The rules applied are as follows:

- If the year has four digits it is left unchanged.
- If the year has two digits in the range 70..99, the previous century, i.e. 1900, will be added giving a year in the range 1970..1999.
- If the year has two digits in the range 0..69, the current century, i.e. 2000, will be added giving a year in the range 2000..2069.
- If the year is in the range 100..999, the current century, i.e. 2000, will be added giving a year in the range 2100..2999.

QDate QDateEdit::maxValue () const

Returns the maximum editor value. See the "maxValue" [p. 51] property for details.

QDate QDateEdit::minValue () const

Returns the minimum editor value. See the "minValue" [p. 52] property for details.

Order QDateEdit::order () const

Returns the order in which the year, month and day appear. See the "order" [p. 52] property for details.

QString QDateEdit::sectionFormattedText (int sec) [virtual protected]

Returns the formatted number for section *sec*. This will correspond to either the year, month or day section, depending on the current display order.

See also order [p. 52].

QString QDateEdit::separator () const

Returns the separator for the editor.

void QDateEdit::setAutoAdvance (bool advance) [virtual]

Sets whether the editor automatically advances to the next section to *advance*. See the "autoAdvance" [p. 51] property for details.

void QDateEdit::setDate (const QDate & date) [virtual]

Sets the date value of the editor to *date*. See the "date" [p. 51] property for details.

void QDateEdit::setDay (int day) [virtual protected]

Sets the day to *day*, which must be a valid day. The function will ensure that the *day* set is valid for the month and year.

void QDateEdit::setMaxValue (const QDate & d) [virtual]

Sets the maximum editor value to *d*. See the "maxValue" [p. 51] property for details.

void QDateEdit::setMinValue (const QDate & d) [virtual]

Sets the minimum editor value to *d*. See the "minValue" [p. 52] property for details.

void QDateEdit::setMonth (int month) [virtual protected]

Sets the month to *month*, which must be a valid month, i.e. between 1 and 12.

void QDateEdit::setOrder (Order order) [virtual]

Sets the order in which the year, month and day appear to *order*. See the "order" [p. 52] property for details.

void QDateEdit::setRange (const QDate & min, const QDate & max) [virtual]

Sets the valid input range for the editor to be from *min* to *max* inclusive. If *min* is invalid no minimum date will be set. Similarly, if *max* is invalid no maximum date will be set.

void QDateEdit::setSeparator (const QString & s) [virtual]

Sets the separator to *s*. Note that currently only the first character of *s* is used.

void QDateEdit::setYear (int year) [virtual protected]

Sets the year to *year*, which must be a valid year. The range currently supported is from 1752 to 8000.

See also QDate [Additional Functionality with Qt].

void QDateEdit::updateButtons () [protected slot]

Enables/disables the push buttons according to the min/max date for this widget.

void QDateEdit::valueChanged (const QDate & date) [signal]

This signal is emitted whenever the editor's value changes. The *date* parameter is the new value.

Property Documentation

bool autoAdvance

This property holds whether the editor automatically advances to the next section.

If autoAdvance is TRUE, the editor will automatically advance focus to the next date section if a user has completed a section. The default is FALSE.

Set this property's value with setAutoAdvance() and get this property's value with autoAdvance().

QDate date

This property holds the date value of the editor.

If the date property is not valid, the editor displays all zeroes and QDateEdit::date() will return an invalid date. It is strongly recommended that the editor be given a default date value. That way, attempts to set the date property to an invalid date will fail.

When changing the date property, if the date is less than minValue(), or is greater than maxValue(), nothing happens.

Set this property's value with setDate() and get this property's value with date().

QDate maxValue

This property holds the maximum editor value.

Setting the maximum date value for the editor is equivalent to calling QDateEdit::setRange(minValue(), *d*), where *d* is the maximum date. Teh default maximum date is 8000-12-31.

Set this property's value with setMaxValue() and get this property's value with maxValue().

QDate minValue

This property holds the minimum editor value.

Setting the minimum date value is equivalent to calling QDateEdit::setRange(*d*, maxValue()), where *d* is the minimum date. The default minimum date is 1752-09-14.

Set this property's value with setMinValue() and get this property's value with minValue().

Order order

This property holds the order in which the year, month and day appear.

The default order is locale dependent.

See also Order [p. 48].

Set this property's value with setOrder() and get this property's value with order().

QDateTimeEdit Class Reference

The QDateTimeEdit class combines a QDateEdit and QTimeEdit widget into a single widget for editing datetimes.

`#include <qdatetimeedit.h>`

Inherits QWidget [p. 412].

Public Members

- **QDateTimeEdit** (QWidget * parent = 0, const char * name = 0)
- **QDateTimeEdit** (const QDateTime & datetime, QWidget * parent = 0, const char * name = 0)
- **~QDateTimeEdit** ()
- virtual void **setDateTime** (const QDateTime & dt)
- QDateTime **dateTime** () const
- QDateEdit * **dateEdit** ()
- QTimeEdit * **timeEdit** ()
- virtual void **setAutoAdvance** (bool advance)
- bool **autoAdvance** () const

Signals

- void **valueChanged** (const QDateTime & datetime)

Properties

- QDateTime **dateTime** — the datetime value of the editor

Detailed Description

The QDateTimeEdit class combines a QDateEdit and QTimeEdit widget into a single widget for editing datetimes.

QDateTimeEdit consists of a QDateEdit and QTimeEdit widget placed side by side and offers the functionality of both. The user can edit the date and time by using the keyboard or the arrow keys to increase/decrease date or time values. The Tab key can be used to move from section to section within the QDateTimeEdit widget, and the user can be moved automatically when they complete a section using setAutoAdvance(). The datetime can be set with setDateTime().

The dateformat is read from the system's locale settings. It is set to year, month, day order if that is not possible. see QDateEdit::setOrder() to change this. Times appear in the order hours, minutes, seconds using the 24 hour clock.

It is recommended that the QDateTimeEdit is initialised with a datetime, e.g.

```
QDateTimeEdit *dateTimeEdit = new QDateTimeEdit( QDateTime::currentDateTime(), this );
dateTimeEdit->dateEdit()->setRange( QDateTime::currentDate(),
                                    QDateTime::currentDate().addDays( 7 ) );
```

Here we've created a new QDateTimeEdit set to the current date and time, and set the date to have a minimum date of now and a maximum date of a week from now.

Terminology: A QDateEdit widget consists of three 'sections', one each for the year, month and day. Similarly a QTimeEdit consists of three sections, one each for the hour, minute and second. The character that separates each date section is specified with setDateSeparator(); similarly setTimeSeparator() is used for the time sections.

See also QDateEdit [p. 47], QTimeEdit [p. 400], Advanced Widgets and Time and Date.

Member Function Documentation

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

Constructs an empty datetime edit with parent *parent* and name *name*.

QDateTimeEdit::QDateTimeEdit (const QDateTime & datetime, QWidget * parent = 0, const char * name = 0)

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

Constructs a datetime edit with the initial value *datetime*, parent *parent* and name *name*.

QDateTimeEdit::~QDateTimeEdit ()

Destroys the object and frees any allocated resources.

bool QDateTimeEdit::autoAdvance () const

Returns TRUE if auto-advance is enabled, otherwise returns FALSE.

See also setAutoAdvance() [p. 54].

QDateEdit * QDateTimeEdit::dateEdit ()

Returns the internal widget used for editing the date part of the datetime.

QDateTime QDateTimeEdit::dateTime () const

Returns the datetime value of the editor. See the "dateTime" [p. 55] property for details.

void QDateTimeEdit::setAutoAdvance (bool advance) [virtual]

Sets the auto advance property of the editor to *advance*. If set to TRUE, the editor will automatically advance focus to the next date or time section if the user has completed a section.

void QDateTimeEdit::setDateTime (const QDateTime & dt) [virtual]

Sets the datetime value of the editor to *dt*. See the "dateTime" [p. 55] property for details.

QTimeEdit * QDateTimeEdit::timeEdit ()

Returns the internal widget used for editing the time part of the datetime.

void QDateTimeEdit::valueChanged (const QDateTime & datetime) [signal]

This signal is emitted every time the date or time changes. The *datetime* argument is the new datetime.

Property Documentation

QDateTime dateTime

This property holds the datetime value of the editor.

The datetime edit's datetime which may be an invalid datetime.

Set this property's value with setDateTime() and get this property's value with dateTime().

QDial Class Reference

The QDial class provides a rounded range control (like a speedometer or potentiometer).

`#include <qdial.h>`

Inherits QWidget [p. 412] and QRangeControl [p. 246].

Public Members

- **QDial** (QWidget * parent = 0, const char * name = 0, WFlags f = 0)
- **QDial** (int minValue, int maxValue, int pageStep, int value, QWidget * parent = 0, const char * name = 0)
- **~QDial** ()
- bool **tracking** () const
- bool **wrapping** () const
- int **notchSize** () const
- virtual void **setNotchTarget** (double)
- double **notchTarget** () const
- bool **notchesVisible** () const
- int **minValue** () const
- int **maxValue** () const
- void **setMinValue** (int)
- void **setMaxValue** (int)
- int **lineStep** () const
- int **pageStep** () const
- void **setLineStep** (int)
- void **setPageStep** (int)
- int **value** () const

Public Slots

- virtual void **setValue** (int)
- void **addLine** ()
- void **subtractLine** ()
- void **addPage** ()
- void **subtractPage** ()
- virtual void **setNotchesVisible** (bool b)
- virtual void **setWrapping** (bool on)
- virtual void **setTracking** (bool enable)

Signals

- void **valueChanged** (int value)
- void **dialPressed** ()
- void **dialMoved** (int value)
- void **dialReleased** ()

Properties

- int **lineStep** — the current line step
- int **maxValue** — the current maximum value
- int **minValue** — the current minimum value
- int **notchSize** — the current notch size *(read only)*
- double **notchTarget** — the target number of pixels between notches
- bool **notchesVisible** — whether the notches are shown
- int **pageStep** — the current page step
- bool **tracking** — whether tracking is enabled
- int **value** — the current dial value
- bool **wrapping** — whether wrapping is enabled

Protected Members

- virtual void **valueChange** ()
- virtual void **rangeChange** ()
- virtual void **repaintScreen** (const QRect * cr = 0)

Detailed Description

The QDial class provides a rounded range control (like a speedometer or potentiometer).

QDial is used when the user needs to control a value within a program-definable range, and the range either wraps around (typically, 0..359 degrees) or the dialog layout needs a square widget.

Both API- and UI-wise, the dial is very similar to a slider. Indeed, when wrapping() is FALSE (the default) there is no real difference between a slider and a dial. They have the same signals, slots and member functions, all of which do the same things. Which one you use depends only on your taste and on the application.

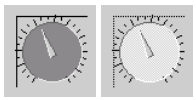
The dial initially emits valueChanged() signals continuously while the slider is being moved; you can make it emit the signal less often by calling setTracking(FALSE). dialMoved() is emitted continuously even when tracking() is FALSE.

The slider also emits dialPressed() and dialReleased() signals when the mouse button is pressed and released. But note that the dial's value can change without these signals being emitted; the keyboard and wheel can be used to change the value.

Unlike the slider, QDial attempts to draw a "nice" number of notches rather than one per lineStep(). If possible, the number of notches drawn is one per lineStep(), but if there aren't enough pixels to draw every one, QDial will draw every second, third etc., notch. notchSize() returns the number of units per notch, hopefully a multiple of lineStep(); setNotchTarget() sets the target distance between neighbouring notches in pixels. The default is 3.75 pixels.

Like the slider, the dial makes the QRangeControl functions setValue(), addLine(), subtractLine(), addPage() and subtractPage() available as slots.

The dial's keyboard interface is fairly simple: The left/up and right/down arrow keys move by lineStep(), page up and page down by pageStep() and Home and End to minValue() and maxValue().



See also QScrollBar [p. 252], QSpinBox [p. 295], GUI Design Handbook: Slider and Basic Widgets.

Member Function Documentation

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

Constructs a dial called *name* with parent *parent*. *f* is propagated to the QWidget constructor. It has the default range of a QRangeControl.

QDial::QDial (int minValue, int maxValue, int pageStep, int value, QWidget * parent = 0, const char * name = 0)

Constructs a dial called *name* with parent *parent*. The dial's value can never be smaller than *minValue* or greater than *maxValue*. Its page step size is *pageStep*, and its initial value is *value*.

value is forced to be within the legal range.

QDial::~QDial ()

Destroys the dial.

void QDial::addLine () [slot]

Increments the dial's value() by one lineStep().

void QDial::addPage () [slot]

Increments the dial's value() by one pageStep() of steps.

void QDial::dialMoved (int value) [signal]

This signal is emitted whenever the dial *value* changes. The frequency of this signal is *not* influenced by setTracking().

See also valueChanged() [p. 61].

void QDial::dialPressed () [signal]

This signal is emitted when the use begins mouse interaction with the dial.

See also dialReleased() [p. 59].

void QDial::dialReleased () [signal]

This signal is emitted when the use ends mouse interaction with the dial.

See also dialPressed() [p. 58].

int QDial::lineStep () const

Returns the current line step. See the "lineStep" [p. 61] property for details.

int QDial::maxValue () const

Returns the current maximum value. See the "maxValue" [p. 61] property for details.

int QDial::minValue () const

Returns the current minimum value. See the "minValue" [p. 61] property for details.

int QDial::notchSize () const

Returns the current notch size. See the "notchSize" [p. 62] property for details.

double QDial::notchTarget () const

Returns the target number of pixels between notches. See the "notchTarget" [p. 62] property for details.

bool QDial::notchesVisible () const

Returns TRUE if the notches are shown; otherwise returns FALSE. See the "notchesVisible" [p. 62] property for details.

int QDial::pageStep () const

Returns the current page step. See the "pageStep" [p. 62] property for details.

void QDial::rangeChange () [virtual protected]

Reimplemented to ensure tick-marks are consistent with the new range.

Reimplemented from QRangeControl [p. 249].

void QDial::repaintScreen (const QRect * cr = 0) [virtual protected]

Paints the dial using clip region *cr*.

void QDial::setLineStep (int)

Sets the current line step. See the "lineStep" [p. 61] property for details.

void QDial::setMaxValue (int)

Sets the current maximum value. See the "maxValue" [p. 61] property for details.

void QDial::setMinValue (int)

Sets the current minimum value. See the "minValue" [p. 61] property for details.

void QDial::setNotchTarget (double) [virtual]

Sets the target number of pixels between notches. See the "notchTarget" [p. 62] property for details.

void QDial::setNotchesVisible (bool b) [virtual slot]

Sets whether the notches are shown to *b*. See the "notchesVisible" [p. 62] property for details.

void QDial::setPageStep (int)

Sets the current page step. See the "pageStep" [p. 62] property for details.

void QDial::setTracking (bool enable) [virtual slot]

Sets whether tracking is enabled to *enable*. See the "tracking" [p. 62] property for details.

void QDial::setValue (int) [virtual slot]

Sets the current dial value. See the "value" [p. 62] property for details.

void QDial::setWrapping (bool on) [virtual slot]

Sets whether wrapping is enabled to *on*. See the "wrapping" [p. 63] property for details.

void QDial::subtractLine () [slot]

Decrements the dial's value() by one lineStep().

void QDial::subtractPage () [slot]

Decrements the dial's value() by one pageStep() of steps.

bool QDial::tracking () const

Returns TRUE if tracking is enabled; otherwise returns FALSE. See the "tracking" [p. 62] property for details.

int QDial::value () const

Returns the current dial value. See the "value" [p. 62] property for details.

void QDial::valueChange () [virtual protected]

Reimplemented to ensure the display is correct and to emit the valueChanged(int) signal when appropriate.
Reimplemented from QRangeControl [p. 251].

void QDial::valueChanged (int value) [signal]

This signal is emitted whenever the dial's *value* changes. The frequency of this signal is influenced by setTracking().

bool QDial::wrapping () const

Returns TRUE if wrapping is enabled; otherwise returns FALSE. See the "wrapping" [p. 63] property for details.

Property Documentation

int lineStep

This property holds the current line step.
setLineStep() calls the virtual stepChange() function if the new line step is different from the previous setting.
See also QRangeControl::setSteps() [p. 250], pageStep [p. 62] and setRange() [p. 250].
Set this property's value with setLineStep() and get this property's value with lineStep().

int maxValue

This property holds the current maximum value.
When setting this property, the QDial::minValue is adjusted so that the range remains valid if necessary.
See also setRange() [p. 250].
Set this property's value with setMaxValue() and get this property's value with maxValue().

int minValue

This property holds the current minimum value.
When setting this property, the QDial::maxValue is adjusted so that the range remains valid if necessary.
See also setRange() [p. 250].
Set this property's value with setMinValue() and get this property's value with minValue().

int notchSize

This property holds the current notch size.

The notch size is in range control units, not pixels, and if possible it is a multiple of lineStep() that results in an on-screen notch size near notchTarget().

See also notchTarget [p. 62] and lineStep [p. 61].

Get this property's value with notchSize().

double notchTarget

This property holds the target number of pixels between notches.

The notch target is the number of pixels QDial attempts to put between each notch.

The actual size may differ from the target size.

Set this property's value with setNotchTarget() and get this property's value with notchTarget().

bool notchesVisible

This property holds whether the notches are shown.

If TRUE, the notches are shown. If FALSE (the default) notches are not shown.

Set this property's value with setNotchesVisible() and get this property's value with notchesVisible().

int pageStep

This property holds the current page step.

setPageStep() calls the virtual stepChange() function if the new page step is different from the previous setting.

See also stepChange() [p. 250].

Set this property's value with setPageStep() and get this property's value with pageStep().

bool tracking

This property holds whether tracking is enabled.

If TRUE (the default), tracking is enabled. This means that the arrow can be moved using the mouse; otherwise the arrow cannot be moved with the mouse.

Set this property's value with setTracking() and get this property's value with tracking().

int value

This property holds the current dial value.

This is guaranteed to be within the range QDial::minValue..QDial::maxValue.

See also minValue [p. 61] and maxValue [p. 61].

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

bool wrapping

This property holds whether wrapping is enabled.

If TRUE, wrapping is enabled. This means that the arrow can be turned around 360ř. Otherwise there is some space at the bottom of the dial which is skipped by the arrow.

This property's default is FALSE.

Set this property's value with setWrapping() and get this property's value with wrapping().

QFilePreview Class Reference

The QFilePreview class provides file previewing in QFileDialog.

`#include <qfiledialog.h>`

Public Members

- **QFilePreview** ()
- virtual void **previewUrl** (const QUrl & url)

Detailed Description

The QFilePreview class provides file previewing in QFileDialog.

This class is an abstract base class which is used to implement widgets that can display a preview of a file in a QFileDialog.

You must derive your preview widget from both QWidget and from this class. Then you must reimplement the previewUrl() function of this class, which is called by the file dialog if the preview of a URL should be shown.

See also QFileDialog::setPreviewMode() [Dialogs and Windows with Qt], QFileDialog::setContentsPreview() [Dialogs and Windows with Qt], QFileDialog::setInfoPreview() [Dialogs and Windows with Qt], QFileDialog::setInfoPreviewEnabled() [Dialogs and Windows with Qt], QFileDialog::setContentsPreviewEnabled() [Dialogs and Windows with Qt].

For an example of a preview widget see qt/examples/qdir/qdir.cpp.

See also Miscellaneous Classes.

Member Function Documentation

QFilePreview::QFilePreview ()

Constructs the QFilePreview.

void QFilePreview::previewUrl (const QUrl & url) [virtual]

This function is called by QFileDialog if a preview for the *url* should be shown. Reimplement this function to perform file/URL previews.

QFrame Class Reference

The QFrame class is the base class of widgets that can have a frame.

`#include <qframe.h>`

Inherits QWidget [p. 412].

Inherited by QGroupBox [p. 78], QScrollView [p. 259], QDockWindow [Dialogs and Windows with Qt], QGrid [Events, Actions, Layouts and Styles with Qt], QHBox [Events, Actions, Layouts and Styles with Qt], QLabel [p. 117], QLCDNumber [p. 125], QLineEdit [p. 132], QMenuBar [Dialogs and Windows with Qt], QPopupMenu [Dialogs and Windows with Qt], QProgressBar [p. 225], QSplitter [p. 306], QtTableView and QWidgetStack [p. 478].

Public Members

- **QFrame** (QWidget * parent = 0, const char * name = 0, WFlags f = 0)
- int **frameStyle** () const
- virtual void **setFrameStyle** (int style)
- int **frameWidth** () const
- QRect **contentsRect** () const
- enum **Shape** { NoFrame = 0, Box = 0x0001, Panel = 0x0002, WinPanel = 0x0003, HLine = 0x0004, VLine = 0x0005, StyledPanel = 0x0006, PopupPanel = 0x0007, MenuBarPanel = 0x0008, ToolBarPanel = 0x0009, MShape = 0x000f }
- enum **Shadow** { Plain = 0x0010, Raised = 0x0020, Sunken = 0x0030, MShadow = 0x00f0 }
- Shape **frameShape** () const
- void **setFrameShape** (Shape)
- Shadow **frameShadow** () const
- void **setFrameShadow** (Shadow)
- int **lineWidth** () const
- virtual void **setLineWidth** (int)
- int **margin** () const
- virtual void **setMargin** (int)
- int **midLineWidth** () const
- virtual void **setMidLineWidth** (int)
- QRect **frameRect** () const
- virtual void **setFrameRect** (const QRect &)

Properties

- QRect **contentsRect** — the rectangle inside the frame *(read only)*
- QRect **frameRect** — the frame rectangle

- Shadow **frameShadow** — the frame shadow value from the frame style
- Shape **frameShape** — the frame shape value from the frame style
- int **frameWidth** — the width of the frame that is drawn *(read only)*
- int **lineWidth** — the line width
- int **margin** — the width of the margin
- int **midLineWidth** — the width of the mid-line

Protected Members

- virtual void **paintEvent** (QPaintEvent * event)
- virtual void **resizeEvent** (QResizeEvent * e)
- virtual void **drawFrame** (QPainter * p)
- virtual void **drawContents** (QPainter *)
- virtual void **frameChanged** ()

Detailed Description

The QFrame class is the base class of widgets that can have a frame.

It draws a frame and calls a virtual function, drawContents(), to fill in the frame. This function is reimplemented by subclasses. There are also two other less useful functions: drawFrame() and frameChanged().

QPopupMenu uses this to "raise" the menu above the surrounding screen. QProgressBar has a "sunken" look. QLabel has a flat look. The frames of widgets such as these can be changed.

```
QLabel label(...);
label.setFrameStyle( QFrame::Panel | QFrame::Raised );
label.setLineWidth( 2 );

QProgressBar pbar(...);
label.setFrameStyle( QFrame::NoFrame );
```

The QFrame class can also be used directly for creating simple frames without any contents, although usually you would use a QHBox or QVBox because they automatically lay out the widgets you put inside the frame.

A frame widget has four attributes: frameStyle(), lineWidth(), midLineWidth(), and margin().

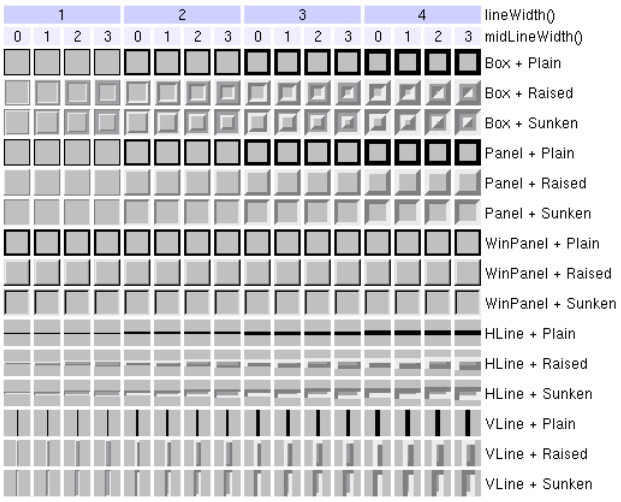
The frame style is specified by a frame shape and a shadow style. The frame shapes are NoFrame, Box, Panel, StyledPanel, PopupPanel, WinPanel, ToolBarPanel, MenuBarPanel, HLine and VLine; the shadow styles are Plain, Raised and Sunken.

The line width is the width of the frame border.

The mid-line width specifies the width of an extra line in the middle of the frame, which uses a third color to obtain a special 3D effect. Notice that a mid-line is only drawn for Box, HLine and VLine frames that are raised or sunken.

The margin is the gap between the frame and the contents of the frame.

This table shows the most useful combinations of styles and widths (and some rather useless ones):



See also Abstract Widget Classes.

Member Type Documentation

QFrame::Shadow

This enum type defines the 3D effect used for QFrame's frame. The currently defined effects are:

- `QFrame::Plain` - the frame and contents appear level with the surroundings
- `QFrame::Raised` - the frame and contents appear raised
- `QFrame::Sunken` - the frame and contents appear sunken
- `QFrame::MShadow` - internal; mask for the shadow

Shadow interacts with QFrame::Shape, the lineWidth() and the midLineWidth(). The picture of the frames in the class documentation may illustrate this better than words.

See also QFrame::Shape [p. 67], lineWidth [p. 72] and midLineWidth [p. 72].

QFrame::Shape

This enum type defines the shapes of a QFrame's frame. The currently defined shapes are:

- NoFrame - QFrame draws nothing
- Box - QFrame draws a box around its contents
- Panel - QFrame draws a panel such that the contents appear raised or sunken
- WinPanel - like Panel, but QFrame draws the 3D effects the way Microsoft Windows 95 (etc.) does
- ToolBarPanel - QFrame calls QStyle::drawToolBarPanel()
- MenuBarPanel - QFrame calls QStyle::drawMenuBarPanel()
- HLine - QFrame draws a horizontal line that frames nothing (useful as separator)
- VLine - QFrame draws a vertical line that frames nothing (useful as separator)
- StyledPanel - QFrame calls QStyle::drawPanel()
- PopupPanel - QFrame calls QStyle::drawPopupPanel()

When it does not call QStyle, Shape interacts with QFrame::Shadow, the lineWidth() and the midLineWidth() to create the total result. The picture of the frames in the class documentation may illustrate this better than words.

See also QFrame::Shadow [p. 67], QFrame::style() [p. 455] and QStyle::drawPrimitive() [Events, Actions, Layouts and Styles with Qt].

Member Function Documentation

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

Constructs a frame widget with frame style NoFrame and a 1-pixel frame width.

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

QRect QFrame::contentsRect () const

Returns the rectangle inside the frame. See the "contentsRect" [p. 71] property for details.

void QFrame::drawContents (QPainter *) [virtual protected]

Virtual function that draws the contents of the frame.

The QPainter is already open when you get it, and you must leave it open. Painter transformations are switched off on entry. If you transform the painter, remember to take the frame into account and reset transformation before returning.

This function is reimplemented by subclasses that draw something inside the frame. It should draw only inside contentsRect(). The default function does nothing.

See also contentsRect [p. 71] and QPainter::setClipRect() [Graphics with Qt].

Reimplemented in QLabel, QLCDNumber, QMenuBar and QPopupMenu.

void QFrame::drawFrame (QPainter * p) [virtual protected]

Draws the frame using the painter *p* and the current frame attributes and color group. The rectangle inside the frame is not affected.

This function is virtual, but in general you do not need to reimplement it. If you do, note that the QPainter is already open and must remain open.

See also frameRect [p. 71], contentsRect [p. 71], drawContents() [p. 68], frameStyle() [p. 69] and palette [p. 468].

void QFrame::frameChanged () [virtual protected]

Virtual function that is called when the frame style, line width or mid-line width changes.

This function can be reimplemented by subclasses that need to know when the frame attributes change.

The default implementation calls update().

QRect QFrame::frameRect () const

Returns the frame rectangle. See the "frameRect" [p. 71] property for details.

Shadow QFrame::frameShadow () const

Returns the frame shadow value from the frame style. See the "frameShadow" [p. 71] property for details.

Shape QFrame::frameShape () const

Returns the frame shape value from the frame style. See the "frameShape" [p. 72] property for details.

int QFrame::frameStyle () const

Returns the frame style.

The default value is QFrame::NoFrame.

See also setFrameStyle() [p. 70], frameShape [p. 72] and frameShadow [p. 71].

Example: scrollview/scrollview.cpp.

int QFrame::frameWidth () const

Returns the width of the frame that is drawn. See the "frameWidth" [p. 72] property for details.

int QFrame::lineWidth () const

Returns the line width. See the "lineWidth" [p. 72] property for details.

int QFrame::margin () const

Returns the width of the margin. See the "margin" [p. 72] property for details.

int QFrame::midLineWidth () const

Returns the width of the mid-line. See the "midLineWidth" [p. 72] property for details.

void QFrame::paintEvent (QPaintEvent * event) [virtual protected]

Processes the paint event *event*.

Paints the frame and the contents.

Opens the painter on the frame and calls drawFrame(), then drawContents().

Examples: life/life.cpp and qfd/fontdisplayer.cpp.

Reimplemented from QWidget [p. 441].

Reimplemented in QtTableView.

void QFrame::resizeEvent (QResizeEvent * e) [virtual protected]

Processes the resize event *e*.

Adjusts the frame rectangle for the resized widget. The frame rectangle is elastic, and the surrounding area is static.

The resulting frame rectangle may be null or invalid. You can use setMinimumSize() to avoid that possibility.

Nothing is done if the frame rectangle is a null rectangle already.

Example: life/life.cpp.

Reimplemented from QWidget [p. 445].

void QFrame::setFrameRect (const QRect &) [virtual]

Sets the frame rectangle. See the "frameRect" [p. 71] property for details.

void QFrame::setFrameShadow (Shadow)

Sets the frame shadow value from the frame style. See the "frameShadow" [p. 71] property for details.

void QFrame::setFrameShape (Shape)

Sets the frame shape value from the frame style. See the "frameShape" [p. 72] property for details.

void QFrame::setFrameStyle (int style) [virtual]

Sets the frame style to *style*.

The *style* is the bitwise OR between a frame shape and a frame shadow style. See the illustration in the class documentation.

The frame shapes are:

- NoFrame draws nothing. Naturally, you should not specify a shadow style if you use this.
- Box draws a rectangular box. The contents appear to be level with the surrounding screen, but the border itself may be raised or sunken.
- Panel draws a rectangular panel that can be raised or sunken.
- StyledPanel draws a rectangular panel with a look that depends on the current GUI style. It can be raised or sunken.
- PopupPanel is used to draw a frame suitable for popup windows. Its look also depends on the current GUI style, usually the same as StyledPanel.
- ToolBarPanel is used to draw a frame suitable for tool bars. The look depends upon the current GUI style.
- MenuBarPanel is used to draw a frame suitable for menu bars. The look depends upon the current GUI style.
- WinPanel draws a rectangular panel that can be raised or sunken like those in Windows 95. Specifying this shape sets the line width to 2 pixels. WinPanel is provided for compatibility. For GUI style independence we recommend using StyledPanel instead.
- HLine draws a horizontal line (vertically centered).
- VLine draws a vertical line (horizontally centered).

The shadow styles are:

- Plain draws using the palette foreground color (without any 3D effect).
- Raised draws a 3D raised line using the light and dark colors of the current color group.
- Sunken draws a 3D sunken line using the light and dark colors of the current color group.

If a mid-line width greater than 0 is specified, an additional line is drawn for Raised or Sunken Box, HLine, and VLine frames. The mid-color of the current color group is used for drawing middle lines.

See also Illustration [p. 66], frameStyle() [p. 69], colorGroup [p. 460] and QColorGroup [Graphics with Qt].

Examples: cursor/cursor.cpp, layout/layout.cpp, listboxcombo/listboxcombo.cpp, rangecontrols/rangecontrols.cpp, scrollview/scrollview.cpp, tabdialog/tabdialog.cpp and tictac/tictac.cpp.

void QFrame::setLineWidth (int) [virtual]

Sets the line width. See the "lineWidth" [p. 72] property for details.

void QFrame::setMargin (int) [virtual]

Sets the width of the margin. See the "margin" [p. 72] property for details.

void QFrame::setMidLineWidth (int) [virtual]

Sets the width of the mid-line. See the "midLineWidth" [p. 72] property for details.

Property Documentation

QRect contentsRect

This property holds the rectangle inside the frame.

Get this property's value with contentsRect().

See also frameRect [p. 71] and drawContents() [p. 68].

QRect frameRect

This property holds the frame rectangle.

The frame rectangle is the rectangle the frame is drawn in. By default, this is the entire widget. Setting this property does *not* cause a widget update.

If this property is set to a null rectangle (for example `QRect(0, 0, 0, 0)`), then the frame rectangle is equivalent to the widget rectangle.

See also contentsRect [p. 71].

Set this property's value with setFrameRect() and get this property's value with frameRect().

Shadow frameShadow

This property holds the frame shadow value from the frame style.

Set this property's value with setFrameShadow() and get this property's value with frameShadow().

See also frameStyle() [p. 69] and frameShape [p. 72].

Shape frameShape

This property holds the frame shape value from the frame style.

Set this property's value with setFrameShape() and get this property's value with frameShape().

See also frameStyle() [p. 69] and frameShadow [p. 71].

int frameWidth

This property holds the width of the frame that is drawn.

Note that the frame width depends on the frame style, not only the line width and the mid-line width. For example, the style NoFrame always has a frame width 0, whereas the style Panel has a frame width equivalent to the line width. The frame width also includes the margin.

See also lineWidth [p. 72], midLineWidth [p. 72], frameStyle() [p. 69] and margin [p. 72].

Get this property's value with frameWidth().

int lineWidth

This property holds the line width.

Note that the *total* line width for HLine and VLine is given by frameWidth(), not lineWidth().

The default value is 1.

See also midLineWidth [p. 72] and frameWidth [p. 72].

Set this property's value with setLineWidth() and get this property's value with lineWidth().

int margin

This property holds the width of the margin.

The margin is the distance between the innermost pixel of the frame and he outermost pixel of contentsRect(). It is included in frameWidth().

The margin is filled according to backgroundMode().

The default value is 0.

See also margin [p. 72], lineWidth [p. 72] and frameWidth [p. 72].

Set this property's value with setMargin() and get this property's value with margin().

int midLineWidth

This property holds the width of the mid-line.

The default value is 0.

See also lineWidth [p. 72] and frameWidth [p. 72].

Set this property's value with setMidLineWidth() and get this property's value with midLineWidth().

QGridView Class Reference

The QGridView class provides an abstract base for fixed-size grids.

`#include <qgridview.h>`

Inherits QScrollView [p. 259].

Public Members

- **QGridView** (QWidget * parent = 0, const char * name = 0, WFlags f = 0)
- **~QGridView** ()
- int **numRows** () const
- virtual void **setNumRows** (int)
- int **numCols** () const
- virtual void **setNumCols** (int)
- int **cellWidth** () const
- virtual void **setCellWidth** (int)
- int **cellHeight** () const
- virtual void **setCellHeight** (int)
- QRect **cellRect** () const
- QRect **cellGeometry** (int row, int column)
- QSize **gridSize** () const
- int **rowAt** (int y) const
- int **columnAt** (int x) const
- void **repaintCell** (int row, int column, bool erase = TRUE)
- void **updateCell** (int row, int column)
- void **ensureCellVisible** (int row, int column)

Properties

- int **cellHeight** — the height of a grid row
- int **cellWidth** — the width of a grid column
- int **numCols** — the number of columns in the grid
- int **numRows** — the number of rows in the grid

Protected Members

- virtual void **paintCell** (QPainter * p, int row, int col)
- virtual void **paintEmptyArea** (QPainter * p, int cx, int cy, int cw, int ch)
- virtual void **dimensionChange** (int oldNumRows, int oldNumCols)

Detailed Description

The QGridView class provides an abstract base for fixed-size grids.

A grid view consists of a number of abstract cells organized in rows and columns. The cells have a fixed size and are identified with a row index and a column index. The top-left cell is in row 0, column 0. The bottom-right cell is in row numRows()-1, column numCols()-1.

You can define numRows, numCols, cellWidth and cellHeight. Reimplement the pure virtual function paintCell() to draw the content of a cell.

With ensureCellVisible(), you can ensure a certain cell is visible. With rowAt() and columnAt() you can find a cell based on the given x- and y-coordinates.

If you need to monitor changes to the grid's dimensions (i.e. when numRows or numCols is changed), reimplement the dimensionChange() change handler.

Note: the row, column indices are always given in the order, row (vertical offset) then column (horizontal offset). This order is the opposite of all pixel operations, which are given in the order x (horizontal offset), y (vertical offset).

QGridView is a very simple abstract class based on QScrollView. It is designed to simplify the task of drawing many cells of the same size in a potentially scrollable canvas. If you need rows and columns in different sizes, use a QTable instead. If you need a simple list of items, use a QListBox. If you need to present hierachical data use a QListView, and if you need random objects at random positions, consider using either a QIconView or a QCanvas.

See also Abstract Widget Classes.

Member Function Documentation

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

Constructs a grid view.

The *parent*, *name* and widget flag, *f*, arguments are passed to the QScrollView constructor.

QGridView::~QGridView ()

Destroys the grid view.

QRect QGridView::cellGeometry (int row, int column)

Returns the geometry of cell (*row*, *column*) in the content coordinate system.

See also cellRect() [p. 74].

int QGridView::cellHeight () const

Returns the height of a grid row. See the "cellHeight" [p. 76] property for details.

QRect QGridView::cellRect () const

Returns the geometry of a cell in a cell's coordinate system. This is a convenience function useful in paintCell(). It is equivalent to QRect(0, 0, cellWidth(), cellHeight()).

See also cellGeometry() [p. 74].

int QGridView::cellWidth () const

Returns the width of a grid column. See the "cellWidth" [p. 77] property for details.

int QGridView::columnAt (int x) const

Returns the number of the column at position *x*. *x* must be given in content coordinates.

See also rowAt() [p. 76].

void QGridView::dimensionChange (int oldNumRows, int oldNumCols) [virtual protected]

This change handler is called whenever any of the grid's dimensions changes. *oldNumRows* and *oldNumCols* contain the old dimensions, numRows() and numCols() contain the new dimensions.

void QGridView::ensureCellVisible (int row, int column)

Ensure cell (*row*, *column*) is visible, scrolling the grid view if necessary.

QSize QGridView::gridSize () const

Returns the size of the grid in pixels.

int QGridView::numCols () const

Returns the number of columns in the grid. See the "numCols" [p. 77] property for details.

int QGridView::numRows () const

Returns the number of rows in the grid. See the "numRows" [p. 77] property for details.

void QGridView::paintCell (QPainter * p, int row, int col) [virtual protected]

This pure virtual function is called to paint the single cell at (*row*, *col*) using painter *p*. The painter must be open when paintCell() is called and must remain open.

The coordinate system is translated so that the origin is at the top-left corner of the cell to be painted, i.e. *cell* coordinates. Do not scale or shear the coordinate system (or if you do, restore the transformation matrix before you return).

The painter is not clipped by default in order to get maximum efficiency. If you want clipping, use

```
p->setClipRect( cellRect(), QPainter::ClipPainter );
//... your drawing code
p->setClipping( FALSE );
```

void QGridView::paintEmptyArea (QPainter * p, int cx, int cy, int cw, int ch) [virtual protected]

This function fills the *cw* pixels wide and *ch* pixels high rectangle starting at position (*cx*, *cy*) with the background color using the painter *p*.

paintEmptyArea() is invoked by drawContents() to erase or fill unused areas.

void QGridView::repaintCell (int row, int column, bool erase = TRUE)

Repaints cell (*row*, *column*).

If *erase* is TRUE, Qt erases the area of the cell before the paintCell() call; otherwise no erasing takes place.

See also QWidget::repaint() [p. 443].

int QGridView::rowAt (int y) const

Returns the number of the row at position *y*. *y* must be given in content coordinates.

See also columnAt() [p. 75].

void QGridView::setCellHeight (int) [virtual]

Sets the height of a grid row. See the "cellHeight" [p. 76] property for details.

void QGridView::setCellWidth (int) [virtual]

Sets the width of a grid column. See the "cellWidth" [p. 77] property for details.

void QGridView::setNumCols (int) [virtual]

Sets the number of columns in the grid. See the "numCols" [p. 77] property for details.

void QGridView::setNumRows (int) [virtual]

Sets the number of rows in the grid. See the "numRows" [p. 77] property for details.

void QGridView::updateCell (int row, int column)

Updates cell (*row*, *column*).

See also QWidget::update() [p. 456].

Property Documentation

int cellHeight

This property holds the height of a grid row.

All rows in a grid view have the same height.

See also cellWidth [p. 77].

Set this property's value with setCellHeight() and get this property's value with cellHeight().

int cellWidth

This property holds the width of a grid column.

All columns in a grid view have the same width.

See also cellHeight [p. 76].

Set this property's value with setCellWidth() and get this property's value with cellWidth().

int numCols

This property holds the number of columns in the grid.

Set this property's value with setNumCols() and get this property's value with numCols().

See also numRows [p. 77].

int numRows

This property holds the number of rows in the grid.

Set this property's value with setNumRows() and get this property's value with numRows().

See also numCols [p. 77].

QGroupBox Class Reference

The QGroupBox widget provides a group box frame with a title.

`#include <qgroupbox.h>`

Inherits QFrame [p. 65].

Inherited by QButtonGroup [p. 14], QHGroupBox [p. 84] and QVGroupBox [p. 407].

Public Members

- **QGroupBox** (QWidget * parent = 0, const char * name = 0)
- **QGroupBox** (const QString & title, QWidget * parent = 0, const char * name = 0)
- **QGroupBox** (int strips, Orientation orientation, QWidget * parent = 0, const char * name = 0)
- **QGroupBox** (int strips, Orientation orientation, const QString & title, QWidget * parent = 0, const char * name = 0)
- virtual void **setColumnLayout** (int strips, Orientation direction)
- QString **title** () const
- virtual void **setTitle** (const QString &)
- int **alignment** () const
- virtual void **setAlignment** (int)
- int **columns** () const
- void **setColumns** (int)
- Orientation **orientation** () const
- void **setOrientation** (Orientation)
- int **insideMargin** () const
- int **insideSpacing** () const
- void **setInsideMargin** (int m)
- void **setInsideSpacing** (int s)
- void **addSpace** (int size)

Properties

- Alignment **alignment** — the alignment of the group box title
- int **columns** — the number of columns or rows (depending on \l orientation) in the group box
- Orientation **orientation** — the current orientation of the group box
- QString **title** — the group box title text

Detailed Description

The QGroupBox widget provides a group box frame with a title.

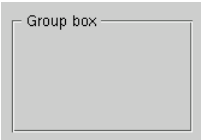
A group box provides a frame, a title and a keyboard shortcut, and displays various other widgets inside itself. The title is on top, the keyboard shortcut moves keyboard focus to one of the group box's child widgets, and the child widgets are arranged in an array inside the frame.

The simplest way to use it is to create a group box with the desired number of columns (or rows) and orientation, and then just create widgets with the group box as parent.

However, it is also possible to change the orientation() and number of columns() after construction, or to ignore all the automatic layout support and manage all that yourself. You can add 'empty' spaces to the group box with addSpace().

QGroupBox also lets you set the title() (normally set in the constructor) and the title's alignment().

You can change the spacing used by the group box with setInsideMargin() and setInsideSpacing().



See also QButtonGroup [p. 14], Widget Appearance and Style, Layout Management and Organizers.

Member Function Documentation

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

Constructs a group box widget with no title.

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

This constructor does not do automatic layout.

QGroupBox::QGroupBox (const QString & title, QWidget * parent = 0, const char * name = 0)

Constructs a group box with the title *title*.

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

This constructor does not do automatic layout.

QGroupBox::QGroupBox (int strips, Orientation orientation, QWidget * parent = 0, const char * name = 0)

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

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

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

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

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

void QGroupBox::addSpace (int size)

Adds an empty cell at the next free position. If *size* is greater than 0, the empty cell has a fixed height or width. If the group box is oriented horizontally, the empty cell has a fixed height; if oriented vertically, it has a fixed width.

Use this method to separate the widgets in the group box or to skip the next free cell. For performance reasons, call this method after calling setColumnLayout() or by changing the QGroupBox::columns or QGroupBox::orientation properties. It is generally a good idea to call these methods first (if needed at all), and insert the widgets and spaces afterwards.

int QGroupBox::alignment () const

Returns the alignment of the group box title. See the "alignment" [p. 81] property for details.

int QGroupBox::columns () const

Returns the number of columns or rows (depending on orientation) in the group box. See the "columns" [p. 82] property for details.

int QGroupBox::insideMargin () const

Returns the width of the blank spacing between the items in the group and the frame of the group.

Only applies if the group box has a defined orientation.

The default is about 11.

See also setInsideMargin() [p. 81] and orientation [p. 82].

int QGroupBox::insideSpacing () const

Returns the width of the blank spacing between each of the items in the group.

Only applies if the group box has a defined orientation.

The default is about 5.

See also setInsideSpacing() [p. 81] and orientation [p. 82].

Orientation QGroupBox::orientation () const

Returns the current orientation of the group box. See the "orientation" [p. 82] property for details.

void QGroupBox::setAlignment (int) [virtual]

Sets the alignment of the group box title. See the "alignment" [p. 81] property for details.

void QGroupBox::setColumnLayout (int strips, Orientation direction) [virtual]

Changes the layout of the group box. This function is useful only in combination with the default constructor that does not take any layout information. This function will put all existing children in the new layout. It is not good Qt programming style to call this function after children have been inserted. Sets the number of columns or rows to be *strips*, depending on *direction*.

See also orientation [p. 82] and columns [p. 82].

void QGroupBox::setColumns (int)

Sets the number of columns or rows (depending on orientation) in the group box. See the "columns" [p. 82] property for details.

void QGroupBox::setInsideMargin (int m)

Sets the the width of the blank spacing between each of the items in the group to *m* pixels.

See also insideSpacing() [p. 80].

void QGroupBox::setInsideSpacing (int s)

Sets the width of the blank spacing between each of the items in the group to *s* pixels.

void QGroupBox::setOrientation (Orientation)

Sets the current orientation of the group box. See the "orientation" [p. 82] property for details.

void QGroupBox::setTitle (const QString &) [virtual]

Sets the group box title text. See the "title" [p. 82] property for details.

QString QGroupBox::title () const

Returns the group box title text. See the "title" [p. 82] property for details.

Property Documentation

Alignment alignment

This property holds the alignment of the group box title.

The title is always placed on the upper frame line; however, the horizontal alignment can be specified by the alignment parameter.

The alignment is one of the following flags:

- AlignAuto aligns the title accroding to the language, usually left.
- AlignLeft aligns the title text to the left.
- AlignRight aligns the title text to the right.
- AlignHCenter aligns the title text centered.

The default alignment is AlignAuto.

See also Qt::AlignmentFlags [Additional Functionality with Qt].

Set this property's value with setAlignment() and get this property's value with alignment().

int columns

This property holds the number of columns or rows (depending on orientation) in the group box.

Usually it is not a good idea to set this property because it is slow (it does a complete layout). It is better to set the number of columns directly in the constructor.

Set this property's value with setColumns() and get this property's value with columns().

Orientation orientation

This property holds the current orientation of the group box.

A horizontal group box arranges it's children in columns, while a vertical group box arranges them in rows. Thus, a horizontal group box with only one column will arrange the children vertically in that column.

Usually it is not a good idea to set this property because it is slow (it does a complete layout). It is better to set the orientation directly in the constructor.

Set this property's value with setOrientation() and get this property's value with orientation().

QString title

This property holds the group box title text.

The group box title text will have a focus-change keyboard accelerator if the title contains &, followed by a letter.

```
g->setTitle( "&User information" );
```

This produces "User information" with the U underlined; Alt+U moves the keyboard focus to the group box.

There is no default title text.

Set this property's value with setTitle() and get this property's value with title().

QHButtonGroup Class Reference

The QHButtonGroup widget organizes QButton widgets in a group with one horizontal row.

`#include <qhbuttongroup.h>`

Inherits QButtonGroup [p. 14].

Public Members

- **QHButtonGroup** (QWidget * parent = 0, const char * name = 0)
- **QHButtonGroup** (const QString & title, QWidget * parent = 0, const char * name = 0)
- **~QHButtonGroup** ()

Detailed Description

The QHButtonGroup widget organizes QButton widgets in a group with one horizontal row.

QHButtonGroup is a convenience class that offers a thin layer on top of QButtonGroup. Think of it as a QHBox that offers a frame with a title and is specifically designed for buttons.

See also Widget Appearance and Style, Layout Management and Organizers.

Member Function Documentation

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

Constructs a horizontal button group with no title.

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

QHButtonGroup::QHButtonGroup (const QString & title, QWidget * parent = 0, const char * name = 0)

Constructs a horizontal button group with the title *title*.

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

QHButtonGroup::~QHButtonGroup ()

Destroys the horizontal button group, deleting its child widgets.

QHGroupBox Class Reference

The QHGroupBox widget organizes widgets in a group with one horizontal row.

`#include <qhgroupbox.h>`

Inherits QGroupBox [p. 78].

Public Members

- **QHGroupBox** (QWidget * parent = 0, const char * name = 0)
- **QHGroupBox** (const QString & title, QWidget * parent = 0, const char * name = 0)
- **~QHGroupBox** ()

Detailed Description

The QHGroupBox widget organizes widgets in a group with one horizontal row.

QHGroupBox is a convenience class that offers a thin layer on top of QGroupBox. Think of it as a QHBox that offers a frame with a title.

See also Widget Appearance and Style, Layout Management and Organizers.

Member Function Documentation

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

Constructs a horizontal group box with no title.

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

QHGroupBox::QHGroupBox (const QString & title, QWidget * parent = 0, const char * name = 0)

Constructs a horizontal group box with the title *title*.

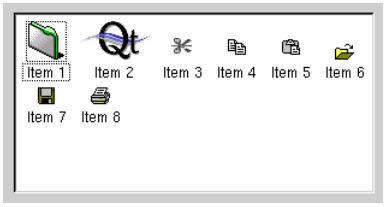
The *parent* and *name* arguments are passed to the QWidget constructor.

QHGroupBox::~QHGroupBox ()

Destroys the horizontal group box, deleting its child widgets.

IconView Module

The icon view module provides a powerful visualization widget dubbed QIconView. API and feature-wise it is similar to QListView and QListBox. It contains optionally labelled pixmap items that the user can select, drag around, rename, delete and more.



Please see the class documentation for details.

QIconView Class Reference

The QIconView class provides an area with movable labelled icons.

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

`#include <qiconview.h>`

Inherits QScrollView [p. 259].

Public Members

- enum **SelectionMode** { Single = 0, Multi, Extended, NoSelection }
- enum **Arrangement** { LeftToRight = 0, TopToBottom }
- enum **ResizeMode** { Fixed = 0, Adjust }
- enum **ItemTextPos** { Bottom = 0, Right }
- **QIconView** (QWidget * parent = 0, const char * name = 0, WFlags f = 0)
- virtual **~QIconView** ()
- virtual void **insertItem** (QIconViewItem * item, QIconViewItem * after = 0L)
- virtual void **takeItem** (QIconViewItem * item)
- int **index** (const QIconViewItem * item) const
- QIconViewItem * **firstItem** () const
- QIconViewItem * **lastItem** () const
- QIconViewItem * **currentItem** () const
- virtual void **setCurrentItem** (QIconViewItem * item)
- virtual void **setSelected** (QIconViewItem * item, bool s, bool cb = FALSE)
- uint **count** () const
- virtual void **setSelectionMode** (SelectionMode m)
- SelectionMode **selectionMode** () const
- QIconViewItem * **findItem** (const QPoint & pos) const
- QIconViewItem * **findItem** (const QString & text, ComparisonFlags compare = BeginsWith) const
- virtual void **selectAll** (bool select)
- virtual void **clearSelection** ()
- virtual void **invertSelection** ()
- virtual void **repaintItem** (QIconViewItem * item)
- void **ensureItemVisible** (QIconViewItem * item)
- QIconViewItem * **findFirstVisibleItem** (const QRect & r) const
- QIconViewItem * **findLastVisibleItem** (const QRect & r) const
- virtual void **clear** ()
- virtual void **setGridX** (int rx)
- virtual void **setGridY** (int ry)
- int **gridX** () const

- int **gridY** () const
- virtual void **setSpacing** (int sp)
- int **spacing** () const
- virtual void **setItemTextPos** (ItemTextPos pos)
- ItemTextPos **itemTextPos** () const
- virtual void **setItemTextBackground** (const QBrush & b)
- QBrush **itemTextBackground** () const
- virtual void **setArrangement** (Arrangement am)
- Arrangement **arrangement** () const
- virtual void **setResizeMode** (ResizeMode am)
- ResizeMode **resizeMode** () const
- virtual void **setMaxItemWidth** (int w)
- int **maxItemWidth** () const
- virtual void **setMaxItemTextLength** (int w)
- int **maxItemTextLength** () const
- virtual void **setAutoArrange** (bool b)
- bool **autoArrange** () const
- virtual void **setShowToolTips** (bool b)
- bool **showToolTips** () const
- void **setSorting** (bool sort, bool ascending = TRUE)
- bool **sorting** () const
- bool **sortDirection** () const
- virtual void **setItemsMovable** (bool b)
- bool **itemsMovable** () const
- virtual void **setWordWrapIconText** (bool b)
- bool **wordWrapIconText** () const
- virtual void **sort** (bool ascending = TRUE)
- bool **isRenaming** () const

Public Slots

- virtual void **arrangeItemsInGrid** (const QSize & grid, bool update = TRUE)
- virtual void **arrangeItemsInGrid** (bool update = TRUE)

Signals

- void **selectionChanged** ()
- void **selectionChanged** (QIconViewItem * item)
- void **currentChanged** (QIconViewItem * item)
- void **clicked** (QIconViewItem * item)
- void **clicked** (QIconViewItem * item, const QPoint & pos)
- void **pressed** (QIconViewItem * item)
- void **pressed** (QIconViewItem * item, const QPoint & pos)
- void **doubleClicked** (QIconViewItem * item)
- void **returnPressed** (QIconViewItem * item)
- void **rightButtonClicked** (QIconViewItem * item, const QPoint & pos)
- void **rightButtonPressed** (QIconViewItem * item, const QPoint & pos)
- void **mouseButtonPressed** (int button, QIconViewItem * item, const QPoint & pos)

- void **mouseButtonClicked** (int button, QIconViewItem * item, const QPoint & pos)
- void **contextMenuRequested** (QIconViewItem * item, const QPoint & pos)
- void **dropped** (QDropEvent * e, const QValueList<QIconDragItem> & lst)
- void **moved** ()
- void **onItem** (QIconViewItem * item)
- void **onViewport** ()
- void **itemRenamed** (QIconViewItem * item, const QString & name)
- void **itemRenamed** (QIconViewItem * item)

Properties

- Arrangement **arrangement** — the arrangement mode of the icon view
- bool **autoArrange** — whether the icon view rearranges its items when a new item is inserted
- uint **count** — the number of items in the icon view *(read only)*
- int **gridX** — the horizontal grid of the icon view
- int **gridY** — the vertical grid of the icon view
- QBrush **itemTextBackground** — the brush that should be used when drawing the background of an item's text
- ItemTextPos **itemTextPos** — the position where the text of each item is drawn
- bool **itemsMovable** — whether the user is allowed to move items around in the icon view
- int **maxItemTextLength** — the maximum length (in characters) that an item's text may have
- int **maxItemWidth** — the maximum width that an item may have
- ResizeMode **resizeMode** — the resize mode of the icon view
- SelectionMode **selectionMode** — the selection mode of the icon view
- bool **showToolTips** — whether the icon view will display a tool tip with the complete text for any truncated item text
- bool **sortDirection** — whether the sort direction for inserting new items is ascending; *(read only)*
- bool **sorting** — whether the icon view sorts on insertion *(read only)*
- int **spacing** — the space in pixels between icon view items
- bool **wordWrapIconText** — whether the item text will be word-wrapped if it is too long

Protected Members

- virtual void **drawRubber** (QPainter * p)
- virtual QDragObject * **dragObject** ()
- virtual void **startDrag** ()
- virtual void **insertInGrid** (QIconViewItem * item)
- virtual void **drawBackground** (QPainter * p, const QRect & r)
- void **emitSelectionChanged** (QIconViewItem * i = 0)
- QIconViewItem * **makeRowLayout** (QIconViewItem * begin, int & y, bool & changed)

Protected Slots

- virtual void **doAutoScroll** ()
- virtual void **adjustItems** ()
- virtual void **slotUpdate** ()

Detailed Description

The QIconView class provides an area with movable labelled icons.

The QIconView can display and manage a grid or other 2D layout of labelled icons. Each labelled icon is a QIconViewItem. Items (QIconViewItems) can be added or deleted at any time; items can be moved within the QIconView. Single or multiple items can be selected. Items can be renamed in-place. QIconView also supports drag and drop.

Each item contains a label string, a pixmap or picture (the icon itself) and optionally an index key. The index key is used for sorting the items and defaults to the label string. The label string can be displayed below or to the right of the icon (see ItemTextPos).

The simplest way to create a QIconView is to create a QIconView object and create some QIconViewItems with the QIconView as their parent, set the icon view's geometry and show it. Below is an example of how such code might look:

```
QIconView *iv = new QIconView( this );
QDir dir( path, "*.xpm" );
for ( uint i = 0; i < dir.count(); i++ ) {
    (void) new QIconViewItem( iv, dir[i], QPixmap( path + dir[i] ) );
}
iv->resize( 600, 400 );
iv->show();
```

The QIconViewItem call passes a pointer to the QIconView we wish to populate followed by the label text and a QPixmap.

When an item is inserted the QIconView allocates a position for it. The default arrangement is LeftToRight — QIconView fills up the *left-most* column from top to bottom, then moves one column *right* and fills that from top to bottom and so on. The arrangement can be modified with any of the following approaches:

- Call setArrangement(), e.g. with TopToBottom which will fill the *top-most* row from left to right, then moves one row *down* and fills that row from left to right and so on.
- Construct each QIconViewItem using a constructor which allows you to specify which item the new one is to follow.
- Call setSorting() or sort() to sort the items.

Items which are *selectable* may be selected depending on the SelectionMode (default is Single). Because QIconView offers multiple selection it has to display keyboard focus and selection state separately. Therefore there are functions to set the selection state of an item (setSelected()) and to select which item displays keyboard focus (setCurrentItem()). When multiple items may be selected the icon view provides a rubberband, too.

When in-place renaming is enabled (it is disabled by default), the user may change the item's label. They do this by selecting the item (single clicking it or navigating to it with the arrow keys), then single clicking it (or pressing F2), and entering their text. If no key has been set with QIconViewItem::setKey() the new text will also serve as the key. (See QIconViewItem::setRenameEnabled().)

QIconView offers functions similar to QListView and QListBox, such as takeItem(), clearSelection(), setSelected(), setCurrentItem(), currentItem() and many more.

Because the internal structure used to store the icon view items is linear (a double-linked list), no iterator class is needed to iterate over all the items. Instead we iterate by getting the first item from the *icon view* and then each subsequent (QIconViewItem::nextItem()) from each *item* in turn:

```
for ( QIconViewItem *item = iv->firstItem(); item; item = item->nextItem() )
    do_something( item );
```

QIconView supports the drag and drop of items within the QIconView itself. It also supports the drag and drop of items out of or into the QIconView and drag and drop onto items themselves. The drag and drop of items outside

the QIconView can be achieved in a simple way with basic functionality, or in a more sophisticated way which provides more power and control.

The simple approach to dragging items out of the icon view is to subclass QIconView and reimplement QIconView::dragObject().

```
QDragObject *MyIconView::dragObject()
{
  return new QTextDrag( currentItem()->text(), this );
}
```

In this example we create a QTextDrag object, (derived from QDragObject), containing the item's label and return it as the drag object. We could just as easily have created a QImageDrag from the item's pixmap and returned that instead.

QIconViews and their QIconViewItems can also be the targets of drag and drops. To make the QIconView itself able to accept drops connect to the dropped() signal. When a drop occurs this signal will be emitted with a QDragEvent and a QValueList of QIconDragItems. To make a QIconViewItem into a drop target subclass QIconViewItem and reimplement QIconViewItem::acceptDrop() and QIconViewItem::dropped().

```
bool MyIconViewItem::acceptDrop( const QMimeSource *mime ) const
{
    if ( mime->provides( "text/plain" ) )
        return TRUE;
    return FALSE;
}

void MyIconViewItem::dropped( QDropEvent *evt, const QValueList& )
{
    QString label;
    if ( QTextDrag::decode( evt, label ) )
        setText( label );
}
```

See iconview/simple_dd/main.h and iconview/simple_dd/main.cpp for a simple drag and drop example which demonstrates drag and drop between a QIconView and a QListBox.

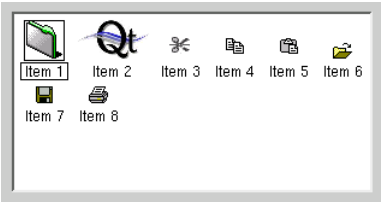
If you want to use extended drag-and-drop or have drag shapes drawn you have to take a more sophisticated approach.

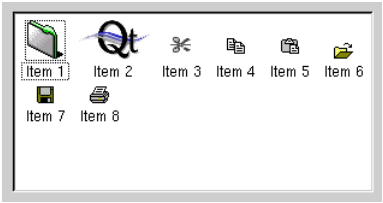
The first part is starting drags — you should use a QIconDrag (or a class derived from it) for the drag object. In dragObject() create the drag object, populate it with QIconDragItems and return it. Normally such a drag should offer each selected item's data. So in dragObject() you should iterate over all the items, and create a QIconDragItem for each selected item, and append these items with QIconDrag::append() to the QIconDrag object. You can use QIconDragItem::setData() to set the data of each item that should be dragged. If you want to offer the data in additional mime-types, it's best to use a class derived from QIconDrag, which implements additional encoding and decoding functions.

When a drag enters the icon view, there is little to do. Simply connect to the dropped() signal and reimplement QIconViewItem::acceptDrop() and QIconViewItem::dropped(). If you've used a QIconDrag (or a subclass of it) the second argument to the dropped signal contains a QValueList of QIconDragItems — you can access their data by calling QIconDragItem::data.

For an example implementation of complex drag-and-drop look at the qfileiconview example (qt/examples/qfileiconview).

See also QIconViewItem::setDragEnabled() [p. 114], QIconViewItem::setDropEnabled() [p. 114], QIconViewItem::acceptDrop() [p. 109], QIconViewItem::dropped() [p. 110] and Advanced Widgets.





Member Type Documentation

QIconView::Arrangement

This enum type determines in which direction the items flow when the view runs out of space.

- `QIconView::LeftToRight` - Items which don't fit onto the view go further down (you get a vertical scrollbar)
- `QIconView::TopToBottom` - Items which don't fit onto the view go further right (you get a horizontal scrollbar)

QIconView::ItemTextPos

This enum type specifies the position of the item text in relation to the icon.

- `QIconView::Bottom` - The text is drawn below the icon.
- `QIconView::Right` - The text is drawn to the right of the icon.

QIconView::ResizeMode

This enum type is used to tell QIconView how it should treat the positions of its icons when the widget is resized. The currently defined modes are:

- `QIconView::Fixed` - The icons' positions are not changed.
- `QIconView::Adjust` - The icons' positions are adjusted to be within the new geometry, if possible.

QIconView::SelectionMode

This enumerated type is used by QIconView to indicate how it reacts to selection by the user. It has four values:

- `QIconView::Single` - When the user selects an item, any already-selected item becomes unselected and the user cannot unselect the selected item. This means that the user can never clear the selection. (The application programmer can, using QIconView::clearSelection().)
- `QIconView::Multi` - When the user selects an item, e.g. by navigating to it with the keyboard arrow keys or by clicking it, the selection status of that item is toggled and the other items are left alone.
- `QIconView::Extended` - When the user selects an item the selection is cleared and the new item selected. However, if the user presses the Ctrl key when clicking on an item, the clicked item gets toggled and all other items are left untouched. If the user presses the Shift key while clicking on an item, all items between the current item and the clicked item get selected or unselected, depending on the state of the clicked item. Also, multiple items can be selected by dragging the mouse while the left mouse button stays pressed.
- `QIconView::NoSelection` - Items cannot be selected.

To summarise: Single is a real single-selection icon view; Multi a real multi-selection icon view; Extended is an icon view in which users can select multiple items but usually want to select either just one or a range of contiguous items; and NoSelection mode is for an icon view where the user can look but not touch.

Member Function Documentation

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

Constructs an empty icon view with the parent *parent* and the name *name*, using the widget flags *f*.

QIconView::~QIconView () [virtual]

Destroys the icon view and deletes all items.

void QIconView::adjustItems () [virtual protected slot]

Adjusts the positions of the items to the geometry of the icon view.

void QIconView::arrangeItemsInGrid (const QSize & grid, bool update = TRUE) [virtual slot]

This variant uses *grid* instead of (gridX(), gridY()). If *grid* is invalid (see QSize::isValid()), arrangeItemsInGrid() calculates a valid grid itself and uses that.

If *update* is TRUE (the default) the viewport is repainted.

Example: fileiconview/qfileiconview.h.

void QIconView::arrangeItemsInGrid (bool update = TRUE) [virtual slot]

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

Arranges all the items in the grid given by gridX() and gridY().

Even if sorting() is enabled, the items are not sorted by this function. If you want to sort or rearrange all items, use iconview->sort(iconview->sortDirection()).

If *update* is TRUE (the default), the viewport is repainted as well.

See also QIconView::gridX [p. 103], QIconView::gridY [p. 103] and QIconView::sort() [p. 101].

Arrangement QIconView::arrangement () const

Returns the arrangement mode of the icon view. See the "arrangement" [p. 102] property for details.

bool QIconView::autoArrange () const

Returns TRUE if the icon view rearranges its items when a new item is inserted; otherwise returns FALSE. See the "autoArrange" [p. 103] property for details.

void QIconView::clear () [virtual]

Clears the icon view. All items are deleted.

void QIconView::clearSelection () [virtual]

Unselects all items.

void QIconView::clicked (QIconViewItem * item) [signal]

This signal is emitted when the user clicks any mouse button. If *item* is non-null, the cursor is on *item*. If *item* is null, the mouse cursor isn't on any item.

See also mouseButtonClicked() [p. 97], rightButtonClicked() [p. 98] and pressed() [p. 98].

void QIconView::clicked (QIconViewItem * item, const QPoint & pos) [signal]

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

This signal is emitted when the user clicks any mouse button. *item* is a pointer to the item that has been clicked. If you click on the iconview, but not on an item, then the signal is not emitted.

pos is the position of the mouse cursor in the global coordinate system (QMouseEvent::globalPos()). (If the click's press and release differ by a pixel or two, *pos* is the position at release time.)

See also mouseButtonClicked() [p. 97], rightButtonClicked() [p. 98] and pressed() [p. 98].

void QIconView::contextMenuRequested (QIconViewItem * item, const QPoint & pos) [signal]

This signal is emitted when the user invokes a context menu with the right mouse button or with special system keys, with *item* being the item under the mouse cursor or the current item, respectively.

pos is the position for the context menu in the global coordinate system.

uint QIconView::count () const

Returns the number of items in the icon view. See the "count" [p. 103] property for details.

void QIconView::currentChanged (QIconViewItem * item) [signal]

This signal is emitted when a new item becomes current. *item* is the new current item (or 0 if no item is current now).

See also currentItem() [p. 93].

QIconViewItem * QIconView::currentItem () const

Returns a pointer to the current item of the icon view, or 0 if no item is current.

void QIconView::doAutoScroll () [virtual protected slot]

Performs autoscrolling when selecting multiple icons with the rubber band.

void QIconView::doubleClicked (QIconViewItem * item) [signal]

This signal is emitted when the user double-clicks on *item*.

QDragObject * QIconView::dragObject () [virtual protected]

Returns the QDragObject that should be used for drag-and-drop. This function is called by the icon view when starting a drag to get the dragobject which should be used for the drag. Subclasses may reimplement this.

See also QIconDrag [Events, Actions, Layouts and Styles with Qt].

Example: fileiconview/qfileiconview.cpp.

void QIconView::drawBackground (QPainter * p, const QRect & r) [virtual protected]

This function is called to draw the rectangle *r* of the background using the painter *p*.

The default implementation fills *r* with the viewport's backgroundBrush(). Subclasses may reimplement this to draw custom backgrounds.

See also contentsX [p. 276], contentsY [p. 276] and drawContents() [p. 269].

void QIconView::drawRubber (QPainter * p) [virtual protected]

Draws the rubber band using the painter *p*.

void QIconView::dropped (QDropEvent * e, const QValueList<QIconDragItem> & lst) [signal]

This signal is emitted when a drop event occurs in the viewport (but not on any icon) which the icon view itself can't handle.

e provides all the information about the drop. If the drag object of the drop was a QIconDrag, *lst* contains the list of the dropped items. You can get the data using QIconDragItem::data() on each item. If the *lst* is empty, i.e. the drag was not a QIconDrag, you have to decode the data in *e* and work with that.

Note QIconViewItems may be drop targets; if a drop event occurs on an item the item handles the drop.

Example: iconview/main.cpp.

void QIconView::emitSelectionChanged (QIconViewItem * i = 0) [protected]

Emits a signal to indicate selection changes. *i* is the QIconViewItem that was selected or de-selected.

You should never need to call this.

void QIconView::ensureItemVisible (QIconViewItem * item)

Makes sure that *item* is entirely visible. If necessary, ensureItemVisible() scrolls the icon view.

See also ensureVisible() [p. 270].

QIconViewItem * QIconView::findFirstVisibleItem (const QRect & r) const

Finds the first item whose bounding rectangle overlaps *r* and returns a pointer to that item. *r* is given in content coordinates.

If you want to find all items that touch *r*, you will need to use this function and nextItem() in a loop ending at findLastVisibleItem() and test QItem::rect() for each of these items.

See also findLastVisibleItem() [p. 95] and QIconViewItem::rect() [p. 113].

QIconViewItem * QIconView::findItem (const QPoint & pos) const

Returns a pointer to the item that contains *pos*, which is given in contents coordinates.

QIconViewItem * QIconView::findItem (const QString & text, ComparisonFlags compare = BeginsWith) const

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

Returns a pointer to the first item whose text begins with *text*, or 0 if no such item could be found. Use the *compare* flag to control the comparison behaviour. (See Qt::StringComparisonMode.)

QIconViewItem * QIconView::findLastVisibleItem (const QRect & r) const

Finds the last item whose bounding rectangle overlaps *r* and returns a pointer to that item. *r* is given in content coordinates.

See also findFirstVisibleItem() [p. 95].

QIconViewItem * QIconView::firstItem () const

Returns a pointer to the first item of the icon view, or 0 if there are no items in the icon view.

int QIconView::gridX () const

Returns the horizontal grid of the icon view. See the "gridX" [p. 103] property for details.

int QIconView::gridY () const

Returns the vertical grid of the icon view. See the "gridY" [p. 103] property for details.

int QIconView::index (const QIconViewItem * item) const

Returns the index of *item*, or -1 if *item* doesn't exist in this icon view.

void QIconView::insertInGrid (QIconViewItem * item) [virtual protected]

Inserts the QIconViewItem *item* in the icon view's grid. *You should never need to call this manually.* Insert QIconViewItems by creating them with a pointer to the QIconView that they are to be inserted into as a parameter.

void QIconView::insertItem (QIconViewItem * item, QIconViewItem * after = 0L) [virtual]

Inserts the icon view item *item* after *after*. If *after* is 0, *item* is appended after the last item.

You should never need to call this function. Instead create QIconViewItem's and associate them with your icon view like this:

```
(void) new QIconViewItem( myIconview, "The text of the item", aPixmap );
```

void QIconView::invertSelection () [virtual]

Inverts the selection. Works only in Multi and Extended selection mode.

bool QIconView::isRenaming () const

Returns TRUE if an iconview item is being renamed; otherwise returns FALSE.

void QIconView::itemRenamed (QIconViewItem * item, const QString & name) [signal]

This signal is emitted when *item* has been renamed to *name*, usually by in-place renaming.

See also QIconViewItem::setRenameEnabled() [p. 115] and QIconViewItem::rename() [p. 113].

void QIconView::itemRenamed (QIconViewItem * item) [signal]

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

This signal is emitted when *item* has been renamed, usually by in-place renaming.

See also QIconViewItem::setRenameEnabled() [p. 115] and QIconViewItem::rename() [p. 113].

QBrush QIconView::itemTextBackground () const

Returns the brush that should be used when drawing the background of an item's text. See the "itemTextBackground" [p. 103] property for details.

ItemTextPos QIconView::itemTextPos () const

Returns the position where the text of each item is drawn. See the "itemTextPos" [p. 103] property for details.

bool QIconView::itemsMovable () const

Returns TRUE if the user is allowed to move items around in the icon view; otherwise returns FALSE. See the "itemsMovable" [p. 103] property for details.

QIconViewItem * QIconView::lastItem () const

Returns a pointer to the last item of the icon view, or 0 if there are no items in the icon view.

QIconViewItem * QIconView::makeRowLayout (QIconViewItem * begin, int & y, bool & changed) [protected]

Lays out a row of icons (if Arrangement == TopToBottom this is a column). Starts laying out with the item *begin*. *y* is the starting coordinate. Returns the last item of the row (column) and sets the new starting coordinate to *y*. The *changed* parameter is used internally.

This function may be made private in a future version of Qt. *We do not recommend calling it.*

int QIconView::maxItemTextLength () const

Returns the maximum length (in characters) that an item's text may have. See the "maxItemTextLength" [p. 104] property for details.

int QIconView::maxItemWidth () const

Returns the maximum width that an item may have. See the "maxItemWidth" [p. 104] property for details.

void QIconView::mouseButtonClicked (int button, QIconViewItem * item, const QPoint & pos) [signal]

This signal is emitted when the user clicks mouse button *button*. If *item* is non-null, the cursor is on *item*. If *item* is null, the mouse cursor isn't on any item.

pos is the position of the mouse cursor in the global coordinate system (QMouseEvent::globalPos()). (If the click's press and release differ by a pixel or two, *pos* is the position at release time.)

See also mouseButtonPressed() [p. 97], rightButtonClicked() [p. 98] and clicked() [p. 93].

void QIconView::mouseButtonPressed (int button, QIconViewItem * item, const QPoint & pos) [signal]

This signal is emitted when the user presses mouse button *button*. If *item* is non-null, the cursor is on *item*. If *item* is null, the mouse cursor isn't on any item.

pos is the position of the mouse cursor in the global coordinate system (QMouseEvent::globalPos()).

See also rightButtonClicked() [p. 98] and pressed() [p. 98].

void QIconView::moved () [signal]

This signal is emitted after successfully dropping one (or more) items of the icon view. If the items should be removed, it's best to do so in a slot connected to this signal.

Example: iconview/main.cpp.

void QIconView::onItem (QIconViewItem * item) [signal]

This signal is emitted when the user moves the mouse cursor onto an *item*, similar to the QWidget::enterEvent() function.

void QIconView::onViewport () [signal]

This signal is emitted when the user moves the mouse cursor from an item to an empty part of the icon view.

See also onItem() [p. 97].

void QIconView::pressed (QIconViewItem * item) [signal]

This signal is emitted when the user presses any mouse button. If *item* is non-null, the cursor is on *item*. If *item* is null, the mouse cursor isn't on any item.

See also mouseButtonPressed() [p. 97], rightButtonPressed() [p. 99] and clicked() [p. 93].

void QIconView::pressed (QIconViewItem * item, const QPoint & pos) [signal]

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

This signal is emitted when the user presses any mouse button. If *item* is non-null, the cursor is on *item*. If *item* is null, the mouse cursor isn't on any item.

pos is the position of the mouse cursor in the global coordinate system (QMouseEvent::globalPos()). (If the click's press and release differ by a pixel or two, *pos* is the position at release time.)

See also mouseButtonPressed() [p. 97], rightButtonPressed() [p. 99] and clicked() [p. 93].

void QIconView::repaintItem (QIconViewItem * item) [virtual]

Repaints the *item*.

ResizeMode QIconView::resizeMode () const

Returns the resize mode of the icon view. See the "resizeMode" [p. 104] property for details.

void QIconView::returnPressed (QIconViewItem * item) [signal]

This signal is emitted if the user presses the Return or Enter key. *item* is the currentItem() at the time of the keypress.

void QIconView::rightButtonClicked (QIconViewItem * item, const QPoint & pos) [signal]

This signal is emitted when the user clicks the right mouse button. If *item* is non-null, the cursor is on *item*. If *item* is null, the mouse cursor isn't on any item.

pos is the position of the mouse cursor in the global coordinate system (QMouseEvent::globalPos()). (If the click's press and release differ by a pixel or two, *pos* is the position at release time.)

See also rightButtonPressed() [p. 99], mouseButtonClicked() [p. 97] and clicked() [p. 93].

void QIconView::rightButtonPressed (QIconViewItem * item, const QPoint & pos) [signal]

This signal is emitted when the user presses the right mouse button. If *item* is non-null, the cursor is on *item*. If *item* is null, the mouse cursor isn't on any item.

pos is the position of the mouse cursor in the global coordinate system (QMouseEvent::globalPos()).

void QIconView::selectAll (bool select) [virtual]

In Multi and Extended modes, this function sets all items to be selected if *select* is TRUE, and to be unselected if *select* is FALSE.

In Single and NoSelection modes, this function only changes the selection status of currentItem().

void QIconView::selectionChanged () [signal]

This signal is emitted when the selection has been changed. It's emitted in each selection mode.

void QIconView::selectionChanged (QIconViewItem * item) [signal]

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

This signal is emitted when the selection changes. *item* is the newly selected item. This signal is emitted only in single selection mode.

SelectionMode QIconView::selectionMode () const

Returns the selection mode of the icon view. See the "selectionMode" [p. 104] property for details.

void QIconView::setArrangement (Arrangement am) [virtual]

Sets the arrangement mode of the icon view to *am*. See the "arrangement" [p. 102] property for details.

void QIconView::setAutoArrange (bool b) [virtual]

Sets whether the icon view rearranges its items when a new item is inserted to *b*. See the "autoArrange" [p. 103] property for details.

void QIconView::setCurrentItem (QIconViewItem * item) [virtual]

Makes *item* the new current item of the icon view.

void QIconView::setGridX (int rx) [virtual]

Sets the horizontal grid of the icon view to *rx*. See the "gridX" [p. 103] property for details.

void QIconView::setGridY (int ry) [virtual]

Sets the vertical grid of the icon view to *ry*. See the "gridY" [p. 103] property for details.

void QIconView::setItemTextBackground (const QBrush & b) [virtual]

Sets the brush that should be used when drawing the background of an item's text to *b*. See the "itemTextBackground" [p. 103] property for details.

void QIconView::setItemTextPos (ItemTextPos pos) [virtual]

Sets the position where the text of each item is drawn to *pos*. See the "itemTextPos" [p. 103] property for details.

void QIconView::setItemsMovable (bool b) [virtual]

Sets whether the user is allowed to move items around in the icon view to *b*. See the "itemsMovable" [p. 103] property for details.

void QIconView::setMaxItemTextLength (int w) [virtual]

Sets the maximum length (in characters) that an item's text may have to *w*. See the "maxItemTextLength" [p. 104] property for details.

void QIconView::setMaxItemWidth (int w) [virtual]

Sets the maximum width that an item may have to *w*. See the "maxItemWidth" [p. 104] property for details.

void QIconView::setResizeMode (ResizeMode am) [virtual]

Sets the resize mode of the icon view to *am*. See the "resizeMode" [p. 104] property for details.

void QIconView::setSelected (QIconViewItem * item, bool s, bool cb = FALSE) [virtual]

Selects or unselects *item* depending on *s*, and may also unselect other items, depending on QIconView::selectionMode() and *cb*.

If *s* is FALSE, *item* is unselected.

If *s* is TRUE and QIconView::selectionMode() is Single, *item* is selected, and the item which was selected is unselected.

If *s* is TRUE and QIconView::selectionMode() is Extended, *item* is selected. If *cb* is TRUE, the selection state of the icon view's other items is left unchanged. If *cb* is FALSE (the default) all other items are unselected.

If *s* is TRUE and QIconView::selectionMode() is Multi *item* is selected.

Note that *cb* is used only if QIconView::selectionMode() is Extended. *cb* defaults to FALSE.

All items whose selection status is changed repaint themselves.

void QIconView::setSelectionMode (SelectionMode m) [virtual]

Sets the selection mode of the icon view to *m*. See the "selectionMode" [p. 104] property for details.

void QIconView::setShowToolTips (bool b) [virtual]

Sets whether the icon view will display a tool tip with the complete text for any truncated item text to *b*. See the "showToolTips" [p. 104] property for details.

void QIconView::setSorting (bool sort, bool ascending = TRUE)

If *sort* is TRUE, this function sets the icon view to sort items when a new item is inserted. If *sort* is FALSE, the icon view will not be sorted.

Note that autoArrange() has to be TRUE for sorting to take place.

If *ascending* is TRUE, items are sorted in ascending order. If *ascending* is FALSE, items are sorted in descending order.

QIconViewItem::compare() is used to compare pairs of items. The sorting is based on the item's keys; these default to the item's text unless specifically set to something else.

See also QIconView::autoArrange [p. 103], QIconView::autoArrange [p. 103], sortDirection [p. 104], sort() [p. 101] and QIconViewItem::setKey() [p. 114].

void QIconView::setSpacing (int sp) [virtual]

Sets the space in pixels between icon view items to *sp*. See the "spacing" [p. 105] property for details.

void QIconView::setWordWrapIconText (bool b) [virtual]

Sets whether the item text will be word-wrapped if it is too long to *b*. See the "wordWrapIconText" [p. 105] property for details.

bool QIconView::showToolTips () const

Returns TRUE if the icon view will display a tool tip with the complete text for any truncated item text; otherwise returns FALSE. See the "showToolTips" [p. 104] property for details.

void QIconView::slotUpdate () [virtual protected slot]

This slot is used for a slightly-delayed update.

The icon view is not redrawn immediately after inserting a new item but after a very small delay using a QTimer. This means that when many items are inserted in a loop the icon view is probably redrawn only once at the end of the loop. This makes the insertions both flicker-free and faster.

void QIconView::sort (bool ascending = TRUE) [virtual]

Sorts and rearranges all items in the icon view. If *ascending* is TRUE, the items are sorted in increasing order, otherwise they are sorted in decreasing order.

QIconViewItem::compare() is used to compare pairs of items. The sorting is based on the item's keys; these default to the item's text unless specifically set to something else.

Note that this function sets the sort order to *ascending*.

See also QIconViewItem::key() [p. 111], QIconViewItem::setKey() [p. 114], QIconViewItem::compare() [p. 109], QIconView::setSorting() [p. 101] and QIconView::sortDirection [p. 104].

bool QIconView::sortDirection () const

Returns TRUE if the sort direction for inserting new items is ascending;; otherwise returns FALSE. See the "sortDirection" [p. 104] property for details.

bool QIconView::sorting () const

Returns TRUE if the icon view sorts on insertion; otherwise returns FALSE. See the "sorting" [p. 104] property for details.

int QIconView::spacing () const

Returns the space in pixels between icon view items. See the "spacing" [p. 105] property for details.

void QIconView::startDrag () [virtual protected]

Starts a drag.

void QIconView::takeItem (QIconViewItem * item) [virtual]

Takes the icon view item *item* out of the icon view and causes an update of the screen display. The item is not deleted. You should normally not need to call this function because QIconViewItem::~QIconViewItem() calls it. The normal way to delete an item is to delete it.

bool QIconView::wordWrapIconText () const

Returns TRUE if the item text will be word-wrapped if it is too long; otherwise returns FALSE. See the "wordWrapIconText" [p. 105] property for details.

Property Documentation

Arrangement arrangement

This property holds the arrangement mode of the icon view.

This can be LeftToRight or TopToBottom. The default is LeftToRight.

Set this property's value with setArrangement() and get this property's value with arrangement().

bool autoArrange

This property holds whether the icon view rearranges its items when a new item is inserted.

The default is set to TRUE.

Note that if the icon view is not visible at the time of insertion, QIconView defers all position-related work until it's shown and then calls arrangeItemsInGrid().

Set this property's value with setAutoArrange() and get this property's value with autoArrange().

uint count

This property holds the number of items in the icon view.

Get this property's value with count().

int gridX

This property holds the horizontal grid of the icon view.

If the value is -1, (the default), QIconView computes suitable column widths based on the icon view's contents.

Note that setting a grid width overrides setMaxItemWidth().

Set this property's value with setGridX() and get this property's value with gridX().

int gridY

This property holds the vertical grid of the icon view.

If the value is -1, (the default), QIconView computes suitable column heights based on the icon view's contents.

Set this property's value with setGridY() and get this property's value with gridY().

QBrush itemTextBackground

This property holds the brush that should be used when drawing the background of an item's text.

By default this brush is set to NoBrush, meaning that only the normal icon view background is used.

Set this property's value with setItemTextBackground() and get this property's value with itemTextBackground().

ItemTextPos itemTextPos

This property holds the position where the text of each item is drawn.

Valid values are Bottom or Right. The default is Bottom.

Set this property's value with setItemTextPos() and get this property's value with itemTextPos().

bool itemsMovable

This property holds whether the user is allowed to move items around in the icon view.

The default is TRUE.

Set this property's value with setItemsMovable() and get this property's value with itemsMovable().

int maxItemTextLength

This property holds the maximum length (in characters) that an item's text may have.

The default is 255 characters.

Set this property's value with setMaxItemTextLength() and get this property's value with maxItemTextLength().

int maxItemWidth

This property holds the maximum width that an item may have.

The default is 100 pixels.

Note that if the gridX() value is set QIconView will ignore this property.

Set this property's value with setMaxItemWidth() and get this property's value with maxItemWidth().

ResizeMode resizeMode

This property holds the resize mode of the icon view.

This can be Fixed or Adjust. The default is Fixed.

Set this property's value with setResizeMode() and get this property's value with resizeMode().

SelectionMode selectionMode

This property holds the selection mode of the icon view.

This can be Single (the default), Extended, Multi or NoSelection.

Set this property's value with setSelectionMode() and get this property's value with selectionMode().

bool showToolTips

This property holds whether the icon view will display a tool tip with the complete text for any truncated item text.

The default is TRUE. Note that this has no effect if setWordWrapIconText() is TRUE, as it is by default.

Set this property's value with setShowToolTips() and get this property's value with showToolTips().

bool sortDirection

This property holds whether the sort direction for inserting new items is ascending;.

The default is TRUE (i.e. ascending). This sort direction only has meaning if sorting() and autoArrange() are both TRUE.

To set the sort direction, use setSorting()

Get this property's value with sortDirection().

bool sorting

This property holds whether the icon view sorts on insertion.

The default is FALSE, i.e. no sorting on insertion.

To set the soring, use setSorting().

Get this property's value with sorting().

int spacing

This property holds the space in pixels between icon view items.

The default is 5 pixels.

Negative values for spacing are illegal.

Set this property's value with setSpacing() and get this property's value with spacing().

bool wordWrapIconText

This property holds whether the item text will be word-wrapped if it is too long.

The default is TRUE.

If this property is FALSE, icon text that is too long is truncated, and an ellipsis (...) appended to indicate that truncation has occurred.

Set this property's value with setWordWrapIconText() and get this property's value with wordWrapIconText().

QIconViewItem Class Reference

The QIconViewItem class provides a single item in a QIconView.

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

`#include <qiconview.h>`

Inherits Qt [Additional Functionality with Qt].

Public Members

- **QIconViewItem** (QIconView * parent)
- **QIconViewItem** (QIconView * parent, QIconViewItem * after)
- **QIconViewItem** (QIconView * parent, const QString & text)
- **QIconViewItem** (QIconView * parent, QIconViewItem * after, const QString & text)
- **QIconViewItem** (QIconView * parent, const QString & text, const QPixmap & icon)
- **QIconViewItem** (QIconView * parent, QIconViewItem * after, const QString & text, const QPixmap & icon)
- **QIconViewItem** (QIconView * parent, const QString & text, const QPicture & picture)
- **QIconViewItem** (QIconView * parent, QIconViewItem * after, const QString & text, const QPicture & picture)
- virtual **~QIconViewItem** ()
- virtual void **setRenameEnabled** (bool allow)
- virtual void **setDragEnabled** (bool allow)
- virtual void **setDropEnabled** (bool allow)
- virtual QString **text** () const
- virtual QPixmap * **pixmap** () const
- virtual QPicture * **picture** () const
- virtual QString **key** () const
- bool **renameEnabled** () const
- bool **dragEnabled** () const
- bool **dropEnabled** () const
- QIconView * **iconView** () const
- QIconViewItem * **prevItem** () const
- QIconViewItem * **nextItem** () const
- int **index** () const
- virtual void **setSelected** (bool s, bool cb)
- virtual void **setSelected** (bool s)
- virtual void **setSelectable** (bool enable)
- bool **isSelected** () const
- bool **isSelectable** () const
- virtual void **repaint** ()
- virtual bool **move** (int x, int y)

- virtual void **moveBy** (int dx, int dy)
- virtual bool **move** (const QPoint & pnt)
- virtual void **moveBy** (const QPoint & pnt)
- QRect **rect** () const
- int **x** () const
- int **y** () const
- int **width** () const
- int **height** () const
- QSize **size** () const
- QPoint **pos** () const
- QRect **textRect** (bool relative = TRUE) const
- QRect **pixmapRect** (bool relative = TRUE) const
- bool **contains** (const QPoint & pnt) const
- bool **intersects** (const QRect & r) const
- virtual bool **acceptDrop** (const QMimeSource * mime) const
- void **rename** ()
- virtual int **compare** (QIconViewItem * i) const
- virtual void **setText** (const QString & text)
- virtual void **setPixmap** (const QPixmap & icon)
- virtual void **setPicture** (const QPicture & icon)
- virtual void **setText** (const QString & text, bool recalc, bool redraw = TRUE)
- virtual void **setPixmap** (const QPixmap & icon, bool recalc, bool redraw = TRUE)
- virtual void **setKey** (const QString & k)
- virtual int **rtti** () const

Protected Members

- virtual void **removeRenameBox** ()
- virtual void **calcRect** (const QString & text_ = QString::null)
- virtual void **paintItem** (QPainter * p, const QColorGroup & cg)
- virtual void **paintFocus** (QPainter * p, const QColorGroup & cg)
- virtual void **dropped** (QDropEvent * e, const QValueList<QIconDragItem> & lst)
- virtual void **dragEntered** ()
- virtual void **dragLeft** ()
- void **setItemRect** (const QRect & r)
- void **setTextRect** (const QRect & r)
- void **setPixmapRect** (const QRect & r)

Detailed Description

The QIconViewItem class provides a single item in a QIconView.

A QIconViewItem contains an icon, a string and optionally a sort key, and can display itself in a QIconView.

The simplest way to create a QIconViewItem and insert it into a QIconView is to construct the item passing the constructor a pointer to the icon view, a string and an icon:

```
(void) new QIconViewItem(
                iconView,   // A pointer to a QIconView
                "This is the text of the item",
                aPixmap );
```

By default the text of an icon view item may not be edited by the user but calling setRenameEnabled(TRUE) will allow the user to perform in-place editing of the item's text.

When the icon view is deleted all items in it are deleted automatically.

The QIconView::firstItem() and QIconViewItem::nextItem() functions provide a means of iterating over all the items in a QIconView:

```
QIconViewItem *item;
for ( item = iconView->firstItem(); item; item = item->nextItem() )
  do_something_with( item );
```

To remove an item from an icon view, just delete the item. The QIconViewItem destructor removes it cleanly from its icon view.

Because the icon view is designed to use drag-and-drop, the icon view item also has functions for drag-and-drop which may be reimplemented.

The class is designed to be very similar to QListView and QListBox in use, both via instantiation and subclassing.

See also Advanced Widgets.

Member Function Documentation

QIconViewItem::QIconViewItem (QIconView * parent)

Constructs a QIconViewItem and inserts it into icon view *parent* with no text and a default icon.

QIconViewItem::QIconViewItem (QIconView * parent, QIconViewItem * after)

Constructs a QIconViewItem and inserts it into the icon view *parent* with no text and a default icon, after the icon view item *after*.

QIconViewItem::QIconViewItem (QIconView * parent, const QString & text)

Constructs an icon view item and inserts it into the icon view *parent* using *text* as the text and a default icon.

QIconViewItem::QIconViewItem (QIconView * parent, QIconViewItem * after, const QString & text)

Constructs an icon view item and inserts it into the icon view *parent* using *text* as the text and a default icon, after the icon view item *after*.

QIconViewItem::QIconViewItem (QIconView * parent, const QString & text, const QPixmap & icon)

Constructs an icon view item and inserts it into the icon view *parent* using *text* as the text and *icon* as the icon.

QIconViewItem::QIconViewItem (QIconView * parent, QIconViewItem * after, const QString & text, const QPixmap & icon)

Constructs an icon view item and inserts it into the icon view *parent* using *text* as the text and *icon* as the icon, after the icon view item *after*.

QIconViewItem::QIconViewItem (QIconView * parent, const QString & text, const QPicture & picture)

Constructs an icon view item and inserts it into the icon view *parent* using *text* as the text and a *picture* as the icon.

QIconViewItem::QIconViewItem (QIconView * parent, QIconViewItem * after, const QString & text, const QPicture & picture)

Constructs an icon view item and inserts it into the icon view *parent* using *text* as the text and *picture* as the icon, after the icon view item *after*.

QIconViewItem::~QIconViewItem () [virtual]

Destroys the icon view item and tells the parent icon view that the item has been destroyed.

bool QIconViewItem::acceptDrop (const QMimeSource * mime) const [virtual]

Returns TRUE if you can drop things with a QMimeSource of *mime* onto this item, and FALSE if you cannot.

The default implementation always returns FALSE. You must subclass QIconViewItem and reimplement acceptDrop() to accept drops.

Example: fileiconview/qfileiconview.cpp.

void QIconViewItem::calcRect (const QString & text_ = QString::null) [virtual protected]

This virtual function is responsible for calculating the rectangles returned by rect(), textRect() and pixmapRect(). setRect(), setTextRect() and setPixmapRect() are provided mainly for reimplementations of this function.

*text_* is an internal parameter which defaults to QString::null.

int QIconViewItem::compare (QIconViewItem * i) const [virtual]

Compares this icon view item to *i*. Returns -1 if this item is less than *i*, 0 if they are equal, and 1 if this icon view item is greater than *i*.

The default implementation compares the item keys (key()) using QString::localeAwareCompare(). A reimplementation may use different values and a different comparison function. Here is a reimplementation that uses plain Unicode comparison:

```
int MyIconViewItem::compare( QIconViewItem *i ) const
{
    return key().compare( i->key() );
}
```

See also key() [p. 111], QString::localeAwareCompare() [Datastructures and String Handling with Qt] and QString::compare() [Datastructures and String Handling with Qt].

bool QIconViewItem::contains (const QPoint & pnt) const

Returns TRUE if the item contains the point *pnt* (in contents coordinates), and FALSE if it does not.

bool QIconViewItem::dragEnabled () const

Returns TRUE if the user is allowed to drag the icon view item, otherwise returns FALSE.

See also setDragEnabled() [p. 114].

void QIconViewItem::dragEntered () [virtual protected]

This function is called when a drag enters the item's bounding rectangle.

The default implementation does nothing; subclasses may reimplement this function.

Example: fileiconview/qfileiconview.cpp.

void QIconViewItem::dragLeft () [virtual protected]

This function is called when a drag leaves the item's bounding rectangle.

The default implementation does nothing; subclasses may reimplement this function.

Example: fileiconview/qfileiconview.cpp.

bool QIconViewItem::dropEnabled () const

Returns TRUE if the user is allowed to drop something onto the item, otherwise returns FALSE.

See also setDropEnabled() [p. 114].

void QIconViewItem::dropped (QDropEvent * e, const QValueList<QIconDragItem> & lst) [virtual protected]

This function is called when something is dropped on the item. *e* provides all the information about the drop. If the drag object of the drop was a QIconDrag, *lst* contains the list of the dropped items. You can get the data using QIconDragItem::data() on each item. If the *lst* is empty, i.e. the drag was not a QIconDrag, you have to decode the data in *e* and work with that.

The default implementation does nothing; subclasses may reimplement this function.

Example: fileiconview/qfileiconview.cpp.

int QIconViewItem::height () const

Returns the height of the item.

QIconView * QIconViewItem::iconView () const

Returns a pointer to this item's icon view parent.

int QIconViewItem::index () const

Returns the index of this item in the icon view, or -1 if an error occurred.

bool QIconViewItem::intersects (const QRect & r) const

Returns TRUE if the item intersects the rectangle *r* (in contents coordinates), and FALSE if it does not.

bool QIconViewItem::isSelectable () const

Returns TRUE if the item is selectable, otherwise returns FALSE.

See also setSelectable() [p. 115].

bool QIconViewItem::isSelected () const

Returns TRUE if the item is selected, otherwise returns FALSE.

See also setSelected() [p. 115].

Example: fileiconview/qfileiconview.cpp.

QString QIconViewItem::key () const [virtual]

Returns the key of the icon view item or the text if no key has been explicitly set.

See also setKey() [p. 114] and compare() [p. 109].

bool QIconViewItem::move (int x, int y) [virtual]

Moves the item to *x* and *y* in the icon view (these are contents coordinates).

bool QIconViewItem::move (const QPoint & pnt) [virtual]

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

Moves the item to the point *pnt*.

void QIconViewItem::moveBy (int dx, int dy) [virtual]

Moves the item *dx* pixels in the x-direction and *dy* pixels in the y-direction.

void QIconViewItem::moveBy (const QPoint & pnt) [virtual]

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

Moves the item by the x, y values in point *pnt*.

QIconViewItem * QIconViewItem::nextItem () const

Returns a pointer to the next item, or 0 if this is the last item in the icon view.

To find the first item use QIconView::firstItem().

Example: fileiconview/qfileiconview.cpp.

void QIconViewItem::paintFocus (QPainter * p, const QColorGroup & cg) [virtual protected]

Paints the focus rectangle of the item using the painter *p* and the color group *cg*.

void QIconViewItem::paintItem (QPainter * p, const QColorGroup & cg) [virtual protected]

Paints the item using the painter *p* and the color group *cg*. If you want the item to be drawn with a different font or color, reimplement this function, change the values of the color group or the painter's font, and then call the QIconViewItem::paintItem() with the changed values.

Example: fileiconview/qfileiconview.cpp.

QPicture * QIconViewItem::picture () const [virtual]

Returns the icon of the icon view item if it is a picture, or 0 if it is a pixmap. In the latter case use pixmap() instead. Normally will set the picture of the item with setPicture(), but sometimes it's inconvenient to call setPicture() for each item. So you can subclass QIconViewItem, reimplement this function and return a pointer to the item's picture. If you do this, you *must* call calcRect() manually each time the size of this picture changes.

See also setPicture() [p. 114].

QPixmap * QIconViewItem::pixmap () const [virtual]

Returns the icon of the icon view item if it is a pixmap, or 0 if it is a picture. In the latter case use picture() instead. Normally you set the pixmap of the item with setPixmap(), but sometimes it's inconvenient to call setPixmap() for each item. So you can subclass QIconViewItem, reimplement this function and return a pointer to the item's pixmap. If you do this, you *must* call calcRect() manually each time the size of this pixmap changes.

See also setPixmap() [p. 114].

Example: fileiconview/qfileiconview.cpp.

QRect QIconViewItem::pixmapRect (bool relative = TRUE) const

Returns the bounding rectangle of the item's icon.

If *relative* is TRUE, (the default), the rectangle is relative to the origin of the item's rectangle. If *relative* is FALSE, the returned rectangle is relative to the origin of the icon view's contents coordinate system.

Example: fileiconview/qfileiconview.cpp.

QPoint QIconViewItem::pos () const

Returns the position of the item (in contents coordinates).

QIconViewItem * QIconViewItem::prevItem () const

Returns a pointer to the previous item, or 0 if this is the first item in the icon view.

To iterate the items in an icon view

QRect QIconViewItem::rect () const

Returns the bounding rectangle of the item (in contents coordinates).

void QIconViewItem::removeRenameBox () [virtual protected]

Removes the editbox that is used for in-place renaming.

void QIconViewItem::rename ()

Starts in-place renaming of an icon, if allowed.

This function sets up the icon view so that the user can edit the item text, and then returns. When the user is done, setText() will be called and QIconView::itemRenamed() will be emitted (unless the user cancelled, e.g. by pressing the Escape key).

See also setRenameEnabled() [p. 115].

Example: fileiconview/qfileiconview.cpp.

bool QIconViewItem::renameEnabled () const

Returns TRUE if the item can be renamed by the user with in-place renaming, or else FALSE.

See also setRenameEnabled() [p. 115].

Example: fileiconview/qfileiconview.cpp.

void QIconViewItem::repaint () [virtual]

Repaints the item.

int QIconViewItem::rtti () const [virtual]

Returns 0.

Make your derived classes return their own values for rtti(), so that you can distinguish between iconview item types. You should use values greater than 1000, preferably a large random number, to allow for extensions to this class.

void QIconViewItem::setDragEnabled (bool allow) [virtual]

If *allow* is TRUE, the icon view permits the user to drag the icon view item either to another position within the icon view or to somewhere outside of it. If *allow* is FALSE, the item cannot be dragged.

void QIconViewItem::setDropEnabled (bool allow) [virtual]

If *allow* is TRUE, the icon view lets the user drop something on this icon view item.

void QIconViewItem::setItemRect (const QRect & r) [protected]

Sets the bounding rectangle of the whole item to *r*. This function is provided for subclasses which reimplement calcRect(), so that they can set the calculated rectangle. *Other use is discouraged.*

See also calcRect() [p. 109], textRect() [p. 116], setTextRect() [p. 116], pixmapRect() [p. 112] and setPixmapRect() [p. 114].

void QIconViewItem::setKey (const QString & k) [virtual]

Sets *k* as the sort key of the icon view item. By default the text itself is used for sorting.

See also compare() [p. 109].

Example: fileiconview/qfileiconview.cpp.

void QIconViewItem::setPicture (const QPicture & icon) [virtual]

Sets *icon* as the item's icon in the icon view. This function might be a no-op if you reimplement picture().

See also picture() [p. 112].

void QIconViewItem::setPixmap (const QPixmap & icon) [virtual]

Sets *icon* as the item's icon in the icon view. This function might be a no-op if you reimplement pixmap().

See also pixmap() [p. 112].

void QIconViewItem::setPixmap (const QPixmap & icon, bool recalc, bool redraw = TRUE) [virtual]

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

Sets *icon* as the item's icon in the icon view. If *recalc* is TRUE, the icon view's layout is recalculated. If *redraw* is TRUE (the default), the icon view is repainted.

See also pixmap() [p. 112].

void QIconViewItem::setPixmapRect (const QRect & r) [protected]

Sets the bounding rectangle of the item's icon to *r*. This function is provided for subclasses which reimplement calcRect(), so that they can set the calculated rectangle. *Other use is discouraged.*

See also calcRect() [p. 109], pixmapRect() [p. 112], setItemRect() [p. 114] and setTextRect() [p. 116].

void QIconViewItem::setRenameEnabled (bool allow) [virtual]

If *allow* is TRUE, the user can rename the icon view item by clicking on the text (or pressing F2) while the item is selected (in-place renaming). If *allow* is FALSE, in-place renaming is not possible.

Examples: fileiconview/qfileiconview.cpp and iconview/main.cpp.

void QIconViewItem::setSelectable (bool enable) [virtual]

Sets this item to be selectable if *enable* is TRUE (the default) or unselectable if *enable* is FALSE.

The user is unable to select a non-selectable item using either the keyboard or the mouse. (The application programmer can select an item in code regardless of this setting.)

See also isSelectable() [p. 111].

void QIconViewItem::setSelected (bool s, bool cb) [virtual]

Selects or unselects the item, depending on *s*; it may also unselect other items, depending on QIconView::selectionMode() and *cb*.

If *s* is FALSE, the item is unselected.

If *s* is TRUE and QIconView::selectionMode() is `Single`, the item is selected and the item previously selected is unselected.

If *s* is TRUE and QIconView::selectionMode() is `Extended`, the item is selected. If *cb* is TRUE, the selection state of the other items is left unchanged. If *cb* is FALSE (the default) all other items are unselected.

If *s* is TRUE and QIconView::selectionMode() is `Multi`, the item is selected.

Note that *cb* is used only if QIconView::selectionMode() is `Extended`; cb defaults to FALSE.

All items whose selection status changes repaint themselves.

Example: fileiconview/qfileiconview.cpp.

void QIconViewItem::setSelected (bool s) [virtual]

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

This variant is equivalent to calling the other variant with *cb* set to FALSE.

void QIconViewItem::setText (const QString & text) [virtual]

Sets *text* as the text of the icon view item. This function might be a no-op if you reimplement text().

See also text() [p. 116].

Example: fileiconview/qfileiconview.cpp.

void QIconViewItem::setText (const QString & text, bool recalc, bool redraw = TRUE) [virtual]

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

Sets *text* as the text of the icon view item. If *recalc* is TRUE, the icon view's layout is recalculated. If *redraw* is TRUE (the default), the icon view is repainted.

See also text() [p. 116].

void QIconViewItem::setTextRect (const QRect & r) [protected]

Sets the bounding rectangle of the item's text to *r*. This function is provided for subclasses which reimplement calcRect(), so that they can set the calculated rectangle. *Other use is discouraged.*

See also calcRect() [p. 109], textRect() [p. 116], setItemRect() [p. 114] and setPixmapRect() [p. 114].

QSize QIconViewItem::size () const

Returns the size of the item.

QString QIconViewItem::text () const [virtual]

Returns the text of the icon view item. Normally you set the text of the item with setText(), but sometimes it's inconvenient to call setText() for each item; so you can subclass QIconViewItem, reimplement this function, and return the text of the item. If you do this, you have to call calcRect() manually each time the text (and therefore its size) changes.

See also setText() [p. 115].

Example: fileiconview/qfileiconview.cpp.

QRect QIconViewItem::textRect (bool relative = TRUE) const

Returns the bounding rectangle of the item's text.

If *relative* is TRUE, (the default), the returned rectangle is relative to the origin of the item's rectangle. If *relative* is FALSE, the returned rectangle is relative to the origin of the icon view's contents coordinate system.

Example: fileiconview/qfileiconview.cpp.

int QIconViewItem::width () const

Returns the width of the item.

int QIconViewItem::x () const

Returns the x-coordinate of the item (in contents coordinates).

int QIconViewItem::y () const

Returns the y-coordinate of the item (in contents coordinates).

QLabel Class Reference

The QLabel widget provides a text or image display.

`#include <qlabel.h>`

Inherits QFrame [p. 65].

Public Members

- **QLabel** (QWidget * parent, const char * name = 0, WFlags f = 0)
- **QLabel** (const QString & text, QWidget * parent, const char * name = 0, WFlags f = 0)
- **QLabel** (QWidget * buddy, const QString & text, QWidget * parent, const char * name = 0, WFlags f = 0)
- **~QLabel** ()
- QString **text** () const
- QPixmap * **pixmap** () const
- QPicture * **picture** () const
- QMovie * **movie** () const
- TextFormat **textFormat** () const
- void **setTextFormat** (TextFormat)
- int **alignment** () const
- virtual void **setAlignment** (int)
- int **indent** () const
- void **setIndent** (int)
- bool autoResize () const *(obsolete)*
- virtual void setAutoResize (bool enable) *(obsolete)*
- bool **hasScaledContents** () const
- void **setScaledContents** (bool)
- virtual void **setBuddy** (QWidget * buddy)
- QWidget * **buddy** () const
- virtual void **setFont** (const QFont & f)

Public Slots

- virtual void **setText** (const QString &)
- virtual void **setPixmap** (const QPixmap &)
- virtual void **setPicture** (const QPicture & picture)
- virtual void **setMovie** (const QMovie & movie)
- virtual void **setNum** (int num)
- virtual void **setNum** (double num)
- void **clear** ()

Properties

- Alignment **alignment** — the alignment of the label's contents
- int **indent** — the label's indent in pixels
- QPixmap **pixmap** — the label's pixmap
- bool **scaledContents** — whether the label will scale its contents to fill all available space
- QString **text** — the label text
- TextFormat **textFormat** — the label's text format

Protected Members

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

Detailed Description

The QLabel widget provides a text or image display.

QLabel is used for displaying information in the form of text or an image. No user interaction functionality is provided. The visual appearance of the label can be configured in various ways, and it can be used for specifying a focus accelerator key for another widget.

A QLabel can contain any of the following content types:

- Plain text: set by passing a QString to setText().
- Rich text: set by passing a QString that contains rich text to setText().
- A pixmap: set by passing a QPixmap to setPixmap().
- A movie: set by passing a QMovie to setMovie().
- A number: set by passing an *int* or a *double* to setNum(), which converts the number to plain text.
- Nothing: the same as an empty plain text. This is the default. Set by clear().

When the content is changed using any of these functions, any previous content is cleared.

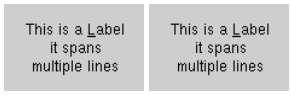
The look of a QLabel can be tuned in several ways. All the settings of QFrame are available for specifying a widget frame. The positioning of the content within the QLabel widget area can be tuned with setAlignment() and setIndent(). For example, this code sets up a sunken panel with a two-line text in the bottom right corner (both lines being flush with the right side of the label):

```
QLabel *label = new QLabel;
label->setFrameStyle( QFrame::Panel | QFrame::Sunken );
label->setText( "first line\nsecond line" );
label->setAlignment( AlignBottom | AlignRight );
```

A QLabel is often used as a label for an interactive widget. For this use QLabel provides a useful mechanism for adding an accelerator key (see QAccel) that will set the keyboard focus to the other widget (called the QLabel's "buddy"). Example:

```
QLineEdit* phoneEdit = new QLineEdit( this, "phoneEdit" );
QLabel* phoneLabel = new QLabel( phoneEdit, "&Phone:", this, "phoneLabel" );
```

In this example, keyboard focus is transferred to the label's buddy (the QLineEdit) when the user presses `Alt-P`. You can also use the setBuddy() function to accomplish the same thing.



See also QLineEdit [p. 132], QTextView, QPixmap [Graphics with Qt], QMovie [Graphics with Qt], GUI Design Handbook: Label, Basic Widgets and Text Related Classes.

Member Function Documentation

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

Constructs an empty label.

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

See also alignment [p. 123], setFrameStyle() [p. 70] and indent [p. 123].

QLabel::QLabel (const QString & text, QWidget * parent, const char * name = 0, WFlags f = 0)

Constructs a label that displays the text, *text*.

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

See also text [p. 124], alignment [p. 123], setFrameStyle() [p. 70] and indent [p. 123].

QLabel::QLabel (QWidget * buddy, const QString & text, QWidget * parent, const char * name = 0, WFlags f = 0)

Constructs a label that displays the text *text*. The label has a buddy widget, *buddy*.

If the *text* contains an underlined letter (a letter preceded by an ampersand, &), and the text is in plain text format, when the user presses Alt+ the underlined letter, focus is passed to the buddy widget.

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

See also text [p. 124], setBuddy() [p. 121], alignment [p. 123], setFrameStyle() [p. 70] and indent [p. 123].

QLabel::~QLabel ()

Destroys the label.

int QLabel::alignment () const

Returns the alignment of the label's contents. See the "alignment" [p. 123] property for details.

bool QLabel::autoResize () const

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

Returns TRUE if auto-resizing is enabled, or FALSE if auto-resizing is disabled.

Auto-resizing is disabled by default.

See also setAutoResize() [p. 121].

QWidget * QLabel::buddy () const

Returns the buddy of this label, or 0 if no buddy is currently set.

See also setBuddy() [p. 121].

void QLabel::clear () [slot]

Clears any label contents. Equivalent to setText("").

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

Draws the label contents using the painter *p*.

Reimplemented from QFrame [p. 68].

bool QLabel::hasScaledContents () const

Returns TRUE if the label will scale its contents to fill all available space; otherwise returns FALSE. See the "scaledContents" [p. 124] property for details.

int QLabel::indent () const

Returns the label's indent in pixels. See the "indent" [p. 123] property for details.

QMovie * QLabel::movie () const

If the label contains a movie, returns a pointer to it. Otherwise, returns 0.

See also setMovie() [p. 122].

QPicture * QLabel::picture () const

Returns the label's picture or 0 if the label doesn't have a picture.

QPixmap * QLabel::pixmap () const

Returns the label's pixmap. See the "pixmap" [p. 123] property for details.

void QLabel::setAlignment (int) [virtual]

Sets the alignment of the label's contents. See the "alignment" [p. 123] property for details.

void QLabel::setAutoResize (bool enable) [virtual]

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

Enables auto-resizing if *enable* is TRUE, or disables it if *enable* is FALSE.

When auto-resizing is enabled the label will resize itself to fit the contents whenever the contents change. The top-left corner is not moved. This is useful for QLabel widgets that are not managed by a QLayout (e.g., top-level widgets).

Auto-resizing is disabled by default.

See also autoResize() [p. 119], adjustSize() [p. 423] and sizeHint [p. 470].

void QLabel::setBuddy (QWidget * buddy) [virtual]

Sets the buddy of this label to *buddy*.

When the user presses the accelerator key indicated by this label, the keyboard focus is transferred to the label's buddy widget.

The buddy mechanism is available only for QLabels that contain plain text in which one letter is prefixed with an ampersand, &. This letter is set as the accelerator key. The letter is displayed underlined, and the '&' is not displayed (i.e. the ShowPrefix alignment flag is turned on; see setAlignment()).

In a dialog, you might create two data entry widgets and a label for each, and set up the geometry layout so each label is just to the left of its data entry widget (its "buddy"), perhaps like this:

```
QLineEdit *nameEd  = new QLineEdit( this );
QLabel    *nameLb  = new QLabel( "&Name:", this );
nameLb->setBuddy( nameEd );
QLineEdit *phoneEd = new QLineEdit( this );
QLabel    *phoneLb = new QLabel( "&Phone:", this );
phoneLb->setBuddy( phoneEd );
// ( layout setup not shown )
```

With the code above, the focus jumps to the Name field when the user presses Alt-N, and to the Phone field when the user presses Alt-P.

To unset a previously set buddy, call this function with *buddy* set to 0.

See also buddy() [p. 120], text [p. 124], QAccel [Events, Actions, Layouts and Styles with Qt] and alignment [p. 123].

Example: addressbook/centralwidget.cpp.

void QLabel::setFont (const QFont & f) [virtual]

Sets the font used on the QLabel to font *f*.

Reimplemented from QWidget [p. 449].

void QLabel::setIndent (int)

Sets the label's indent in pixels. See the "indent" [p. 123] property for details.

void QLabel::setMovie (const QMovie & movie) [virtual slot]

Sets the label contents to *movie*. Any previous content is cleared.

The buddy accelerator, if any, is disabled.

The label resizes itself if auto-resizing is enabled.

See also movie() [p. 120] and setBuddy() [p. 121].

void QLabel::setNum (int num) [virtual slot]

Sets the label contents to plain text containing the printed representation of integer *num*. Any previous content is cleared. Does nothing if the integer's string representation is the same as the current contents of the label.

The buddy accelerator, if any, is disabled.

The label resizes itself if auto-resizing is enabled.

See also text [p. 124], QString::setNum() [Datastructures and String Handling with Qt] and setBuddy() [p. 121].

void QLabel::setNum (double num) [virtual slot]

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

Sets the label contents to plain text containing the printed representation of double *num*. Any previous content is cleared. Does nothing if the double's string representation is the same as the current contents of the label.

The buddy accelerator, if any, is disabled.

The label resizes itself if auto-resizing is enabled.

See also text [p. 124], QString::setNum() [Datastructures and String Handling with Qt] and setBuddy() [p. 121].

void QLabel::setPicture (const QPicture & picture) [virtual slot]

Sets the label contents to *picture*. Any previous content is cleared.

The buddy accelerator, if any, is disabled.

See also picture() [p. 120] and setBuddy() [p. 121].

void QLabel::setPixmap (const QPixmap &) [virtual slot]

Sets the label's pixmap. See the "pixmap" [p. 123] property for details.

void QLabel::setScaledContents (bool)

Sets whether the label will scale its contents to fill all available space. See the "scaledContents" [p. 124] property for details.

void QLabel::setText (const QString &) [virtual slot]

Sets the label text. See the "text" [p. 124] property for details.

void QLabel::setTextFormat (TextFormat)

Sets the label's text format. See the "textFormat" [p. 124] property for details.

QString QLabel::text () const

Returns the label text. See the "text" [p. 124] property for details.

TextFormat QLabel::textFormat () const

Returns the label's text format. See the "textFormat" [p. 124] property for details.

Property Documentation

Alignment alignment

This property holds the alignment of the label's contents.

The alignment is a bitwise OR of Qt::AlignmentFlags and Qt::TextFlags values. The ExpandTabs, SingleLine and ShowPrefix flags apply only if the label contains plain text; otherwise they are ignored. The DontClip flag is always ignored. WordBreak applies to both rich text and plain text labels.

If the label has a buddy, the ShowPrefix flag is forced to TRUE.

The default alignment is `AlignAuto | AlignVCenter | ExpandTabs` if the label doesn't have a buddy and `AlignAuto | AlignVCenter | ExpandTabs | ShowPrefix` if the label has a buddy. If the label contains rich text, additionally WordBreak is turned on.

See also Qt::AlignmentFlags [Additional Functionality with Qt], alignment [p. 123], setBuddy() [p. 121] and text [p. 124].

Set this property's value with setAlignment() and get this property's value with alignment().

int indent

This property holds the label's indent in pixels.

The indent applies to the left edge if alignment() is AlignLeft, to the right edge if alignment() is AlignRight, to the top edge if alignment() is AlignTop, and to to the bottom edge if alignment() is AlignBottom.

If the indent is negative, or if no indent has been set, the label computes the effective indent as follows: if frameWidth() is 0, the effective indent becomes 0. If frameWidth() is greater than 0, the effective indent becomes half the width of the "x" character of the widget's current font().

See also alignment [p. 123], frameWidth [p. 72] and font [p. 462].

Set this property's value with setIndent() and get this property's value with indent().

QPixmap pixmap

This property holds the label's pixmap.

If no pixmap has been set this will return an invalid pixmap.

Setting the pixmap clears any previous content, and resizes the label if QLabel::autoResize() is TRUE. The buddy accelerator, if any, is disabled.

Set this property's value with setPixmap() and get this property's value with pixmap().

bool scaledContents

This property holds whether the label will scale its contents to fill all available space.

When enabled and the label shows a pixmap, it will scale the pixmap to fill the available space.

This property's default is FALSE.

See also scaledContents [p. 124].

Set this property's value with setScaledContents() and get this property's value with hasScaledContents().

QString text

This property holds the label text.

If no text has been set this will return an empty string. Setting the text clears any previous content, unless they are the same.

The text will be interpreted either as a plain text or as a rich text, depending on the text format setting; see setTextFormat(). The default setting is AutoText, i.e. QLabel will try to auto-detect the format of the text set.

If the text is interpreted as a plain text and a buddy has been set, the buddy accelerator key is updated from the new text.

The label resizes itself if auto-resizing is enabled.

Note that Qlabel is well-suited to display small rich text documents only. For large documents, use QTextView instead. QTextView will flicker less on resize and can also provide a scrollbar, when necessary.

See also text [p. 124], textFormat [p. 124], setBuddy() [p. 121] and alignment [p. 123].

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

TextFormat textFormat

This property holds the label's text format.

See the Qt::TextFormat enum for an explanation of the possible options.

The default format is AutoText.

See also text [p. 124].

Set this property's value with setTextFormat() and get this property's value with textFormat().

QLCDNumber Class Reference

The QLCDNumber widget displays a number with LCD-like digits.

`#include <qlcdnumber.h>`

Inherits QFrame [p. 65].

Public Members

- **QLCDNumber** (QWidget * parent = 0, const char * name = 0)
- **QLCDNumber** (uint numDigits, QWidget * parent = 0, const char * name = 0)
- **~QLCDNumber** ()
- enum **Mode** { Hex, Dec, Oct, Bin, HEX = Hex, DEC = Dec, OCT = Oct, BIN = Bin }
- enum **SegmentStyle** { Outline, Filled, Flat }
- bool **smallDecimalPoint** () const
- int **numDigits** () const
- virtual void **setNumDigits** (int nDigits)
- bool **checkOverflow** (double num) const
- bool **checkOverflow** (int num) const
- Mode **mode** () const
- virtual void **setMode** (Mode)
- SegmentStyle **segmentStyle** () const
- virtual void **setSegmentStyle** (SegmentStyle)
- double **value** () const
- int **intValue** () const

Public Slots

- void **display** (const QString & s)
- void **display** (int num)
- void **display** (double num)
- virtual void **setHexMode** ()
- virtual void **setDecMode** ()
- virtual void **setOctMode** ()
- virtual void **setBinMode** ()
- virtual void **setSmallDecimalPoint** (bool)

Signals

- void **overflow** ()

Properties

- int **intValue** — the displayed value rounded to the nearest integer
- Mode **mode** — the current display mode (number base)
- int **numDigits** — the current number of digits displayed
- SegmentStyle **segmentStyle** — the style of the LCDNumber
- bool **smallDecimalPoint** — the style of the decimal point
- double **value** — the displayed value

Protected Members

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

Detailed Description

The QLCDNumber widget displays a number with LCD-like digits.

It can display a number in just about any size. It can display decimal, hexadecimal, octal or binary numbers. It is easy to connect to data sources using the display() slot, which is overloaded to take any of five argument types.

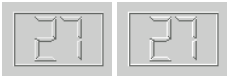
There are also slots to change the base with setMode() and the decimal point with setSmallDecimalPoint().

QLCDNumber emits the overflow() signal when it is asked to display something beyond its range. The range is set by setNumDigits(), but setSmallDecimalPoint() also influences it.

These digits and other symbols can be shown: 0/O, 1, 2, 3, 4, 5/S, 6, 7, 8, 9/g, minus, decimal point, A, B, C, D, E, F, h, H, L, o, P, r, u, U, Y, colon, degree sign (which is specified as single quote in the string) and space. QLCDNumber substitutes spaces for illegal characters.

It is not possible to retrieve the contents of a QLCDNumber object, although you can retrieve the numeric value with value(). If you really need the text, we recommend that you connect the signals that feed the display() slot to another slot as well and store the value there.

Incidentally, QLCDNumber is the very oldest part of Qt, tracing back to a BASIC program on the Sinclair Spectrum.



See also QLabel [p. 117], QFrame [p. 65] and Basic Widgets.

Member Type Documentation

QLCDNumber::Mode

This type determines how numbers are shown. The possible values are:

- `QLCDNumber::Hex` - Hexadecimal
- `QLCDNumber::Dec` - Decimal
- `QLCDNumber::Oct` - Octal
- `QLCDNumber::Bin` - Binary

QLCDNumber::SegmentStyle

This type determines the visual appearance of the QLCDNumber widget. The possible values are:

- `QLCDNumber::Outline` - gives raised segments filled with the background brush.
- `QLCDNumber::Filled` - gives raised segments filled with the foreground brush.
- `QLCDNumber::Flat` - gives flat segments filled with the foreground brush.

Member Function Documentation

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

Constructs an LCD number, sets the number of digits to 5, the base to decimal, the decimal point mode to 'small' and the frame style to a raised box. The segmentStyle() is set to Outline.

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

See also numDigits [p. 130] and smallDecimalPoint [p. 130].

QLCDNumber::QLCDNumber (uint numDigits, QWidget * parent = 0, const char * name = 0)

Constructs an LCD number, sets the number of digits to *numDigits*, the base to decimal, the decimal point mode to 'small' and the frame style to a raised box. The segmentStyle() is set to Outline.

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

See also numDigits [p. 130] and smallDecimalPoint [p. 130].

QLCDNumber::~QLCDNumber ()

Destroys the LCD number.

bool QLCDNumber::checkOverflow (double num) const

Returns TRUE if *num* is too big to be displayed in its entirety; otherwise returns FALSE.

See also intValue [p. 130], numDigits [p. 130] and smallDecimalPoint [p. 130].

bool QLCDNumber::checkOverflow (int num) const

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

Returns TRUE if *num* is too big to be displayed in its entirety; otherwise returns FALSE.

See also intValue [p. 130], numDigits [p. 130] and smallDecimalPoint [p. 130].

void QLCDNumber::display (int num) [slot]

Sets the displayed value rounded to the nearest integer to *num*. See the "intValue" [p. 130] property for details.

void QLCDNumber::display (const QString & s) [slot]

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

Displays the number represented by the string *s*.

This version of the function disregards mode() and smallDecimalPoint().

These digits and other symbols can be shown: 0/O, 1, 2, 3, 4, 5/S, 6, 7, 8, 9/g, minus, decimal point, A, B, C, D, E, F, h, H, L, o, P, r, u, U, Y, colon, degree sign (which is specified as single quote in the string) and space. QLCDNumber substitutes spaces for illegal characters.

void QLCDNumber::display (double num) [slot]

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

Displays the number *num*.

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

Draws the LCD number using painter *p*. This function is called from QFrame::paintEvent().

Reimplemented from QFrame [p. 68].

int QLCDNumber::intValue () const

Returns the displayed value rounded to the nearest integer. See the "intValue" [p. 130] property for details.

Mode QLCDNumber::mode () const

Returns the current display mode (number base). See the "mode" [p. 130] property for details.

int QLCDNumber::numDigits () const

Returns the current number of digits displayed. See the "numDigits" [p. 130] property for details.

void QLCDNumber::overflow () [signal]

This signal is emitted whenever the QLCDNumber is asked to display a too-large number or a too-long string.

It is never emitted by setNumDigits().

SegmentStyle QLCDNumber::segmentStyle () const

Returns the style of the LCDNumber. See the "segmentStyle" [p. 130] property for details.

void QLCDNumber::setBinMode () [virtual slot]

Calls setMode(BIN). Provided for convenience (e.g. for connecting buttons to it).

See also mode [p. 130], setHexMode() [p. 129], setDecMode() [p. 129], setOctMode() [p. 129] and mode [p. 130].

void QLCDNumber::setDecMode () [virtual slot]

Calls setMode(DEC). Provided for convenience (e.g. for connecting buttons to it).

See also mode [p. 130], setHexMode() [p. 129], setOctMode() [p. 129], setBinMode() [p. 128] and mode [p. 130].

void QLCDNumber::setHexMode () [virtual slot]

Calls setMode(HEX). Provided for convenience (e.g. for connecting buttons to it).

See also mode [p. 130], setDecMode() [p. 129], setOctMode() [p. 129], setBinMode() [p. 128] and mode [p. 130].

void QLCDNumber::setMode (Mode) [virtual]

Sets the current display mode (number base). See the "mode" [p. 130] property for details.

void QLCDNumber::setNumDigits (int nDigits) [virtual]

Sets the current number of digits displayed to *nDigits*. See the "numDigits" [p. 130] property for details.

void QLCDNumber::setOctMode () [virtual slot]

Calls setMode(OCT). Provided for convenience (e.g. for connecting buttons to it).

See also mode [p. 130], setHexMode() [p. 129], setDecMode() [p. 129], setBinMode() [p. 128] and mode [p. 130].

void QLCDNumber::setSegmentStyle (SegmentStyle) [virtual]

Sets the style of the LCDNumber. See the "segmentStyle" [p. 130] property for details.

void QLCDNumber::setSmallDecimalPoint (bool) [virtual slot]

Sets the style of the decimal point. See the "smallDecimalPoint" [p. 130] property for details.

bool QLCDNumber::smallDecimalPoint () const

Returns the style of the decimal point. See the "smallDecimalPoint" [p. 130] property for details.

double QLCDNumber::value () const

Returns the displayed value. See the "value" [p. 131] property for details.

Property Documentation

int intValue

This property holds the displayed value rounded to the nearest integer.

This property corresponds to the nearest integer to the current value displayed by the LCDNumber.

If the displayed value is not a number, the property has a value of 0.

Set this property's value with display() and get this property's value with intValue().

Mode mode

This property holds the current display mode (number base).

Corresponds to the current display mode, which is one of BIN, OCT, DEC (the default) and HEX. All four modes can display both integers, floating-point numbers and strings (subject to character set limitations).

See also smallDecimalPoint [p. 130], setHexMode() [p. 129], setDecMode() [p. 129], setOctMode() [p. 129] and setBinMode() [p. 128].

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

int numDigits

This property holds the current number of digits displayed.

Corresponds to the current number of digits. If QLCDNumber::smallDecimalPoint is FALSE, the decimal point occupies one digit position.

See also numDigits [p. 130] and smallDecimalPoint [p. 130].

Set this property's value with setNumDigits() and get this property's value with numDigits().

SegmentStyle segmentStyle

This property holds the style of the LCDNumber.

The style of the QLCDNumber is one of:

- Outline gives raised segments filled with the background color (this is the default).
- Filled gives raised segments filled with the foreground color.
- Flat gives flat segments filled with the foreground color.

Outline and Filled will additionally use QColorGroup::light() and QColorGroup::dark() for shadow effects.

Set this property's value with setSegmentStyle() and get this property's value with segmentStyle().

bool smallDecimalPoint

This property holds the style of the decimal point.

If TRUE the decimal point is drawn between two digit positions. Otherwise it occupies a digit position of its own, i.e. is drawn in a digit position. The default is FALSE.

The inter-digit space is made slightly wider when the decimal point is drawn between the digits.

See also mode [p. 130].

Set this property's value with setSmallDecimalPoint() and get this property's value with smallDecimalPoint().

double value

This property holds the displayed value.

This property corresponds to the current value displayed by the LCDNumber.

If the displayed value is not a number, the property has a value of 0.

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

QLineEdit Class Reference

The QLineEdit widget is a one-line text editor.

`#include <qlineedit.h>`

Inherits QFrame [p. 65].

Public Members

- **QLineEdit** (QWidget * parent, const char * name = 0)
- **QLineEdit** (const QString & contents, QWidget * parent, const char * name = 0)
- **~QLineEdit** ()
- QString **text** () const
- QString **displayText** () const
- int **maxLength** () const
- bool **frame** () const
- enum **EchoMode** { Normal, NoEcho, Password }
- EchoMode **echoMode** () const
- bool **isReadOnly** () const
- const QValidator * **validator** () const
- virtual QSize **sizeHint** () const
- virtual QSize **minimumSizeHint** () const
- int **cursorPosition** () const
- bool **validateAndSet** (const QString & newText, int newPos, int newMarkAnchor, int newMarkDrag)
- int **alignment** () const
- void cursorLeft (bool mark, int steps = 1) *(obsolete)*
- void cursorRight (bool mark, int steps = 1) *(obsolete)*
- void **cursorForward** (bool mark, int steps = 1)
- void **cursorBackward** (bool mark, int steps = 1)
- void **cursorWordForward** (bool mark)
- void **cursorWordBackward** (bool mark)
- void **backspace** ()
- void **del** ()
- void **home** (bool mark)
- void **end** (bool mark)
- void **setEdited** (bool)
- bool **edited** () const
- bool **hasSelectedText** () const
- QString **selectedText** () const
- bool **getSelection** (int * start, int * end)
- bool **isUndoAvailable** () const

- bool **isRedoAvailable** () const
- bool hasMarkedText () const *(obsolete)*
- QString markedText () const *(obsolete)*
- bool **dragEnabled** () const
- int **characterAt** (int xpos, QChar * chr) const

Public Slots

- virtual void **setText** (const QString &)
- virtual void **selectAll** ()
- virtual void **deselect** ()
- virtual void **clearValidator** ()
- virtual void **insert** (const QString & newText)
- virtual void **clear** ()
- virtual void **undo** ()
- virtual void **redo** ()
- virtual void **setMaxLength** (int)
- virtual void **setFrame** (bool)
- virtual void **setEchoMode** (EchoMode)
- virtual void **setReadOnly** (bool)
- virtual void **setValidator** (const QValidator * v)
- virtual void **setSelection** (int start, int length)
- virtual void **setCursorPosition** (int)
- virtual void **setAlignment** (int flag)
- virtual void **cut** ()
- virtual void **copy** () const
- virtual void **paste** ()
- virtual void **setDragEnabled** (bool b)

Signals

- void **textChanged** (const QString &)
- void **returnPressed** ()
- void **selectionChanged** ()

Properties

- Alignment **alignment** — the alignment of the line edit
- int **cursorPosition** — the current cursor position for this line edit
- QString **displayText** — the text that is displayed *(read only)*
- bool **dragEnabled** — whether the lineedit starts a drag if the user presses and moves the mouse on some selected text
- EchoMode **echoMode** — the echo mode of the line edit
- bool **edited** — the edited flag of the line edit
- bool **frame** — whether the line edit draws itself with a frame
- bool hasMarkedText — whether part of the text has been selected by the user (e.g. by clicking and dragging) *(read only)* *(obsolete)*

- bool **hasSelectedText** — whether there is any text selected *(read only)*
- QString markedText — the text selected by the user (e.g. by clicking and dragging), or QString::null if no text is selected *(read only)* *(obsolete)*
- int **maxLength** — the maximum permitted length of the text in the editor
- bool **readOnly** — whether the line edit is read only
- bool **redoAvailable** — whether redo is available *(read only)*
- QString **selectedText** — any text selected by the user or QString::null *(read only)*
- QString **text** — the text in the line
- bool **undoAvailable** — whether undo is available *(read only)*

Protected Members

- virtual void **keyPressEvent** (QKeyEvent * e)
- void repaintArea (int from, int to) *(obsolete)*
- virtual QPopupMenu * **createPopupMenu** ()

Detailed Description

The QLineEdit widget is a one-line text editor.

A line edit allows the user to enter and edit a single line of plain text with a useful collection of editing functions, including undo and redo, cut and paste, and drag and drop.

By changing the echoMode() of a line edit, it can also be used as a "write-only" field, for inputs such as passwords.

The length of the field can be constrained to maxLength(), or the value can be arbitrarily constrained by setting a validator().

A closely related class is QTextEdit which allows multi-line, rich-text editing.

You can change the text with setText() or insert(). The text is retrieved with text(); the displayed text (which may be different, see EchoMode) is retrieved with displayText(). Text can be selected with setSelection() or selectAll(), and the selection can be cut(), copy()ied and paste()d. The text can be aligned with setAlignment().

When the text changes the textChanged() signal is emitted; when the Return or Enter key is pressed the returnPressed() signal is emitted.

The default QLineEdit object has its own frame as specified by the Windows/Motif style guides; you can turn off the frame by calling setFrame(FALSE).

The default key bindings are described below. A right-mouse-button menu presents some of the editing commands to the user.

- *Left* Arrow - moves the cursor one character to the left.
- *Right* Arrow - moves the cursor one character to the right.
- *Backspace* - deletes the character to the left of the cursor.
- *Home* - moves the cursor to the beginning of the line.
- *End* - moves the cursor to the end of the line.
- *Delete* - deletes the character to the right of the cursor.
- *Shift+Left* Arrow - moves and selects text one character to the left.
- *Shift+Right* Arrow - moves and selects text one character to the right.
- *Ctrl+A* - moves the cursor to the beginning of the line.
- *Ctrl+B* - moves the cursor one character to the left.

- *Ctrl+C* - copies the selected text to the clipboard. (Windows also supports Ctrl+Insert for this operation.)
- *Ctrl+D* - deletes the character to the right of the cursor.
- *Ctrl+E* - moves the cursor to the end of the line.
- *Ctrl+F* - moves the cursor one character to the right.
- *Ctrl+H* - deletes the character to the left of the cursor.
- *Ctrl+K* - deletes to the end of the line.
- *Ctrl+V* - pastes the clipboard text into line edit. (Windows also supports Shift+Insert for this operation.)
- *Ctrl+X* - deletes the selected text and copies it to the clipboard. (Windows also supports Shift+Delete for this operation.)
- *Ctrl+Z* - undoes the last operation.
- *Ctrl+Y* - redoes the last undone operation.

Any other key sequence, that represents a valid character, will cause the character to be inserted into the line.

See also QTextEdit [p. 372], QLabel [p. 117], QComboBox [p. 32], GUI Design Handbook: Field, Entry, GUI Design Handbook: Field, Required and Basic Widgets.

Member Type Documentation

QLineEdit::EchoMode

This enum type describes how a line edit should display its contents. The defined values are:

- `QLineEdit::Normal` - display characters as they are entered. This is the default.
- `QLineEdit::NoEcho` - do not display anything. This may be appropriate for passwords where even the length of the password should be kept secret.
- `QLineEdit::Password` - display asterisks instead of the characters actually entered.

See also echoMode [p. 143] and echoMode [p. 143].

Member Function Documentation

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

Constructs a line edit with no text.

The maximum text length is set to 32767 characters.

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

See also text [p. 145] and maxLength [p. 144].

QLineEdit::QLineEdit (const QString & contents, QWidget * parent, const char * name = 0)

Constructs a line edit containing the text *contents*.

The cursor position is set to the end of the line and the maximum text length to 32767 characters.

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

See also text [p. 145] and maxLength [p. 144].

QLineEdit::~QLineEdit ()

Destroys the line edit.

int QLineEdit::alignment () const

Returns the alignment of the line edit. See the "alignment" [p. 143] property for details.

void QLineEdit::backspace ()

Deletes the character to the left of the text cursor and moves the cursor one position to the left. If any text has been selected by the user (e.g. by clicking and dragging), the cursor will be put at the beginning of the selected text and the selected text will be removed.

See also del() [p. 138].

int QLineEdit::characterAt (int xpos, QChar * chr) const

Returns the index position of the character which is at *xpos* (in logical coordinates from the left). If *chr* is not 0, *chr is populated with the character at this position.

void QLineEdit::clear () [virtual slot]

Syntactic sugar for setText(""), provided to match no-argument signals.

void QLineEdit::clearValidator () [virtual slot]

This slot is equivalent to setValidator(0).

void QLineEdit::copy () const [virtual slot]

Copies the selected text to the clipboard, if there is any, and if echoMode() is Normal.

See also cut() [p. 137] and paste() [p. 140].

QPopupMenu * QLineEdit::createPopupMenu () [virtual protected]

This function is called to create the popup menu which is shown when the user clicks on the lineedit with the right mouse button. If you want to create a custom popup menu, reimplement this function and return the popup menu you create. The popup menu's ownership is transferred to the caller.

void QLineEdit::cursorBackward (bool mark, int steps = 1)

Moves the cursor back *steps* characters. If *mark* is TRUE each character moved over is added to the selection; if *mark* is FALSE the selection is cleared.

See also cursorForward() [p. 137].

void QLineEdit::cursorForward (bool mark, int steps = 1)

Moves the cursor forward *steps* characters. If *mark* is TRUE each character moved over is added to the selection; if *mark* is FALSE the selection is cleared.

See also cursorBackward() [p. 136].

void QLineEdit::cursorLeft (bool mark, int steps = 1)

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

For compatibilty with older applications only. Use cursorBackward() instead.

See also cursorBackward() [p. 136].

int QLineEdit::cursorPosition () const

Returns the current cursor position for this line edit. See the "cursorPosition" [p. 143] property for details.

void QLineEdit::cursorRight (bool mark, int steps = 1)

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

Use cursorForward() instead.

See also cursorForward() [p. 137].

void QLineEdit::cursorWordBackward (bool mark)

Moves the cursor one word backward. If *mark* is TRUE, the word is also selected.

See also cursorWordForward() [p. 137].

void QLineEdit::cursorWordForward (bool mark)

Moves the cursor one word forward. If *mark* is TRUE, the word is also selected.

See also cursorWordBackward() [p. 137].

void QLineEdit::cut () [virtual slot]

Copies the selected text to the clipboard and deletes it, if there is any, and if echoMode() is Normal.

If the current validator disallows deleting the selected text, cut() will copy it but not delete it.

See also copy() [p. 136] and paste() [p. 140].

void QLineEdit::del ()

Deletes the character on the right side of the text cursor. If any text has been selected by the user (e.g. by clicking and dragging), the cursor will be put at the beginning of the selected text and the selected text will be removed.

See also backspace() [p. 136].

void QLineEdit::deselect () [virtual slot]

De-selects all text (i.e. removes highlighting) and leaves the cursor at the current position.

See also setSelection() [p. 141] and selectAll() [p. 140].

QString QLineEdit::displayText () const

Returns the text that is displayed. See the "displayText" [p. 143] property for details.

bool QLineEdit::dragEnabled () const

Returns TRUE if the lineedit starts a drag if the user presses and moves the mouse on some selected text; otherwise returns FALSE. See the "dragEnabled" [p. 143] property for details.

EchoMode QLineEdit::echoMode () const

Returns the echo mode of the line edit. See the "echoMode" [p. 143] property for details.

bool QLineEdit::edited () const

Returns the edited flag of the line edit. See the "edited" [p. 143] property for details.

void QLineEdit::end (bool mark)

Moves the text cursor to the end of the line. If *mark* is TRUE, text is selected towards the last position; otherwise, any selected text is unselected if the cursor is moved.

See also home() [p. 139].

bool QLineEdit::frame () const

Returns TRUE if the line edit draws itself with a frame; otherwise returns FALSE. See the "frame" [p. 144] property for details.

bool QLineEdit::getSelection (int * start, int * end)

This function sets `*<em>start</em>` to the position in the text where the selection starts and `*<em>end</em>` to the position where the selection ends. Returns TRUE if *start* and *end* are not null and if there is some selected text; otherwise returns FALSE.

See also setSelection() [p. 141].

bool QLineEdit::hasMarkedText () const

Returns TRUE if part of the text has been selected by the user (e.g. by clicking and dragging); otherwise returns FALSE. See the "hasMarkedText" [p. 144] property for details.

bool QLineEdit::hasSelectedText () const

Returns TRUE if there is any text selected; otherwise returns FALSE. See the "hasSelectedText" [p. 144] property for details.

void QLineEdit::home (bool mark)

Moves the text cursor to the beginning of the line. If *mark* is TRUE, text is selected towards the first position; otherwise, any selected text is unselected if the cursor is moved.

See also end() [p. 138].

void QLineEdit::insert (const QString & newText) [virtual slot]

Removes any selected text, inserts *newText*, and validates the result. If it is valid, it sets it as the new contents of the line edit.

bool QLineEdit::isReadOnly () const

Returns TRUE if the line edit is read only; otherwise returns FALSE. See the "readOnly" [p. 145] property for details.

bool QLineEdit::isRedoAvailable () const

Returns TRUE if redo is available; otherwise returns FALSE. See the "redoAvailable" [p. 145] property for details.

bool QLineEdit::isUndoAvailable () const

Returns TRUE if undo is available; otherwise returns FALSE. See the "undoAvailable" [p. 145] property for details.

void QLineEdit::keyPressEvent (QKeyEvent * e) [virtual protected]

Converts key press event *e* into a line edit action.

If Return or Enter is pressed and the current text is valid (or can be made valid by the validator), the signal returnPressed is emitted.

The default key bindings are listed in the detailed description.

Reimplemented from QWidget [p. 436].

QString QLineEdit::markedText () const

Returns the text selected by the user (e.g. by clicking and dragging), or QString::null if no text is selected. See the "markedText" [p. 144] property for details.

int QLineEdit::maxLength () const

Returns the maximum permitted length of the text in the editor. See the "maxLength" [p. 144] property for details.

QSize QLineEdit::minimumSizeHint () const [virtual]

Returns a minimum size for the line edit.

The width returned is enough for at least one character.

Reimplemented from QWidget [p. 439].

void QLineEdit::paste () [virtual slot]

Inserts the clipboard's text at the cursor position, deleting any selected text.

If the end result is not acceptable for the current validator, nothing happens.

See also copy() [p. 136] and cut() [p. 137].

void QLineEdit::redo () [virtual slot]

Redoes the last operation

void QLineEdit::repaintArea (int from, int to) [protected]

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

Repaints all characters from *from* to *to*. If cursorPos is between from and to, ensures that cursorPos is visible.

void QLineEdit::returnPressed () [signal]

This signal is emitted when the Return or Enter key is pressed.

Example: popup/popup.cpp.

void QLineEdit::selectAll () [virtual slot]

Selects all the text (i.e. highlights it) and moves the cursor to the end. This is useful when a default value has been inserted because if the user types before clicking on the widget, the selected text will be erased.

See also setSelection() [p. 141] and deselect() [p. 138].

QString QLineEdit::selectedText () const

Returns any text selected by the user or QString::null. See the "selectedText" [p. 145] property for details.

void QLineEdit::selectionChanged () [signal]

This signal is emitted whenever the selection changes.

See also hasSelectedText [p. 144] and selectedText [p. 145].

void QLineEdit::setAlignment (int flag) [virtual slot]

Sets the alignment of the line edit to *flag*. See the "alignment" [p. 143] property for details.

void QLineEdit::setCursorPosition (int) [virtual slot]

Sets the current cursor position for this line edit. See the "cursorPosition" [p. 143] property for details.

void QLineEdit::setDragEnabled (bool b) [virtual slot]

Sets whether the lineedit starts a drag if the user presses and moves the mouse on some selected text to *b*. See the "dragEnabled" [p. 143] property for details.

void QLineEdit::setEchoMode (EchoMode) [virtual slot]

Sets the echo mode of the line edit. See the "echoMode" [p. 143] property for details.

void QLineEdit::setEdited (bool)

Sets the edited flag of the line edit. See the "edited" [p. 143] property for details.

void QLineEdit::setFrame (bool) [virtual slot]

Sets whether the line edit draws itself with a frame. See the "frame" [p. 144] property for details.

void QLineEdit::setMaxLength (int) [virtual slot]

Sets the maximum permitted length of the text in the editor. See the "maxLength" [p. 144] property for details.

void QLineEdit::setReadOnly (bool) [virtual slot]

Sets whether the line edit is read only. See the "readOnly" [p. 145] property for details.

void QLineEdit::setSelection (int start, int length) [virtual slot]

Sets the selected area of this line edit to start at position *start* and be *length* characters long.

See also deselect() [p. 138], selectAll() [p. 140] and getSelection() [p. 138].

void QLineEdit::setText (const QString &) [virtual slot]

Sets the text in the line. See the "text" [p. 145] property for details.

void QLineEdit::setValidator (const QValidator * v) [virtual slot]

Sets this line edit to accept input only as accepted by the validator, *v*, allowing arbitrary constraints on the text which may be entered.

If *v* == 0, setValidator() removes the current input validator. The initial setting is to have no input validator (i.e. any input is accepted up to maxLength()).

See also validator() [p. 142] and QValidator [Additional Functionality with Qt].

Examples: lineedits/lineedits.cpp and wizard/wizard.cpp.

QSize QLineEdit::sizeHint () const [virtual]

Returns a recommended size for the widget.

The width returned, in pixels, is usually enough for about 15 to 20 characters.

Example: addressbook/centralwidget.cpp.

QString QLineEdit::text () const

Returns the text in the line. See the "text" [p. 145] property for details.

void QLineEdit::textChanged (const QString &) [signal]

This signal is emitted whenever the text changes. The argument is the new text.

Examples: wizard/wizard.cpp and xform/xform.cpp.

void QLineEdit::undo () [virtual slot]

Undoes the last operation

bool QLineEdit::validateAndSet (const QString & newText, int newPos, int newMarkAnchor, int newMarkDrag)

Validates and perhaps sets this line edit to contain *newText* with the cursor at position *newPos*, with selected text from *newMarkAnchor* to *newMarkDrag*. Returns TRUE if it changes the line edit; otherwise returns FALSE.

Linebreaks in *newText* are converted to spaces, and the text is truncated to maxLength() before its validity is tested.

Repaints and emits textChanged() if appropriate.

const QValidator * QLineEdit::validator () const

Returns a pointer to the current input validator, or 0 if no validator has been set.

See also setValidator() [p. 142].

Example: wizard/wizard.cpp.

Property Documentation

Alignment alignment

This property holds the alignment of the line edit.

Possible Values are Qt::AlignAuto, Qt::AlignLeft, Qt::AlignRight and Qt::AlignHCenter.

Attempting to set the alignment to an illegal flag combination does nothing.

See also Qt::AlignmentFlags [Additional Functionality with Qt].

Set this property's value with setAlignment() and get this property's value with alignment().

int cursorPosition

This property holds the current cursor position for this line edit.

Setting the cursor position causes a repaint when appropriate.

Set this property's value with setCursorPosition() and get this property's value with cursorPosition().

QString displayText

This property holds the text that is displayed.

If EchoMode is Normal this returns the same as text(); if EchoMode is Password it returns a string of asterisks the text().length() characters long, e.g. "******"; if EchoMode is NoEcho returns an empty string, "".

See also echoMode [p. 143], text [p. 145] and EchoMode [p. 135].

Get this property's value with displayText().

bool dragEnabled

This property holds whether the lineedit starts a drag if the user presses and moves the mouse on some selected text.

Set this property's value with setDragEnabled() and get this property's value with dragEnabled().

EchoMode echoMode

This property holds the echo mode of the line edit.

The initial setting is Normal, but QLineEdit also supports NoEcho and Password modes.

The widget's display and the ability to copy or drag the text is affected by this setting.

See also EchoMode [p. 135] and displayText [p. 143].

Set this property's value with setEchoMode() and get this property's value with echoMode().

bool edited

This property holds the edited flag of the line edit.

The edited flag is never read by QLineEdit; it has a default value of FALSE and is changed to TRUE whenever the user changes the line edit's contents.

This is useful for things that need to provide a default value but cannot find the default at once. Just start the line edit without the best default; when the default is known, check the edited() return value and set the line edit's contents if the user has not started editing the line edit.

Calling setText() resets the edited flag to FALSE.

Set this property's value with setEdited() and get this property's value with edited().

bool frame

This property holds whether the line edit draws itself with a frame.

If enabled (the default) the line edit draws itself inside a two-pixel frame, otherwise the line edit draws itself without any frame.

Set this property's value with setFrame() and get this property's value with frame().

bool hasMarkedText

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

This property holds whether part of the text has been selected by the user (e.g. by clicking and dragging).

Get this property's value with hasMarkedText().

See also selectedText [p. 145].

bool hasSelectedText

This property holds whether there is any text selected.

hasSelectedText() returns TRUE if some or all of the text has been selected by the user (e.g. by clicking and dragging); otherwise returns FALSE.

See also selectedText [p. 145].

Get this property's value with hasSelectedText().

QString markedText

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

This property holds the text selected by the user (e.g. by clicking and dragging), or QString::null if no text is selected.

Get this property's value with markedText().

See also hasSelectedText [p. 144].

int maxLength

This property holds the maximum permitted length of the text in the editor.

If the text is too long, it is truncated at the limit.

If truncation occurs any selected text will be unselected, the cursor position is set to 0 and the first part of the string is shown.

Set this property's value with setMaxLength() and get this property's value with maxLength().

bool readOnly

This property holds whether the line edit is read only.

In read-only mode, the user can still copy the text to the clipboard or drag-and-drop the text, but cannot edit it.

QLineEdit does not show a cursor in read-only mode.

See also enabled [p. 461].

Set this property's value with setReadOnly() and get this property's value with isReadOnly().

bool redoAvailable

This property holds whether redo is available.

Get this property's value with isRedoAvailable().

QString selectedText

This property holds any text selected by the user or QString::null.

Get this property's value with selectedText().

See also hasSelectedText [p. 144].

QString text

This property holds the text in the line.

Setting this property clears the selection and moves the cursor to the end of the line.

The text is truncated to maxLength() length.

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

bool undoAvailable

This property holds whether undo is available.

Get this property's value with isUndoAvailable().

QListBox Class Reference

The QListBox widget provides a list of selectable, read-only items.

`#include <qlistbox.h>`

Inherits QScrollView [p. 259].

Public Members

- **QListBox** (QWidget * parent = 0, const char * name = 0, WFlags f = 0)
- **~QListBox** ()
- uint **count** () const
- void **insertStringList** (const QStringList & list, int index = -1)
- void **insertStrList** (const QStrList * list, int index = -1)
- void **insertStrList** (const QStrList & list, int index = -1)
- void **insertStrList** (const char ** strings, int numStrings = -1, int index = -1)
- void **insertItem** (const QListBoxItem * lbi, int index = -1)
- void **insertItem** (const QListBoxItem * lbi, const QListBoxItem * after)
- void **insertItem** (const QString & text, int index = -1)
- void **insertItem** (const QPixmap & pixmap, int index = -1)
- void **insertItem** (const QPixmap & pixmap, const QString & text, int index = -1)
- void **removeItem** (int index)
- QString **text** (int index) const
- const QPixmap * **pixmap** (int index) const
- void **changeItem** (const QListBoxItem * lbi, int index)
- void **changeItem** (const QString & text, int index)
- void **changeItem** (const QPixmap & pixmap, int index)
- void **changeItem** (const QPixmap & pixmap, const QString & text, int index)
- void **takeItem** (const QListBoxItem * item)
- int **numItemsVisible** () const
- int **currentItem** () const
- QString **currentText** () const
- virtual void **setCurrentItem** (int index)
- virtual void **setCurrentItem** (QListBoxItem * i)
- void **centerCurrentItem** ()
- int **topItem** () const
- virtual void **setTopItem** (int index)
- virtual void **setBottomItem** (int index)
- long **maxItemWidth** () const
- enum **SelectionMode** { Single, Multi, Extended, NoSelection }
- virtual void **setSelectionMode** (SelectionMode)

- SelectionMode **selectionMode** () const
- void setMultiSelection (bool multi) *(obsolete)*
- bool isMultiSelection () const *(obsolete)*
- virtual void **setSelected** (QListBoxItem * item, bool select)
- void **setSelected** (int index, bool select)
- bool **isSelected** (int i) const
- bool **isSelected** (const QListBoxItem * i) const
- QListBoxItem * **item** (int index) const
- int **index** (const QListBoxItem * lbi) const
- QListBoxItem * **findItem** (const QString & text, ComparisonFlags compare = BeginsWith) const
- void **triggerUpdate** (bool doLayout)
- bool **itemVisible** (int index)
- bool **itemVisible** (const QListBoxItem * item)
- enum **LayoutMode** { FixedNumber, FitToWidth, FitToHeight = FitToWidth, Variable }
- virtual void **setColumnMode** (LayoutMode)
- virtual void **setColumnMode** (int columns)
- virtual void **setRowMode** (LayoutMode)
- virtual void **setRowMode** (int rows)
- LayoutMode **columnMode** () const
- LayoutMode **rowMode** () const
- int **numColumns** () const
- int **numRows** () const
- bool **variableWidth** () const
- virtual void **setVariableWidth** (bool)
- bool **variableHeight** () const
- virtual void **setVariableHeight** (bool)
- bool **autoScrollBar** () const
- void **setAutoScrollBar** (bool enable)
- bool **scrollBar** () const
- void **setScrollBar** (bool enable)
- bool **autoBottomScrollBar** () const
- void **setAutoBottomScrollBar** (bool enable)
- bool **bottomScrollBar** () const
- void **setBottomScrollBar** (bool enable)
- int inSort (const QListBoxItem * lbi) *(obsolete)*
- int inSort (const QString & text) *(obsolete)*
- int cellHeight (int i) const *(obsolete)*
- int cellHeight () const *(obsolete)*
- int cellWidth () const *(obsolete)*
- int numCols () const *(obsolete)*
- int **itemHeight** (int index = 0) const
- QListBoxItem * **itemAt** (const QPoint & p) const
- QRect **itemRect** (QListBoxItem * item) const
- QListBoxItem * **firstItem** () const
- void **sort** (bool ascending = TRUE)

Public Slots

- void **clear** ()
- virtual void **ensureCurrentVisible** ()
- virtual void **clearSelection** ()
- virtual void **selectAll** (bool select)
- virtual void **invertSelection** ()

Signals

- void **highlighted** (int index)
- void **selected** (int index)
- void **highlighted** (const QString &)
- void **selected** (const QString &)
- void **highlighted** (QListBoxItem *)
- void **selected** (QListBoxItem *)
- void **selectionChanged** ()
- void **selectionChanged** (QListBoxItem * item)
- void **currentChanged** (QListBoxItem * item)
- void **clicked** (QListBoxItem * item)
- void **clicked** (QListBoxItem * item, const QPoint & pnt)
- void **pressed** (QListBoxItem * item)
- void **pressed** (QListBoxItem * item, const QPoint & pnt)
- void **doubleClicked** (QListBoxItem * item)
- void **returnPressed** (QListBoxItem *)
- void **rightButtonClicked** (QListBoxItem *, const QPoint &)
- void **rightButtonPressed** (QListBoxItem *, const QPoint &)
- void **mouseButtonPressed** (int button, QListBoxItem * item, const QPoint & pos)
- void **mouseButtonClicked** (int button, QListBoxItem * item, const QPoint & pos)
- void **contextMenuRequested** (QListBoxItem * item, const QPoint & pos)
- void **onItem** (QListBoxItem * i)
- void **onViewport** ()

Properties

- LayoutMode **columnMode** — the column layout mode for this list box
- uint **count** — the number of items in the list box *(read only)*
- int **currentItem** — the current highlighted item
- QString **currentText** — the text of the current item *(read only)*
- bool multiSelection — whether or not the list box is in Multi selection mode *(obsolete)*
- int **numColumns** — the number of columns in the list box *(read only)*
- int **numItemsVisible** — the number of visible items *(read only)*
- int **numRows** — the number of rows in the list box *(read only)*
- LayoutMode **rowMode** — the row layout mode for this list box
- SelectionMode **selectionMode** — the selection mode of the list box
- int **topItem** — the index of an item at the top of the screen
- bool **variableHeight** — whether this list box has variable-height rows
- bool **variableWidth** — whether this list box has variable-width columns

Protected Members

- void **updateItem** (int index)
- void **updateItem** (QListBoxItem * i)
- int totalWidth () const *(obsolete)*
- int totalHeight () const *(obsolete)*
- virtual void **paintCell** (QPainter * p, int row, int col)
- void **toggleCurrentItem** ()
- bool **isRubberSelecting** () const
- void **doLayout** () const
- bool itemYPos (int index, int * yPos) const *(obsolete)*
- int findItem (int yPos) const *(obsolete)*

Detailed Description

The QListBox widget provides a list of selectable, read-only items.

This is typically a single-column list in which zero or one item is selected, but it can also be used in many other ways.

QListBox will add scroll bars as necessary, but it isn't intended for *really* big lists. If you want more than a few thousand items, it's probably better to use a different widget mainly because the scroll bars won't provide very good navigation, but also because QListBox may become slow with huge lists.

There are a variety of selection modes described in the QListBox::SelectionMode documentation. The default is Single selection mode, but you can change it using setSelectionMode(). (setMultiSelection() is still provided for compatibility with Qt 1.x. We recomment using setSelectionMode() in all code.)

Because QListBox offers multiple selection it must display keyboard focus and selection state separately. Therefore there are functions both to set the selection state of an item, i.e. setSelected(), and to select which item displays keyboard focus, i.e. setCurrentItem().

The list box normally arranges its items in a single column and adds a vertical scroll bar if required. It is possible to have a different fixed number of columns (setColumnMode()), or as many columns as will fit in the list box's assigned screen space (setColumnMode(FitToWidth)), or to have a fixed number of rows (setRowMode()) or as many rows as will fit in the list box's assigned screen space (setRowMode(FitToHeight)). In all these cases QListBox will add scroll bars, as appropriate, in at least one direction.

If multiple rows are used, each row can be as high as necessary (the normal setting), or you can request that all items will have the same height by calling setVariableHeight(FALSE). The same applies to a column's width, see setVariableWidth().

The items discussed are QListBoxItem objects. QListBox provides methods to insert new items as strings, as pixmaps, and as QListBoxItem * (insertItem() with various arguments), and to replace an existing item with a new string, pixmap or QListBoxItem (changeItem() with various arguments). You can also remove items singly with removeItem() or clear() the entire list box. Note that if you create a QListBoxItem yourself and insert it, it becomes the property of QListBox and you must not delete it. (QListBox will delete it when appropriate.)

You can also create a QListBoxItem, such as QListBoxText or QListBoxPixmap, with the list box as first parameter. The item will then append itself. When you delete an item it is automatically removed from the list box.

The list of items can be arbitrarily large; QListBox will add scroll bars if necessary. QListBox can display a single-column (the common case) or multiple-columns, and offers both single and multiple selection. (QListBox does not support multiple-column items, or tree hierarchies; use QListView if you require such functionality.)

The list box items can be accessed both as QListBoxItem objects (recommended) and using integer indexes (the original QListBox implementation used an array of strings internally, and the API still supports this mode of operation). Everything can be done using the new objects; most things can be done using the indexes, too, but unfortunately not everything.

Each item in a QListBox contains a QListBoxItem. One of the items can be the current item. The highlighted() signal is emitted when a new item gets highlighted, e.g. because the user clicks on it or QListBox::setCurrentItem() is called. The selected() signal is emitted when the user double-clicks on an item or presses Enter when an item is highlighted.

If the user does not select anything, no signals are emitted and currentItem() returns -1.

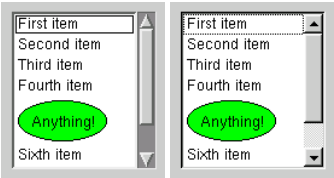
A list box has WheelFocus as a default focusPolicy(), i.e. it can get keyboard focus by tabbing, clicking and through the use of the mouse wheel.

New items can be inserted using insertItem(), insertStrList() or insertStringList(). inSort() is obsolete because this method is quite inefficient. It's preferable to insert the items normally and call sort() afterwards, or to insert a sorted QStringList().

By default, vertical and horizontal scroll bars are added and removed as necessary. setHScrollBarMode() and setVScrollBarMode() can be used to change this policy.

If you need to insert types other than strings and pixmaps, you must define new classes which inherit QListBoxItem.

Warning: The list box assumes ownership of all list box items and will delete them when it does not need them any more.



See also QListView [p. 177], QComboBox [p. 32], QButtonGroup [p. 14], GUI Design Handbook: List Box (two sections) and Advanced Widgets.

Member Type Documentation

QListBox::LayoutMode

This enum type is used to specify how QListBox lays out its rows and columns.

The possible values for each row or column mode are:

- `QListBox::FixedNumber` - There is a fixed number of rows (or columns).
- `QListBox::FitToWidth` - There are as many columns as will fit on-screen.
- `QListBox::FitToHeight` - There are as many rows as will fit on-screen.
- `QListBox::Variable` - There are as many rows as are required by the column mode. (Or as many columns as required by the row mode.)

Example: When you call setRowMode(FitToHeight), columnMode() automatically becomes Variable to accommodate the row mode you've set.

QListBox::SelectionMode

This enumerated type is used by QListBox to indicate how it reacts to selection by the user. It has four values:

- `QListBox::Single` - When the user selects an item, any already-selected item becomes unselected and the user cannot unselect the selected item. This means that the user can never clear the selection, even though the selection may be cleared by the application programmer using QListBox::clearSelection().

- `QListBox::Multi` - When the user selects an item the selection status of that item is toggled and the other items are left alone.
- `QListBox::Extended` - When the user selects an item the selection is cleared and the new item selected. However, if the user presses the Ctrl key when clicking on an item, the clicked item gets toggled and all other items are left untouched. And if the user presses the Shift key while clicking on an item, all items between the current item and the clicked item get selected or unselected, depending on the state of the clicked item. Also, multiple items can be selected by dragging the mouse while the left mouse button is kept pressed.
- `QListBox::NoSelection` - Items cannot be selected.

In other words, Single is a real single-selection list box, Multi is a real multi-selection list box, Extended is a list box in which users can select multiple items but usually want to select either just one or a range of contiguous items, and NoSelection is for a list box where the user can look but not touch.

Member Function Documentation

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

Constructs a new empty list box, with *parent* as a parent and *name* as object name.

Performance is boosted by modifying the widget flags *f* so that only part of the QListBoxItem children is redrawn. This may be unsuitable for custom QListBoxItem classes, in which case WStaticContents and WRepaintNoErase should be cleared immediately after construction.

See also QWidget::clearWFlags() [p. 424] and Qt::WidgetFlags [Additional Functionality with Qt].

QListBox::~QListBox ()

Destroys the list box. Deletes all list box items.

bool QListBox::autoBottomScrollBar () const

Returns TRUE if hScrollBarMode() is Auto; otherwise returns FALSE.

bool QListBox::autoScrollBar () const

Returns TRUE if vScrollBarMode() is Auto; otherwise returns FALSE.

bool QListBox::bottomScrollBar () const

Returns FALSE if vScrollBarMode() is AlwaysOff; otherwise returns TRUE.

int QListBox::cellHeight (int i) const

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

Returns the item height of item *i*.

See also itemHeight() [p. 158].

int QListBox::cellHeight () const

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

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

Returns the item height of the first item, item 0.

See also itemHeight() [p. 158].

int QListBox::cellWidth () const

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

Returns the maximum item width.

See also maxItemWidth() [p. 159].

void QListBox::centerCurrentItem ()

If there is a current item, the list box is scrolled so that this item is displayed centered.

See also QListBox::ensureCurrentVisible() [p. 154].

void QListBox::changeItem (const QListBoxItem * lbi, int index)

Replaces the item at position *index* with *lbi*. If *index* is negative or too large, changeItem() does nothing.

See also insertItem() [p. 156] and removeItem() [p. 160].

void QListBox::changeItem (const QString & text, int index)

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

Replaces the item at position *index* with a new list box text item with text *text*.

The operation is ignored if *index* is out of range.

See also insertItem() [p. 156] and removeItem() [p. 160].

void QListBox::changeItem (const QPixmap & pixmap, int index)

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

Replaces the item at position *index* with a new list box pixmap item with pixmap *pixmap*.

The operation is ignored if *index* is out of range.

See also insertItem() [p. 156] and removeItem() [p. 160].

void QListBox::changeItem (const QPixmap & pixmap, const QString & text, int index)

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

Replaces the item at position *index* with a new list box pixmap item with pixmap *pixmap* and text *text*.

The operation is ignored if *index* is out of range.

See also insertItem() [p. 156] and removeItem() [p. 160].

void QListBox::clear () [slot]

Deletes all the items in the list.

See also removeItem() [p. 160].

void QListBox::clearSelection () [virtual slot]

Deselects all items, if possible.

Note that a Single selection list box will automatically select an item if it has keyboard focus.

Example: listbox/listbox.cpp.

void QListBox::clicked (QListBoxItem * item) [signal]

This signal is emitted when the user clicks any mouse button. If *item* is non-null, the cursor is on *item*. If *item* is null, the mouse cursor isn't on any item.

Note that you must not delete any QListBoxItem objects in slots connected to this signal.

void QListBox::clicked (QListBoxItem * item, const QPoint & pnt) [signal]

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

This signal is emitted when the user clicks any mouse button. If *item* is non-null, the cursor is on *item*. If *item* is null, the mouse cursor isn't on any item.

pnt is the position of the mouse cursor in the global coordinate system (QMouseEvent::globalPos()). (If the click's press and release differ by a pixel or two, *pnt* is the position at release time.)

Note that you must not delete any QListBoxItem objects in slots connected to this signal.

LayoutMode QListBox::columnMode () const

Returns the column layout mode for this list box. See the "columnMode" [p. 165] property for details.

void QListBox::contextMenuRequested (QListBoxItem * item, const QPoint & pos) [signal]

This signal is emitted when the user invokes a context menu with the right mouse button or with special system keys, with *item* being the item under the mouse cursor or the current item, respectively.

pos is the position for the context menu in the global coordinate system.

uint QListBox::count () const

Returns the number of items in the list box. See the "count" [p. 165] property for details.

void QListBox::currentChanged (QListBoxItem * item) [signal]

This signal is emitted when the user highlights a new current item. *item* is the new current list box item.

See also currentItem [p. 166] and currentItem [p. 166].

int QListBox::currentItem () const

Returns the current highlighted item. See the "currentItem" [p. 166] property for details.

QString QListBox::currentText () const

Returns the text of the current item. See the "currentText" [p. 166] property for details.

void QListBox::doLayout () const [protected]

This function does the hard layout work. You should never need to call it.

void QListBox::doubleClicked (QListBoxItem * item) [signal]

This signal is emitted whenever an item is double-clicked. It's emitted on the second button press, not the second button release. *item* is the item item on which the user did the double-click. *item* may be 0.

void QListBox::ensureCurrentVisible () [virtual slot]

Ensures that the current item is visible.

QListBoxItem * QListBox::findItem (const QString & text, ComparisonFlags compare = BeginsWith) const

Finds the first list box item that has the text *text* and returns it, or returns 0 of no such item could be found. If ComparisonFlags are specified in *compare* then these flags are used, otherwise the default is a case-sensitive, exact match search.

See also Qt::StringComparisonMode [Additional Functionality with Qt].

int QListBox::findItem (int yPos) const [protected]

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

Returns the index of the item a point (0, *yPos*).

See also index() [p. 155] and itemAt() [p. 158].

QListBoxItem * QListBox::firstItem () const

Returns the first item in this list box. If the list box is empty this will be 0.

void QListBox::highlighted (int index) [signal]

This signal is emitted when the user highlights a new current item. *index* is the index of the highlighted item.

See also selected() [p. 161], currentItem [p. 166] and selectionChanged() [p. 162].

void QListBox::highlighted (const QString &) [signal]

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

This signal is emitted when the user highlights a new current item and the new item is a string. The argument is the text of the new current item.

See also selected() [p. 161], currentItem [p. 166] and selectionChanged() [p. 162].

void QListBox::highlighted (QListBoxItem *) [signal]

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

This signal is emitted when the user highlights a new current item. The argument is a pointer to the new current item.

See also selected() [p. 161], currentItem [p. 166] and selectionChanged() [p. 162].

int QListBox::inSort (const QListBoxItem * lbi)

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

Using this method is quite inefficient. We suggest to use insertItem() for inserting and sort() afterwards.

Inserts *lbi* at its sorted position in the list box and returns the position.

All items must be inserted with inSort() to maintain the sorting order. inSort() treats any pixmap (or user-defined type) as lexicographically less than any string.

See also insertItem() [p. 156] and sort() [p. 164].

int QListBox::inSort (const QString & text)

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

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

Using this method is quite inefficient. We suggest to use insertItem() for inserting and sort() afterwards.

Inserts a new item of *text* at its sorted position in the list box and returns the position.

All items must be inserted with inSort() to maintain the sorting order. inSort() treats any pixmap (or user-defined type) as lexicographically less than any string.

See also insertItem() [p. 156] and sort() [p. 164].

int QListBox::index (const QListBoxItem * lbi) const

Returns the index of *lbi*, or -1 if the item is not in this list box or *lbi* is a null pointer.

See also item() [p. 158].

void QListBox::insertItem (const QListBoxItem * lbi, int index = -1)

Inserts the item *lbi* into the list at position *index*.

If *index* is negative or larger than the number of items in the list box, *lbi* is inserted at the end of the list.

See also insertStrList() [p. 156].

Examples: i18n/mywidget.cpp, listbox/listbox.cpp, listboxcombo/listboxcombo.cpp and tabdialog/tabdialog.cpp.

void QListBox::insertItem (const QListBoxItem * lbi, const QListBoxItem * after)

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

Inserts the item *lbi* into the list after the item *after*.

If *after* is NULL, *lbi* is inserted at the beginning.

See also insertStrList() [p. 156].

void QListBox::insertItem (const QString & text, int index = -1)

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

Inserts a new list box text item with the text *text* into the list at position *index*.

If *index* is negative, *text* is inserted at the end of the list.

See also insertStrList() [p. 156].

void QListBox::insertItem (const QPixmap & pixmap, int index = -1)

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

Inserts a new list box pixmap item with the pixmap *pixmap* into the list at position *index*.

If *index* is negative, *pixmap* is inserted at the end of the list.

See also insertStrList() [p. 156].

void QListBox::insertItem (const QPixmap & pixmap, const QString & text, int index = -1)

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

Inserts a new list box pixmap item with the pixmap *pixmap* and the text *text* into the list at position *index*.

If *index* is negative, *pixmap* is inserted at the end of the list.

See also insertStrList() [p. 156].

void QListBox::insertStrList (const QStrList * list, int index = -1)

Inserts the string list *list* into the list at position *index*.

If *index* is negative, *list* is inserted at the end of the list. If *index* is too large, the operation is ignored.

Warning: This function uses `const char *` rather than QString, so we recommend against using it. It is provided so that legacy code will continue to work, and so that programs that certainly will not need to handle code outside

a single 8-bit locale can use it. See insertStringList() which uses real QStrings.

Warning: This function is never significantly faster than a loop around insertItem().

See also insertItem() [p. 156] and insertStringList() [p. 157].

void QListBox::insertStrList (const QStrList & list, int index = -1)

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

Inserts the string list *list* into the list at position *index*.

If *index* is negative, *list* is inserted at the end of the list. If *index* is too large, the operation is ignored.

Warning: This function uses `const char *` rather than QString, so we recommend against using it. It is provided so that legacy code will continue to work, and so that programs that certainly will not need to handle code outside a single 8-bit locale can use it. See insertStringList() which uses real QStrings.

Warning: This function is never significantly faster than a loop around insertItem().

See also insertItem() [p. 156] and insertStringList() [p. 157].

void QListBox::insertStrList (const char ** strings, int numStrings = -1, int index = -1)

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

Inserts the *numStrings* strings of the array *strings* into the list at position *index*.

If *index* is negative, insertStrList() inserts *strings* at the end of the list. If *index* is too large, the operation is ignored.

Warning: This function uses `const char *` rather than QString, so we recommend against using it. It is provided so that legacy code will continue to work, and so that programs that certainly will not need to handle code outside a single 8-bit locale can use it. See insertStringList() which uses real QStrings.

Warning: This function is never significantly faster than a loop around insertItem().

See also insertItem() [p. 156] and insertStringList() [p. 157].

void QListBox::insertStringList (const QStringList & list, int index = -1)

Inserts the string list *list* into the list at position *index*.

If *index* is negative, *list* is inserted at the end of the list. If *index* is too large, the operation is ignored.

Warning: This function is never significantly faster than a loop around insertItem().

See also insertItem() [p. 156] and insertStrList() [p. 156].

void QListBox::invertSelection () [virtual slot]

Inverts the selection. Only works in Multi and Extended selection mode.

bool QListBox::isMultiSelection () const

Returns TRUE if or not the list box is in Multi selection mode; otherwise returns FALSE. See the "multiSelection" [p. 166] property for details.

bool QListBox::isRubberSelecting () const [protected]

Returns whether the user is selecting items using a rubber band rectangle.

bool QListBox::isSelected (int i) const

Returns TRUE if item *i* is selected; otherwise returns FALSE.

bool QListBox::isSelected (const QListBoxItem * i) const

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

Returns TRUE if item *i* is selected; otherwise returns FALSE.

QListBoxItem * QListBox::item (int index) const

Returns a pointer to the item at position *index*, or 0 if *index* is out of bounds.

See also index() [p. 155].

Example: listboxcombo/listboxcombo.cpp.

QListBoxItem * QListBox::itemAt (const QPoint & p) const

Returns a pointer to the item at point *p*, which is in on-screen coordinates, or a null pointer if there is no item at *p*.

int QListBox::itemHeight (int index = 0) const

Returns the height in pixels of the item with index *index*. *index* defaults to 0.

If *index* is too large, this function returns 0.

QRect QListBox::itemRect (QListBoxItem * item) const

Returns the rectangle on the screen that *item* occupies in viewport()'s coordinates, or an invalid rectangle if *item* is a null pointer or is not currently visible.

bool QListBox::itemVisible (int index)

Returns TRUE if the item at position *index* is at least partly visible; otherwise returns FALSE.

bool QListBox::itemVisible (const QListBoxItem * item)

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

Returns TRUE if *item* is at least partly visible; otherwise returns FALSE.

bool QListBox::itemYPos (int index, int * yPos) const [protected]

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

Returns the vertical pixel-coordinate in **yPos*, of the list box item at position *index* in the list. Returns FALSE if the item is outside the visible area.

long QListBox::maxItemWidth () const

Returns the width of the widest item in the list box.

void QListBox::mouseButtonClicked (int button, QListBoxItem * item, const QPoint & pos) [signal]

This signal is emitted when the user clicks mouse button *button*. If *item* is non-null, the cursor is on *item*. If *item* is null, the mouse cursor isn't on any item.

pos is the position of the mouse cursor in the global coordinate system (QMouseEvent::globalPos()). (If the click's press and release differ by a pixel or two, *pos* is the position at release time.)

Note that you must not delete any QListBoxItem objects in slots connected to this signal.

void QListBox::mouseButtonPressed (int button, QListBoxItem * item, const QPoint & pos) [signal]

This signal is emitted when the user presses mouse button *button*. If *item* is non-null, the cursor is on *item*. If *item* is null, the mouse cursor isn't on any item.

pos is the position of the mouse cursor in the global coordinate system (QMouseEvent::globalPos()).

Note that you must not delete any QListBoxItem objects in slots connected to this signal.

int QListBox::numCols () 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.

See also numColumns [p. 166].

int QListBox::numColumns () const

Returns the number of columns in the list box. See the "numColumns" [p. 166] property for details.

int QListBox::numItemsVisible () const

Returns the number of visible items. See the "numItemsVisible" [p. 166] property for details.

int QListBox::numRows () const

Returns the number of rows in the list box. See the "numRows" [p. 166] property for details.

void QListBox::onItem (QListBoxItem * i) [signal]

This signal is emitted when the user moves the mouse cursor onto an item, similar to the QWidget::enterEvent() function. *i* is the QListBoxItem that the mouse has moved on.

void QListBox::onViewport () [signal]

This signal is emitted when the user moves the mouse cursor from an item to an empty part of the list box.

void QListBox::paintCell (QPainter * p, int row, int col) [virtual protected]

Provided for compatibility with the old QListBox. We recommend using QListBoxItem::paint()

Repaints the cell at *row*, *col* using painter *p*.

const QPixmap * QListBox::pixmap (int index) const

Returns a pointer to the pixmap at position *index*, or 0 if there is no pixmap there.

See also text() [p. 164].

void QListBox::pressed (QListBoxItem * item) [signal]

This signal is emitted when the user presses any mouse button. If *item* is non-null, the cursor is on *item*. If *item* is null, the mouse cursor isn't on any item.

Note that you must not delete any QListBoxItem objects in slots connected to this signal.

void QListBox::pressed (QListBoxItem * item, const QPoint & pnt) [signal]

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

This signal is emitted when the user presses any mouse button. If *item* is non-null, the cursor is on *item*. If *item* is null, the mouse cursor isn't on any item.

pnt is the position of the mouse cursor in the global coordinate system (QMouseEvent::globalPos()). (If the click's press and release differ by a pixel or two, *pnt* is the position at release time.)

Note that you must not delete any QListBoxItem objects in slots connected to this signal.

See also mouseButtonPressed() [p. 159], rightButtonPressed() [p. 161] and clicked() [p. 153].

void QListBox::removeItem (int index)

Removes and deletes the item at position *index*. If *index* is equal to currentItem(), a new item gets highlighted and the highlighted() signal is emitted.

See also insertItem() [p. 156] and clear() [p. 153].

void QListBox::returnPressed (QListBoxItem *) [signal]

This signal is emitted when Enter or Return is pressed. The argument is currentItem().

void QListBox::rightButtonClicked (QListBoxItem *, const QPoint &) [signal]

This signal is emitted when the right button is clicked (i.e. when it's released at the same point where it was pressed). The arguments are the relevant QListBoxItem (may be 0) and the point in global coordinates.

void QListBox::rightButtonPressed (QListBoxItem *, const QPoint &) [signal]

This signal is emitted when the right button is pressed. The arguments are the relevant QListBoxItem (may be 0) and the point in global coordinates.

LayoutMode QListBox::rowMode () const

Returns the row layout mode for this list box. See the "rowMode" [p. 167] property for details.

bool QListBox::scrollBar () const

Returns FALSE if vScrollBarMode() is AlwaysOff; otherwise returns TRUE.

void QListBox::selectAll (bool select) [virtual slot]

In Multi and Extended modes, this function sets all items to be selected if *select* is TRUE, and to be unselected if *select* is FALSE.

In Single and NoSelection modes, this function only changes the selection status of currentItem().

void QListBox::selected (int index) [signal]

This signal is emitted when the user double-clicks on an item or presses Enter when an item is highlighted. *index* is the index of the selected item.

See also highlighted() [p. 155] and selectionChanged() [p. 162].

void QListBox::selected (const QString &) [signal]

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

This signal is emitted when the user double-clicks on an item or presses Enter while an item is highlighted, and the selected item is (or has) a string. The argument is the text of the selected item.

See also highlighted() [p. 155] and selectionChanged() [p. 162].

void QListBox::selected (QListBoxItem *) [signal]

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

This signal is emitted when the user double-clicks on an item or presses Enter when an item is highlighted. The argument is a pointer to the new selected item.

See also highlighted() [p. 155] and selectionChanged() [p. 162].

void QListBox::selectionChanged () [signal]

This signal is emitted when the selection set of a list box changes. This signal is emitted in each selection mode. If the user selects five items by drag-selecting, QListBox tries to emit just one selectionChanged() signal so the signal can be connected to computationally expensive slots.

See also selected() [p. 161] and currentItem [p. 166].

void QListBox::selectionChanged (QListBoxItem * item) [signal]

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

This signal is emitted when the selection in a Single selection list box changes. *item* is the new selected list box item.

See also selected() [p. 161] and currentItem [p. 166].

SelectionMode QListBox::selectionMode () const

Returns the selection mode of the list box. See the "selectionMode" [p. 167] property for details.

void QListBox::setAutoBottomScrollBar (bool enable)

If *enable* is TRUE sets setHScrollBarMode() to AlwaysOn; otherwise sets setHScrollBarMode() to AlwaysOff.

void QListBox::setAutoScrollBar (bool enable)

If *enable* is TRUE sets setVScrollBarMode() to AlwaysOn; otherwise sets setVScrollBarMode() to AlwaysOff.

void QListBox::setBottomItem (int index) [virtual]

Scrolls the list box so the item at position *index* in the list is displayed in the bottom row of the list box.

See also topItem [p. 167].

void QListBox::setBottomScrollBar (bool enable)

If *enable* is TRUE sets setHScrollBarMode() to AlwaysOn; otherwise sets setHScrollBarMode() to AlwaysOff.

void QListBox::setColumnMode (LayoutMode) [virtual]

Sets the column layout mode for this list box. See the "columnMode" [p. 165] property for details.

void QListBox::setColumnMode (int columns) [virtual]

Sets the column layout mode for this list box to *columns*. See the "columnMode" [p. 165] property for details.

void QListBox::setCurrentItem (int index) [virtual]

Sets the current highlighted item to *index*. See the "currentItem" [p. 166] property for details.

void QListBox::setCurrentItem (QListBoxItem * i) [virtual]

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

Sets the current item to the QListBoxItem *i*.

void QListBox::setMultiSelection (bool multi)

Sets whether or not the list box is in Multi selection mode to *multi*. See the "multiSelection" [p. 166] property for details.

void QListBox::setRowMode (LayoutMode) [virtual]

Sets the row layout mode for this list box. See the "rowMode" [p. 167] property for details.

void QListBox::setRowMode (int rows) [virtual]

Sets the row layout mode for this list box to *rows*. See the "rowMode" [p. 167] property for details.

void QListBox::setScrollBar (bool enable)

If *enable* is TRUE sets setVScrollBarMode() to AlwaysOn; otherwise sets setVScrollBarMode() to AlwaysOff.

void QListBox::setSelected (QListBoxItem * item, bool select) [virtual]

Selects *item* if *select* is TRUE or unselects it if *select* is FALSE, and repaints the item appropriately.

If the list box is a Single selection list box and *select* is TRUE, setSelected() calls setCurrentItem().

If the list box is a Single selection list box, *select* is FALSE, setSelected() calls clearSelection().

Note that for this function NoSelection means Multi selection. The user cannot select items in a NoSelection list box, but the application programmer can.

See also multiSelection [p. 166], currentItem [p. 166], clearSelection() [p. 153] and currentItem [p. 166].

void QListBox::setSelected (int index, bool select)

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

If *select* is TRUE the item at position *index* is selected; otherwise the item is deselected.

void QListBox::setSelectionMode (SelectionMode) [virtual]

Sets the selection mode of the list box. See the "selectionMode" [p. 167] property for details.

void QListBox::setTopItem (int index) [virtual]

Sets the index of an item at the top of the screen to *index*. See the "topItem" [p. 167] property for details.

void QListBox::setVariableHeight (bool) [virtual]

Sets whether this list box has variable-height rows. See the "variableHeight" [p. 167] property for details.

void QListBox::setVariableWidth (bool) [virtual]

Sets whether this list box has variable-width columns. See the "variableWidth" [p. 167] property for details.

void QListBox::sort (bool ascending = TRUE)

If *ascending* is TRUE sorts the items in ascending order; otherwise sorts in descending order.

To compare the items, the text (QListBoxItem::text()) of the items is used.

Example: listbox/listbox.cpp.

void QListBox::takeItem (const QListBoxItem * item)

Removes *item* from the list box and causes an update of the screen display. The item is not deleted. You should normally not need to call this function because QListBoxItem::~QListBoxItem() calls it. The normal way to delete an item is with `delete`.

See also QListBox::insertItem() [p. 156].

QString QListBox::text (int index) const

Returns the text at position *index*, or a null string if there is no text at that position.

See also pixmap() [p. 160].

void QListBox::toggleCurrentItem () [protected]

Toggles the selection status of currentItem() and repaints if the list box is a Multi selection list box.

See also multiSelection [p. 166].

int QListBox::topItem () const

Returns the index of an item at the top of the screen. See the "topItem" [p. 167] property for details.

int QListBox::totalHeight () const [protected]

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

Returns contentsHeight().

int QListBox::totalWidth () const [protected]

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

Returns contentsWidth().

void QListBox::triggerUpdate (bool doLayout)

Ensures that a single paint event will occur at the end of the current event loop iteration. If *doLayout* is TRUE, the layout is also redone.

void QListBox::updateItem (int index) [protected]

Repaints the item at position *index* in the list.

void QListBox::updateItem (QListBoxItem * i) [protected]

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

Repaints the QListBoxItem *i*.

bool QListBox::variableHeight () const

Returns TRUE if this list box has variable-height rows; otherwise returns FALSE. See the "variableHeight" [p. 167] property for details.

bool QListBox::variableWidth () const

Returns TRUE if this list box has variable-width columns; otherwise returns FALSE. See the "variableWidth" [p. 167] property for details.

Property Documentation

LayoutMode columnMode

This property holds the column layout mode for this list box.

Set this property's value with setColumnMode() and get this property's value with columnMode().

See also rowMode [p. 167] and columnMode [p. 165].

setColumnMode() sets the layout mode and adjusts the number of displayed columns. The row layout mode automatically becomes Variable, unless the column mode is Variable.

See also rowMode [p. 167] and columnMode [p. 165].

uint count

This property holds the number of items in the list box.

Get this property's value with count().

int currentItem

This property holds the current highlighted item.

When setting this property, the highlighting is moved and the list box scrolled as necessary.

If no item has been selected, currentItem() returns -1.

Set this property's value with setCurrentItem() and get this property's value with currentItem().

QString currentText

This property holds the text of the current item.

This is equivalent to text(currentItem()).

Get this property's value with currentText().

bool multiSelection

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

This property holds whether or not the list box is in Multi selection mode.

Consider using the QListBox::selectionMode property instead of this property.

When setting this property, Multi selection mode is used if set to TRUE and to Single selection mode if set to FALSE.

When getting this property, TRUE is returned if the list box is in Multi selection mode or Extended selection mode, and FALSE if it is in Single selection mode or NoSelection mode.

See also selectionMode [p. 167].

Set this property's value with setMultiSelection() and get this property's value with isMultiSelection().

int numColumns

This property holds the number of columns in the list box.

This is normally 1, but can be different if QListBox::columnMode or QListBox::rowMode has been set.

See also columnMode [p. 165], rowMode [p. 167] and numRows [p. 166].

Get this property's value with numColumns().

int numItemsVisible

This property holds the number of visible items.

Both partially and entirely visible items are counted.

Get this property's value with numItemsVisible().

int numRows

This property holds the number of rows in the list box.

This is equal to the number of items in the default single-column layout, but can be different.

See also columnMode [p. 165], rowMode [p. 167] and numColumns [p. 166].

Get this property's value with numRows().

LayoutMode rowMode

This property holds the row layout mode for this list box.

This property is normally Variable.

setRowMode() sets the layout mode and adjusts the number of displayed rows. The column layout mode automatically becomes Variable, unless the row mode is Variable.

See also columnMode [p. 165] and rowMode [p. 167].

Set this property's value with setRowMode() and get this property's value with rowMode().

SelectionMode selectionMode

This property holds the selection mode of the list box.

Sets the list box's selection mode, which may be one of Single (the default), Extended, Multi or NoSelection.

See also SelectionMode [p. 150].

Set this property's value with setSelectionMode() and get this property's value with selectionMode().

int topItem

This property holds the index of an item at the top of the screen.

When getting this property and the listbox has multiple columns, an arbitrary item is selected and returned.

When setting this property, the list box is scrolled so the item at position *index* in the list is displayed in the top row of the list box.

Set this property's value with setTopItem() and get this property's value with topItem().

bool variableHeight

This property holds whether this list box has variable-height rows.

When the list box has variable-height rows (the default), each row is as high as the highest item in that row. When it has same-sized rows, all rows are as high as the highest item in the list box.

See also variableWidth [p. 167].

Set this property's value with setVariableHeight() and get this property's value with variableHeight().

bool variableWidth

This property holds whether this list box has variable-width columns.

When the list box has variable-width columns, each column is as wide as the widest item in that column. When it has same-sized columns (the default), all columns are as wide as the widest item in the list box.

See also variableHeight [p. 167].

Set this property's value with setVariableWidth() and get this property's value with variableWidth().

QListBoxItem Class Reference

The QListBoxItem class is the base class of all list box items.

`#include <qlistbox.h>`

Inherited by QListBoxText [p. 175] and QListBoxPixmap [p. 172].

Public Members

- **QListBoxItem** (QListBox * listbox = 0)
- **QListBoxItem** (QListBox * listbox, QListBoxItem * after)
- virtual **~QListBoxItem** ()
- virtual QString **text** () const
- virtual const QPixmap * **pixmap** () const
- virtual int **height** (const QListBox * lb) const
- virtual int **width** (const QListBox * lb) const
- bool **isSelected** () const
- bool **isCurrent** () const
- bool selected () const *(obsolete)*
- bool current () const *(obsolete)*
- QListBox * **listBox** () const
- void **setSelectable** (bool b)
- bool **isSelectable** () const
- QListBoxItem * **next** () const
- QListBoxItem * **prev** () const
- virtual int **rtti** () const

Protected Members

- virtual void **paint** (QPainter * p)
- virtual void **setText** (const QString & text)
- void **setCustomHighlighting** (bool b)

Detailed Description

The QListBoxItem class is the base class of all list box items.

This class is an abstract base class used for all list box items. If you need to insert customized items into a QListBox you must inherit this class and reimplement paint(), height() and width().

See also QListBox [p. 146] and Advanced Widgets.

Member Function Documentation

QListBoxItem::QListBoxItem (QListBox * listbox = 0)

Constructs an empty list box item in the list box *listbox*.

QListBoxItem::QListBoxItem (QListBox * listbox, QListBoxItem * after)

Constructs an empty list box item in the list box *listbox* and inserts it after the item *after*.

QListBoxItem::~QListBoxItem () [virtual]

Destroys the list box item.

bool QListBoxItem::current () const

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

int QListBoxItem::height (const QListBox * lb) const [virtual]

Implement this function to return the height of your item. The *lb* parameter is the same as listBox() and is provided for convenience and compatibility.

See also paint() [p. 170] and width() [p. 171].

Reimplemented in QListBoxText and QListBoxPixmap.

bool QListBoxItem::isCurrent () const

Returns TRUE if the item is the current item; otherwise returns FALSE.

See also QListBox::currentItem [p. 166], QListBox::item() [p. 158] and isSelected() [p. 169].

bool QListBoxItem::isSelectable () const

Returns TRUE if this item is selectable; otherwise returns FALSE.

See also setSelectable() [p. 171].

bool QListBoxItem::isSelected () const

Returns TRUE if the item is selected; otherwise returns FALSE.

See also QListBox::isSelected() [p. 158] and isCurrent() [p. 169].

Example: listboxcombo/listboxcombo.cpp.

QListBox * QListBoxItem::listBox () const

Returns a pointer to the list box containing this item.

QListBoxItem * QListBoxItem::next () const

Returns the item that comes after this in the list box. If this is the last item, a null pointer is returned.

See also prev() [p. 170].

void QListBoxItem::paint (QPainter * p) [virtual protected]

Implement this function to draw your item. You will need to pass the QPainter that will draw the item in *p*.

See also height() [p. 169] and width() [p. 171].

Example: listboxcombo/listboxcombo.cpp.

Reimplemented in QListBoxText and QListBoxPixmap.

const QPixmap * QListBoxItem::pixmap () const [virtual]

Returns the pixmap associated with the item, if any.

The default implementation returns a null pointer.

Example: listboxcombo/listboxcombo.cpp.

Reimplemented in QListBoxPixmap.

QListBoxItem * QListBoxItem::prev () const

Returns the item which comes before this in the list box. If this is the first item, a null pointer is returned.

See also next() [p. 170].

int QListBoxItem::rtti () const [virtual]

Returns 0.

Although often frowned upon by purists, Run Time Type Identification is very useful in this case, as it allows a QListBox to be an efficient indexed storage mechanism.

Make your derived classes return their own values for rtti(), and you can distinguish between listbox items. You should use values greater than 1000 preferably a large random number, to allow for extensions to this class.

bool QListBoxItem::selected () const

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

void QListBoxItem::setCustomHighlighting (bool b) [protected]

Defines whether the list box item is responsible for drawing itself in a highlighted state when being selected.

If *b* is FALSE (the default), the list box will draw some default highlight indicator before calling paint().

See also selected() [p. 170] and paint() [p. 170].

void QListBoxItem::setSelectable (bool b)

If *b* is TRUE then this item can be selected by the user; otherwise this item cannot be selected by the user.

See also isSelectable() [p. 169].

void QListBoxItem::setText (const QString & text) [virtual protected]

Sets the text of the QListBoxItem to *text*. This *text* is also used for sorting. The text is not shown unless explicitly drawn in paint().

See also text() [p. 171].

QString QListBoxItem::text () const [virtual]

Returns the text of the item. This text is also used for sorting.

See also setText() [p. 171].

Example: listboxcombo/listboxcombo.cpp.

int QListBoxItem::width (const QListBox * lb) const [virtual]

Implement this function to return the width of your item. The *lb* parameter is the same as listBox() and is provided for convenience and compatibility.

See also paint() [p. 170] and height() [p. 169].

Reimplemented in QListBoxText and QListBoxPixmap.

QListBoxPixmap Class Reference

The QListBoxPixmap class provides list box items with a pixmap and optional text.

`#include <qlistbox.h>`

Inherits QListBoxItem [p. 168].

Public Members

- **QListBoxPixmap** (QListBox * listbox, const QPixmap & pixmap)
- **QListBoxPixmap** (const QPixmap & pixmap)
- **QListBoxPixmap** (QListBox * listbox, const QPixmap & pixmap, QListBoxItem * after)
- **QListBoxPixmap** (QListBox * listbox, const QPixmap & pix, const QString & text)
- **QListBoxPixmap** (const QPixmap & pix, const QString & text)
- **QListBoxPixmap** (QListBox * listbox, const QPixmap & pix, const QString & text, QListBoxItem * after)
- **~QListBoxPixmap** ()
- virtual const QPixmap * **pixmap** () const
- virtual int **height** (const QListBox * lb) const
- virtual int **width** (const QListBox * lb) const

Protected Members

- virtual void **paint** (QPainter * painter)

Detailed Description

The QListBoxPixmap class provides list box items with a pixmap and optional text.

Items of this class are drawn with the pixmap on the left with the optional text to the right of the pixmap.

See also QListBox [p. 146], QListBoxItem [p. 168] and Advanced Widgets.

Member Function Documentation

QListBoxPixmap::QListBoxPixmap (QListBox * listbox, const QPixmap & pixmap)

Constructs a new list box item in list box *listbox* showing the pixmap *pixmap*.

QListBoxPixmap::QListBoxPixmap (const QPixmap & pixmap)

Constructs a new list box item showing the pixmap *pixmap*.

QListBoxPixmap::QListBoxPixmap (QListBox * listbox, const QPixmap & pixmap, QListBoxItem * after)

Constructs a new list box item in list box *listbox* showing the pixmap *pixmap*. The item gets inserted after the item *after*.

QListBoxPixmap::QListBoxPixmap (QListBox * listbox, const QPixmap & pix, const QString & text)

Constructs a new list box item in list box *listbox* showing the pixmap *pix* and the text *text*.

QListBoxPixmap::QListBoxPixmap (const QPixmap & pix, const QString & text)

Constructs a new list box item showing the pixmap *pix* and the text to *text*.

QListBoxPixmap::QListBoxPixmap (QListBox * listbox, const QPixmap & pix, const QString & text, QListBoxItem * after)

Constructs a new list box item in list box *listbox* showing the pixmap *pix* and the string *text*. The item gets inserted after the item *after*.

QListBoxPixmap::~QListBoxPixmap ()

Destroys the item.

int QListBoxPixmap::height (const QListBox * lb) const [virtual]

Returns the height of the pixmap in list box *lb*.

See also paint() [p. 173] and width() [p. 174].

Reimplemented from QListBoxItem [p. 169].

void QListBoxPixmap::paint (QPainter * painter) [virtual protected]

Draws the pixmap using *painter*.

Reimplemented from QListBoxItem [p. 170].

const QPixmap * QListBoxPixmap::pixmap () const [virtual]

Returns the pixmap associated with the item.

Reimplemented from QListBoxItem [p. 170].

int QListBoxPixmap::width (const QListBox * lb) const [virtual]

Returns the width of the pixmap plus some margin in list box *lb*.

See also paint() [p. 173] and height() [p. 173].

Reimplemented from QListBoxItem [p. 171].

QListBoxText Class Reference

The QListBoxText class provides list box items that display text.

`#include <qlistbox.h>`

Inherits QListBoxItem [p. 168].

Public Members

- **QListBoxText** (QListBox * listbox, const QString & text = QString::null)
- **QListBoxText** (const QString & text = QString::null)
- **QListBoxText** (QListBox * listbox, const QString & text, QListBoxItem * after)
- **~QListBoxText** ()
- virtual int **height** (const QListBox * lb) const
- virtual int **width** (const QListBox * lb) const

Protected Members

- virtual void **paint** (QPainter * painter)

Detailed Description

The QListBoxText class provides list box items that display text.

The text is drawn in the widget's current font. If you need several different fonts, you must implement your own subclass of QListBoxItem.

See also QListBox [p. 146], QListBoxItem [p. 168] and Advanced Widgets.

Member Function Documentation

QListBoxText::QListBoxText (QListBox * listbox, const QString & text = QString::null)

Constructs a list box item in list box *listbox* showing the text *text*.

QListBoxText::QListBoxText (const QString & text = QString::null)

Constructs a list box item showing the text *text*.

QListBoxText::QListBoxText (QListBox * listbox, const QString & text, QListBoxItem * after)

Constructs a list box item in list box *listbox* showing the text *text*. The item gets inserted after the item *after*.

QListBoxText::~QListBoxText ()

Destroys the item.

int QListBoxText::height (const QListBox * lb) const [virtual]

Returns the height of a line of text in list box *lb*.

See also paint() [p. 176] and width() [p. 176].

Reimplemented from QListBoxItem [p. 169].

void QListBoxText::paint (QPainter * painter) [virtual protected]

Draws the text using *painter*.

Reimplemented from QListBoxItem [p. 170].

int QListBoxText::width (const QListBox * lb) const [virtual]

Returns the width of this line in list box *lb*.

See also paint() [p. 176] and height() [p. 176].

Reimplemented from QListBoxItem [p. 171].

QListView Class Reference

The QListView class implements a list/tree view.

`#include <qlistview.h>`

Inherits QScrollView [p. 259].

Public Members

- **QListView** (QWidget * parent = 0, const char * name = 0, WFlags f = 0)
- **~QListView** ()
- int **treeStepSize** () const
- virtual void **setTreeStepSize** (int)
- virtual void **insertItem** (QListViewItem * i)
- virtual void **takeItem** (QListViewItem * i)
- virtual void removeItem (QListViewItem * item) *(obsolete)*
- QHeader * **header** () const
- virtual int **addColumn** (const QString & label, int width = -1)
- virtual int **addColumn** (const QIconSet & iconset, const QString & label, int width = -1)
- virtual void **removeColumn** (int index)
- virtual void **setColumnText** (int column, const QString & label)
- virtual void **setColumnText** (int column, const QIconSet & iconset, const QString & label)
- QString **columnText** (int c) const
- virtual void **setColumnWidth** (int column, int w)
- int **columnWidth** (int c) const
- enum **WidthMode** { Manual, Maximum }
- virtual void **setColumnWidthMode** (int c, WidthMode mode)
- WidthMode **columnWidthMode** (int c) const
- int **columns** () const
- virtual void **setColumnAlignment** (int column, int align)
- int **columnAlignment** (int column) const
- QListViewItem * **itemAt** (const QPoint & viewPos) const
- QRect **itemRect** (const QListViewItem * i) const
- int **itemPos** (const QListViewItem * item)
- void **ensureItemVisible** (const QListViewItem * i)
- void **repaintItem** (const QListViewItem * item) const
- virtual void **setMultiSelection** (bool enable)
- bool **isMultiSelection** () const
- enum **SelectionMode** { Single, Multi, Extended, NoSelection }
- void **setSelectionMode** (SelectionMode mode)
- SelectionMode **selectionMode** () const

- virtual void **clearSelection** ()
- virtual void **setSelected** (QListViewItem * item, bool selected)
- bool **isSelected** (const QListViewItem * i) const
- QListViewItem * **selectedItem** () const
- virtual void **setOpen** (QListViewItem * item, bool open)
- bool **isOpen** (const QListViewItem * item) const
- virtual void **setCurrentItem** (QListViewItem * i)
- QListViewItem * **currentItem** () const
- QListViewItem * **firstChild** () const
- QListViewItem * **lastItem** () const
- int **childCount** () const
- virtual void **setAllColumnsShowFocus** (bool)
- bool **allColumnsShowFocus** () const
- virtual void **setItemMargin** (int)
- int **itemMargin** () const
- virtual void **setRootIsDecorated** (bool)
- bool **rootIsDecorated** () const
- virtual void **setSorting** (int column, bool ascending = TRUE)
- virtual void **sort** ()
- virtual bool **eventFilter** (QObject * o, QEvent * e)
- virtual void **setShowSortIndicator** (bool show)
- bool **showSortIndicator** () const
- virtual void **setShowToolTips** (bool b)
- bool **showToolTips** () const
- enum **ResizeMode** { NoColumn, AllColumns, LastColumn }
- virtual void **setResizeMode** (ResizeMode m)
- ResizeMode **resizeMode** () const
- QListViewItem * **findItem** (const QString & text, int column, ComparisonFlags compare = ExactMatch | CaseSensitive) const
- enum **RenameAction** { Accept, Reject }
- virtual void **setDefaultRenameAction** (RenameAction a)
- RenameAction **defaultRenameAction** () const
- bool **isRenaming** () const

Public Slots

- virtual void **clear** ()
- virtual void **invertSelection** ()
- virtual void **selectAll** (bool select)
- void **triggerUpdate** ()

Signals

- void **selectionChanged** ()
- void **selectionChanged** (QListViewItem *)
- void **currentChanged** (QListViewItem *)
- void **clicked** (QListViewItem * item)
- void **clicked** (QListViewItem * item, const QPoint & pnt, int c)

- void **pressed** (QListViewItem * item)
- void **pressed** (QListViewItem * item, const QPoint & pnt, int c)
- void **doubleClicked** (QListViewItem * item)
- void **returnPressed** (QListViewItem *)
- void **spacePressed** (QListViewItem *)
- void **rightButtonClicked** (QListViewItem *, const QPoint &, int)
- void **rightButtonPressed** (QListViewItem *, const QPoint &, int)
- void **mouseButtonPressed** (int button, QListViewItem * item, const QPoint & pos, int c)
- void **mouseButtonClicked** (int button, QListViewItem * item, const QPoint & pos, int c)
- void **contextMenuRequested** (QListViewItem * item, const QPoint & pos, int col)
- void **onItem** (QListViewItem * i)
- void **onViewport** ()
- void **expanded** (QListViewItem * item)
- void **collapsed** (QListViewItem * item)
- void **dropped** (QDropEvent * e)
- void **itemRenamed** (QListViewItem * item, int col, const QString & text)
- void **itemRenamed** (QListViewItem * item, int col)

Properties

- bool **allColumnsShowFocus** — whether items should show keyboard focus using all columns
- int **childCount** — the number of parentless (top level) QListViewItem objects in this QListView *(read only)*
- int **columns** — the number of columns in this list view *(read only)*
- RenameAction **defaultRenameAction** — whether the list view accepts the rename operation by default
- int **itemMargin** — the advisory item margin that list items may use
- bool **multiSelection** — whether the list view is in multi-selection or single selection mode
- ResizeMode **resizeMode** — whether all, none or the last column should be resized
- bool **rootIsDecorated** — whether the list view show open/close signs on root items
- SelectionMode **selectionMode** — the list view's multi-selection mode
- bool **showSortIndicator** — whether the list view header should display a sort indicator
- bool **showToolTips** — whether this list view should show tooltips for truncated column texts
- int **treeStepSize** — the number of pixels a child is offset from its parent

Protected Members

- virtual void **contentsMousePressEvent** (QMouseEvent * e)
- virtual void **contentsMouseReleaseEvent** (QMouseEvent * e)
- virtual void **contentsMouseMoveEvent** (QMouseEvent * e)
- virtual void **contentsMouseDoubleClickEvent** (QMouseEvent * e)
- virtual QDragObject * **dragObject** ()
- virtual void **startDrag** ()
- virtual void **resizeEvent** (QResizeEvent * e)
- virtual void **drawContentsOffset** (QPainter * p, int ox, int oy, int cx, int cy, int cw, int ch)
- virtual void **paintEmptyArea** (QPainter * p, const QRect & rect)

Protected Slots

- void **updateContents** ()
- void **doAutoScroll** ()

Detailed Description

The QListView class implements a list/tree view.

It can display and control a hierarchy of multi-column items, and provides the ability to add new items at any time. Among others the user may select one or many items and sort the list in increasing or decreasing order by any column.

The simplest mode of use is to create a QListView, add some column headers using addColumn() and create one or more QListViewItem objects with the QListView as parent:

```
QListView * table;

table->addColumn( "Qualified name" );
table->addColumn( "Namespace" );

    element = new QListViewItem( table, qName, namespaceURI );
```

Further nodes can be added to the listview object (the root of the tree) or as child nodes to QListViewItems:

```
    for ( int i = 0 ; i < attributes.length(); i++ ) {
        new QListViewItem( element, attributes.qName(i), attributes.uri(i) );
    }
```

(From xml/tagreader-with-features/structureparser.cpp)

The main setup functions are

- addColumn() - adds a column with text and perhaps width.
- setColumnWidthMode() - sets the column to be resized automatically or not.
- setAllColumnsShowFocus() - sets whether items should show keyboard focus using all columns or just column 0. The default is to show focus using just column 0.
- setRootIsDecorated() - sets whether root items can be opened and closed by the user and have open/close decoration to their left. The default is FALSE.
- setTreeStepSize() - sets how many pixels an item's children are indented relative to their parent. The default is 20. This is mostly a matter of taste.
- setSorting() - sets whether the items should be sorted, whether it should be in ascending or descending order, and by what column it should be sorted. By default the list view is sorted by the first column; to switch this off call setSorting(-1).

To handle events such as mouse presses on the list view, derived classes can reimplement the QScrollView functions contentsMousePressEvent, contentsMouseReleaseEvent, contentsMouseDoubleClickEvent, contentsMouseMoveEvent, contentsDragEnterEvent, contentsDragMoveEvent, contentsDragLeaveEvent, contentsDropEvent, and contentsWheelEvent.

There are also several functions for mapping between items and coordinates. itemAt() returns the item at a position on-screen, itemRect() returns the rectangle an item occupies on the screen, and itemPos() returns the position of any item (not on-screen in the list view). firstChild() returns the item at the top of the view (not

necessarily on-screen) so you can iterate over the items using either QListViewItem::itemBelow() or a combination of QListViewItem::firstChild() and QListViewItem::nextSibling().

If you need to move a list view item you can use takeItem() and insertItem(). Item's are deleted with delete; to delete all items use clear(). See the QListViewItem documentation for examples of traversal.

There are a variety of selection modes described in the QListView::SelectionMode documentation. The default is Single selection, which you can change using setSelectionMode().

Because QListView offers multiple selection it has to display keyboard focus and selection state separately. Therefore there are functions both to set the selection state of an item (setSelected()) and to select which item displays keyboard focus (setCurrentItem()).

QListView emits two groups of signals; one group signals changes in selection/focus state and one signals selection. The first group consists of selectionChanged() (applicable to all list views), selectionChanged(QListViewItem *) (applicable only to Single selection list view), and currentChanged(QListViewItem *). The second group consists of doubleClicked(QListViewItem *), returnPressed(QListViewItem *) and rightButtonClicked(QListViewItem *, const QPoint&, int), etc.

In Motif style, QListView deviates fairly strongly from the look and feel of the Motif hierarchical tree view. This is done mostly to provide a usable keyboard interface and to make the list view look better with a white background.

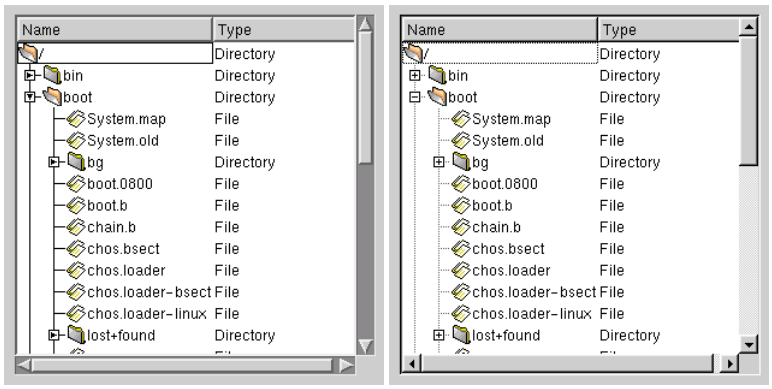
If selectionMode() is Single (the default) the user can select one item at a time, e.g. by clicking an item with the mouse, see QListView::SelectionMode for details.

The listview can be navigated either using the mouse or the keyboard. Clicking an - icon closes an item (hides its children) and clicking an + icon opens an item (shows its children). The keyboard controls are these:

- *Home* - Make the first item current and visible.
- *End* - Make the last item current and visible.
- *Page Up* - Make the item above the top visible item current and visible.
- *Page Down* - Make the item below the bottom visible item current and visible.
- *Up Arrow* - Make the item above the current item current and visible.
- *Down Arrow* - Make the item below the current item current and visible.
- *Left Arrow* - If the current item is closed (+ icon) or has no children make its parent item current and visible. If the current item is open (- icon) close it, i.e. hide its children. Exception: if the current item is the first item and is closed and the horizontal scrollbar is offset to the right the listview will be scrolled left.
- *Right Arrow* - If the current item is closed (+ icon) and has children the item is opened. If the current item is opened (- icon) and has children the item's first child is made current and visible. If the current item has no children the listview is scrolled right.

If the user starts typing letters with the focus in the listview an incremental search will occur. For example if the user types 'd' the current item will change to the first item that begins with the letter 'd'; if they then type 'a', the current item will change to the first item that begins with 'da', and so on. If no item begins with the letters they type the current item doesn't change.

Warning: The list view assumes ownership of all list view items and will delete them when it does not need them any more.



See also Advanced Widgets.

Member Type Documentation

QListView::RenameAction

This enum describes whether a rename operation is accepted if the rename editor loses focus without the user pressing Enter.

- `QListView::Accept` - Rename if Enter is pressed or focus is lost.
- `QListView::Reject` - Discard the rename operation if focus is lost (and Enter has not been pressed).

QListView::ResizeMode

This enum describes how the header adjusts to resize events which affect the width of the listview.

- `QListView::NoColumn` - The columns do not get resized in resize events.
- `QListView::AllColumns` - All columns are resized equally to fit the width of the listview.
- `QListView::LastColumn` - The last columns is resized to fit the with of the listview.

QListView::SelectionMode

This enumerated type is used by QListView to indicate how it reacts to selection by the user. It has four values:

- `QListView::Single` - When the user selects an item, any already-selected item becomes unselected, and the user cannot unselect the selected item. This means that the user can never clear the selection, even though the selection may be cleared by the application programmer using QListView::clearSelection().
- `QListView::Multi` - When the user selects an item in the most ordinary way, the selection status of that item is toggled and the other items are left alone.
- `QListView::Extended` - When the user selects an item in the most ordinary way, the selection is cleared and the new item selected. However, if the user presses the CTRL key when clicking on an item, the clicked item gets toggled and all other items are left untouched. And if the user presses the SHIFT key while clicking on an item, all items between the current item and the clicked item get selected or unselected, depending on the state of the clicked item. Also, multiple items can be selected by dragging the mouse while the left mouse button stays pressed.
- `QListView::NoSelection` - Items cannot be selected.

In other words, Single is a real single-selection list view, Multi a real multi-selection list view, Extended is a list view where users can select multiple items but usually want to select either just one or a range of contiguous items, and NoSelection is a list view where the user can look but not touch.

QListView::WidthMode

This enum type describes how the width of a column in the view changes. The currently defined modes are:

- `QListView::Manual` - the column width does not change automatically.
- `QListView::Maximum` - the column is automatically sized according to the widths of all items in the column. (Note: The column never shrinks in this case.) This means the column is always resized to the width of the item with the largest width in the column.

See also setColumnWidth() [p. 193], setColumnWidthMode() [p. 193] and columnWidth() [p. 185].

Member Function Documentation

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

Constructs a new empty list view, with *parent* as a parent and *name* as object name.

Performance is boosted by modifying the widget flags *f* so that only part of the QListViewItem children is redrawn. This may be unsuitable for custom QListViewItem classes, in which case WStaticContents and WRepaintNoErase should be cleared.

See also QWidget::clearWFlags() [p. 424] and Qt::WidgetFlags [Additional Functionality with Qt].

QListView::~QListView ()

Destroys the list view, deleting all its items, and frees up all allocated resources.

int QListView::addColumn (const QString & label, int width = -1) [virtual]

Adds a *width* pixels wide column with the column header *label* to *this* QListView, and returns the index of the new column.

All columns apart from the first one are inserted to the right of the existing ones.

If *width* is negative, the new column's WidthMode is set to Maximum instead of Manual.

See also setColumnText() [p. 193], setColumnWidth() [p. 193] and setColumnWidthMode() [p. 193].

Examples: addressbook/centralwidget.cpp, checklists/checklists.cpp, dirview/main.cpp, fileiconview/mainwindow.cpp, listviews/listviews.cpp and qdir/qdir.cpp.

int QListView::addColumn (const QIconSet & iconset, const QString & label, int width = -1) [virtual]

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

Adds a *width* pixels wide new column with the header *label* and *iconset* to *this* QListView, and returns the index of the column.

If *width* is negative, the new column's WidthMode is set to Maximum, and to Manual otherwise.

See also setColumnText() [p. 193], setColumnWidth() [p. 193] and setColumnWidthMode() [p. 193].

bool QListView::allColumnsShowFocus () const

Returns TRUE if items should show keyboard focus using all columns; otherwise returns FALSE. See the "allColumnsShowFocus" [p. 196] property for details.

int QListView::childCount () const

Returns the number of parentless (top level) QListViewItem objects in this QListView. See the "childCount" [p. 196] property for details.

void QListView::clear () [virtual slot]

Removes and deletes all the items in this list view and triggers an update.

See also triggerUpdate() [p. 196].

Examples: addressbook/centralwidget.cpp, checklists/checklists.cpp, listviews/listviews.cpp and network/ftpclient/ftpmainwindow.cpp.

void QListView::clearSelection () [virtual]

Sets all items to be not selected, updates the list view as necessary and emits the selectionChanged() signals. Note that for Multi selection list views this function needs to iterate over *all* items.

See also setSelected() [p. 194] and multiSelection [p. 197].

Example: addressbook/centralwidget.cpp.

void QListView::clicked (QListViewItem * item) [signal]

This signal is emitted whenever the user clicks (mouse pressed *and* mouse released) in the list view. *item* is the pointer to the clicked list view item, or 0 if the user didn't click on an item.

Note that you may not delete any QListViewItem objects in slots connected to this signal.

Example: addressbook/centralwidget.cpp.

void QListView::clicked (QListViewItem * item, const QPoint & pnt, int c) [signal]

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

This signal is emitted whenever the user clicks (mouse pressed *and* mouse released) in the list view. *item* is the pointer to the clicked list view item, or 0 if the user didn't click on an item. *pnt* is the position where the user has clicked. If *item* is not 0, *c* is the list view column into which the user pressed; if *item* is 0 *c*'s value is undefined.

Note that you may not delete any QListViewItem objects in slots connected to this signal.

void QListView::collapsed (QListViewItem * item) [signal]

This signal is emitted when the *item* has been collapsed, i.e. when the children of *item* are hidden.

See also setOpen() [p. 194] and expanded() [p. 187].

int QListView::columnAlignment (int column) const

Returns the alignment of column *column*. The default is AlignAuto.

See also Qt::AlignmentFlags [Additional Functionality with Qt].

QString QListView::columnText (int c) const

Returns the text of column *c*.

See also setColumnText() [p. 193].

int QListView::columnWidth (int c) const

Returns the width of column *c*.

See also setColumnWidth() [p. 193].

WidthMode QListView::columnWidthMode (int c) const

Returns the WidthMode for column *c*.

See also setColumnWidthMode() [p. 193].

int QListView::columns () const

Returns the number of columns in this list view. See the "columns" [p. 196] property for details.

void QListView::contentsMouseDoubleClickEvent (QMouseEvent * e) [virtual protected]

Processes the mouse double-click event *e* on behalf of the viewed widget.

Reimplemented from QScrollView [p. 268].

void QListView::contentsMouseMoveEvent (QMouseEvent * e) [virtual protected]

Processes the mouse move event *e* on behalf of the viewed widget.

Example: dirview/dirview.cpp.

Reimplemented from QScrollView [p. 268].

void QListView::contentsMousePressEvent (QMouseEvent * e) [virtual protected]

Processes the mouse move event *e* on behalf of the viewed widget.

Example: dirview/dirview.cpp.

Reimplemented from QScrollView [p. 268].

void QListView::contentsMouseReleaseEvent (QMouseEvent * e) [virtual protected]

Processes the mouse move event *e* on behalf of the viewed widget.

Example: dirview/dirview.cpp.

Reimplemented from QScrollView [p. 268].

void QListView::contextMenuRequested (QListViewItem * item, const QPoint & pos, int col) [signal]

This signal is emitted when the user invokes a context menu with the right mouse button or with special system keys, with *item* being the item under the mouse cursor or the current item, respectively.

pos is the position for the context menu in the global coordinate system.

col is the column on which the user pressed, or -1 if the signal was triggered by a key event.

void QListView::currentChanged (QListViewItem *) [signal]

This signal is emitted whenever the current item has changed (normally after the screen update). The current item is the item responsible for indicating keyboard focus.

The argument is the newly current item, or 0 if the change was to make no item current. This can happen, for example, if all items in the list view are deleted.

Note that you may not delete any QListViewItem objects in slots connected to this signal.

See also setCurrentItem() [p. 193] and currentItem() [p. 186].

Example: listviews/listviews.cpp.

QListViewItem * QListView::currentItem () const

Returns a pointer to the currently highlighted item, or 0 if there isn't one.

See also setCurrentItem() [p. 193].

Examples: addressbook/centralwidget.cpp and listviews/listviews.cpp.

RenameAction QListView::defaultRenameAction () const

Returns TRUE if the list view accepts the rename operation by default; otherwise returns FALSE. See the "defaultRenameAction" [p. 196] property for details.

void QListView::doAutoScroll () [protected slot]

This slot handles auto-scrolling when the mouse button is pressed and the mouse is outside the widget.

void QListView::doubleClicked (QListViewItem * item) [signal]

This signal is emitted whenever an item is double-clicked. It's emitted on the second button press, not the second button release. *item* is the list view item on which the user did the double-click.

QDragObject * QListView::dragObject () [virtual protected]

If the user presses the mouse on an item and starts moving the mouse, and the items allow dragging (see QListViewItem::setDragEnabled()), this function is called to get a drag object and a drag is started unless dragObject() returns 0.

By default this function returns 0. You should reimplement it and create a QDragObject depending on the selected items.

void QListView::drawContentsOffset (QPainter * p, int ox, int oy, int cx, int cy, int cw, int ch) [virtual protected]

Calls QListViewItem::paintCell() and/or QListViewItem::paintBranches() for all list view items that require repainting in the *cw* pixels wide and *ch* pixels high bounding rectangle starting at position *cx*, *cy* with offset *ox*, *oy*. Uses the painter *p*.

Reimplemented from QScrollView [p. 270].

void QListView::dropped (QDropEvent * e) [signal]

This signal is emitted, when a drop event occurred onto the viewport (not onto an item).

e gives you all information about the drop.

void QListView::ensureItemVisible (const QListViewItem * i)

Ensures that item *i* is made visible, scrolling the list view vertically as required and also opening (expanding) any parent items if this is necessary to show the item.

See also itemRect() [p. 189] and QScrollView::ensureVisible() [p. 270].

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

Redirects the event *e* relating to object *o*, for the viewport to mousePressEvent(), keyPressEvent() and friends.

Reimplemented from QScrollView [p. 270].

void QListView::expanded (QListViewItem * item) [signal]

This signal is emitted when *item* has been expanded, i.e. when the children of *item* are shown.

See also setOpen() [p. 194] and collapsed() [p. 184].

QListViewItem * QListView::findItem (const QString & text, int column, ComparisonFlags compare = ExactMatch | CaseSensitive) const

Finds the first list view item in column *column*, that matches *text* and returns it, or returns 0 of no such item could be found. If ComparisonFlags are specified in *compare* then these flags are used, otherwise the default is a case-sensitive, exact match search.

QListViewItem * QListView::firstChild () const

Returns the first item in this QListView. You can use its firstChild() and nextSibling() functions to traverse the entire tree of items.

Returns 0 if there is no first item.

See also itemAt() [p. 189], QListViewItem::itemBelow() [p. 206] and QListViewItem::itemAbove() [p. 206].

Examples: addressbook/centralwidget.cpp and listviews/listviews.cpp.

QHeader * QListView::header () const

Returns a pointer to the QHeader object that manages this list view's columns. Please don't modify the header behind the list view's back.

You may safely call QHeader::setClickEnabled(), QHeader::setResizeEnabled(), QHeader::setMovingEnabled() and all the const QHeader functions.

Examples: listviews/listviews.cpp and qdir/qdir.cpp.

void QListView::insertItem (QListViewItem * i) [virtual]

Inserts item *i* into the list view as a top-level item. You do not need to call this unless you've called takeItem(*i*) or QListViewItem::takeItem(*i*) and need to reinsert *i* elsewhere.

See also QListViewItem::takeItem() [p. 212] and takeItem() [p. 195].

void QListView::invertSelection () [virtual slot]

Inverts the selection. Works only in Multi and Extended selection mode.

bool QListView::isMultiSelection () const

Returns TRUE if the list view is in multi-selection or single selection mode; otherwise returns FALSE. See the "multiSelection" [p. 197] property for details.

bool QListView::isOpen (const QListViewItem * item) const

Identical to *item*->isOpen(). Provided for completeness.

See also setOpen() [p. 194].

bool QListView::isRenaming () const

Returns whether currently an item of the listview is being renamed

bool QListView::isSelected (const QListViewItem * i) const

Returns TRUE if the list view item *i* is selected; otherwise returns FALSE.

See also QListViewItem::isSelected() [p. 206].

QListViewItem * QListView::itemAt (const QPoint & viewPos) const

Returns a pointer to the QListViewItem at *viewPos*. Note that *viewPos* is in the coordinate system of viewport(), not in the list view's own, much larger, coordinate system.

itemAt() returns 0 if there is no such item.

Note that you also get the pointer to the item if *viewPos* points to the root decoration (see setRootIsDecorated()) of the item. To check whether or not *viewPos* is on the root decoration of the item, you can do something like this:

```
QListViewItem *i = itemAt( p );
if ( i ) {
    if ( p.x() > header()->cellPos( header()->mapToActual( 0 ) ) +
           treeStepSize() * ( i->depth() + ( rootIsDecorated() ? 1 : 0) ) + itemMargin() ||
           p.x() cellPos( header()->mapToActual( 0 ) ) ) {
        ; // p is not on root decoration
    else
        ; // p is on the root decoration
}
```

This might be interesting if you use this function to find out where the user clicked and if you want to start a drag (which you do not want to do if the user clicked onto the root decoration of an item).

See also itemPos() [p. 189] and itemRect() [p. 189].

int QListView::itemMargin () const

Returns the advisory item margin that list items may use. See the "itemMargin" [p. 197] property for details.

int QListView::itemPos (const QListViewItem * item)

Returns the y-coordinate of *item* in the list view's coordinate system. This function is normally much slower than itemAt() but it works for all items, whereas itemAt() normally works only for items on the screen.

This is a thin wrapper around QListViewItem::itemPos().

See also itemAt() [p. 189] and itemRect() [p. 189].

QRect QListView::itemRect (const QListViewItem * i) const

Returns the rectangle on the screen that item *i* occupies in viewport()'s coordinates, or an invalid rectangle if *i* is a null pointer or is not currently visible.

The rectangle returned does not include any children of the rectangle (i.e. it uses QListViewItem::height(), rather than QListViewItem::totalHeight()). If you want the rectangle to include children you can use something like this:

```
QRect r( listView->itemRect( item ) );
r.setHeight( (QCOORD)(QMIN( item->totalHeight(),
                            listView->viewport->height() - r.y() ) ) )
```

Note the way it avoids too-high rectangles. totalHeight() can be much larger than the window system's coordinate system allows.

itemRect() is comparatively slow. It's best to call it only for items that are probably on-screen.

void QListView::itemRenamed (QListViewItem * item, int col, const QString & text) [signal]

This signal is emitted when *item* has been renamed to *text*, e.g. by in in-place renaming, in column *col*.

void QListView::itemRenamed (QListViewItem * item, int col) [signal]

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

This signal is emitted when *item* has been renamed, e.g. by in-place renaming, in column *col*.

QListViewItem * QListView::lastItem () const

Returns the last item in the list view tree.

Returns 0 if there are no items in the QListView.

This function is slow.

void QListView::mouseButtonClicked (int button, QListViewItem * item, const QPoint & pos, int c) [signal]

This signal is emitted whenever the user clicks (mouse pressed *and* mouse released) in the list view at position *pos*. *button* is the mouse button that the user pressed, *item* is the pointer to the clicked list view item or 0 if the user didn't click on an item. If *item* is not 0, *c* is the list view column into which the user pressed; if *item* is 0 *c*'s value is undefined.

Note that you may not delete any QListViewItem objects in slots connected to this signal.

void QListView::mouseButtonPressed (int button, QListViewItem * item, const QPoint & pos, int c) [signal]

This signal is emitted whenever the user pressed the mouse button in the list view at position *pos*. *button* is the mouse button which the user pressed, *item* is the pointer to the pressed list view item or 0 if the user didn't press on an item. If *item* is not 0, *c* is the list view column into which the user pressed; if *item* is 0 *c*'s value is undefined.

Note that you may not delete any QListViewItem objects in slots connected to this signal.

void QListView::onItem (QListViewItem * i) [signal]

This signal is emitted when the user moves the mouse cursor onto the item *i*, similar to the QWidget::enterEvent() function.

void QListView::onViewport () [signal]

This signal is emitted when the user moves the mouse cursor from an item to an empty part of the list view.

void QListView::paintEmptyArea (QPainter * p, const QRect & rect) [virtual protected]

Paints *rect* so that it looks like empty background using painter *p*. *rect* is is widget coordinates, ready to be fed to *p*.

The default function fills *rect* with the viewport()->backgroundBrush()

void QListView::pressed (QListViewItem * item) [signal]

This signal is emitted whenever the user presses the mouse button in a list view. *item* is the pointer to the list view item on which the user pressed the mouse button, or 0 if the user didn't press the mouse on an item.

Note that you may not delete any QListViewItem objects in slots connected to this signal.

void QListView::pressed (QListViewItem * item, const QPoint & pnt, int c) [signal]

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

This signal is emitted whenever the user presses the mouse button in a list view. *item* is the pointer to the list view item on which the user pressed the mouse button, or 0 if the user didn't press the mouse on an item. *pnt* is the position of the mouse cursor, and *c* is the column where the mouse cursor was when the user pressed the mouse button.

Note that you may not delete any QListViewItem objects in slots connected to this signal.

void QListView::removeColumn (int index) [virtual]

Removes the column at position *index*.

void QListView::removeItem (QListViewItem * item) [virtual]

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

This function has been renamed takeItem().

void QListView::repaintItem (const QListViewItem * item) const

Repaints *item* on the screen if *item* is currently visible. Takes care to avoid multiple repaints.

void QListView::resizeEvent (QResizeEvent * e) [virtual protected]

Ensures that the header is correctly sized and positioned when the resize event *e* occurs.

ResizeMode QListView::resizeMode () const

Returns TRUE if all, none or the last column should be resized; otherwise returns FALSE. See the "resizeMode" [p. 197] property for details.

void QListView::returnPressed (QListViewItem *) [signal]

This signal is emitted when Enter or Return is pressed. The argument is the currentItem().

void QListView::rightButtonClicked (QListViewItem *, const QPoint &, int) [signal]

This signal is emitted when the right button is clicked (i.e. when it's released). The arguments are the relevant QListViewItem (may be 0), the point in global coordinates and the relevant column (or -1 if the click was outside

the list).

void QListView::rightButtonPressed (QListViewItem *, const QPoint &, int) [signal]

This signal is emitted when the right button is pressed. Arguments are then the relevant QListViewItem (may be 0), the point in global coordinates and the relevant column (or -1 if the click was outside the list).

Example: listviews/listviews.cpp.

bool QListView::rootIsDecorated () const

Returns TRUE if the list view show open/close signs on root items; otherwise returns FALSE. See the "rootIsDecorated" [p. 197] property for details.

void QListView::selectAll (bool select) [virtual slot]

If *select* is TRUE, all items get selected; otherwise all items get unselected. This works only in the selection modes Multi and Extended. In Single and NoSelection mode the selection of the current item is just set to *select*.

QListViewItem * QListView::selectedItem () const

Returns a pointer to the selected item if the list view is in single-selection mode and an item is selected.

If no items are selected or the list view is in multi-selection mode this function returns 0.

See also setSelected() [p. 194] and multiSelection [p. 197].

void QListView::selectionChanged () [signal]

This signal is emitted whenever the set of selected items has changed (normally before the screen update). It is available both in Single selection and Multi selection mode but is most useful in Multi selection mode.

Note that you may not delete any QListViewItem objects in slots connected to this signal.

See also setSelected() [p. 194] and QListViewItem::setSelected() [p. 211].

Example: listviews/listviews.cpp.

void QListView::selectionChanged (QListViewItem *) [signal]

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

This signal is emitted whenever the selected item has changed in Single selection mode (normally after the screen update). The argument is the newly selected item.

The no argument overload of this signal is more useful in Multi selection mode.

Note that you may not delete any QListViewItem objects in slots connected to this signal.

See also setSelected() [p. 194], QListViewItem::setSelected() [p. 211] and currentChanged() [p. 186].

SelectionMode QListView::selectionMode () const

Returns the list view's multi-selection mode. See the "selectionMode" [p. 197] property for details.

void QListView::setAllColumnsShowFocus (bool) [virtual]

Sets whether items should show keyboard focus using all columns. See the "allColumnsShowFocus" [p. 196] property for details.

void QListView::setColumnAlignment (int column, int align) [virtual]

Sets column *column*'s alignment to *align*. The alignment is ultimately passed to QListViewItem::paintCell() for each item in the view.

See also Qt::AlignmentFlags [Additional Functionality with Qt].

Example: listviews/listviews.cpp.

void QListView::setColumnText (int column, const QString & label) [virtual]

Sets the heading of column *column* to *label*. The leftmost column is 0.

See also columnText() [p. 185].

void QListView::setColumnText (int column, const QIconSet & iconset, const QString & label) [virtual]

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

Sets the heading of column *column* to *iconset* and *label*. The leftmost column is 0.

See also columnText() [p. 185].

void QListView::setColumnWidth (int column, int w) [virtual]

Sets the width of column *column* to *w* pixels. Note that if the column has a WidthMode other than Manual, this width setting may be subsequently overridden. The leftmost column is 0.

See also columnWidth() [p. 185].

void QListView::setColumnWidthMode (int c, WidthMode mode) [virtual]

Sets column *c*'s width mode to *mode*. The default depends on whether the width argument to addColumn was positive or negative.

See also QListViewItem::width() [p. 212].

void QListView::setCurrentItem (QListViewItem * i) [virtual]

Sets item *i* to be the current highlighted item and repaints appropriately. This highlighted item is used for keyboard navigation and focus indication; it doesn't mean anything else, e.g. it is different from selection.

See also currentItem() [p. 186] and setSelected() [p. 194].

Example: listviews/listviews.cpp.

void QListView::setDefaultRenameAction (RenameAction a) [virtual]

Sets whether the list view accepts the rename operation by default to *a*. See the "defaultRenameAction" [p. 196] property for details.

void QListView::setItemMargin (int) [virtual]

Sets the advisory item margin that list items may use. See the "itemMargin" [p. 197] property for details.

void QListView::setMultiSelection (bool enable) [virtual]

Sets whether the list view is in multi-selection or single selection mode to *enable*. See the "multiSelection" [p. 197] property for details.

void QListView::setOpen (QListViewItem * item, bool open) [virtual]

Sets *item* to be open if *open* is TRUE and *item* is expandable, and to be closed if *open* is FALSE. Repaints accordingly.

Does nothing if *item* is not expandable.

See also QListViewItem::setOpen() [p. 210] and QListViewItem::setExpandable() [p. 210].

void QListView::setResizeMode (ResizeMode m) [virtual]

Sets whether all, none or the last column should be resized to *m*. See the "resizeMode" [p. 197] property for details.

void QListView::setRootIsDecorated (bool) [virtual]

Sets whether the list view show open/close signs on root items. See the "rootIsDecorated" [p. 197] property for details.

void QListView::setSelected (QListViewItem * item, bool selected) [virtual]

If *selected* is TRUE the *item* is selected; otherwise it is unselected.

If the list view is in Single selection mode and *selected* is TRUE, the currently selected item is unselected and *item* is made current. Unlike QListViewItem::setSelected(), this function updates the list view as necessary and emits the selectionChanged() signals.

See also isSelected() [p. 188], multiSelection [p. 197], multiSelection [p. 197] and setCurrentItem() [p. 193].

Example: listviews/listviews.cpp.

void QListView::setSelectionMode (SelectionMode mode)

Sets the list view's multi-selection mode to *mode*. See the "selectionMode" [p. 197] property for details.

void QListView::setShowSortIndicator (bool show) [virtual]

Sets whether the list view header should display a sort indicator to *show*. See the "showSortIndicator" [p. 197] property for details.

void QListView::setShowToolTips (bool b) [virtual]

Sets whether this list view should show tooltips for truncated column texts to *b*. See the "showToolTips" [p. 198] property for details.

void QListView::setSorting (int column, bool ascending = TRUE) [virtual]

Sets the list view to be sorted by *column* and in ascending order if *ascending* is TRUE or descending order if it is FALSE.

If *column* is -1, sorting is disabled and the user cannot sort columns by clicking on the column headers.

void QListView::setTreeStepSize (int) [virtual]

Sets the number of pixels a child is offset from its parent. See the "treeStepSize" [p. 198] property for details.

bool QListView::showSortIndicator () const

Returns TRUE if the list view header should display a sort indicator; otherwise returns FALSE. See the "showSortIndicator" [p. 197] property for details.

bool QListView::showToolTips () const

Returns TRUE if this list view should show tooltips for truncated column texts; otherwise returns FALSE. See the "showToolTips" [p. 198] property for details.

void QListView::sort () [virtual]

(Re)sorts the list view using the last sorting configuration (sort column and ascending/descending).

void QListView::spacePressed (QListViewItem *) [signal]

This signal is emitted when Space is pressed. The argument is currentItem().

void QListView::startDrag () [virtual protected]

Starts a drag.

void QListView::takeItem (QListViewItem * i) [virtual]

Removes item *i* from the list view; *i* must be a top-level item. The warnings regarding QListViewItem::takeItem() apply to this function, too.

See also insertItem() [p. 188].

int QListView::treeStepSize () const

Returns the number of pixels a child is offset from its parent. See the "treeStepSize" [p. 198] property for details.

void QListView::triggerUpdate () [slot]

Triggers a size, geometry and content update during the next iteration of the event loop. Ensures that there'll be just one update to avoid flicker.

void QListView::updateContents () [protected slot]

Updates the sizes of the viewport, header, scroll bars and so on. Don't call this directly; call triggerUpdate() instead.

Property Documentation

bool allColumnsShowFocus

This property holds whether items should show keyboard focus using all columns.

If this property is TRUE all columns will show focus and selection states, otherwise only column 0 will show focus.

The default is FALSE.

Setting this to TRUE if it's not necessary may cause noticeable flicker.

Set this property's value with setAllColumnsShowFocus() and get this property's value with allColumnsShowFocus().

int childCount

This property holds the number of parentless (top level) QListViewItem objects in this QListView.

Represents the current number of parentless (top level) QListViewItem objects in this QListView, like QListViewItem::childCount() returns the number of child items for a QListViewItem.

See also QListViewItem::childCount() [p. 204].

Get this property's value with childCount().

int columns

This property holds the number of columns in this list view.

Get this property's value with columns().

See also addColumn() [p. 183] and removeColumn() [p. 191].

RenameAction defaultRenameAction

This property holds whether the list view accepts the rename operation by default.

If this property is Accept, and the user renames an item and the editor looses focus (without the user pressing Enter), the item will still be renamed. If the property's value is Reject, the item will not be renamed unless the user presses Enter. The default is Reject.

Set this property's value with setDefaultRenameAction() and get this property's value with defaultRenameAction().

int itemMargin

This property holds the advisory item margin that list items may use.

The item margin defaults to one pixel and is the margin between the item's edges and the area where it draws its contents. QListViewItem::paintFocus() draws in the margin.

See also QListViewItem::paintCell() [p. 208].

Set this property's value with setItemMargin() and get this property's value with itemMargin().

bool multiSelection

This property holds whether the list view is in multi-selection or single selection mode.

If you enable multi-selection mode, it is possible to specify whether or not this mode should be extended. Extended means that the user can select multiple items only when pressing the Shift or Ctrl key at the same time.

The default selection mode is Single.

See also selectionMode [p. 197].

Set this property's value with setMultiSelection() and get this property's value with isMultiSelection().

ResizeMode resizeMode

This property holds whether all, none or the last column should be resized.

Specifies whether all, none or the last column should be resized to fit the full width of the listview. The values for this property can be one of the following: NoColumn (the default), AllColumns or LastColumn.

See also QHeader [Additional Functionality with Qt] and header() [p. 188].

Set this property's value with setResizeMode() and get this property's value with resizeMode().

bool rootIsDecorated

This property holds whether the list view show open/close signs on root items.

Open/close signs are small + or - symbols in windows style, or arrows in Motif style. The default is FALSE.

Set this property's value with setRootIsDecorated() and get this property's value with rootIsDecorated().

SelectionMode selectionMode

This property holds the list view's multi-selection mode.

The mode can be Single (the default), Extended, Multi or NoSelection.

See also multiSelection [p. 197].

Set this property's value with setSelectionMode() and get this property's value with selectionMode().

bool showSortIndicator

This property holds whether the list view header should display a sort indicator.

If this property is TRUE, an arrow is drawn in the header of the list view to indicate the sort order of the list view contents. The arrow will be drawn in the correct column and will point up or down, depending on the current sort direction. The default is FALSE (don't show an indicator).

See also QHeader::setSortIndicator() [Additional Functionality with Qt].

Set this property's value with setShowSortIndicator() and get this property's value with showSortIndicator().

bool showToolTips

This property holds whether this list view should show tooltips for truncated column texts.

The default is TRUE.

Set this property's value with setShowToolTips() and get this property's value with showToolTips().

int treeStepSize

This property holds the number of pixels a child is offset from its parent.

The default is 20 pixels.

Of course, this property is only meaningful for hierarchical list views.

Set this property's value with setTreeStepSize() and get this property's value with treeStepSize().

QListViewItem Class Reference

The QListViewItem class implements a list view item.

`#include <qlistview.h>`

Inherits Qt [Additional Functionality with Qt].

Inherited by QCheckListItem [p. 26].

Public Members

- **QListViewItem** (QListView * parent)
- **QListViewItem** (QListViewItem * parent)
- **QListViewItem** (QListView * parent, QListViewItem * after)
- **QListViewItem** (QListViewItem * parent, QListViewItem * after)
- **QListViewItem** (QListView * parent, QString label1, QString label2 = QString::null, QString label3 = QString::null, QString label4 = QString::null, QString label5 = QString::null, QString label6 = QString::null, QString label7 = QString::null, QString label8 = QString::null)
- **QListViewItem** (QListViewItem * parent, QString label1, QString label2 = QString::null, QString label3 = QString::null, QString label4 = QString::null, QString label5 = QString::null, QString label6 = QString::null, QString label7 = QString::null, QString label8 = QString::null)
- **QListViewItem** (QListView * parent, QListViewItem * after, QString label1, QString label2 = QString::null, QString label3 = QString::null, QString label4 = QString::null, QString label5 = QString::null, QString label6 = QString::null, QString label7 = QString::null, QString label8 = QString::null)
- **QListViewItem** (QListViewItem * parent, QListViewItem * after, QString label1, QString label2 = QString::null, QString label3 = QString::null, QString label4 = QString::null, QString label5 = QString::null, QString label6 = QString::null, QString label7 = QString::null, QString label8 = QString::null)
- virtual **~QListViewItem** ()
- virtual void **insertItem** (QListViewItem * newChild)
- virtual void **takeItem** (QListViewItem * item)
- virtual void removeItem (QListViewItem * item) *(obsolete)*
- int **height** () const
- virtual void **invalidateHeight** ()
- int **totalHeight** () const
- virtual int **width** (const QFontMetrics & fm, const QListView * lv, int c) const
- void **widthChanged** (int c = -1) const
- int **depth** () const
- virtual void **setText** (int column, const QString & text)
- virtual QString **text** (int column) const
- virtual void **setPixmap** (int column, const QPixmap & pm)
- virtual const QPixmap * **pixmap** (int column) const
- virtual QString **key** (int column, bool ascending) const

- virtual int **compare** (QListViewItem * i, int col, bool ascending) const
- virtual void **sortChildItems** (int column, bool ascending)
- int **childCount** () const
- bool **isOpen** () const
- virtual void **setOpen** (bool o)
- virtual void **setup** ()
- virtual void **setSelected** (bool s)
- bool **isSelected** () const
- virtual void **paintCell** (QPainter * p, const QColorGroup & cg, int column, int width, int align)
- virtual void **paintBranches** (QPainter * p, const QColorGroup & cg, int w, int y, int h)
- virtual void **paintFocus** (QPainter * p, const QColorGroup & cg, const QRect & r)
- QListViewItem * **firstChild** () const
- QListViewItem * **nextSibling** () const
- QListViewItem * **parent** () const
- QListViewItem * **itemAbove** ()
- QListViewItem * **itemBelow** ()
- int **itemPos** () const
- QListView * **listView** () const
- virtual void **setSelectable** (bool enable)
- bool **isSelectable** () const
- virtual void **setExpandable** (bool enable)
- bool **isExpandable** () const
- void **repaint** () const
- virtual void **sort** ()
- void **moveItem** (QListViewItem * after)
- virtual void **setDragEnabled** (bool allow)
- virtual void **setDropEnabled** (bool allow)
- bool **dragEnabled** () const
- bool **dropEnabled** () const
- virtual bool **acceptDrop** (const QMimeSource * mime) const
- void **setVisible** (bool b)
- bool **isVisible** () const
- virtual void **setRenameEnabled** (int col, bool b)
- bool **renameEnabled** (int col) const
- virtual void **startRename** (int col)
- virtual void **setEnabled** (bool b)
- bool **isEnabled** () const
- virtual int **rtti** () const
- virtual void **setMultiLinesEnabled** (bool b)
- bool **multiLinesEnabled** () const

Protected Members

- virtual void **enforceSortOrder** () const
- virtual void **setHeight** (int height)
- virtual void **activate** ()
- bool **activatedPos** (QPoint & pos)
- virtual void **dropped** (QDropEvent * e)
- virtual void **dragEntered** ()
- virtual void **dragLeft** ()
- virtual void **okRename** (int col)
- virtual void **cancelRename** (int col)

Detailed Description

The QListViewItem class implements a list view item.

A list view item is a multi-column object capable of displaying itself. Its design has the following main goals:

- Work quickly and well for *large* sets of data.
- Be easy to use in the simple case.

The easiest way to use QListViewItem is to construct one with a few constant strings. This creates an item that is a child of *parent* with two fixed-content strings, and discards the pointer to it:

```
(void) new QListViewItem( parent, "first column", "second column" );
```

This object will be deleted when *parent* is deleted, as for QObjects.

The parent is either another QListViewItem or a QListView. If the parent is a QListView, this item is a top-level item within that QListView. If the parent is another QListViewItem, this item becomes a child of that list view item.

If you keep the pointer, you can set or change the texts using setText(), add pixmaps using setPixmap(), change its mode using setSelectable(), setSelected(), setOpen() and setExpandable(). You'll also be able to change its height using setHeight(), and traverse the tree. There's no need to retain the pointer however, since you can get a pointer to any QListViewItem in a QListView using QListView::selectedItem(), QListView::currentItem(), QListView::firstChild(), QListView::lastItem(), QListView::findItem().

You can traverse the tree as if it were a doubly-linked list using itemAbove() and itemBelow(); they return pointers to the items directly above and below this item on the screen (even if none of the three are actually visible at the moment).

You can also traverse it as a tree by using parent(), firstChild(), and nextSibling().

Example:

```
QListViewItem * myChild = myItem->firstChild();
while( myChild ) {
    doSomething( myChild );
    myChild = myChild->nextSibling();
}
```

There is also an interator class to traverse a tree of list view items. To iterate over all items of a list view, do the following:

```
QListViewItemIterator it( listview );
for ( ; it.current(); ++it )
  doSomething( it.current() ); // it.current() is a QListViewItem*
```

Note that the order of the children will change when the sorting order changes and is undefined if the items are not visible. You can, however, call enforceSortOrder() at any time; QListView will always call it before it needs to show an item.

Many programs will need to reimplement QListViewItem. The most commonly reimplemented functions are:

- text() returns the text in a column. Many subclasses will compute that on the fly.
- key() is used for sorting. The default key() simply calls text(), but judicious use of key can be used to sort by date, for example (as QFileDialog does).
- setup() is called before showing the item and whenever the font changes, for example.
- activate() is called whenever the user clicks on the item or presses space when the item is the currently highlighted item.

Some subclasses call setExpandable(TRUE) even when they have no children, and populate themselves when setup() or setOpen(TRUE) is called. The dirview/dirview.cpp example program uses this technique to start up quickly: The files and subdirectories in a directory aren't inserted into the tree until they're actually needed.

See also Advanced Widgets.

Member Function Documentation

QListViewItem::QListViewItem (QListView * parent)

Constructs a new top-level list view item in the QListView *parent*.

QListViewItem::QListViewItem (QListViewItem * parent)

Constructs a new list view item that is a child of *parent* and first in the parent's list of children.

QListViewItem::QListViewItem (QListView * parent, QListViewItem * after)

Constructs an empty list view item that is a child of *parent* and is after *after* in the parent's list of children. Since *parent* is a QListView the item will be a top-level item.

QListViewItem::QListViewItem (QListViewItem * parent, QListViewItem * after)

Constructs an empty list view item that is a child of *parent* and is after *after* in the parent's list of children.

QListViewItem::QListViewItem (QListView * parent, QString label1, QString label2 = QString::null, QString label3 = QString::null, QString label4 = QString::null, QString label5 = QString::null, QString label6 = QString::null, QString label7 = QString::null, QString label8 = QString::null)

Constructs a new list view item in the QListView *parent*, *parent*, with up to eight constant strings *label1*, *label2*, *label3*, *label4*, *label5*, *label6*, *label7* and *label8* defining its column contents.

See also setText() [p. 211].

QListViewItem::QListViewItem (QListViewItem * parent, QString label1, QString label2 = QString::null, QString label3 = QString::null, QString label4 = QString::null, QString label5 = QString::null, QString label6 = QString::null, QString label7 = QString::null, QString label8 = QString::null)

Constructs a new list view item as a child of the QListViewItem *parent* with optional constant strings *label1*, *label2*, *label3*, *label4*, *label5*, *label6*, *label7* and *label8* as column contents.

See also setText() [p. 211].

QListViewItem::QListViewItem (QListView * parent, QListViewItem * after, QString label1, QString label2 = QString::null, QString label3 = QString::null, QString label4 = QString::null, QString label5 = QString::null, QString label6 = QString::null, QString label7 = QString::null, QString label8 = QString::null)

Constructs a new list view item in the QListView *parent* that is included after item *after* and can contain up to eight column texts *label1*, *label2*, *label3*, *label4*, *label5*, *label6*, *label7* and*label8*.

Note that the order is changed according to QListViewItem::key() unless the list view's sorting is disabled using QListView::setSorting(-1).

See also setText() [p. 211].

QListViewItem::QListViewItem (QListViewItem * parent, QListViewItem * after, QString label1, QString label2 = QString::null, QString label3 = QString::null, QString label4 = QString::null, QString label5 = QString::null, QString label6 = QString::null, QString label7 = QString::null, QString label8 = QString::null)

Constructs a new list view item as a child of the QListViewItem *parent*. It is inserted after item *after* and may contain up to eight strings *label1*, *label2*, *label3*, *label4*, *label5*, *label6*, *label7* and *label8* as column entries.

Note that the order is changed according to QListViewItem::key() unless the list view's sorting is disabled using QListView::setSorting(-1).

See also setText() [p. 211].

QListViewItem::~QListViewItem () [virtual]

Destroys the item, deleting all its children and freeing up all allocated resources.

bool QListViewItem::acceptDrop (const QMimeSource * mime) const [virtual]

Returns TRUE if the item can accept drops of type QMimeSource *mime*; otherwise returns FALSE.

The default implementation does nothing and returns FALSE. A subclass must reimplement this to accept drops.

void QListViewItem::activate () [virtual protected]

This virtual function is called whenever the user clicks on this item or presses Space on it.

See also activatedPos() [p. 203].

Reimplemented in QCheckListItem.

bool QListViewItem::activatedPos (QPoint & pos) [protected]

When called from a reimplementation of activate(), this function gives information on how the item was activated. Otherwise the behavior is undefined.

If activate() was caused by a mouse press, the function sets *pos* to where the user clicked and returns TRUE; otherwise it returns FALSE and does not change *pos*.

pos is relative to the top-left corner of this item.

We recommend not using this function; it is scheduled to become obsolete.

See also activate() [p. 203].

void QListViewItem::cancelRename (int col) [virtual protected]

This function is called if the user cancels in-place renaming of this item in column *col*.

See also okRename() [p. 207].

int QListViewItem::childCount () const

Returns how many children this item has.

int QListViewItem::compare (QListViewItem * i, int col, bool ascending) const [virtual]

Compares this listview item to *i* using the column *col* in *ascending* order. Returns -1 if this item is less than *i*, 0 if they are equal and 1 if this item is greater than *i*.

This function is used for sorting.

The default implementation compares the item keys (key()) using QString::localeAwareCompare(). A reimplementation may use different values and a different comparison function. Here is a reimplementation that uses plain Unicode comparison:

```
int MyListViewItem::compare( QListViewItem *i, int col,
                             bool ascending ) const
{
    return key( col, ascending ).compare( i->key(col, ascending) );
}
```

We don't recommend using *ascending* so your code can safely ignore it.

See also key() [p. 207], QString::localeAwareCompare() [Datastructures and String Handling with Qt] and QString::compare() [Datastructures and String Handling with Qt].

Example: network/ftpclient/ftpview.cpp.

int QListViewItem::depth () const

Returns the depth of this item.

Example: dirview/dirview.cpp.

bool QListViewItem::dragEnabled () const

Returns TRUE if this item can be dragged; otherwise returns FALSE.

See also setDragEnabled() [p. 209].

void QListViewItem::dragEntered () [virtual protected]

This method is called when a drag entered the item's bounding rectangle.

The default implementation does nothing, subclasses may need to reimplement this method.

void QListViewItem::dragLeft () [virtual protected]

This method is called when a drag left the item's bounding rectangle.

The default implementation does nothing, subclasses may need to reimplement this method.

bool QListViewItem::dropEnabled () const

Returns TRUE if this item accepts drops; otherwise returns FALSE.

See also setDropEnabled() [p. 209] and acceptDrop() [p. 203].

void QListViewItem::dropped (QDropEvent * e) [virtual protected]

This method is called when something was dropped on the item. *e* contains all the information about the drop.

The default implementation does nothing, subclasses may need to reimplement this method.

void QListViewItem::enforceSortOrder () const [virtual protected]

Makes sure that this object's children are sorted appropriately.

This works only if every item from the root item down to this item is already sorted.

See also sortChildItems() [p. 211].

QListViewItem * QListViewItem::firstChild () const

Returns a pointer to the first (top) child of this item, or a null pointer if this item has no children.

Note that the children are not guaranteed to be sorted properly. QListView and QListViewItem try to postpone or avoid sorting to the greatest degree possible, in order to keep the user interface snappy.

See also nextSibling() [p. 207].

Example: checklists/checklists.cpp.

int QListViewItem::height () const

Returns the height of this item in pixels. This does not include the height of any children; totalHeight() returns that.

void QListViewItem::insertItem (QListViewItem * newChild) [virtual]

Inserts *newChild* into this list view item's list of children. You should not need to call this function; it is called automatically by the constructor of *newChild*.

void QListViewItem::invalidateHeight () [virtual]

Invalidates the cached total height of this item, including all open children.

See also setHeight() [p. 210], height() [p. 205] and totalHeight() [p. 212].

bool QListViewItem::isEnabled () const

Returns TRUE if this item is enabled; otherwise returns FALSE.

See also setEnabled() [p. 209].

bool QListViewItem::isExpandable () const

Returns TRUE if this item is expandable even when it has no children; otherwise returns FALSE.

bool QListViewItem::isOpen () const

Returns TRUE if this list view item has children *and* they are potentially visible. Returns FALSE if the item has no children or they are hidden.

See also setOpen() [p. 210].

bool QListViewItem::isSelectable () const

Returns TRUE if the item is selectable (as it is by default); otherwise returns FALSE

See also setSelectable() [p. 210].

bool QListViewItem::isSelected () const

Returns TRUE if this item is selected; otherwise returns FALSE.

See also setSelected() [p. 211], QListView::setSelected() [p. 194] and QListView::selectionChanged() [p. 192].

Example: listviews/listviews.cpp.

bool QListViewItem::isVisible () const

Returns TRUE if the item is visible; otherwise returns FALSE.

See also setVisible() [p. 211].

QListViewItem * QListViewItem::itemAbove ()

Returns a pointer to the item immediately above this item on the screen. This is usually the item's closest older sibling, but it may also be its parent or its next older sibling's youngest child, or something else if anyoftheabove->height() returns 0. Returns a null pointer if there is no item immediately above this item.

This function assumes that all parents of this item are open (i.e. that this item is visible, or can be made visible by scrolling).

See also itemBelow() [p. 206] and QListView::itemRect() [p. 189].

QListViewItem * QListViewItem::itemBelow ()

Returns a pointer to the item immediately below this item on the screen. This is usually the item's eldest child, but it may also be its next younger sibling, its parent's next younger sibling, grandparent's, etc., or something else if anyoftheabove->height() returns 0. Returns a null pointer if there is no item immediately above this item.

This function assumes that all parents of this item are open (i.e. that this item is visible or can be made visible by scrolling).

See also itemAbove() [p. 206] and QListView::itemRect() [p. 189].

Example: dirview/dirview.cpp.

int QListViewItem::itemPos () const

Returns the y coordinate of *this* item in the list view's coordinate system. This function is normally much slower than QListView::itemAt(), but it works for all items whereas QListView::itemAt() normally only works for items on the screen.

See also QListView::itemAt() [p. 189], QListView::itemRect() [p. 189] and QListView::itemPos() [p. 189].

QString QListViewItem::key (int column, bool ascending) const [virtual]

Returns a key that can be used for sorting by column *column*. The default implementation returns text(). Derived classes may also incorporate the order indicated by *ascending* into this key, although this is not recommended.

If you want to sort on non-alphabetical data, e.g. dates, numbers, etc., reimplement compare().

See also compare() [p. 204] and sortChildItems() [p. 211].

QListView * QListViewItem::listView () const

Returns a pointer to the list view containing this item.

void QListViewItem::moveItem (QListViewItem * after)

Moves this item after the item *after*. This means it will get the sibling exactly after the item *after*. To move an item in the hierarchy, use takeItem() and insertItem().

bool QListViewItem::multiLinesEnabled () const

Returns TRUE if the item can display multiple lines of text; otherwise returns FALSE.

QListViewItem * QListViewItem::nextSibling () const

Returns a pointer to the sibling item below this item, or a null pointer if there is no sibling item after this item.

Note that the siblings are not guaranteed to be sorted properly. QListView and QListViewItem try to postpone or avoid sorting to the greatest degree possible, in order to keep the user interface snappy.

See also firstChild() [p. 205].

void QListViewItem::okRename (int col) [virtual protected]

This function is called if the user presses Enter during in-place renaming of the item in column *col*.

See also cancelRename() [p. 204].

void QListViewItem::paintBranches (QPainter * p, const QColorGroup & cg, int w, int y, int h) [virtual]

Paints a set of branches from this item to (some of) its children.

Painter *p* is set up with clipping and translation so that you can draw only in the rectangle you need to; *cg* is the color group to use; the update rectangle is at (0, 0) and has size width *w* by height *h*. The top of the rectangle you own is at *y* (which is never greater than 0 but can be outside the window system's allowed coordinate range).

The update rectangle is in an undefined state when this function is called; this function must draw on *all* of the pixels.

See also paintCell() [p. 208] and QListView::drawContentsOffset() [p. 187].

void QListViewItem::paintCell (QPainter * p, const QColorGroup & cg, int column, int width, int align) [virtual]

This virtual function paints the contents of one column of an item and aligns it as described by *align*.

p is a QPainter open on the relevant paint device. *p* is translated so (0, 0) is the top-left pixel in the cell and *width-1*, height()-1 is the bottom-right pixel *in* the cell. The other properties of *p* (pen, brush, etc) are undefined. *cg* is the color group to use. *column* is the logical column number within the item that is to be painted; 0 is the column which may contain a tree.

This function may use QListView::itemMargin() for readability spacing on the left and right sides of data such as text, and should honor isSelected() and QListView::allColumnsShowFocus().

If you reimplement this function, you should also reimplement width().

The rectangle to be painted is in an undefined state when this function is called, so you *must* draw on all the pixels. The painter *p* has the right font on entry.

See also paintBranches() [p. 208] and QListView::drawContentsOffset() [p. 187].

Example: listviews/listviews.cpp.

Reimplemented in QCheckListItem.

void QListViewItem::paintFocus (QPainter * p, const QColorGroup & cg, const QRect & r) [virtual]

Paints a focus indication on the rectangle *r* using painter *p* and colors *cg*.

p is already clipped.

See also paintCell() [p. 208], paintBranches() [p. 208] and QListView::allColumnsShowFocus [p. 196].

Reimplemented in QCheckListItem.

QListViewItem * QListViewItem::parent () const

Returns a pointer to the parent of this item, or a null pointer if this item has no parent.

See also firstChild() [p. 205] and nextSibling() [p. 207].

Example: dirview/dirview.cpp.

const QPixmap * QListViewItem::pixmap (int column) const [virtual]

Returns a pointer to the pixmap for *column*, or a null pointer if there is no pixmap for *column*.

See also setText() [p. 211] and setPixmap() [p. 210].

Examples: dirview/dirview.cpp and network/ftpclient/ftpview.cpp.

void QListViewItem::removeItem (QListViewItem * item) [virtual]

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

This function has been renamed takeItem().

bool QListViewItem::renameEnabled (int col) const

Returns TRUE if this item can be in-place renamed in column *col*; otherwise returns FALSE.

void QListViewItem::repaint () const

Repaints this item on the screen if it is currently visible.

Example: addressbook/centralwidget.cpp.

int QListViewItem::rtti () const [virtual]

Returns 0.

Make your derived classes return their own values for rtti(), and you can distinguish between listview items. You should use values greater than 1000 preferably a large random number, to allow for extensions to this class.

Reimplemented in QCheckListItem.

void QListViewItem::setDragEnabled (bool allow) [virtual]

If *allow* is TRUE, the listview starts a drag (see QListView::dragObject()) when the user presses and moves the mouse on this item.

void QListViewItem::setDropEnabled (bool allow) [virtual]

If *allow* is TRUE, the listview accepts drops onto the item; otherwise drops are not allowed..

void QListViewItem::setEnabled (bool b) [virtual]

If *b* is TRUE the item is enabled; otherwise it is disabled. Disabled items are drawn grayed-out and are not accessable by the user.

void QListViewItem::setExpandable (bool enable) [virtual]

Sets this item to be expandable even if it has no children if *enable* is TRUE, and to be expandable only if it has children if *enable* is FALSE (the default).

The dirview example uses this in the canonical fashion. It checks whether the directory is empty in setup() and calls setExpandable(TRUE) if not; in setOpen() it reads the contents of the directory and inserts items accordingly. This strategy means that dirview can display the entire file system without reading very much at startup.

Note that root items are not expandable by the user unless QListView::setRootIsDecorated() is set to TRUE.

See also setSelectable() [p. 210].

void QListViewItem::setHeight (int height) [virtual protected]

Sets this item's height to *height* pixels. This implicitly changes totalHeight(), too.

Note that a font change causes this height to be overwritten unless you reimplement setup().

For best results in Windows style we suggest using an even number of pixels.

See also height() [p. 205], totalHeight() [p. 212] and isOpen() [p. 206].

void QListViewItem::setMultiLinesEnabled (bool b) [virtual]

If *b* is TRUE items may contain multiple lines of text; otherwise they may only contain a single line.

void QListViewItem::setOpen (bool o) [virtual]

Opens or closes an item, i.e. shows or hides an item's children.

If *o* is TRUE all child items are shown initially. The user can hide them by clicking the - icon to the left of the item. If *o* is FALSE, the children of this item are initially hidden. The user can show them by clicking the + icon to the left of the item.

See also height() [p. 205], totalHeight() [p. 212] and isOpen() [p. 206].

Examples: checklists/checklists.cpp, dirview/dirview.cpp, dirview/main.cpp, fileiconview/mainwindow.cpp and xml/tagreader-with-features/structureparser.cpp.

void QListViewItem::setPixmap (int column, const QPixmap & pm) [virtual]

Sets the pixmap in column *column* to *pm*, if *pm* is non-null and different from the current pixmap, and if *column* is non-negative.

See also pixmap() [p. 209] and setText() [p. 211].

Example: dirview/dirview.cpp.

void QListViewItem::setRenameEnabled (int col, bool b) [virtual]

If *b* is TRUE, this item can be in-place renamed in the column *col* by the user, otherwise it is not possible.

void QListViewItem::setSelectable (bool enable) [virtual]

Sets this items to be selectable if *enable* is TRUE (the default) or not to be selectable if *enable* is FALSE.

The user is not able to select a non-selectable item using either the keyboard or mouse. The application programmer still can, e.g. using setSelected().

See also isSelectable() [p. 206].

Example: network/ftpclient/ftpview.cpp.

void QListViewItem::setSelected (bool s) [virtual]

If *s* is TRUE this item is selected; otherwise it is deselected.

This function does not maintain any invariants or repaint anything — QListView::setSelected() does that.

See also height() [p. 205] and totalHeight() [p. 212].

Example: addressbook/centralwidget.cpp.

void QListViewItem::setText (int column, const QString & text) [virtual]

Sets the text in column *column* to *text*, if *column* is a valid column number and *text* is different from the existing text.

If *text()* has been reimplemented, this function may be a no-op.

See also text() [p. 212] and key() [p. 207].

Examples: addressbook/centralwidget.cpp and xml/outliner/outlinetree.cpp.

void QListViewItem::setVisible (bool b)

If *b* is TRUE, the item is made visible; otherwise it is hidden.

If the item is not visible, itemAbove() and itemBelow() will never hit this item, although you still can reach it by using e.g. the QListViewItemIterator.

void QListViewItem::setup () [virtual]

This virtual function is called before the first time QListView needs to know the height or any other graphical attribute of this object, and whenever the font, GUI style, or colors of the list view change.

The default calls widthChanged() and sets the item's height to the height of a single line of text in the list view's font. (If you use icons, multi-line text, etc., you will probably need to call setHeight() yourself or reimplement it.)

Example: dirview/dirview.cpp.

void QListViewItem::sort () [virtual]

(Re)sorts all child items of this item using the last sorting configuration (sort column and direction).

See also enforceSortOrder() [p. 205].

void QListViewItem::sortChildItems (int column, bool ascending) [virtual]

Sorts the children of this item using column *column*. This is done in ascending order if *ascending* is TRUE and in descending order if *ascending* is FALSE.

Asks some of the children to sort their children. (QListView and QListViewItem ensure that all on-screen objects are properly sorted but may avoid or defer sorting other objects in order to be more responsive.)

See also key() [p. 207] and compare() [p. 204].

void QListViewItem::startRename (int col) [virtual]

If in-place renaming of this item is enabled (see renameEnabled()), this function starts renaming the item in cloumn *col*, by creating and initializing an edit box.

void QListViewItem::takeItem (QListViewItem * item) [virtual]

Removes *item* from this object's list of children and causes an update of the screen display. The item is not deleted. You should normally not need to call this function because QListViewItem::~QListViewItem() calls it.

The normal way to delete an item is `delete`.

If you need to move an item from one place in the hierarchy to another you can use takeItem() to remove the item from the list view and then insertItem() to put the item back in its new position.

Warning: This function leaves *item* and its children in a state where most member functions are unsafe. Only a few functions work correctly on an item in this state, most notably insertItem(). The functions that work on detached items are explicitly documented as such.

See also QListViewItem::insertItem() [p. 205].

QString QListViewItem::text (int column) const [virtual]

Returns the text in column *column*, or a null string if there is no text in that column.

See also key() [p. 207] and paintCell() [p. 208].

Examples: addressbook/centralwidget.cpp, dirview/dirview.cpp and network/ftpclient/ftpview.cpp.

int QListViewItem::totalHeight () const

Returns the total height of this object, including any visible children. This height is recomputed lazily and cached for as long as possible.

Functions which can affect the total height are, setHeight() which is used to set an item's height, setOpen() to show or hide an item's children, and invalidateHeight() to invalidate the cached height.

See also height() [p. 205].

int QListViewItem::width (const QFontMetrics & fm, const QListView * lv, int c) const [virtual]

Returns the number of pixels of width required to draw column *c* of list view *lv*, using the metrics *fm* without cropping. The list view containing this item may use this information depending on the QListView::WidthMode settings for the column.

The default implementation returns the width of the bounding rectangle of the text of column *c*.

See also listView() [p. 207], widthChanged() [p. 213], QListView::setColumnWidthMode() [p. 193] and QListView::itemMargin [p. 197].

void QListViewItem::widthChanged (int c = -1) const

Call this function when the value of width() may have changed for column *c*. Normally, you should call this if text(c) changes. Passing -1 for *c* indicates that all columns may have changed. For efficiency, you should do this if more than one call to widthChanged() is required.

See also width() [p. 212].

QListViewItemIterator Class Reference

The QListViewItemIterator class provides an iterator for collections of QListViewItems.

`#include <qlistview.h>`

Public Members

- **QListViewItemIterator** ()
- **QListViewItemIterator** (QListViewItem * item)
- **QListViewItemIterator** (const QListViewItemIterator & it)
- **QListViewItemIterator** (QListView * lv)
- QListViewItemIterator & **operator=** (const QListViewItemIterator & it)
- **~QListViewItemIterator** ()
- QListViewItemIterator & **operator++** ()
- const QListViewItemIterator **operator++** (int)
- QListViewItemIterator & **operator+=** (int j)
- QListViewItemIterator & **operator--** ()
- const QListViewItemIterator **operator--** (int)
- QListViewItemIterator & **operator-=** (int j)
- QListViewItem * **current** () const

Detailed Description

The QListViewItemIterator class provides an iterator for collections of QListViewItems.

Construct an instance of a QListViewItemIterator, with either a QListView* or a QListViewItem* as argument, to operate on the tree of QListViewItems.

A QListViewItemIterator iterates over all items of a list view. This means that it always makes the first child of the current item the new current item. If there is no child, the next sibling becomes the new current item; and if there is no next sibling, the next sibling of the parent becomes current.

The following example function gets a list of all the items that have been selected by the user, storing pointers to the items in a QPtrList:

```
QPtrList * getSelectedItems( QListView *lv ) {
  if ( !lv )
    return 0;

  // Create the list
  QPtrList *lst = new QPtrList;
  lst->setAutoDelete( FALSE );
```

```
    // Create an iterator and give the list view as argument
    QListViewItemIterator it( lv );
    // iterate through all items of the list view
    for ( ; it.current(); ++it ) {
      if ( it.current()->isSelected() )
        lst->append( it.current() );
    }

    return lst;
  }
```

A QListViewItemIterator provides a convenient and easy way to traverse a hierarchical QListView.

Multiple QListViewItemIterators can operate on the tree of QListViewItems. A QListView knows about all iterators operating on its QListViewItems. So when a QListViewItem gets removed all iterators that point to this item are updated and point to the following item.

See also QListView [p. 177], QListViewItem [p. 199] and Advanced Widgets.

Member Function Documentation

QListViewItemIterator::QListViewItemIterator ()

Constructs an empty iterator.

QListViewItemIterator::QListViewItemIterator (QListViewItem * item)

Constructs an iterator for the QListView of the *item*. The current iterator item is set to point to the *item*.

QListViewItemIterator::QListViewItemIterator (const QListViewItemIterator & it)

Constructs an iterator for the same QListView as *it*. The current iterator item is set to point on the current item of *it*.

QListViewItemIterator::QListViewItemIterator (QListView * lv)

Constructs an iterator for the QListView *lv*. The current iterator item is set to point on the first child (QListViewItem) of *lv*.

QListViewItemIterator::~QListViewItemIterator ()

Destroys the iterator.

QListViewItem * QListViewItemIterator::current () const

Returns a pointer to the current item of the iterator.

Examples: addressbook/centralwidget.cpp, checklists/checklists.cpp, dirview/dirview.cpp and network/ftpclient/ftpview.cpp.

QListViewItemIterator & QListViewItemIterator::operator++ ()

Prefix ++ makes the next item in the QListViewItem tree of the QListView of the iterator the current item and returns it. If the current item was the last item in the QListView or null, null is returned.

const QListViewItemIterator QListViewItemIterator::operator++ (int)

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

Postfix ++ makes the next item in the QListViewItem tree of the QListView of the iterator the current item and returns the item which was previously current.

QListViewItemIterator & QListViewItemIterator::operator+= (int j)

Sets the current item to the item *j* positions after the current item in the QListViewItem hierarchy. If this item is beyond the last item, the current item is set to null.

The new current item (or null, if the new current item is null) is returned.

QListViewItemIterator & QListViewItemIterator::operator-- ()

Prefix — makes the previous item in the QListViewItem tree of the QListView of the iterator the current item and returns it. If the current item was the last first in the QListView or null, null is returned.

const QListViewItemIterator QListViewItemIterator::operator-- (int)

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

Postfix — makes the previous item in the QListViewItem tree of the QListView of the iterator the current item and returns the item.

QListViewItemIterator & QListViewItemIterator::operator-= (int j)

Sets the current item to the item *j* positions before the current item in the QListViewItem hierarchy. If this item is before the first item, the current item is set to null. The new current item (or null, if the new current item is null) is returned.

QListViewItemIterator & QListViewItemIterator::operator= (const QListViewItemIterator & it)

Assignment. Makes a copy of *it* and returns a reference to its iterator.

QMultiLineEdit Class Reference (obsolete)

The QMultiLineEdit widget is a simple editor for inputting text.

`#include <qmultilineedit.h>`

Inherits QTextEdit [p. 372].

Public Members

- **QMultiLineEdit** (QWidget * parent = 0, const char * name = 0)
- QString **textLine** (int line) const
- int **numLines** () const
- virtual void **insertLine** (const QString & txt, int line = -1)
- virtual void **insertAt** (const QString & s, int line, int col, bool mark = FALSE)
- virtual void **removeLine** (int paragraph)
- virtual void **setCursorPosition** (int line, int col, bool mark = FALSE)
- bool **atBeginning** () const
- bool **atEnd** () const
- virtual void **setAlignment** (int flags)
- int **alignment** () const
- void **setEdited** (bool)
- bool **edited** () const
- bool **hasMarkedText** () const
- QString **markedText** () const
- void **cursorWordForward** (bool mark)
- void **cursorWordBackward** (bool mark)
- bool autoUpdate () const *(obsolete)*
- virtual void setAutoUpdate (bool) *(obsolete)*
- int totalWidth () const *(obsolete)*
- int totalHeight () const *(obsolete)*
- int maxLines () const *(obsolete)*
- void setMaxLines (int) *(obsolete)*

Public Slots

- void deselect () *(obsolete)*

Properties

- Alignment **alignment** — the editor's paragraph alignment
- bool **atBeginning** — whether the cursor is placed at the beginning of the text *(read only)*
- bool **atEnd** — whether the cursor is placed at the end of the text *(read only)*
- bool **edited** — whether the document has been edited by the user
- int **numLines** — the number of paragraphs in the editor *(read only)*

Protected Members

- QPoint **cursorPoint** () const
- virtual void **insertAndMark** (const QString & str, bool mark)
- virtual void **newLine** ()
- virtual void **killLine** ()
- virtual void **pageUp** (bool mark = FALSE)
- virtual void **pageDown** (bool mark = FALSE)
- virtual void **cursorLeft** (bool mark = FALSE, bool wrap = TRUE)
- virtual void **cursorRight** (bool mark = FALSE, bool wrap = TRUE)
- virtual void **cursorUp** (bool mark = FALSE)
- virtual void **cursorDown** (bool mark = FALSE)
- virtual void **backspace** ()
- virtual void **home** (bool mark = FALSE)
- virtual void **end** (bool mark = FALSE)
- bool **getMarkedRegion** (int * line1, int * col1, int * line2, int * col2) const
- int **lineLength** (int row) const

Detailed Description

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

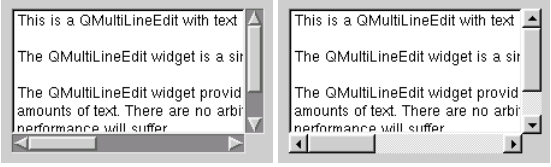
The QMultiLineEdit widget is a simple editor for inputting text.

The QMultiLineEdit was a simple editor widget in former Qt versions. Qt 3.0 includes a new richtext engine which obsoletes QMultiLineEdit. It is still included for compatibility reasons. It is now a subclass of QTextEdit, and provides enough of the old QMultiLineEdit API to keep old applications working.

If you implement something new with QMultiLineEdit, we suggest using QTextEdit instead and call QTextEdit::setTextFormat(Qt::PlainText).

Although most of the old QMultiLineEdit API is still available, there is a few difference. The old QMultiLineEdit operated on lines, not on paragraphs. As lines change all the time during wordwrap, the new richtext engine uses paragraphs as basic elements in the data structure. All functions (numLines(), textLine(), etc.) that operated on lines, now operate on paragraphs. Further, getString() has been removed completely. It revealed too much of the internal data structure.

Applications which made normal and reasonable use of QMultiLineEdit should still work without problems. Some odd usage will require some porting. In these cases, it may be better to use QTextEdit now.



See also QTextEdit [p. 372] and Advanced Widgets.

Member Function Documentation

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

Constructs a new, empty, QMultiLineEdit with parent *parent* called *name*.

int QMultiLineEdit::alignment () const

Returns the editor's paragraph alignment. See the "alignment" [p. 223] property for details.

bool QMultiLineEdit::atBeginning () const

Returns TRUE if the cursor is placed at the beginning of the text; otherwise returns FALSE. See the "atBeginning" [p. 223] property for details.

bool QMultiLineEdit::atEnd () const

Returns TRUE if the cursor is placed at the end of the text; otherwise returns FALSE. See the "atEnd" [p. 223] property for details.

bool QMultiLineEdit::autoUpdate () const

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

void QMultiLineEdit::backspace () [virtual protected]

Deletes the character on the left side of the text cursor and moves the cursor one position to the left. If a text has been selected by the user (e.g. by clicking and dragging) the cursor is put at the beginning of the selected text and the selected text is removed. del()

void QMultiLineEdit::cursorDown (bool mark = FALSE) [virtual protected]

Moves the cursor one line down. If *mark* is TRUE, the text is selected.

See also cursorUp() [p. 220], cursorLeft() [p. 219] and cursorRight() [p. 220].

void QMultiLineEdit::cursorLeft (bool mark = FALSE, bool wrap = TRUE) [virtual protected]

Moves the cursor one character to the left. If *mark* is TRUE, the text is selected. The *wrap* parameter is currently ignored.

See also cursorRight() [p. 220], cursorUp() [p. 220] and cursorDown() [p. 219].

QPoint QMultiLineEdit::cursorPoint () const [protected]

Returns the top center point where the cursor is drawn.

void QMultiLineEdit::cursorRight (bool mark = FALSE, bool wrap = TRUE) [virtual protected]

Moves the cursor one character to the right. If *mark* is TRUE, the text is selected. The *wrap* parameter is currently ignored.

See also cursorLeft() [p. 219], cursorUp() [p. 220] and cursorDown() [p. 219].

void QMultiLineEdit::cursorUp (bool mark = FALSE) [virtual protected]

Moves the cursor up one line. If *mark* is TRUE, the text is selected.

See also cursorDown() [p. 219], cursorLeft() [p. 219] and cursorRight() [p. 220].

void QMultiLineEdit::cursorWordBackward (bool mark)

Moves the cursor one word to the left. If *mark* is TRUE, the text is selected.

See also cursorWordForward() [p. 220].

void QMultiLineEdit::cursorWordForward (bool mark)

Moves the cursor one word to the right. If *mark* is TRUE, the text is selected.

See also cursorWordBackward() [p. 220].

void QMultiLineEdit::deselect () [slot]

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

bool QMultiLineEdit::edited () const

Returns TRUE if the document has been edited by the user; otherwise returns FALSE. See the "edited" [p. 223] property for details.

void QMultiLineEdit::end (bool mark = FALSE) [virtual protected]

Moves the text cursor to the right end of the line. If *mark* is TRUE, text is selected toward the last position. If it is FALSE and the cursor is moved, all selected text is unselected.

See also home() [p. 221].

bool QMultiLineEdit::getMarkedRegion (int * line1, int * col1, int * line2, int * col2) const [protected]

If there is selected text, sets *line1*, *col1*, *line2* and *col2* to the start and end of the selected region and returns TRUE. Returns FALSE if there is no selected text.

bool QMultiLineEdit::hasMarkedText () const

Returns TRUE if there is selected text.

void QMultiLineEdit::home (bool mark = FALSE) [virtual protected]

Moves the text cursor to the left end of the line. If *mark* is TRUE, text is selected toward the first position. If it is FALSE and the cursor is moved, all selected text is unselected.

See also end() [p. 220].

void QMultiLineEdit::insertAndMark (const QString & str, bool mark) [virtual protected]

Inserts *str* at the current cursor position and selects the text if *mark* is TRUE.

void QMultiLineEdit::insertAt (const QString & s, int line, int col, bool mark = FALSE) [virtual]

Inserts string *s* at paragraph number *line*, after character number *col* in the paragraph. If *s* contains newline characters, new lines are inserted. If *mark* is TRUE the inserted string will be selected.

The cursor position is adjusted.

void QMultiLineEdit::insertLine (const QString & txt, int line = -1) [virtual]

Inserts *txt* at paragraph number *line*. If *line* is less than zero, or larger than the number of paragraphs, the new text is put at the end. If *txt* contains newline characters, several paragraphs are inserted.

The cursor position is not changed.

void QMultiLineEdit::killLine () [virtual protected]

Deletes text from the current cursor position to the end of the line. (Note that this function still operates on lines, not paragraphs.)

int QMultiLineEdit::lineLength (int row) const [protected]

Returns the number of characters at paragraph number *row*. If *row* is out of range, -1 is returned.

QString QMultiLineEdit::markedText () const

Returns a copy of the selected text.

int QMultiLineEdit::maxLines () const

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

void QMultiLineEdit::newLine () [virtual protected]

Splits the paragraph at the current cursor position.

int QMultiLineEdit::numLines () const

Returns the number of paragraphs in the editor. See the "numLines" [p. 224] property for details.

void QMultiLineEdit::pageDown (bool mark = FALSE) [virtual protected]

Moves the cursor one page down. If *mark* is TRUE, the text is selected.

void QMultiLineEdit::pageUp (bool mark = FALSE) [virtual protected]

Moves the cursor one page up. If *mark* is TRUE, the text is selected.

void QMultiLineEdit::removeLine (int paragraph) [virtual]

Deletes the paragraph at paragraph number *paragraph*. If *paragraph* is less than zero or larger than the number of paragraphs, nothing is deleted.

void QMultiLineEdit::setAlignment (int flags) [virtual]

Sets the editor's paragraph alignment to *flags*. See the "alignment" [p. 223] property for details.

Reimplemented from QTextEdit [p. 390].

void QMultiLineEdit::setAutoUpdate (bool) [virtual]

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

Example: qwerty/qwerty.cpp.

void QMultiLineEdit::setCursorPosition (int line, int col, bool mark = FALSE) [virtual]

Sets the cursor position to character number *col* in paragraph number *line*. The parameters are adjusted to lie within the legal range.

If *mark* is FALSE, the selection is cleared. otherwise it is extended.

void QMultiLineEdit::setEdited (bool)

Sets whether the document has been edited by the user. See the "edited" [p. 223] property for details.

void QMultiLineEdit::setMaxLines (int)

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

QString QMultiLineEdit::textLine (int line) const

Returns the text at line number *line* (possibly the empty string), or a null string if *line* is invalid.

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

int QMultiLineEdit::totalHeight () const

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

int QMultiLineEdit::totalWidth () const

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

Property Documentation

Alignment alignment

This property holds the editor's paragraph alignment.

Sets the alignment to flag, which must be AlignLeft, AlignHCenter or AlignRight.

If flag is an illegal flag nothing happens.

See also Qt::AlignmentFlags [Additional Functionality with Qt].

Set this property's value with setAlignment() and get this property's value with alignment().

bool atBeginning

This property holds whether the cursor is placed at the beginning of the text.

Get this property's value with atBeginning().

See also atEnd [p. 223].

bool atEnd

This property holds whether the cursor is placed at the end of the text.

Get this property's value with atEnd().

See also atBeginning [p. 223].

bool edited

This property holds whether the document has been edited by the user.

This is the same as QTextEdit's "modifed" property.

See also QTextEdit::modified [p. 397].

Set this property's value with setEdited() and get this property's value with edited().

int numLines

This property holds the number of paragraphs in the editor.

The count includes any empty paragraph at top and bottom, so for an empty editor this method returns 1.

Get this property's value with numLines().

QProgressBar Class Reference

The QProgressBar widget provides a horizontal progress bar.

`#include <qprogressbar.h>`

Inherits QFrame [p. 65].

Public Members

- **QProgressBar** (QWidget * parent = 0, const char * name = 0, WFlags f = 0)
- **QProgressBar** (int totalSteps, QWidget * parent = 0, const char * name = 0, WFlags f = 0)
- int **totalSteps** () const
- int **progress** () const
- const QString & **progressString** () const
- void **setCenterIndicator** (bool on)
- bool **centerIndicator** () const
- void **setIndicatorFollowsStyle** (bool)
- bool **indicatorFollowsStyle** () const
- bool **percentageVisible** () const
- void **setPercentageVisible** (bool)

Public Slots

- void **reset** ()
- virtual void **setTotalSteps** (int totalSteps)
- virtual void **setProgress** (int progress)

Properties

- bool **centerIndicator** — whether the indicator string should be centered
- bool **indicatorFollowsStyle** — whether the display of the indicator string should follow the GUI style
- bool **percentageVisible** — whether the current progress value is displayed
- int **progress** — the current amount of progress
- QString **progressString** — the current amount of progress as a string *(read only)*
- int **totalSteps** — the total number of steps

Protected Members

- virtual bool **setIndicator** (QString & indicator, int progress, int totalSteps)

Detailed Description

The QProgressBar widget provides a horizontal progress bar.

A progress bar is used to give the user an indication of the progress of an operation and to reassure them that the application is still running.

The progress bar uses the concept of *steps*; you give it the total number of steps and the number of steps completed so far and it will display the percentage of steps that have been completed. You can specify the total number of steps in the constructor or later with setTotalSteps(). The current number of steps is set with setProgress(). The progress bar can be rewound to the beginning with reset().

See also QProgressDialog [Dialogs and Windows with Qt], GUI Design Handbook: Progress Indicator and Advanced Widgets.

See also QProgressDialog [Dialogs and Windows with Qt], GUI Design Handbook: Progress Indicator and Advanced Widgets.

Member Function Documentation

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

Constructs a progress bar.

The total number of steps is set to 100 by default.

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

See also totalSteps [p. 229].

QProgressBar::QProgressBar (int totalSteps, QWidget * parent = 0, const char * name = 0, WFlags f = 0)

Constructs a progress bar.

The *totalSteps* is the total number of steps that need to be completed for the operation which this progress bar represents. For example, if the operation is to examine 50 files, this value would be 50. Before examining the first file, call setProgress(0); call setProgress(50) after examining the last file .

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

See also totalSteps [p. 229] and progress [p. 228].

bool QProgressBar::centerIndicator () const

Returns TRUE if the indicator string should be centered; otherwise returns FALSE. See the "centerIndicator" [p. 228] property for details.

bool QProgressBar::indicatorFollowsStyle () const

Returns TRUE if the display of the indicator string should follow the GUI style; otherwise returns FALSE. See the "indicatorFollowsStyle" [p. 228] property for details.

bool QProgressBar::percentageVisible () const

Returns TRUE if the current progress value is displayed; otherwise returns FALSE. See the "percentageVisible" [p. 228] property for details.

int QProgressBar::progress () const

Returns the current amount of progress. See the "progress" [p. 228] property for details.

const QString & QProgressBar::progressString () const

Returns the current amount of progress as a string. See the "progressString" [p. 228] property for details.

void QProgressBar::reset () [slot]

Reset the progress bar. The progress bar "rewinds" and shows no progress.

Examples: fileiconview/mainwindow.cpp, network/ftpclient/ftpmainwindow.cpp and progressbar/progressbar.cpp.

void QProgressBar::setCenterIndicator (bool on)

Sets whether the indicator string should be centered to *on*. See the "centerIndicator" [p. 228] property for details.

bool QProgressBar::setIndicator (QString & indicator, int progress, int totalSteps) [virtual protected]

This method is called to generate the text displayed in the center (or in some styles, to the left) of the progress bar.

The *progress* may be negative, indicating that the progress bar is in the "reset" state before any progress is set.

The default implementation is the percentage of completion or blank in the reset state. The percentage is calculated based on the *progress* and *totalSteps*. You can set the *indicator* text if you wish.

To allow efficient repainting of the progress bar, this method should return FALSE if the string is unchanged from the last call to this function.

void QProgressBar::setIndicatorFollowsStyle (bool)

Sets whether the display of the indicator string should follow the GUI style. See the "indicatorFollowsStyle" [p. 228] property for details.

void QProgressBar::setPercentageVisible (bool)

Sets whether the current progress value is displayed. See the "percentageVisible" [p. 228] property for details.

void QProgressBar::setProgress (int progress) [virtual slot]

Sets the current amount of progress to *progress*. See the "progress" [p. 228] property for details.

void QProgressBar::setTotalSteps (int totalSteps) [virtual slot]

Sets the total number of steps to *totalSteps*. See the "totalSteps" [p. 229] property for details.

int QProgressBar::totalSteps () const

Returns the total number of steps. See the "totalSteps" [p. 229] property for details.

Property Documentation

bool centerIndicator

This property holds whether the indicator string should be centered.

Changing this property sets QProgressBar::indicatorFollowsStyle to FALSE. The default is TRUE.

Set this property's value with setCenterIndicator() and get this property's value with centerIndicator().

bool indicatorFollowsStyle

This property holds whether the display of the indicator string should follow the GUI style.

The default is TRUE.

See also centerIndicator [p. 228].

Set this property's value with setIndicatorFollowsStyle() and get this property's value with indicatorFollowsStyle().

bool percentageVisible

This property holds whether the current progress value is displayed.

The default is TRUE.

\se centerIndicator, indicatorFollowsStyle

Set this property's value with setPercentageVisible() and get this property's value with percentageVisible().

int progress

This property holds the current amount of progress.

This property is -1 if the progress counting has not started.

Set this property's value with setProgress() and get this property's value with progress().

QString progressString

This property holds the current amount of progress as a string.

This property is QString::null if the progress counting has not started.

Get this property's value with progressString().

int totalSteps

This property holds the total number of steps.

If totalSteps is null, the progress bar will display a busy indicator.

See also totalSteps [p. 229].

Set this property's value with setTotalSteps() and get this property's value with totalSteps().

QPushButton Class Reference

The QPushButton widget provides a command button.

`#include <qpushbutton.h>`

Inherits QButton [p. 4].

Public Members

- **QPushButton** (QWidget * parent, const char * name = 0)
- **QPushButton** (const QString & text, QWidget * parent, const char * name = 0)
- **QPushButton** (const QIconSet & icon, const QString & text, QWidget * parent, const char * name = 0)
- **~QPushButton** ()
- void **setToggleButton** (bool)
- bool **autoDefault** () const
- virtual void **setAutoDefault** (bool autoDef)
- bool **isDefault** () const
- virtual void **setDefault** (bool def)
- virtual void setIsMenuButton (bool enable) *(obsolete)*
- bool isMenuButton () const *(obsolete)*
- void **setPopup** (QPopupMenu * popup)
- QPopupMenu * **popup** () const
- void **setIconSet** (const QIconSet &)
- QIconSet * **iconSet** () const
- void **setFlat** (bool)
- bool **isFlat** () const

Public Slots

- virtual void **setOn** (bool)

Important Inherited Members

- QString **text** () const
- virtual void **setText** (const QString &)
- const QPixmap * **pixmap** () const
- virtual void **setPixmap** (const QPixmap &)
- QKeySequence **accel** () const
- virtual void **setAccel** (const QKeySequence &)

- bool **isToggleButton** () const
- virtual void **setDown** (bool)
- bool **isDown** () const
- bool **isOn** () const
- ToggleState **state** () const
- bool **autoRepeat** () const
- virtual void **setAutoRepeat** (bool)
- bool **isExclusiveToggle** () const
- QButtonGroup * **group** () const
- void **toggle** ()
- void **pressed** ()
- void **released** ()
- void **clicked** ()
- void **toggled** (bool on)
- void **stateChanged** (int state)

Properties

- bool **autoDefault** — whether the push button is the auto default button
- bool **autoMask** — whether the button is automatically masked *(read only)*
- bool **default** — whether the push button is the default button
- bool **flat** — whether the border is disabled
- QIconSet **iconSet** — the icon set on the push button
- bool menuButton — whether the push button has a menu button on it *(read only)* *(obsolete)*
- bool **on** — whether the push button is toggled
- bool **toggleButton** — whether the button is a toggle button

Detailed Description

The QPushButton widget provides a command button.

The push button, or command button, is perhaps the most commonly used widget in any graphical user interface. Push (click) a button to command the computer to perform some action, or to answer a question. Typical buttons are OK, Apply, Cancel, Close, Yes, No and Help.

A command button is rectangular and typically displays a text label describing its action. An underscored character in the label (signified by preceding it with an ampersand in the text) indicates an accelerator key, e.g.

```
QPushButton *pb = new QPushButton( "&Download", this );
```

In this example the accelerator is *Ctrl+D*, and the label text will be displayed as **Download**.

Push buttons can display a textual label or a pixmap, and optionally a small icon. These can be set using the constructors and changed later using setText(), setPixmap() and setIconSet(). If the button is disabled the appearance of the text or pixmap and iconset will be manipulated with respect to the GUI style to make the button look "disabled".

A push button emits the signal clicked() when it is activated by the mouse, the Spacebar or by a keyboard accelerator. Connect to this signal to perform the button's action. Push buttons also provide less commonly used signals, for example, pressed() and released().

Command buttons in dialogs are by default auto-default buttons, i.e. they become the default push button automatically when they receive the keyboard input focus. A default button is a push button that is activated when the

user hits the Enter or Return key in a dialog. You can change this with setAutoDefault(). Note that auto-default buttons reserve a little extra space which is necessary to draw a default-button indicator. If you do not want this space around your buttons, call setAutoDefault(FALSE).

Being so central, the button widget has grown to accommodate a great many variations in the past decade. The Microsoft style guide now shows about ten different states of Windows push buttons and the text implies that there are dozens more when all the combinations of features are taken into consideration.

The most important modes or states are:

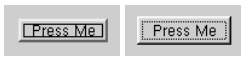
- Available or not (grayed out, disabled).
- Standard push button, toggling push button or menu button.
- On or off (only for toggling push buttons).
- Default or normal. The default button in a dialog can generally be "clicked" using the Enter or Return key.
- Auto-repeat or not.
- Pressed down or not.

As a general rule, use a push button when the application or dialog window performs an action when the user clicks on it (such as Apply, Cancel, Close and Help) *and* when the widget is supposed to have a wide, rectangular shape with a text label. Small, typically square buttons that change the state of the window rather than performing an action (such as the buttons in the top-right corner of the QFileDialog) are not command buttons, but tool buttons. Qt provides a special class (QToolButton) for these buttons.

If you need toggle behavior (see setToggleButton()) or a button that auto-repeats the activation signal when being pushed down like the arrows in a scroll bar (see setAutoRepeat()), a command button is probably not what you want. When in doubt, use a tool button.

A variation of a command button is a menu button. These provide not just one command, but several, since when they are clicked they pop up a menu of options. Use the method setPopup() to associate a popup menu with a push button.

Other classes of buttons are option buttons (see QRadioButton) and check boxes (see QCheckBox).



In Qt, the QButton base class provides most of the modes and other API, and QPushButton provides GUI logic. See QButton for more information about the API.

See also QToolButton [Dialogs and Windows with Qt], QRadioButton [p. 240], QCheckBox [p. 19], GUI Design Handbook: Push Button and Basic Widgets.

Member Function Documentation

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

Constructs a push button with no text.

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

QPushButton::QPushButton (const QString & text, QWidget * parent, const char * name = 0)

Constructs a push button called *name* with the parent *parent* and the text *text*.

QPushButton::QPushButton (const QIconSet & icon, const QString & text, QWidget * parent, const char * name = 0)

Constructs a push button with an *icon* and a *text*.

Note that you can also pass a QPixmap object as an icon (thanks to the implicit type conversion provided by C++).

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

QPushButton::~QPushButton ()

Destroys the push button

QKeySequence QButton::accel () const

Returns the accelerator associated with the button. See the "accel" [p. 11] property for details.

bool QPushButton::autoDefault () const

Returns TRUE if the push button is the auto default button; otherwise returns FALSE. See the "autoDefault" [p. 237] property for details.

bool QButton::autoRepeat () const

Returns TRUE if autoRepeat is enabled; otherwise returns FALSE. See the "autoRepeat" [p. 11] property for details.

void QButton::clicked () [signal]

This signal is emitted when the button is activated (i.e. first pressed down and then released when the mouse cursor is inside the button), when the accelerator key is typed or when animateClick() is called.

The QButtonGroup::clicked() signal does the same job, if you want to connect several buttons to the same slot.

See also pressed() [p. 9], released() [p. 9] and toggled() [p. 11].

Examples: fonts/simple-qfont-demo/viewer.cpp, listbox/listbox.cpp, network/clientserver/client/client.cpp, network/ftpclient/ftpmainwindow.cpp, richtext/richtext.cpp, t2/main.cpp and t4/main.cpp.

QButtonGroup * QButton::group () const

Returns a pointer to the group of which this button is a member.

If the button is not a member of any QButtonGroup, this function returns 0.

See also QButtonGroup [p. 14].

QIconSet * QPushButton::iconSet () const

Returns the icon set on the push button. See the "iconSet" [p. 238] property for details.

bool QPushButton::isDefault () const

Returns TRUE if the push button is the default button; otherwise returns FALSE. See the "default" [p. 237] property for details.

bool QButton::isDown () const

Returns TRUE if the button is pressed; otherwise returns FALSE. See the "down" [p. 11] property for details.

bool QButton::isExclusiveToggle () const

Returns TRUE if the button is an exclusive toggle; otherwise returns FALSE. See the "exclusiveToggle" [p. 12] property for details.

bool QPushButton::isFlat () const

Returns TRUE if the border is disabled; otherwise returns FALSE. See the "flat" [p. 237] property for details.

bool QPushButton::isMenuButton () const

Returns TRUE if the push button has a menu button on it; otherwise returns FALSE. See the "menuButton" [p. 238] property for details.

bool QButton::isOn () const

Returns TRUE if the button is toggled; otherwise returns FALSE. See the "on" [p. 12] property for details.

bool QButton::isToggleButton () const

Returns TRUE if the button is a toggle button; otherwise returns FALSE. See the "toggleButton" [p. 12] property for details.

const QPixmap * QButton::pixmap () const

Returns the pixmap shown on the button. See the "pixmap" [p. 12] property for details.

QPopupMenu * QPushButton::popup () const

Returns the button's associated popup menu or 0 if no popup menu has been defined.

See also setPopup() [p. 236].

void QButton::pressed () [signal]

This signal is emitted when the button is pressed down.

See also released() [p. 9] and clicked() [p. 8].

Examples: network/httpd/httpd.cpp and popup/popup.cpp.

void QButton::released () [signal]

This signal is emitted when the button is released.

See also pressed() [p. 9], clicked() [p. 8] and toggled() [p. 11].

void QButton::setAccel (const QKeySequence &) [virtual]

Sets the accelerator associated with the button. See the "accel" [p. 11] property for details.

void QPushButton::setAutoDefault (bool autoDef) [virtual]

Sets whether the push button is the auto default button to *autoDef*. See the "autoDefault" [p. 237] property for details.

void QButton::setAutoRepeat (bool) [virtual]

Sets whether autoRepeat is enabled. See the "autoRepeat" [p. 11] property for details.

void QPushButton::setDefault (bool def) [virtual]

Sets whether the push button is the default button to *def*. See the "default" [p. 237] property for details.

void QButton::setDown (bool) [virtual]

Sets whether the button is pressed. See the "down" [p. 11] property for details.

void QPushButton::setFlat (bool)

Sets whether the border is disabled. See the "flat" [p. 237] property for details.

void QPushButton::setIconSet (const QIconSet &)

Sets the icon set on the push button. See the "iconSet" [p. 238] property for details.

void QPushButton::setIsMenuButton (bool enable) [virtual]

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

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

Sets whether the push button is toggled. See the "on" [p. 238] property for details.

void QButton::setPixmap (const QPixmap &) [virtual]

Sets the pixmap shown on the button. See the "pixmap" [p. 12] property for details.

void QPushButton::setPopup (QPopupMenu * popup)

Associates the popup menu *popup* with this push button and thus turns it into a menu button.

Ownership of the popup menu is *not* transferred to the push button.

See also popup() [p. 234].

Example: qdir/qdir.cpp.

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

Sets the text shown on the button. See the "text" [p. 12] property for details.

void QPushButton::setToggleButton (bool)

Sets whether the button is a toggle button. See the "toggleButton" [p. 238] property for details.

ToggleState QButton::state () const

Returns TRUE if the button is toggled; otherwise returns FALSE. See the "toggleState" [p. 12] property for details.

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

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

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

See also clicked() [p. 8].

QString QButton::text () const

Returns the text shown on the button. See the "text" [p. 12] property for details.

void QButton::toggle () [slot]

Toggles the state of a toggle button.

See also on [p. 238], on [p. 238], toggled() [p. 11] and toggleButton [p. 238].

void QButton::toggled (bool on) [signal]

This signal is emitted whenever a toggle button changes status. *on* is TRUE if the button is on, or FALSE if the button is off.

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

See also clicked() [p. 8].

Example: listbox/listbox.cpp.

Property Documentation

QKeySequence accel

This property holds the accelerator associated with the button.

This property is 0 if there is no accelerator set. If you set this property to 0 then any current accelerator is removed.

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

bool autoDefault

This property holds whether the push button is the auto default button.

If this property is set to TRUE then the push button will be the focused item in a dialog when the dialog is first shown.

This property's default is FALSE.

Set this property's value with setAutoDefault() and get this property's value with autoDefault().

bool autoMask

This property holds whether the button is automatically masked.

See also QWidget::autoMask [p. 459].

bool autoRepeat

This property holds whether autoRepeat is enabled.

If autoRepeat is enabled then the clicked() signal is emitted at regular intervals if the button is down. This property has no effect on toggle buttons. autoRepeat is off by default.

Set this property's value with setAutoRepeat() and get this property's value with autoRepeat().

bool default

This property holds whether the push button is the default button.

If this property is set to TRUE then the push button will be pressed if the user hits the Enter key in a dialog.

This property's default is FALSE.

Set this property's value with setDefault() and get this property's value with isDefault().

bool flat

This property holds whether the border is disabled.

This property's default is FALSE.

Set this property's value with setFlat() and get this property's value with isFlat().

QIconSet iconSet

This property holds the icon set on the push button.

This property will return 0 if the push button has no iconset

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

bool menuButton

This property holds whether the push button has a menu button on it.

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

If this property is set to TRUE, then a down arrow is drawn on the push button to indicate that a menu will pop up if the user clicks on the arrow.

Get this property's value with isMenuButton().

bool on

This property holds whether the push button is toggled.

This property should only be set for toggle push buttons. The default value is FALSE.

See also on [p. 238], toggle() [p. 11], toggled() [p. 11] and toggleButton [p. 238].

Set this property's value with setOn().

QPixmap pixmap

This property holds the pixmap shown on the button.

If the pixmap is monochrome (i.e., it is a QBitmap or its depth is 1) and it does not have a mask, this property will set the pixmap to be its own mask. The purpose of this is to draw transparent bitmaps which are important for toggle buttons, for example.

pixmap() returns 0 if no pixmap was set.

Set this property's value with setPixmap() and get this property's value with pixmap().

QString text

This property holds the text shown on the button.

This property will return a null string if the button has no text. If the text has an ampersand ('&') in it, then an accelerator is automatically created for it using the character after the '&' as the accelerator key.

There is no default text.

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

bool toggleButton

This property holds whether the button is a toggle button.

Toggle buttons have an on/off state similar to check boxes. A push button is initially not a toggle button.

See also on [p. 238], toggle() [p. 11], toggleButton [p. 238] and toggled() [p. 11].

Set this property's value with setToggleButton().

QRadioButton Class Reference

The QRadioButton widget provides a radio button with a text or pixmap label.

`#include <qradiobutton.h>`

Inherits QButton [p. 4].

Public Members

- **QRadioButton** (QWidget * parent, const char * name = 0)
- **QRadioButton** (const QString & text, QWidget * parent, const char * name = 0)
- bool **isChecked** () const

Public Slots

- virtual void **setChecked** (bool check)

Important Inherited Members

- QString **text** () const
- virtual void **setText** (const QString &)
- const QPixmap * **pixmap** () const
- virtual void **setPixmap** (const QPixmap &)
- QKeySequence **accel** () const
- virtual void **setAccel** (const QKeySequence &)
- bool **isToggleButton** () const
- virtual void **setDown** (bool)
- bool **isDown** () const
- bool **isOn** () const
- ToggleState **state** () const
- bool **autoRepeat** () const
- virtual void **setAutoRepeat** (bool)
- bool **isExclusiveToggle** () const
- QButtonGroup * **group** () const
- void **toggle** ()
- void **pressed** ()
- void **released** ()
- void **clicked** ()
- void **toggled** (bool on)
- void **stateChanged** (int state)

Properties

- bool **autoMask** — whether the radio button is automatically masked *(read only)*
- bool **checked** — whether the radio button is checked

Detailed Description

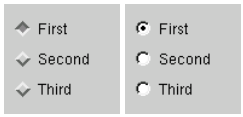
The QRadioButton widget provides a radio button with a text or pixmap label.

QRadioButton and QCheckBox are both option buttons. That is, they can be switched on (checked) or off (unchecked). The classes differ in how the choices for the user are restricted. Check boxes define "many of many" choices, whereas radio buttons provide a "one of many" choice. In a group of radio buttons only one button at a time can be checked; if the user selects another button, the previously selected button is switched off.

The easiest way to implement a "one of many" choice is simply to put the radio buttons into QButtonGroup.

Whenever a button is switched on or off it emits the signal toggled(). Connect to this signal if you want to trigger an action each time the button changes state. Otherwise, use isChecked() to query whether or not a particular button is selected.

Just like QPushButton, a radio button can display text or a pixmap. The text can be set in the constructor or with setText(); the pixmap is set with setPixmap().



See also QPushButton [p. 230], QToolButton [Dialogs and Windows with Qt], GUI Design Handbook: Radio Button and Basic Widgets.

Member Function Documentation

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

Constructs a radio button with no text.

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

QRadioButton::QRadioButton (const QString & text, QWidget * parent, const char * name = 0)

Constructs a radio button with the text *text*.

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

QKeySequence QButton::accel () const

Returns the accelerator associated with the button. See the "accel" [p. 11] property for details.

bool QButton::autoRepeat () const

Returns TRUE if autoRepeat is enabled; otherwise returns FALSE. See the "autoRepeat" [p. 11] property for details.

void QButton::clicked () [signal]

This signal is emitted when the button is activated (i.e. first pressed down and then released when the mouse cursor is inside the button), when the accelerator key is typed or when animateClick() is called.

The QButtonGroup::clicked() signal does the same job, if you want to connect several buttons to the same slot.

See also pressed() [p. 9], released() [p. 9] and toggled() [p. 11].

Examples: fonts/simple-qfont-demo/viewer.cpp, listbox/listbox.cpp, network/clientserver/client/client.cpp, network/ftpclient/ftpmainwindow.cpp, richtext/richtext.cpp, t2/main.cpp and t4/main.cpp.

QButtonGroup * QButton::group () const

Returns a pointer to the group of which this button is a member.

If the button is not a member of any QButtonGroup, this function returns 0.

See also QButtonGroup [p. 14].

bool QRadioButton::isChecked () const

Returns TRUE if the radio button is checked; otherwise returns FALSE. See the "checked" [p. 244] property for details.

bool QButton::isDown () const

Returns TRUE if the button is pressed; otherwise returns FALSE. See the "down" [p. 11] property for details.

bool QButton::isExclusiveToggle () const

Returns TRUE if the button is an exclusive toggle; otherwise returns FALSE. See the "exclusiveToggle" [p. 12] property for details.

bool QButton::isOn () const

Returns TRUE if the button is toggled; otherwise returns FALSE. See the "on" [p. 12] property for details.

bool QButton::isToggleButton () const

Returns TRUE if the button is a toggle button; otherwise returns FALSE. See the "toggleButton" [p. 12] property for details.

const QPixmap * QButton::pixmap () const

Returns the pixmap shown on the button. See the "pixmap" [p. 12] property for details.

void QButton::pressed () [signal]

This signal is emitted when the button is pressed down.

See also released() [p. 9] and clicked() [p. 8].

Examples: network/httpd/httpd.cpp and popup/popup.cpp.

void QButton::released () [signal]

This signal is emitted when the button is released.

See also pressed() [p. 9], clicked() [p. 8] and toggled() [p. 11].

void QButton::setAccel (const QKeySequence &) [virtual]

Sets the accelerator associated with the button. See the "accel" [p. 11] property for details.

void QButton::setAutoRepeat (bool) [virtual]

Sets whether autoRepeat is enabled. See the "autoRepeat" [p. 11] property for details.

void QRadioButton::setChecked (bool check) [virtual slot]

Sets whether the radio button is checked to *check*. See the "checked" [p. 244] property for details.

void QButton::setDown (bool) [virtual]

Sets whether the button is pressed. See the "down" [p. 11] property for details.

void QButton::setPixmap (const QPixmap &) [virtual]

Sets the pixmap shown on the button. See the "pixmap" [p. 12] property for details.

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

Sets the text shown on the button. See the "text" [p. 12] property for details.

ToggleState QButton::state () const

Returns TRUE if the button is toggled; otherwise returns FALSE. See the "toggleState" [p. 12] property for details.

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

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

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

See also clicked() [p. 8].

QString QButton::text () const

Returns the text shown on the button. See the "text" [p. 12] property for details.

void QButton::toggle () [slot]

Toggles the state of a toggle button.

See also on [p. 12], setOn() [p. 10], toggled() [p. 11] and toggleButton [p. 12].

void QButton::toggled (bool on) [signal]

This signal is emitted whenever a toggle button changes status. *on* is TRUE if the button is on, or FALSE if the button is off.

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

See also clicked() [p. 8].

Example: listbox/listbox.cpp.

Property Documentation

QKeySequence accel

This property holds the accelerator associated with the button.

This property is 0 if there is no accelerator set. If you set this property to 0 then any current accelerator is removed.

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

bool autoMask

This property holds whether the radio button is automatically masked.

See also QWidget::autoMask [p. 459].

bool autoRepeat

This property holds whether autoRepeat is enabled.

If autoRepeat is enabled then the clicked() signal is emitted at regular intervals if the button is down. This property has no effect on toggle buttons. autoRepeat is off by default.

Set this property's value with setAutoRepeat() and get this property's value with autoRepeat().

bool checked

This property holds whether the radio button is checked.

This property will not effect any other radio buttons unless they have been placed in the same QButtonGroup. The default value is FALSE (unchecked).

Set this property's value with setChecked() and get this property's value with isChecked().

QPixmap pixmap

This property holds the pixmap shown on the button.

If the pixmap is monochrome (i.e., it is a QBitmap or its depth is 1) and it does not have a mask, this property will set the pixmap to be its own mask. The purpose of this is to draw transparent bitmaps which are important for toggle buttons, for example.

pixmap() returns 0 if no pixmap was set.

Set this property's value with setPixmap() and get this property's value with pixmap().

QString text

This property holds the text shown on the button.

This property will return a null string if the button has no text. If the text has an ampersand ('&') in it, then an accelerator is automatically created for it using the character after the '&' as the accelerator key.

There is no default text.

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

QRangeControl Class Reference

The QRangeControl class provides an integer value within a range.

`#include <qrangecontrol.h>`

Inherited by QDial [p. 56], QScrollBar [p. 252], QSlider [p. 285] and QSpinBox [p. 295].

Public Members

- **QRangeControl** ()
- **QRangeControl** (int minValue, int maxValue, int lineStep, int pageStep, int value)
- virtual **~QRangeControl** ()
- int **value** () const
- void **setValue** (int value)
- void **addPage** ()
- void **subtractPage** ()
- void **addLine** ()
- void **subtractLine** ()
- int **minValue** () const
- int **maxValue** () const
- void **setRange** (int minValue, int maxValue)
- void **setMinValue** (int minVal)
- void **setMaxValue** (int maxVal)
- int **lineStep** () const
- int **pageStep** () const
- void **setSteps** (int lineStep, int pageStep)
- int **bound** (int v) const

Protected Members

- int **positionFromValue** (int logical_val, int span) const
- int **valueFromPosition** (int pos, int span) const
- void **directSetValue** (int value)
- int **prevValue** () const
- virtual void **valueChange** ()
- virtual void **rangeChange** ()
- virtual void **stepChange** ()

Detailed Description

The QRangeControl class provides an integer value within a range.

Although originally designed for the QScrollBar widget, the QRangeControl can also be used in conjunction with other widgets such as QSlider and QSpinBox. Here are the five main concepts in the class:

1. Current value. This is the bounded integer that QRangeControl maintains. value() returns this and several functions, including setValue(), set it.
2. Minimum. This is the lowest value that value() can ever return. Returned by minValue() and set by setRange() or one of the constructors.
3. Maximum. This is the highest value that value() can ever return. Returned by maxValue() and set by setRange() or one of the constructors.
4. Line step. This is the smaller of two natural steps that QRangeControl provides and typically corresponds to the user pressing an arrow key. The line step is returned by lineStep() and set using setSteps(). The functions addLine() and subtractLine() respectively increment and decrement the current value by lineStep().
5. Page step. This is the larger of two natural steps that QRangeControl provides and typically corresponds to the user pressing PageUp or PageDown. The page step is returned by pageStep() and set using setSteps(). The functions addPage() and substractPage() respectively increment and decrement the current value by pageStep().

Unity (1) may be viewed as a third step size. setValue() lets you set the current value to any integer in the allowed range, not just minValue() + *n* * lineStep() for integer values of *n*. Some widgets may allow the user to set any value at all; others may just provide multiples of lineStep() or pageStep().

QRangeControl provides three virtual functions that are well suited for updating the on-screen representation of range controls and emitting signals, namely valueChange(), rangeChange() and stepChange().

QRangeControl also provides a function called bound() which lets you force arbitrary integers to be within the allowed range of the range control.

We recommend that all widgets that inherit QRangeControl provide at least a signal called valueChanged(); many widgets will want to provide addStep(), addPage(), substractStep() and substractPage() as slots.

Note that you have to use multiple inheritance if you plan to implement a widget using QRangeControl because QRangeControl is not derived from QWidget.

See also Miscellaneous Classes.

Member Function Documentation

QRangeControl::QRangeControl ()

Constructs a range control with min value 0, max value 99, line step 1, page step 10 and initial value 0.

QRangeControl::QRangeControl (int minValue, int maxValue, int lineStep, int pageStep, int value)

Constructs a range control whose value can never be smaller than *minValue* or greater than *maxValue*, whose line step size is *lineStep* and page step size is *pageStep* and whose value is initially *value* (which is guaranteed to be in range using bound()).

QRangeControl::~QRangeControl () [virtual]

Destroys the range control

void QRangeControl::addLine ()

Equivalent to `setValue( value() + lineStep() )`.

If the value is changed, then valueChange() is called.

See also subtractLine() [p. 250], addPage() [p. 248] and setValue() [p. 250].

void QRangeControl::addPage ()

Equivalent to `setValue( value() + pageStep() )`.

If the value is changed, then valueChange() is called.

See also subtractPage() [p. 251], addLine() [p. 248] and setValue() [p. 250].

int QRangeControl::bound (int v) const

Forces the value *v* to be within the range from minValue() to maxValue() inclusive, and returns the result.

This function is provided so that you can easily force other numbers than value() into the allowed range. You do not need to call it in order to use QRangeControl itself.

See also setValue() [p. 250], value() [p. 251], minValue() [p. 249] and maxValue() [p. 248].

void QRangeControl::directSetValue (int value) [protected]

Sets the range control *value* directly without calling valueChange().

Forces the new *value* to be within the legal range.

You will rarely have to call this function. However, if you want to change the range control's value inside the overloaded method valueChange(), setValue() would call the function valueChange() again. To avoid this recursion you must use directSetValue() instead.

See also setValue() [p. 250].

int QRangeControl::lineStep () const

Returns the current line step.

See also setSteps() [p. 250] and pageStep() [p. 249].

int QRangeControl::maxValue () const

Returns the current maximum value of the range.

See also setMaxValue() [p. 249], setRange() [p. 250] and minValue() [p. 249].

int QRangeControl::minValue () const

Returns the current minimum value of the range.

See also setMinValue() [p. 250], setRange() [p. 250] and maxValue() [p. 248].

int QRangeControl::pageStep () const

Returns the current page step.

See also setSteps() [p. 250] and lineStep() [p. 248].

int QRangeControl::positionFromValue (int logical_val, int span) const [protected]

Converts *logical_val* to a pixel position. minValue() maps to 0, maxValue() maps to *span* and other values are distributed evenly in-between.

This function can handle the entire integer range without overflow.

Calling this method is useful when actually drawing a range control such as a QScrollBar on-screen.

See also valueFromPosition() [p. 251].

int QRangeControl::prevValue () const [protected]

Returns the previous value of the range control. "Previous value" means the value before the last change occurred. Setting a new range may affect the value, too, because the value is forced to be inside the specified range. When the range control is initially created, this is the same as value().

prevValue() can be outside the current legal range if a call to setRange() causes the current value to change. For example, if the range was [0, 1000] and the current value is 500, setRange(0, 400) makes value() return 400 and prevValue() return 500.

See also value() [p. 251] and setRange() [p. 250].

void QRangeControl::rangeChange () [virtual protected]

This virtual function is called whenever the range control's range changes. You can reimplement it if you want to be notified when the range changes. The default implementation does nothing.

Note that this method is called after the range changed.

See also setRange() [p. 250], valueChange() [p. 251] and stepChange() [p. 250].

Reimplemented in QDial, QSlider and QSpinBox.

void QRangeControl::setMaxValue (int maxVal)

Sets the current minimum value of the range to *maxVal*.

If necessary, the minValue() is adjusted so that the range remains valid.

See also maxValue() [p. 248] and setMinValue() [p. 250].

void QRangeControl::setMinValue (int minVal)

Sets the current minimum value of the range to *minVal*.

If necessary, the maxValue() is adjusted so that the range remains valid.

See also minValue() [p. 249] and setMaxValue() [p. 249].

void QRangeControl::setRange (int minValue, int maxValue)

Sets the range control's min value to *minValue* and its max value to *maxValue*.

Calls the virtual rangeChange() function if one or both of the new min and max values are different from the previous setting. Calls the virtual valueChange() function if the current value is adjusted because it was outside the new range.

If *maxValue* is smaller than *minValue*, *minValue* becomes the only legal value.

See also minValue() [p. 249] and maxValue() [p. 248].

Examples: listbox/listbox.cpp, t12/lcdrange.cpp, t5/main.cpp, t6/main.cpp, t8/lcdrange.cpp and xform/xform.cpp.

void QRangeControl::setSteps (int lineStep, int pageStep)

Sets the range line step to *lineStep* and page step to *pageStep*.

Calls the virtual stepChange() function if the new line step and/or page step are different from the previous settings.

See also lineStep() [p. 248], pageStep() [p. 249] and setRange() [p. 250].

void QRangeControl::setValue (int value)

Sets the range control's value to *value* and forces it to be within the legal range.

Calls the virtual valueChange() function if the new value is different from the previous value. The old value can still be retrieved using prevValue().

See also value() [p. 251].

void QRangeControl::stepChange () [virtual protected]

This virtual function is called whenever the range control's line/page step settings change. You can reimplement it if you want to be notified when the step changes. The default implementation does nothing.

Note that this method is called after the step settings change.

See also setSteps() [p. 250], rangeChange() [p. 249] and valueChange() [p. 251].

void QRangeControl::subtractLine ()

Equivalent to `setValue( value() - lineStep() )`.

If the value is changed, then valueChange() is called.

See also addLine() [p. 248], subtractPage() [p. 251] and setValue() [p. 250].

void QRangeControl::subtractPage ()

Equivalent to `setValue( value() - pageStep() )`.

If the value is changed, then valueChange() is called.

See also addPage() [p. 248], subtractLine() [p. 250] and setValue() [p. 250].

int QRangeControl::value () const

Returns the current range control value. This is guaranteed to be within the range [minValue(), maxValue()].

See also setValue() [p. 250] and prevValue() [p. 249].

void QRangeControl::valueChange () [virtual protected]

This virtual function is called whenever the range control value changes. You can reimplement it if you want to be notified when the value changes. The default implementation does nothing.

Note that this method is called after the value changed. The previous value can be retrieved using prevValue().

See also setValue() [p. 250], addPage() [p. 248], subtractPage() [p. 251], addLine() [p. 248], subtractLine() [p. 250], rangeChange() [p. 249] and stepChange() [p. 250].

Reimplemented in QDial, QSlider and QSpinBox.

int QRangeControl::valueFromPosition (int pos, int span) const [protected]

Converts the pixel position *pos* to a value. 0 maps to minValue(), *span* maps to maxValue() and other values are distributed evenly in-between.

This function can handle the entire integer range without overflow.

Calling this method is useful if you actually implemented a range control widget such as QScrollBar and want to handle mouse press events. This function then maps screen coordinates to the logical values.

See also positionFromValue() [p. 249].

QScrollBar Class Reference

The QScrollBar widget provides a vertical or horizontal scroll bar.

`#include <qscrollbar.h>`

Inherits QWidget [p. 412] and QRangeControl [p. 246].

Public Members

- **QScrollBar** (QWidget * parent, const char * name = 0)
- **QScrollBar** (Orientation orientation, QWidget * parent, const char * name = 0)
- **QScrollBar** (int minValue, int maxValue, int lineStep, int pageStep, int value, Orientation orientation, QWidget * parent, const char * name = 0)
- virtual void **setOrientation** (Orientation)
- Orientation **orientation** () const
- virtual void **setTracking** (bool enable)
- bool **tracking** () const
- bool **draggingSlider** () const
- virtual void **setPalette** (const QPalette & p)
- int **minValue** () const
- int **maxValue** () const
- void **setMinValue** (int)
- void **setMaxValue** (int)
- int **lineStep** () const
- int **pageStep** () const
- void **setLineStep** (int)
- void **setPageStep** (int)
- int **value** () const
- int **sliderStart** () const
- QRect **sliderRect** () const

Public Slots

- void **setValue** (int)

Signals

- void **valueChanged** (int value)
- void **sliderPressed** ()

- void **sliderMoved** (int value)
- void **sliderReleased** ()
- void **nextLine** ()
- void **prevLine** ()
- void **nextPage** ()
- void **prevPage** ()

Properties

- bool **draggingSlider** — whether the user has clicked the mouse on the slider and is currently dragging it *(read only)*
- int **lineStep** — the current line step
- int **maxValue** — the current maximum value of the scroll bar
- int **minValue** — the current minimum value of the scroll bar
- Orientation **orientation** — the orientation of the scroll bar
- int **pageStep** — the current line step
- bool **tracking** — whether scroll bar tracking is enabled
- int **value** — the current scroll bar value

Detailed Description

The QScrollBar widget provides a vertical or horizontal scroll bar.

A scroll bar allows the user to control a value within a program-definable range and gives users a visible indication of the current value of a range control.

Scroll bars include four separate controls:

- The *line-up* and *line-down* controls are little buttons which the user can use to move one line up or down. The meaning of "line" is configurable. In editors and list boxes it means one line of text; in an image viewer it might mean 20 pixels.
- The *slider* is the handle that indicates the current value of the scroll bar, which the user can drag to change the value. This part of the scroll bar is sometimes called the "thumb".
- The *page-up/page-down* control is the area on which the slider slides (the scroll bar's background). Clicking here moves the scroll bar towards the click. The meaning of "page" is also configurable: in editors and list boxes it means as many lines as there is space for in the widget.

QScrollBar has very few of its own functions; it mostly relies on QRangeControl. The most useful functions are setValue() to set the scroll bar directly to some value; addPage(), addLine(), subtractPage(), and subtractLine() to simulate the effects of clicking (useful for accelerator keys); setSteps() to define the values of pageStep() and lineStep(); and setRange() to set the minValue() and maxValue() of the scroll bar. QScrollBar has a convenience constructor with which you can set most of these properties.

Some GUI styles (for example, the Windows and Motif styles provided), also use the pageStep() value to calculate the size of the slider.

In addition to the access functions from QRangeControl, QScrollBar has a comprehensive set of signals:

- valueChanged() - emitted when the scroll bar's value has changed. The tracking() determines whether this signal is emitted during user interaction.
- sliderPressed() - emitted when the user starts to drag the slider.
- sliderMoved() - emitted when the user drags the slider.

- sliderReleased() - emitted when the user releases the slider.
- nextLine() - emitted when the scroll bar has moved one line down or right. Line is defined in QRangeControl.
- prevLine() - emitted when the scroll bar has moved one line up or left.
- nextPage() - emitted when the scroll bar has moved one page down or right.
- prevPage() - emitted when the scroll bar has moved one page up or left.

QScrollBar only provides integer ranges. Note that although QScrollBar handles very large numbers, scroll bars on current screens cannot usefully control ranges above about 100,000 pixels. Beyond that, it becomes difficult for the user to control the scroll bar using either the keyboard or the mouse.

A scroll bar can be controlled by the keyboard, but it has a default focusPolicy() of NoFocus. Use setFocusPolicy() to enable keyboard focus. See keyPressEvent() for a list of key bindings.

If you need to add scroll bars to an interface, consider using the QScrollView class, which encapsulates the common uses for scroll bars.

See also QSlider [p. 285], QSpinBox [p. 295], QScrollView [p. 259], GUI Design Handbook: Scroll Bar and Basic Widgets.

Member Function Documentation

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

Constructs a vertical scroll bar.

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

QScrollBar::QScrollBar (Orientation orientation, QWidget * parent, const char * name = 0)

Constructs a scroll bar.

The *orientation* must be Qt::Vertical or Qt::Horizontal.

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

QScrollBar::QScrollBar (int minValue, int maxValue, int lineStep, int pageStep, int value, Orientation orientation, QWidget * parent, const char * name = 0)

Constructs a scroll bar whose value can never be smaller than *minValue* or greater than *maxValue*, whose line step size is *lineStep* and page step size is *pageStep* and whose value is initially *value* (which is guaranteed to be in range using bound()).

If *orientation* is Vertical the scroll bar is vertical and if it is Horizontal the scroll bar is horizontal.

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

bool QScrollBar::draggingSlider () const

Returns TRUE if the user has clicked the mouse on the slider and is currently dragging it; otherwise returns FALSE. See the "draggingSlider" [p. 257] property for details.

int QScrollBar::lineStep () const

Returns the current line step. See the "lineStep" [p. 257] property for details.

int QScrollBar::maxValue () const

Returns the current maximum value of the scroll bar. See the "maxValue" [p. 257] property for details.

int QScrollBar::minValue () const

Returns the current minimum value of the scroll bar. See the "minValue" [p. 257] property for details.

void QScrollBar::nextLine () [signal]

This signal is emitted when the scroll bar scrolls one line down or right.

void QScrollBar::nextPage () [signal]

This signal is emitted when the scroll bar scrolls one page down or right.

Orientation QScrollBar::orientation () const

Returns the orientation of the scroll bar. See the "orientation" [p. 258] property for details.

int QScrollBar::pageStep () const

Returns the current line step. See the "pageStep" [p. 258] property for details.

void QScrollBar::prevLine () [signal]

This signal is emitted when the scroll bar scrolls one line up or left.

void QScrollBar::prevPage () [signal]

This signal is emitted when the scroll bar scrolls one page up or left.

void QScrollBar::setLineStep (int)

Sets the current line step. See the "lineStep" [p. 257] property for details.

void QScrollBar::setMaxValue (int)

Sets the current maximum value of the scroll bar. See the "maxValue" [p. 257] property for details.

void QScrollBar::setMinValue (int)

Sets the current minimum value of the scroll bar. See the "minValue" [p. 257] property for details.

void QScrollBar::setOrientation (Orientation) [virtual]

Sets the orientation of the scroll bar. See the "orientation" [p. 258] property for details.

void QScrollBar::setPageStep (int)

Sets the current line step. See the "pageStep" [p. 258] property for details.

void QScrollBar::setPalette (const QPalette & p) [virtual]

Reimplements the virtual function QWidget::setPalette().

Sets the background color to the mid color for Motif style scroll bars using palette *p*.

Reimplemented from QWidget [p. 451].

void QScrollBar::setTracking (bool enable) [virtual]

Sets whether scroll bar tracking is enabled to *enable*. See the "tracking" [p. 258] property for details.

void QScrollBar::setValue (int) [slot]

Sets the current scroll bar value. See the "value" [p. 258] property for details.

void QScrollBar::sliderMoved (int value) [signal]

This signal is emitted when the slider is moved by the user, with the new scroll bar *value* as an argument.

This signal is emitted even when tracking is turned off.

See also tracking [p. 258], valueChanged() [p. 257], nextLine() [p. 255], prevLine() [p. 255], nextPage() [p. 255] and prevPage() [p. 255].

void QScrollBar::sliderPressed () [signal]

This signal is emitted when the user presses the slider with the mouse.

QRect QScrollBar::sliderRect () const

Returns the scroll bar slider rectangle.

See also sliderStart() [p. 257].

void QScrollBar::sliderReleased () [signal]

This signal is emitted when the user releases the slider with the mouse.

int QScrollBar::sliderStart () const

Returns the pixel position where the scroll bar slider starts.

This is equivalent to sliderRect().y() for vertical scroll bars or sliderRect().x() for horizontal scroll bars.

bool QScrollBar::tracking () const

Returns TRUE if scroll bar tracking is enabled; otherwise returns FALSE. See the "tracking" [p. 258] property for details.

int QScrollBar::value () const

Returns the current scroll bar value. See the "value" [p. 258] property for details.

void QScrollBar::valueChanged (int value) [signal]

This signal is emitted when the scroll bar value has changed, with the new scroll bar *value* as an argument.

Property Documentation

bool draggingSlider

This property holds whether the user has clicked the mouse on the slider and is currently dragging it.

Get this property's value with draggingSlider().

int lineStep

This property holds the current line step.

When setting lineStep, the virtual stepChange() function will be called if the new line step is different from the previous setting.

See also setSteps() [p. 250], QRangeControl::pageStep() [p. 249] and setRange() [p. 250].

Set this property's value with setLineStep() and get this property's value with lineStep().

int maxValue

This property holds the current maximum value of the scroll bar.

When setting this property, the QScrollBar::minValue is adjusted so that the range remains valid if necessary.

See also setRange() [p. 250].

Set this property's value with setMaxValue() and get this property's value with maxValue().

int minValue

This property holds the current minimum value of the scroll bar.

When setting this property, the QScrollBar::maxValue is adjusted so that the range remains valid if necessary.

See also setRange() [p. 250].

Set this property's value with setMinValue() and get this property's value with minValue().

Orientation orientation

This property holds the orientation of the scroll bar.

The orientation must be Qt::Vertical (the default) or Qt::Horizontal.

Set this property's value with setOrientation() and get this property's value with orientation().

int pageStep

This property holds the current line step.

When setting pageStep, the virtual stepChange() function will be called if the new page step is different from the previous setting.

See also QRangeControl::setSteps() [p. 250], lineStep [p. 257] and setRange() [p. 250].

Set this property's value with setPageStep() and get this property's value with pageStep().

bool tracking

This property holds whether scroll bar tracking is enabled.

If tracking is enabled (the default), the scroll bar emits the valueChanged() signal while the slider is being dragged. If tracking is disabled, the scroll bar emits the valueChanged() signal only when the user releases the mouse button after moving the slider.

Set this property's value with setTracking() and get this property's value with tracking().

int value

This property holds the current scroll bar value.

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

See also QRangeControl::value() [p. 251] and prevValue() [p. 249].

QScrollView Class Reference

The QScrollView widget provides a scrolling area with on-demand scroll bars.

`#include <qscrollview.h>`

Inherits QFrame [p. 65].

Inherited by QCanvasView [Graphics with Qt], QTable [p. 325], QGridView [p. 73], QIconView [p. 86], QListBox [p. 146], QListView [p. 177] and QTextEdit [p. 372].

Public Members

- **QScrollView** (QWidget * parent = 0, const char * name = 0, WFlags f = 0)
- **~QScrollView** ()
- enum **ResizePolicy** { Default, Manual, AutoOne, AutoOneFit }
- virtual void **setResizePolicy** (ResizePolicy)
- ResizePolicy **resizePolicy** () const
- void **removeChild** (QWidget * child)
- virtual void **addChild** (QWidget * child, int x = 0, int y = 0)
- virtual void **moveChild** (QWidget * child, int x, int y)
- int **childX** (QWidget * child)
- int **childY** (QWidget * child)
- bool childIsVisible (QWidget * child) *(obsolete)*
- void showChild (QWidget * child, bool y = TRUE) *(obsolete)*
- enum **ScrollBarMode** { Auto, AlwaysOff, AlwaysOn }
- ScrollBarMode **vScrollBarMode** () const
- virtual void **setVScrollBarMode** (ScrollBarMode)
- ScrollBarMode **hScrollBarMode** () const
- virtual void **setHScrollBarMode** (ScrollBarMode)
- QWidget * **cornerWidget** () const
- virtual void **setCornerWidget** (QWidget * corner)
- QScrollBar * **horizontalScrollBar** () const
- QScrollBar * **verticalScrollBar** () const
- QWidget * **viewport** () const
- QWidget * **clipper** () const
- int **visibleWidth** () const
- int **visibleHeight** () const
- int **contentsWidth** () const
- int **contentsHeight** () const
- int **contentsX** () const
- int **contentsY** () const
- void **updateContents** (int x, int y, int w, int h)

- void **updateContents** (const QRect & r)
- void **updateContents** ()
- void **repaintContents** (int x, int y, int w, int h, bool erase = TRUE)
- void **repaintContents** (const QRect & r, bool erase = TRUE)
- void **repaintContents** (bool erase = TRUE)
- void **contentsToViewport** (int x, int y, int & vx, int & vy) const
- void **viewportToContents** (int vx, int vy, int & x, int & y) const
- QPoint **contentsToViewport** (const QPoint & p) const
- QPoint **viewportToContents** (const QPoint & vp) const
- void **enableClipper** (bool y)
- void **setStaticBackground** (bool y)
- bool **hasStaticBackground** () const
- QSize **viewportSize** (int x, int y) const
- virtual void **setDragAutoScroll** (bool b)
- bool **dragAutoScroll** () const

Public Slots

- virtual void **resizeContents** (int w, int h)
- void **scrollBy** (int dx, int dy)
- virtual void **setContentsPos** (int x, int y)
- void **ensureVisible** (int x, int y)
- void **ensureVisible** (int x, int y, int xmargin, int ymargin)
- void **center** (int x, int y)
- void **center** (int x, int y, float xmargin, float ymargin)
- void **updateScrollBars** ()

Signals

- void **contentsMoving** (int x, int y)

Properties

- int **contentsHeight** — the height of the contents area *(read only)*
- int **contentsWidth** — the width of the contents area *(read only)*
- int **contentsX** — the X coordinate of the contents that are at the left edge of the viewport *(read only)*
- int **contentsY** — the Y coordinate of the contents that are at the top edge of the viewport *(read only)*
- bool **dragAutoScroll** — whether autoscrolling in drag move events is enabled
- ScrollBarMode **hScrollBarMode** — the mode for the horizontal scroll bar
- ResizePolicy **resizePolicy** — the current resize policy
- ScrollBarMode **vScrollBarMode** — the mode for the vertical scroll bar
- int **visibleHeight** — the vertical amount of the content that is visible *(read only)*
- int **visibleWidth** — the horizontal amount of the content that is visible *(read only)*

Protected Members

- virtual void **drawContents** (QPainter * p, int clipx, int clipy, int clipw, int cliph)
- virtual void **drawContentsOffset** (QPainter * p, int offsetx, int offsety, int clipx, int clipy, int clipw, int cliph)
- virtual void **contentsMousePressEvent** (QMouseEvent *)
- virtual void **contentsMouseReleaseEvent** (QMouseEvent *)
- virtual void **contentsMouseDoubleClickEvent** (QMouseEvent *)
- virtual void **contentsMouseMoveEvent** (QMouseEvent *)
- virtual void **contentsDragEnterEvent** (QDragEnterEvent *)
- virtual void **contentsDragMoveEvent** (QDragMoveEvent *)
- virtual void **contentsDragLeaveEvent** (QDragLeaveEvent *)
- virtual void **contentsDropEvent** (QDropEvent *)
- virtual void **contentsWheelEvent** (QWheelEvent * e)
- virtual void **contentsContextMenuEvent** (QContextMenuEvent * e)
- virtual void **viewportPaintEvent** (QPaintEvent * pe)
- virtual void **viewportResizeEvent** (QResizeEvent *)
- virtual void **setMargins** (int left, int top, int right, int bottom)
- int **leftMargin** () const
- int **topMargin** () const
- int **rightMargin** () const
- int **bottomMargin** () const
- virtual void **setHBarGeometry** (QScrollBar & hbar, int x, int y, int w, int h)
- virtual void **setVBarGeometry** (QScrollBar & vbar, int x, int y, int w, int h)
- virtual bool **eventFilter** (QObject * obj, QEvent * e)

Detailed Description

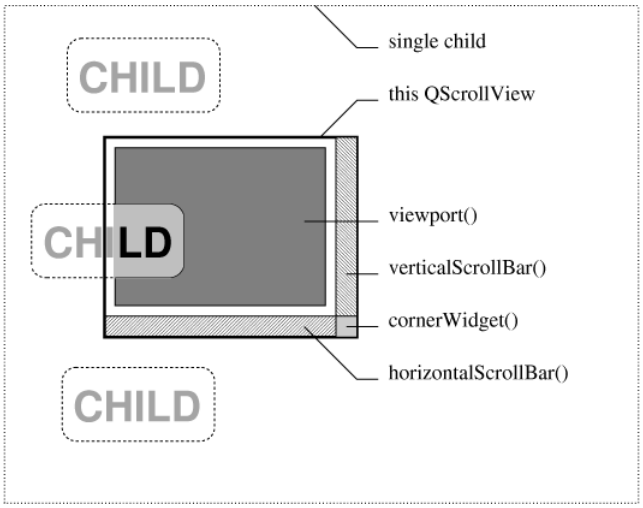
The QScrollView widget provides a scrolling area with on-demand scroll bars.

The QScrollView is a large canvas - potentially larger than the coordinate system normally supported by the underlying window system. This is important because it is quite easy to go beyond these limitations (e.g. many web pages are more than 32000 pixels high). Additionally, the QScrollView can have QWidgets positioned on it that scroll around with the drawn content. These subwidgets can also have positions outside the normal coordinate range (but they are still limited in size).

To provide content for the widget, inherit from QScrollView, reimplement drawContents() and use resizeContents() to set the size of the viewed area. Use addChild() and moveChild() to position widgets on the view.

To use QScrollView effectively it is important to understand its widget structure in the three styles of use: a single large child widget, a large panning area with some widgets and a large panning area with many widgets.

Using One Big Widget



The first, simplest usage of QScrollView (depicted above), is appropriate for scrolling areas that are never more than about 4000 pixels in either dimension (this is about the maximum reliable size on X11 servers). In this usage, you just make one large child in the QScrollView. The child should be a child of the viewport() of the scrollview and be added with addChild():

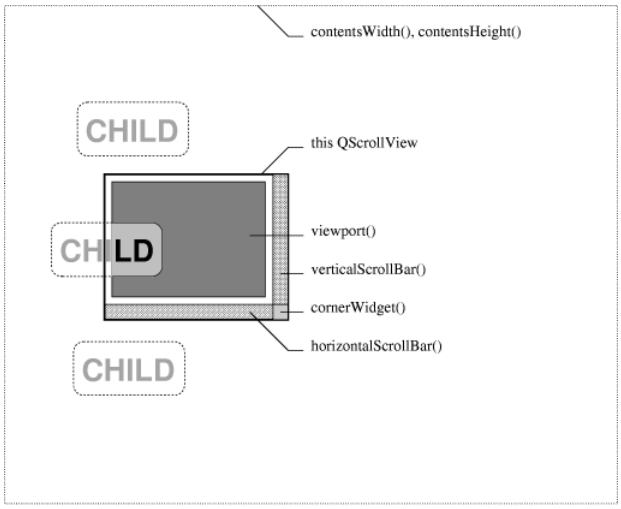
```
QScrollView* sv = new QScrollView(...);
QVBox* big_box = new QVBox(sv->viewport());
sv->addChild(big_box);
```

You may go on to add arbitrary child widgets to the single child in the scrollview as you would with any widget:

```
QLabel* child1 = new QLabel("CHILD", big_box);
QLabel* child2 = new QLabel("CHILD", big_box);
QLabel* child3 = new QLabel("CHILD", big_box);
...
```

Here the QScrollView has four children: the viewport(), the verticalScrollBar(), the horizontalScrollBar() and a small cornerWidget(). The viewport() has one child — the big QVBox. The QVBox has the three QLabel objects as child widgets. When the view is scrolled, the QVBox is moved; its children move with it as child widgets normally do.

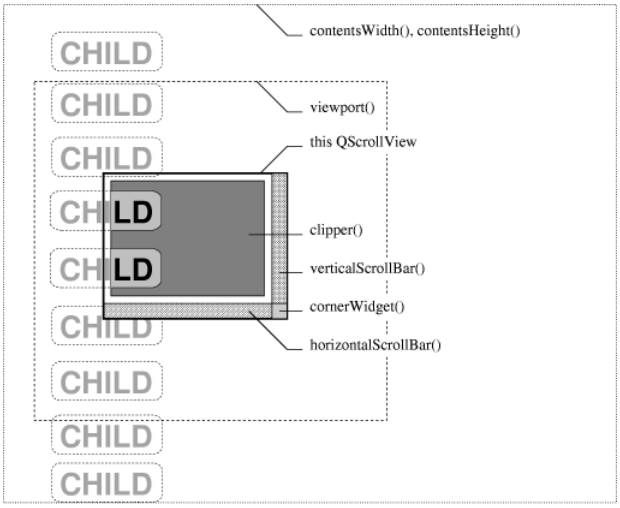
Using a Very Big View with Some Widgets



The second usage of QScrollView (depicted above) is appropriate when few, if any, widgets are on a very large scrolling area that is potentially larger than 4000 pixels in either dimension. In this usage you call resizeContents() to set the size of the area and reimplement drawContents() to paint the contents. You may also add some widgets by making them children of the viewport() and adding them with addChild() (this is the same as the process for the single large widget in the previous example):

```
QScrollView* sv = new QScrollView(...);
QLabel* child1 = new QLabel("CHILD", sv->viewport());
sv->addChild(child1);
QLabel* child2 = new QLabel("CHILD", sv->viewport());
sv->addChild(child2);
QLabel* child3 = new QLabel("CHILD", sv->viewport());
sv->addChild(child3);
```

Here, the QScrollView has the same four children: the viewport(), the verticalScrollBar(), the horizontalScrollBar() and a small cornerWidget(). The viewport() has the three QLabel objects as child widgets. When the view is scrolled, the scrollview moves the child widgets individually.



Using a Very Big View with Many Widgets

The final usage of QScrollView (depicted above) is appropriate when many widgets are on a very large scrolling area that is potentially larger than 4000 pixels in either dimension. In this usage you call resizeContents() to set

the size of the area and reimplement drawContents() to paint the contents. You then call enableClipper(TRUE) and add widgets, again by making them children of the viewport(), and adding them with addChild():

```
QScrollView* sv = new QScrollView(...);
sv->enableClipper(TRUE);
QLabel* child1 = new QLabel("CHILD", sv->viewport());
sv->addChild(child1);
QLabel* child2 = new QLabel("CHILD", sv->viewport());
sv->addChild(child2);
QLabel* child3 = new QLabel("CHILD", sv->viewport());
sv->addChild(child3);
```

Here, the QScrollView has four children: the clipper() (not the viewport() this time), the verticalScrollBar(), the horizontalScrollBar() and a small cornerWidget(). The clipper() has one child: the viewport(). The viewport() has the same three labels as child widgets. When the view is scrolled the viewport() is moved; its children move with it as child widgets normally do.

Details Relevant for All Views

Normally you will use the first or third method if you want any child widgets in the view.

Note that the widget you see in the scrolled area is the viewport() widget, not the QScrollView itself. So to turn mouse tracking on, for example, use viewport()->setMouseTracking(TRUE).

To enable drag-and-drop, you would setAcceptDrops(TRUE) on the QScrollView (because drag-and-drop events propagate to the parent). But to work out the logical position in the view, you would need to map the drop co-ordinate from being relative to the QScrollView to being relative to the contents; use the function viewportTo-Contents() for this.

To handle mouse events on the scrolling area, subclass scrollview as you would subclass other widgets, but rather than reimplementing mousePressEvent(), reimplement contentsMousePressEvent() instead. The contents specific event handers provide translated events in the coordinate system of the scrollview. If you reimplement mousePres-sEvent(), you'll get called only when part of the QScrollView is clicked — and the only such part is the "corner" (if you don't set a cornerWidget()) and the frame; everything else is covered up by the viewport, clipper or scroll bars.

When you construct a QScrollView, some of the widget flags apply to the viewport() instead of being sent to the QWidget constructor for the QScrollView. This applies to WResizeNoErase, WStaticContents, WRepaintNoErase and WPaintClever. See Qt::WidgetFlags for documentation about these flags. Here are some examples:

- An image-manipulation widget would use `WResizeNoErase|WStaticContents` because the widget draws all pixels itself, and when its size increases, it only needs a paint event for the new part because the old part remains unchanged.
- A word processing widget might use WResizeNoErase and repaint itself line by line to get a less-flickery resizing. If the widget is in a mode in which no text justification can take place, it might use WStaticContents too, so that it would only get a repaint for the newly visible parts.
- A scrolling game widget in which the background scrolls as the characters move might use WRepaintNoErase (in addition to WStaticContents and WResizeNoErase) so that the window system background does not flash in and out during scrolling.

Child widgets may be moved using addChild() or moveChild(). Use childX() and childY() to get the position of a child widget.

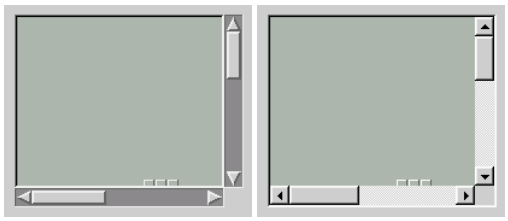
A widget may be placed in the corner between the vertical and horizontal scrollbars with setCornerWidget(). You can get access to the scrollbars using horizontalScrollBar() and verticalScrollBar(), and to the viewport with viewport(). The scroll view can be scrolled using scrollBy(), ensureVisible(), setContentsPos() or center().

The visible area is given by visibleWidth() and visibleHeight(), and the contents area by contentsWidth() and con-tentsHeight(). The contents may be repainted using one of the repaintContents() or updateContents() functions.

Coordinate conversion is provided by contentsToViewport() and viewportToContents().

The contentsMoving() signal is emitted just before the contents are moved to a new position.

Warning: WResizeNoErase is currently set by default, i.e. you always have to clear the background manually in scrollview subclasses. This will change in a future version of Qt and we recommend specifying the flag explicitly.



See also Abstract Widget Classes.

Member Type Documentation

QScrollView::ResizePolicy

This enum type is used to control a QScrollView's reaction to resize events. There are four possible settings:

- `QScrollView::Default` - the QScrollView selects one of the other settings automatically when it has to. In this version of Qt, QScrollView changes to Manual if you resize the contents with resizeContents() and to AutoOne if a child is added.
- `QScrollView::Manual` - the view stays the size set by resizeContents().
- `QScrollView::AutoOne` - if there is only one child widget the view stays the size of that widget. Otherwise the behaviour is undefined.
- `QScrollView::AutoOneFit` - if there is only one child widget the view stays the size of that widget's sizeHint(). If the scrollview is resized larger than the child's sizeHint(), the child will be resized to fit. If there is more than one child, the behaviour is undefined.

QScrollView::ScrollBarMode

This enum type describes the various modes of QScrollView's scroll bars. The defined modes are:

- `QScrollView::Auto` - QScrollView shows a scroll bar when the content is too large to fit and not otherwise. This is the default.
- `QScrollView::AlwaysOff` - QScrollView never shows a scroll bar.
- `QScrollView::AlwaysOn` - QScrollView always shows a scroll bar.

(The modes for the horizontal and vertical scroll bars are independent.)

Member Function Documentation

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

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

The widget flags WStaticContents, WRepaintNoErase and WPaintClever are propagated to the viewport() widget. The other widget flags are propagated to the parent constructor as usual.

QScrollView::~QScrollView ()

Destroys the QScrollView. Any children added with addChild() will be deleted.

void QScrollView::addChild (QWidget * child, int x = 0, int y = 0) [virtual]

Inserts the widget, *child*, into the scrolled area positioned at (*x*, *y*). The position defaults to (0, 0). If the child is already in the view, it is just moved.

You may want to call enableClipper(TRUE) if you add a large number of widgets.

Example: scrollview/scrollview.cpp.

int QScrollView::bottomMargin () const [protected]

Returns the bottom margin.

See also setMargins() [p. 273].

void QScrollView::center (int x, int y) [slot]

Scrolls the content so that the point *(x, y)* is in the center of visible area.

Example: scrollview/scrollview.cpp.

void QScrollView::center (int x, int y, float xmargin, float ymargin) [slot]

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

Scrolls the content so that the point *(x, y)* is visible with the *xmargin* and *ymargin* margins (as fractions of visible area).

For example:

- Margin 0.0 allows (x, y) to be on the edge of the visible area.
- Margin 0.5 ensures that (x, y) is in middle 50% of the visible area.
- Margin 1.0 ensures that (x, y) is in the center of the the visible area.

bool QScrollView::childIsVisible (QWidget * child)

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

Returns TRUE if *child* is visible. This is equivalent to child->isVisible().

int QScrollView::childX (QWidget * child)

Returns the X position of the given *child* widget. Use this rather than QWidget::x() for widgets added to the view.

int QScrollView::childY (QWidget * child)

Returns the Y position of the given *child* widget. Use this rather than QWidget::y() for widgets added to the view.

QWidget * QScrollView::clipper () const

Returns the clipper widget. Contents in the scrollview are ultimately clipped to be inside the clipper widget.

You should not need to use this function.

See also visibleWidth [p. 277] and visibleHeight [p. 276].

void QScrollView::contentsContextMenuEvent (QContextMenuEvent * e) [virtual protected]

This event handler is called whenever the QScrollView receives a contextMenuEvent() in *e* - the mouse position is translated to be a point on the contents.

void QScrollView::contentsDragEnterEvent (QDragEnterEvent *) [virtual protected]

This event handler is called whenever the QScrollView receives a dragEnterEvent() - the drag position is translated to be a point on the contents.

Example: dirview/dirview.cpp.

Reimplemented in QTable.

void QScrollView::contentsDragLeaveEvent (QDragLeaveEvent *) [virtual protected]

This event handler is called whenever the QScrollView receives a dragLeaveEvent() - the drag position is translated to be a point on the contents.

Example: dirview/dirview.cpp.

Reimplemented in QTable.

void QScrollView::contentsDragMoveEvent (QDragMoveEvent *) [virtual protected]

This event handler is called whenever the QScrollView receives a dragMoveEvent() - the drag position is translated to be a point on the contents.

Example: dirview/dirview.cpp.

Reimplemented in QTable.

void QScrollView::contentsDropEvent (QDropEvent *) [virtual protected]

This event handler is called whenever the QScrollView receives a dropEvent() - the drop position is translated to be a point on the contents.

Example: dirview/dirview.cpp.

Reimplemented in QTable.

int QScrollView::contentsHeight () const

Returns the height of the contents area. See the "contentsHeight" [p. 275] property for details.

void QScrollView::contentsMouseDoubleClickEvent (QMouseEvent *) [virtual protected]

This event handler is called whenever the QScrollView receives a mouseDoubleClickEvent() - the click position is translated to be a point on the contents.

Reimplemented in QListView.

void QScrollView::contentsMouseMoveEvent (QMouseEvent *) [virtual protected]

This event handler is called whenever the QScrollView receives a mouseMoveEvent() - the mouse position is translated to be a point on the contents.

Reimplemented in QListView.

void QScrollView::contentsMousePressEvent (QMouseEvent *) [virtual protected]

This event handler is called whenever the QScrollView receives a mousePressEvent() - the press position is translated to be a point on the contents.

Reimplemented in QListView.

void QScrollView::contentsMouseReleaseEvent (QMouseEvent *) [virtual protected]

This event handler is called whenever the QScrollView receives a mouseReleaseEvent() - the release position is translated to be a point on the contents.

Reimplemented in QListView.

void QScrollView::contentsMoving (int x, int y) [signal]

This signal is emitted just before the contents are moved to position *(x, y)*.

See also contentsX [p. 276] and contentsY [p. 276].

void QScrollView::contentsToViewport (int x, int y, int & vx, int & vy) const

Translates a point (*x*, *y*) in the contents to a point (*vx*, *vy*) on the viewport() widget.

QPoint QScrollView::contentsToViewport (const QPoint & p) const

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

Returns the point *p* translated to a point on the viewport() widget.

void QScrollView::contentsWheelEvent (QWheelEvent * e) [virtual protected]

This event handler is called whenever the QScrollView receives a wheelEvent() in *e* - the mouse position is translated to be a point on the contents.

int QScrollView::contentsWidth () const

Returns the width of the contents area. See the "contentsWidth" [p. 275] property for details.

int QScrollView::contentsX () const

Returns the X coordinate of the contents that are at the left edge of the viewport. See the "contentsX" [p. 276] property for details.

int QScrollView::contentsY () const

Returns the Y coordinate of the contents that are at the top edge of the viewport. See the "contentsY" [p. 276] property for details.

QWidget * QScrollView::cornerWidget () const

Returns the widget in the corner between the two scroll bars.

By default, no corner widget is present.

Example: scrollview/scrollview.cpp.

bool QScrollView::dragAutoScroll () const

Returns TRUE if autoscrolling in drag move events is enabled; otherwise returns FALSE. See the "dragAutoScroll" [p. 276] property for details.

void QScrollView::drawContents (QPainter * p, int clipx, int clipy, int clipw, int cliph) [virtual protected]

Reimplement this function if you are viewing a drawing area rather than a widget.

The function should draw the rectangle (*clipx*, *clipy*, *clipw*, *cliph*) of the contents using painter *p*. The clip rectangle is in the scrollview's coordinates.

For example:

```
{
  // Fill a 40000 by 50000 rectangle at (100000,150000)

  // Calculate the coordinates...
  int x1 = 100000, y1 = 150000;
  int x2 = x1+40000-1, y2 = y1+50000-1;

  // Clip the coordinates so X/Windows will not have problems...
  if (x1 < clipx) x1=clipx;
  if (y1  clipx+clipw-1) x2=clipx+clipw-1;
  if (y2 > clipy+cliph-1) y2=clipy+cliph-1;

  // Paint using the small coordinates...
  if ( x2 >= x1 && y2 >= y1 )
      p->fillRect(x1, y1, x2-x1+1, y2-y1+1, red);
}
```

The clip rectangle and translation of the painter *p* is already set appropriately.

Example: qdir/qdir.cpp.

Reimplemented in QCanvasView and QTable.

void QScrollView::drawContentsOffset (QPainter * p, int offsetx, int offsety, int clipx, int clipy, int clipw, int cliph) [virtual protected]

For backward-compatibility only. It is easier to use drawContents(QPainter*,int,int,int,int).

The default implementation translates the painter appropriately and calls drawContents(QPainter*,int,int,int,int). See drawContents for an explanation of the parameters *p*, *offsetx*, *offsety*, *clipx*, *clipy*, *clipw* and *cliph*.

Reimplemented in QListView.

void QScrollView::enableClipper (bool y)

When a large numbers of child widgets are in a scrollview, especially if they are close together, the scrolling performance can suffer greatly. If *y* is TRUE the scrollview will use an extra widget to group child widgets.

Note that you may only call enableClipper() prior to adding widgets.

For a full discussion, see this class's detailed description.

Example: scrollview/scrollview.cpp.

void QScrollView::ensureVisible (int x, int y) [slot]

Scrolls the content so that the point *(x, y)* is visible with at least 50-pixel margins (if possible, otherwise centered).

void QScrollView::ensureVisible (int x, int y, int xmargin, int ymargin) [slot]

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

Scrolls the content so that the point *(x, y)* is visible with at least the *xmargin* and *ymargin* margins (if possible, otherwise centered).

bool QScrollView::eventFilter (QObject * obj, QEvent * e) [virtual protected]

This event filter ensures the scroll bars are updated when a single contents widget is resized, shown, hidden or destroyed; it passes mouse events to the QScrollView. The event is in *e* and the object is in *obj*.

Reimplemented from QObject [Additional Functionality with Qt].

Reimplemented in QListView.

ScrollBarMode QScrollView::hScrollBarMode () const

Returns the mode for the horizontal scroll bar. See the "hScrollBarMode" [p. 276] property for details.

bool QScrollView::hasStaticBackground () const

Returns TRUE if QScrollView uses a static background; otherwise returns FALSE.

See also setStaticBackground() [p. 273].

QScrollBar * QScrollView::horizontalScrollBar () const

Returns the component horizontal scroll bar. It is made available to allow accelerators, autoscrolling, etc. and to allow changing arrow scroll rates, e.g. bar->setSteps(rate, bar->pageStep()).

It should not be otherwise manipulated.

This function never returns 0.

int QScrollView::leftMargin () const [protected]

Returns the left margin.

See also setMargins() [p. 273].

void QScrollView::moveChild (QWidget * child, int x, int y) [virtual]

Repositions the *child* widget to (*x*, *y*). This function is the same as addChild().

void QScrollView::removeChild (QWidget * child)

Removes the *child* widget from the scrolled area. Note that this happens automatically if the *child* is deleted.

void QScrollView::repaintContents (int x, int y, int w, int h, bool erase = TRUE)

Calls repaint() on a rectangle defined by *x*, *y*, *w*, *h*, translated appropriately. If the rectangle in not visible, nothing is repainted. If *erase* is TRUE the background is cleared using the background color.

See also updateContents() [p. 274].

void QScrollView::repaintContents (const QRect & r, bool erase = TRUE)

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

Repaints the contents of rectangle *r*. If *erase* is TRUE the background is cleared using the background color.

void QScrollView::repaintContents (bool erase = TRUE)

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

Repaints the contents. If *erase* is TRUE the background is cleared using the background color.

void QScrollView::resizeContents (int w, int h) [virtual slot]

Sets the size of the contents area to *w* pixels wide and *h* pixels high and updates the viewport accordingly.

ResizePolicy QScrollView::resizePolicy () const

Returns the current resize policy. See the "resizePolicy" [p. 276] property for details.

int QScrollView::rightMargin () const [protected]

Returns the right margin.

See also setMargins() [p. 273].

void QScrollView::scrollBy (int dx, int dy) [slot]

Scrolls the content by *dx* to the left and *dy* upwards.

void QScrollView::setContentsPos (int x, int y) [virtual slot]

Scrolls the content so that the point *(x, y)* is in the top-left corner.

Example: process/process.cpp.

void QScrollView::setCornerWidget (QWidget * corner) [virtual]

Sets the widget in the *corner* between the two scroll bars.

You will probably also want to set at least one of the scroll bar modes to AlwaysOn.

Passing 0 shows no widget in the corner.

Any previous *corner* widget is hidden.

You may call setCornerWidget() with the same widget at different times.

All widgets set here will be deleted by the QScrollView when it is destroyed unless you separately reparent the widget after setting some other corner widget (or 0).

Any *newly* set widget should have no current parent.

By default, no corner widget is present.

See also vScrollBarMode [p. 276] and hScrollBarMode [p. 276].

Example: scrollview/scrollview.cpp.

void QScrollView::setDragAutoScroll (bool b) [virtual]

Sets whether autoscrolling in drag move events is enabled to *b*. See the "dragAutoScroll" [p. 276] property for details.

void QScrollView::setHBarGeometry (QScrollBar & hbar, int x, int y, int w, int h) [virtual protected]

Called when the horizontal scroll bar geometry changes. This is provided as a protected function so that subclasses can do interesting things such as providing extra buttons in some of the space normally used by the scroll bars.

The default implementation simply gives all the space to *hbar*. The new geometry is given by *x*, *y*, *w* and *h*.

See also setVBarGeometry() [p. 273].

void QScrollView::setHScrollBarMode (ScrollBarMode) [virtual]

Sets the mode for the horizontal scroll bar. See the "hScrollBarMode" [p. 276] property for details.

void QScrollView::setMargins (int left, int top, int right, int bottom) [virtual protected]

Sets the margins around the scrolling area to *left*, *top*, *right* and *bottom*. This is useful for applications such as spreadsheets with "locked" rows and columns. The marginal space is *inside* the frameRect() and is left blank; reimplement drawContents() or put widgets in the unused area.

By default all margins are zero.

See also frameChanged() [p. 68].

void QScrollView::setResizePolicy (ResizePolicy) [virtual]

Sets the current resize policy. See the "resizePolicy" [p. 276] property for details.

void QScrollView::setStaticBackground (bool y)

Sets the scrollview to have a static background if *y* is TRUE, or a scrolling background if *y* is FALSE. By default, the background is scrolling.

Be aware that this mode is quite slow, as a full repaint of the visible area has to be triggered on every contents move.

See also hasStaticBackground() [p. 270].

void QScrollView::setVBarGeometry (QScrollBar & vbar, int x, int y, int w, int h) [virtual protected]

Called when the vertical scroll bar geometry changes. This is provided as a protected function so that subclasses can do interesting things such as providing extra buttons in some of the space normally used by the scroll bars.

The default implementation simply gives all the space to *vbar*. The new geometry is given by *x*, *y*, *w* and *h*.

See also setHBarGeometry() [p. 272].

void QScrollView::setVScrollBarMode (ScrollBarMode) [virtual]

Sets the mode for the vertical scroll bar. See the "vScrollBarMode" [p. 276] property for details.

void QScrollView::showChild (QWidget * child, bool y = TRUE)

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

Sets the visibility of *child*. Equivalent to QWidget::show() or QWidget::hide().

int QScrollView::topMargin () const [protected]

Returns the top margin.

See also setMargins() [p. 273].

void QScrollView::updateContents (int x, int y, int w, int h)

Calls update() on a rectangle defined by *x*, *y*, *w*, *h*, translated appropriately. If the rectangle is not visible, nothing is repainted.

See also repaintContents() [p. 271].

void QScrollView::updateContents (const QRect & r)

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

Updates the contents in rectangle *r*

void QScrollView::updateContents ()

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

void QScrollView::updateScrollBars () [slot]

Updates scroll bars - all possibilities considered. You should never need to call this in your code.

ScrollBarMode QScrollView::vScrollBarMode () const

Returns the mode for the vertical scroll bar. See the "vScrollBarMode" [p. 276] property for details.

QScrollBar * QScrollView::verticalScrollBar () const

Returns the component vertical scroll bar. It is made available to allow accelerators, autoscrolling, etc. and to allow changing arrow scroll rates, e.g. bar->setSteps(rate, bar->pageStep()).

It should not be otherwise manipulated.

This function never returns 0.

QWidget * QScrollView::viewport () const

Returns the viewport widget of the scrollview. This is the widget containing the contents widget or which is the drawing area.

Example: scrollview/scrollview.cpp.

void QScrollView::viewportPaintEvent (QPaintEvent * pe) [virtual protected]

This is a low-level painting routine that draws the viewport contents. Reimplement this if drawContents() is too high-level (for example, if you don't want to open a QPainter on the viewport). The paint event is passed in *pe*.

void QScrollView::viewportResizeEvent (QResizeEvent *) [virtual protected]

To provide simple processing of events on the contents, this function receives all resize events sent to the viewport.

See also QWidget::resizeEvent() [p. 445].

QSize QScrollView::viewportSize (int x, int y) const

Returns the viewport size for size (*x*, *y*).

The viewport size depends on *(x, y)* (the size of the contents), the size of this widget and the modes of the horizontal and vertical scroll bars.

This function permits widgets that can trade vertical and horizontal space for each other to control scroll bar appearance better. For example, a word processor or web browser can control the width of the right margin accurately, whether or not there needs to be a vertical scroll bar.

void QScrollView::viewportToContents (int vx, int vy, int & x, int & y) const

Translates a point (*vx*, *vy*) on the viewport() widget to a point (*x*, *y*) in the contents.

QPoint QScrollView::viewportToContents (const QPoint & vp) const

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

Returns the point on the viewport *vp* translated to a point in the contents.

int QScrollView::visibleHeight () const

Returns the vertical amount of the content that is visible. See the "visibleHeight" [p. 276] property for details.

int QScrollView::visibleWidth () const

Returns the horizontal amount of the content that is visible. See the "visibleWidth" [p. 277] property for details.

Property Documentation

int contentsHeight

This property holds the height of the contents area.

Get this property's value with contentsHeight().

int contentsWidth

This property holds the width of the contents area.

Get this property's value with contentsWidth().

int contentsX

This property holds the X coordinate of the contents that are at the left edge of the viewport.

Get this property's value with contentsX().

int contentsY

This property holds the Y coordinate of the contents that are at the top edge of the viewport.

Get this property's value with contentsY().

bool dragAutoScroll

This property holds whether autoscrolling in drag move events is enabled.

If this property is set to TRUE (the default), the QScrollView automatically scrolls the contents in drag move events if the user moves the cursor close to a border of the view. Of course this works only if the viewport accepts drops. Specifying FALSE disables this autoscroll feature.

Set this property's value with setDragAutoScroll() and get this property's value with dragAutoScroll().

ScrollBarMode hScrollBarMode

This property holds the mode for the horizontal scroll bar.

The default mode is QScrollView::Auto.

See also vScrollBarMode [p. 276].

Set this property's value with setHScrollBarMode() and get this property's value with hScrollBarMode().

ResizePolicy resizePolicy

This property holds the current resize policy.

The default is Default.

See also ResizePolicy [p. 265].

Set this property's value with setResizePolicy() and get this property's value with resizePolicy().

ScrollBarMode vScrollBarMode

This property holds the mode for the vertical scroll bar.

The default mode is QScrollView::Auto.

See also hScrollBarMode [p. 276].

Set this property's value with setVScrollBarMode() and get this property's value with vScrollBarMode().

int visibleHeight

This property holds the vertical amount of the content that is visible.

Get this property's value with visibleHeight().

int visibleWidth

This property holds the horizontal amount of the content that is visible.

Get this property's value with visibleWidth().

QSizeGrip Class Reference

The QSizeGrip class provides corner-grip for resizing a top-level window.

`#include <qsizegrip.h>`

Inherits QWidget [p. 412].

Public Members

- **QSizeGrip** (QWidget * parent, const char * name = 0)
- **~QSizeGrip** ()
- virtual QSize **sizeHint** () const

Protected Members

- virtual void **paintEvent** (QPaintEvent * e)
- virtual void **mousePressEvent** (QMouseEvent * e)
- virtual void **mouseMoveEvent** (QMouseEvent * e)

Detailed Description

The QSizeGrip class provides corner-grip for resizing a top-level window.

This widget works like the standard Windows resize handle. In the X11 version this resize handle generally works differently from the one provided by the system; we hope to reduce this difference in the future.

Put this widget anywhere in a tree and the user can use it to resize the top-level window. Generally, this should be in the lower right-hand corner. Note that QStatusBar already uses this widget, so if you have a status bar (e.g. you are using QMainWindow), then you don't need to use this widget explicitly.

See also QStatusBar [p. 311], Widget Appearance and Style, Main Window and Related Classes and Basic Widgets.

Member Function Documentation

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

Constructs a resize corner as a child widget of *parent* with the name *name*.

QSizeGrip::~QSizeGrip ()

Destroys the size grip.

void QSizeGrip::mouseMoveEvent (QMouseEvent * e) [virtual protected]

Resizes the top-level widget containing this widget. The event is in *e*.

Reimplemented from QWidget [p. 439].

void QSizeGrip::mousePressEvent (QMouseEvent * e) [virtual protected]

Primes the resize operation. The event is in *e*.

Reimplemented from QWidget [p. 440].

void QSizeGrip::paintEvent (QPaintEvent * e) [virtual protected]

Paints the resize grip. Resize grips are usually rendered as small diagonal textured lines in the lower-right corner. The event is in *e*.

Reimplemented from QWidget [p. 441].

QSize QSizeGrip::sizeHint () const [virtual]

Returns the size grip's size hint; this is a small size.

Reimplemented from QWidget [p. 454].

QSizePolicy Class Reference

The QSizePolicy class is a layout attribute describing horizontal and vertical resizing.

`#include <qsizepolicy.h>`

Public Members

- enum **SizeType** { Fixed = 0, Minimum = MayGrow, Maximum = MayShrink, Preferred = MayGrow | MayShrink, MinimumExpanding = MayGrow | ExpMask, Expanding = MayGrow | MayShrink | ExpMask, Ignored = ExpMask }
- enum **ExpandData** { NoDirection = 0, Horizontally = 1, Vertically = 2, BothDirections = Horizontally | Vertically, Horizontal = Horizontally, Vertical = Vertically }
- **QSizePolicy** ()
- **QSizePolicy** (SizeType hor, SizeType ver, bool hfw = FALSE)
- **QSizePolicy** (SizeType hor, SizeType ver, uchar horStretch, uchar verStretch, bool hfw = FALSE)
- SizeType **horData** () const
- SizeType **verData** () const
- bool **mayShrinkHorizontally** () const
- bool **mayShrinkVertically** () const
- bool **mayGrowHorizontally** () const
- bool **mayGrowVertically** () const
- ExpandData **expanding** () const
- void **setHorData** (SizeType d)
- void **setVerData** (SizeType d)
- void **setHeightForWidth** (bool b)
- bool **hasHeightForWidth** () const
- bool **operator==** (const QSizePolicy & s) const
- bool **operator!=** (const QSizePolicy & s) const
- uint **horStretch** () const
- uint **verStretch** () const
- void **setHorStretch** (uchar sf)
- void **setVerStretch** (uchar sf)

Detailed Description

The QSizePolicy class is a layout attribute describing horizontal and vertical resizing.

The size policy of a widget is an expression of its willingness to be resized in various ways.

Widgets that reimplement QWidget::sizePolicy() return a QSizePolicy describing the horizontal and vertical resizing policy best used when laying out the widget. Only one of the constructors is of interest in most applications.

QSizePolicy contains two independent SizeType objects; one describes the widgets's horizontal size policy, and the other describes its vertical size policy. It also contains a flag to indicate whether the height and width of its preferred size are related.

The per-dimension SizeType objects are set in the usual constructor and can be queried using a variety of functions, none of which are really interesting to application programmers.

The hasHeightForWidth() flag indicates whether the widget's sizeHint() is width-dependent (such as a word-wrapping label).

See also QSizePolicy::SizeType [p. 281], Widget Appearance and Style and Layout Management.

Member Type Documentation

QSizePolicy::ExpandData

This enum type describes in which directions a widget can make use of extra space. There are four possible values:

- `QSizePolicy::NoDirection` - the widget cannot make use of extra space in any direction.
- `QSizePolicy::Horizontally` - the widget can usefully be wider than the sizeHint().
- `QSizePolicy::Vertically` - the widget can usefully be taller than the sizeHint().
- `QSizePolicy::BothDirections` - the widget can usefully be both wider and taller than the sizeHint().

QSizePolicy::SizeType

The per-dimension sizing types used when constructing a QSizePolicy are:

- `QSizePolicy::Fixed` - the QWidget::sizeHint() is the only acceptable alternative, so the widget can never grow or shrink (e.g. the vertical direction of a push button).
- `QSizePolicy::Minimum` - the sizeHint() is minimal, and sufficient. The widget can be expanded, but there is no advantage to it being larger (e.g. the horizontal direction of a push button).
- `QSizePolicy::Maximum` - the sizeHint() is a maximum. The widget can be shrunk any amount without detriment if other widgets need the space (e.g. a separator line).
- `QSizePolicy::Preferred` - the sizeHint() is best, but the widget can be shrunk and still be useful. The widget can be expanded, but there is no advantage to it being larger than sizeHint() (the default QWidget policy).
- `QSizePolicy::Expanding` - the sizeHint() is a sensible size, but the widget can be shrunk and still be useful. The widget can make use of extra space, so it should get as much space as possible (e.g. the horizontal direction of a slider).
- `QSizePolicy::MinimumExpanding` - the sizeHint() is minimal, and sufficient. The widget can make use of extra space, so it should get as much space as possible (e.g. the horizontal direction of a slider).
- `QSizePolicy::Ignored` - the sizeHint() is ignored. The widget will get as much space as possible.

In any case, QLayout never shrinks a widget below the QWidget::minimumSizeHint().

Member Function Documentation

QSizePolicy::QSizePolicy ()

Default constructor; produces a minimally initialized QSizePolicy.

QSizePolicy::QSizePolicy (SizeType hor, SizeType ver, bool hfw = FALSE)

This is the constructor normally used to return a value in the overridden QWidget::sizePolicy() function of a QWidget subclass.

It constructs a QSizePolicy with independent horizontal and vertical sizing types, *hor* and *ver* respectively. These sizing types affect how the widget is treated by the layout engine.

If *hfw* is TRUE, the preferred height of the widget is dependent on the width of the widget (for example, a QLabel with line wrapping).

See also horData() [p. 282], verData() [p. 284] and hasHeightForWidth() [p. 282].

QSizePolicy::QSizePolicy (SizeType hor, SizeType ver, uchar horStretch, uchar verStretch, bool hfw = FALSE)

Constructs a QSizePolicy with independent horizontal and vertical sizing types *hor* and *ver*, and stretch factors *horStretch* and *verStretch*.

If *hfw* is TRUE, the preferred height of the widget is dependent on the width of the widget.

See also horStretch() [p. 282] and verStretch() [p. 284].

ExpandData QSizePolicy::expanding () const

Returns a value indicating whether the widget can make use of extra space (i.e. if it "wants" to grow) horizontally and/or vertically.

See also mayShrinkHorizontally() [p. 283], mayGrowHorizontally() [p. 282], mayShrinkVertically() [p. 283] and mayGrowVertically() [p. 283].

bool QSizePolicy::hasHeightForWidth () const

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

See also setHeightForWidth() [p. 283].

SizeType QSizePolicy::horData () const

Returns the horizontal component of the size policy.

See also setHorData() [p. 283], verData() [p. 284] and horStretch() [p. 282].

uint QSizePolicy::horStretch () const

Returns the horizontal stretch factor of the size policy.

See also setHorStretch() [p. 283] and verStretch() [p. 284].

bool QSizePolicy::mayGrowHorizontally () const

Returns TRUE if the widget can sensibly be wider than its sizeHint(); otherwise returns FALSE.

See also mayGrowVertically() [p. 283] and mayShrinkHorizontally() [p. 283].

bool QSizePolicy::mayGrowVertically () const

Returns TRUE if the widget can sensibly be taller than its sizeHint(); otherwise returns FALSE.

See also mayGrowHorizontally() [p. 282] and mayShrinkVertically() [p. 283].

bool QSizePolicy::mayShrinkHorizontally () const

Returns TRUE if the widget can sensibly be narrower than its sizeHint(); otherwise returns FALSE.

See also mayShrinkVertically() [p. 283] and mayGrowHorizontally() [p. 282].

bool QSizePolicy::mayShrinkVertically () const

Returns TRUE if the widget can sensibly be shorter than its sizeHint(); otherwise returns FALSE.

See also mayShrinkHorizontally() [p. 283] and mayGrowVertically() [p. 283].

bool QSizePolicy::operator!= (const QSizePolicy & s) const

Returns TRUE if this policy is different from *s*; otherwise returns FALSE.

See also operator==() [p. 283].

bool QSizePolicy::operator== (const QSizePolicy & s) const

Returns TRUE if this policy is equal to *s*; otherwise returns FALSE.

See also operator!=() [p. 283].

void QSizePolicy::setHeightForWidth (bool b)

Sets the hasHeightForWidth() flag to *b*.

See also hasHeightForWidth() [p. 282].

void QSizePolicy::setHorData (SizeType d)

Sets the horizontal component of the size policy to size type *d*.

See also horData() [p. 282] and setVerData() [p. 283].

void QSizePolicy::setHorStretch (uchar sf)

Sets the horizontal stretch factor of the size policy to *sf*.

See also horStretch() [p. 282] and setVerStretch() [p. 284].

void QSizePolicy::setVerData (SizeType d)

Sets the vertical component of the size policy to size type *d*.

See also verData() [p. 284] and setHorData() [p. 283].

void QSizePolicy::setVerStretch (uchar sf)

Sets the vertical stretch factor of the size policy to *sf*.

See also verStretch() [p. 284] and setHorStretch() [p. 283].

SizeType QSizePolicy::verData () const

Returns the vertical component of the size policy.

See also setVerData() [p. 283], horData() [p. 282] and verStretch() [p. 284].

uint QSizePolicy::verStretch () const

Returns the vertical stretch factor of the size policy.

See also setVerStretch() [p. 284] and horStretch() [p. 282].

QSlider Class Reference

The QSlider widget provides a vertical or horizontal slider.

`#include <qslider.h>`

Inherits QWidget [p. 412] and QRangeControl [p. 246].

Public Members

- enum **TickSetting** { NoMarks = 0, Above = 1, Left = Above, Below = 2, Right = Below, Both = 3 }
- **QSlider** (QWidget * parent, const char * name = 0)
- **QSlider** (Orientation orientation, QWidget * parent, const char * name = 0)
- **QSlider** (int minValue, int maxValue, int pageStep, int value, Orientation orientation, QWidget * parent, const char * name = 0)
- virtual void **setOrientation** (Orientation)
- Orientation **orientation** () const
- virtual void **setTracking** (bool enable)
- bool **tracking** () const
- virtual void **setPalette** (const QPalette & p)
- int **sliderStart** () const
- QRect **sliderRect** () const
- virtual void **setTickmarks** (TickSetting)
- TickSetting **tickmarks** () const
- virtual void **setTickInterval** (int)
- int **tickInterval** () const
- int **minValue** () const
- int **maxValue** () const
- void **setMinValue** (int)
- void **setMaxValue** (int)
- int **lineStep** () const
- int **pageStep** () const
- void **setLineStep** (int)
- void **setPageStep** (int)
- int **value** () const

Public Slots

- virtual void **setValue** (int)
- void **addStep** ()
- void **subtractStep** ()

Signals

- void **valueChanged** (int value)
- void **sliderPressed** ()
- void **sliderMoved** (int value)
- void **sliderReleased** ()

Properties

- int **lineStep** — the current line step
- int **maxValue** — the current maximum value of the slider
- int **minValue** — the current minimum value of the slider
- Orientation **orientation** — the orientation of the slider
- int **pageStep** — the current page step
- int **tickInterval** — the interval between tickmarks
- TickSetting **tickmarks** — the tickmark settings for this slider
- bool **tracking** — whether slider tracking is enabled
- int **value** — the current slider value

Protected Members

- virtual void **valueChange** ()
- virtual void **rangeChange** ()

Detailed Description

The QSlider widget provides a vertical or horizontal slider.

The slider is the classic widget for controlling a bounded value. It lets the user move a slider along a horizontal or vertical groove and translates the slider's position into an integer value within the legal range.

QSlider inherits QRangeControl, which provides the "integer" side of the slider. setRange() and value() are likely to be used by practically all slider users; see the QRangeControl documentation for information about the many other functions that class provides.

The main functions offered by the slider itself are tickmark and orientation control; you can use setTickmarks() to indicate where you want the tickmarks to be, setTickInterval() to indicate how many of them you want and setOrientation() to indicate whether the slider is to be horizontal or vertical.

A slider has a default focusPolicy() of `WeakWheelFocus`, i.e. it accepts focus on Tab and uses the mouse wheel and a suitable keyboard interface.

See also QScrollBar [p. 252], QSpinBox [p. 295], GUI Design Handbook: Slider and Basic Widgets.

Member Type Documentation

QSlider::TickSetting

This enum specifies where the tickmarks are to be drawn relative to the slider's groove and the handle the user moves. The possible values are:

- `QSlider::NoMarks` - do not draw any tickmarks.
- `QSlider::Both` - draw tickmarks on both sides of the groove.
- `QSlider::Above` - draw tickmarks above the (horizontal) slider
- `QSlider::Below` - draw tickmarks below the (horizontal) slider
- `QSlider::Left` - draw tickmarks to the left of the (vertical) slider
- `QSlider::Right` - draw tickmarks to the right of the (vertical) slider

Member Function Documentation

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

Constructs a vertical slider.

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

QSlider::QSlider (Orientation orientation, QWidget * parent, const char * name = 0)

Constructs a slider.

The *orientation* must be Qt::Vertical or Qt::Horizontal.

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

QSlider::QSlider (int minValue, int maxValue, int pageStep, int value, Orientation orientation, QWidget * parent, const char * name = 0)

Constructs a slider whose value can never be smaller than *minValue* or greater than *maxValue*, whose page step size is *pageStep* and whose value is initially *value* (which is guaranteed to be in range using bound()).

If *orientation* is Qt::Vertical the slider is vertical and if it is Qt::Horizontal the slider is horizontal.

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

void QSlider::addStep () [slot]

Moves the slider one pageStep() up or right.

int QSlider::lineStep () const

Returns the current line step. See the "lineStep" [p. 290] property for details.

int QSlider::maxValue () const

Returns the current maximum value of the slider. See the "maxValue" [p. 290] property for details.

int QSlider::minValue () const

Returns the current minimum value of the slider. See the "minValue" [p. 290] property for details.

Orientation QSlider::orientation () const

Returns the orientation of the slider. See the "orientation" [p. 291] property for details.

int QSlider::pageStep () const

Returns the current page step. See the "pageStep" [p. 291] property for details.

void QSlider::rangeChange () [virtual protected]

Implements the virtual QRangeControl function.

Reimplemented from QRangeControl [p. 249].

void QSlider::setLineStep (int)

Sets the current line step. See the "lineStep" [p. 290] property for details.

void QSlider::setMaxValue (int)

Sets the current maximum value of the slider. See the "maxValue" [p. 290] property for details.

void QSlider::setMinValue (int)

Sets the current minimum value of the slider. See the "minValue" [p. 290] property for details.

void QSlider::setOrientation (Orientation) [virtual]

Sets the orientation of the slider. See the "orientation" [p. 291] property for details.

void QSlider::setPageStep (int)

Sets the current page step. See the "pageStep" [p. 291] property for details.

void QSlider::setPalette (const QPalette & p) [virtual]

Reimplements the virtual function QWidget::setPalette().

Sets the background color to the mid color for Motif style sliders using palette *p*.

Reimplemented from QWidget [p. 451].

void QSlider::setTickInterval (int) [virtual]

Sets the interval between tickmarks. See the "tickInterval" [p. 291] property for details.

void QSlider::setTickmarks (TickSetting) [virtual]

Sets the tickmark settings for this slider. See the "tickmarks" [p. 291] property for details.

void QSlider::setTracking (bool enable) [virtual]

Sets whether slider tracking is enabled to *enable*. See the "tracking" [p. 291] property for details.

void QSlider::setValue (int) [virtual slot]

Sets the current slider value. See the "value" [p. 292] property for details.

void QSlider::sliderMoved (int value) [signal]

This signal is emitted when the slider is dragged, with the new slider *value* as an argument.

void QSlider::sliderPressed () [signal]

This signal is emitted when the user presses the slider with the mouse.

QRect QSlider::sliderRect () const

Returns the slider handle rectangle. (This is the visual marker that the user can move.)

void QSlider::sliderReleased () [signal]

This signal is emitted when the user releases the slider with the mouse.

int QSlider::sliderStart () const

Returns the start position of the slider.

void QSlider::subtractStep () [slot]

Moves the slider one pageStep() down or left.

int QSlider::tickInterval () const

Returns the interval between tickmarks. See the "tickInterval" [p. 291] property for details.

TickSetting QSlider::tickmarks () const

Returns the tickmark settings for this slider. See the "tickmarks" [p. 291] property for details.

bool QSlider::tracking () const

Returns TRUE if slider tracking is enabled; otherwise returns FALSE. See the "tracking" [p. 291] property for details.

int QSlider::value () const

Returns the current slider value. See the "value" [p. 292] property for details.

void QSlider::valueChange () [virtual protected]

Implements the virtual QRangeControl function.

Reimplemented from QRangeControl [p. 251].

void QSlider::valueChanged (int value) [signal]

This signal is emitted when the slider value is changed, with the new slider *value* as an argument.

Examples: rangecontrols/rangecontrols.cpp, t12/lcdrange.cpp, t5/main.cpp, t6/main.cpp, t7/lcdrange.cpp and xform/xform.cpp.

Property Documentation

int lineStep

This property holds the current line step.

When setting lineStep, the virtual stepChange() function will be called if the new line step is different from the previous setting.

See also setSteps() [p. 250], QRangeControl::pageStep() [p. 249] and setRange() [p. 250].

Set this property's value with setLineStep() and get this property's value with lineStep().

int maxValue

This property holds the current maximum value of the slider.

When setting this property, the QSlider::minValue is adjusted, if necessary, to ensure that the range remains valid.

See also setRange() [p. 250].

Set this property's value with setMaxValue() and get this property's value with maxValue().

int minValue

This property holds the current minimum value of the slider.

When setting this property, the QSlider::maxValue is adjusted, if necessary, to ensure that the range remains valid.

See also setRange() [p. 250].

Set this property's value with setMinValue() and get this property's value with minValue().

Orientation orientation

This property holds the orientation of the slider.

The orientation must be Qt::Vertical (the default) or Qt::Horizontal.

Set this property's value with setOrientation() and get this property's value with orientation().

int pageStep

This property holds the current page step.

When setting pageStep, the virtual stepChange() function will be called if the new page step is different from the previous setting.

See also QRangeControl::setSteps() [p. 250], lineStep [p. 290] and setRange() [p. 250].

Set this property's value with setPageStep() and get this property's value with pageStep().

int tickInterval

This property holds the interval between tickmarks.

This is a value interval, not a pixel interval. If it is 0, the slider will choose between lineStep() and pageStep(). The initial value of tickInterval is 0.

See also QRangeControl::lineStep() [p. 248] and QRangeControl::pageStep() [p. 249].

Set this property's value with setTickInterval() and get this property's value with tickInterval().

TickSetting tickmarks

This property holds the tickmark settings for this slider.

The valid values are in QSlider::TickSetting. The default is NoMarks.

See also tickInterval [p. 291].

Set this property's value with setTickmarks() and get this property's value with tickmarks().

bool tracking

This property holds whether slider tracking is enabled.

If tracking is enabled (the default), the slider emits the valueChanged() signal whenever the slider is being dragged. If tracking is disabled, the slider emits the valueChanged() signal when the user releases the mouse button (unless the value happens to be the same as before).

Set this property's value with setTracking() and get this property's value with tracking().

int value

This property holds the current slider value.

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

See also QRangeControl::value() [p. 251] and prevValue() [p. 249].

QSpacerItem Class Reference

The QSpacerItem class provides blank space in a layout.

`#include <qlayout.h>`

Inherits QLayoutItem [Events, Actions, Layouts and Styles with Qt].

Public Members

- **QSpacerItem** (int w, int h, QSizePolicy::SizeType hData = QSizePolicy::Minimum, QSizePolicy::SizeType vData = QSizePolicy::Minimum)
- void **changeSize** (int w, int h, QSizePolicy::SizeType hData = QSizePolicy::Minimum, QSizePolicy::SizeType vData = QSizePolicy::Minimum)
- virtual QSize **sizeHint** () const
- virtual QSize **minimumSize** () const
- virtual QSize **maximumSize** () const
- virtual QSizePolicy::ExpandData **expanding** () const
- virtual bool **isEmpty** () const
- virtual void **setGeometry** (const QRect & r)

Detailed Description

The QSpacerItem class provides blank space in a layout.

This class is used by custom layouts.

See also QLayout [Events, Actions, Layouts and Styles with Qt], Widget Appearance and Style and Layout Management.

Member Function Documentation

QSpacerItem::QSpacerItem (int w, int h, QSizePolicy::SizeType hData = QSizePolicy::Minimum, QSizePolicy::SizeType vData = QSizePolicy::Minimum)

Constructs a spacer item with preferred width *w*, preferred height *h*, horizontal size policy *hData* and vertical size policy *vData*.

The default values provide a gap that is able to stretch if nothing else wants the space.

void QSpacerItem::changeSize (int w, int h, QSizePolicy::SizeType hData = QSizePolicy::Minimum, QSizePolicy::SizeType vData = QSizePolicy::Minimum)

Changes this spacer item to have preferred width *w*, preferred height *h*, horizontal size policy *hData* and vertical size policy *vData*.

The default values provide a gap that is able to stretch if nothing else wants the space.

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

Returns TRUE if this spacer item is expanding; otherwise returns FALSE.

Reimplemented from QLayoutItem [Events, Actions, Layouts and Styles with Qt].

bool QSpacerItem::isEmpty () const [virtual]

Returns TRUE because a spacer item never contains widgets.

Reimplemented from QLayoutItem [Events, Actions, Layouts and Styles with Qt].

QSize QSpacerItem::maximumSize () const [virtual]

Returns the maximum size of this space item.

Reimplemented from QLayoutItem [Events, Actions, Layouts and Styles with Qt].

QSize QSpacerItem::minimumSize () const [virtual]

Returns the minimum size of this spacer item.

Reimplemented from QLayoutItem [Events, Actions, Layouts and Styles with Qt].

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

This function stores the spacer item's rect *r* so that it can be returned by geometry().

Reimplemented from QLayoutItem [Events, Actions, Layouts and Styles with Qt].

QSize QSpacerItem::sizeHint () const [virtual]

Returns the preferred size of this spacer item.

Reimplemented from QLayoutItem [Events, Actions, Layouts and Styles with Qt].

QSpinBox Class Reference

The QSpinBox class provides a spin box widget (spin button).

`#include <qspinbox.h>`

Inherits QWidget [p. 412] and QRangeControl [p. 246].

Public Members

- **QSpinBox** (QWidget * parent = 0, const char * name = 0)
- **QSpinBox** (int minValue, int maxValue, int step = 1, QWidget * parent = 0, const char * name = 0)
- **~QSpinBox** ()
- QString **text** () const
- virtual QString **prefix** () const
- virtual QString **suffix** () const
- virtual QString **cleanText** () const
- virtual void **setSpecialValueText** (const QString & text)
- QString **specialValueText** () const
- virtual void **setWrapping** (bool on)
- bool **wrapping** () const
- enum **ButtonSymbols** { UpDownArrows, PlusMinus }
- virtual void **setButtonSymbols** (ButtonSymbols)
- ButtonSymbols **buttonSymbols** () const
- virtual void **setValidator** (const QValidator * v)
- const QValidator * **validator** () const
- int **minValue** () const
- int **maxValue** () const
- void **setMinValue** (int)
- void **setMaxValue** (int)
- int **lineStep** () const
- void **setLineStep** (int)
- int **value** () const
- QRect **upRect** () const
- QRect **downRect** () const

Public Slots

- virtual void **setValue** (int value)
- virtual void **setPrefix** (const QString & text)
- virtual void **setSuffix** (const QString & text)

- virtual void **stepUp** ()
- virtual void **stepDown** ()
- virtual void **selectAll** ()

Signals

- void **valueChanged** (int value)
- void **valueChanged** (const QString & valueText)

Properties

- ButtonSymbols **buttonSymbols** — the current button symbol mode
- QString **cleanText** — the text of the spin box with any prefix() or suffix() and with any whitespace at the start and end removed *(read only)*
- int **lineStep** — the line step
- int **maxValue** — the maximum value of the spin box
- int **minValue** — the minimum value of the spin box
- QString **prefix** — the prefix of the spin box
- QString **specialValueText** — the special-value text
- QString **suffix** — the suffix of the spin box
- QString **text** — the text of the spin box, including any prefix() and suffix() *(read only)*
- int **value** — the value of the spin box
- bool **wrapping** — whether it is possible to step the value from the highest value to the lowest value and vice versa

Protected Members

- virtual QString **mapValueToText** (int v)
- virtual int **mapTextToValue** (bool * ok)
- QString **currentValueText** ()
- virtual void **updateDisplay** ()
- virtual void **interpretText** ()
- QLineEdit * **editor** () const
- virtual void **valueChange** ()
- virtual void **rangeChange** ()
- virtual bool **eventFilter** (QObject * obj, QEvent * ev)

Protected Slots

- void **textChanged** ()

Detailed Description

The QSpinBox class provides a spin box widget (spin button).

QSpinBox allows the user to choose a value either by clicking the up/down buttons to increase/decrease the value currently displayed or by typing the value directly into the spin box. The value is usually an integer.

Every time the value changes QSpinBox emits the valueChanged() signal. The current value can be fetched with value() and set with setValue().

The spin box keeps the value within a numeric range, and to multiples of the lineStep() size (see QRangeControl for details). Clicking the up/down buttons or using the keyboard accelerator's up and down arrows will increase or decrease the current value in steps of size lineStep(). The minimum and maximum value and the step size can be set using one of the constructors, and can be changed later with setMinValue(), setMaxValue() and setLineStep().

Most spin boxes are directional, but QSpinBox can also operate as a circular spin box, i.e. if the range is 0-99 and the current value is 99, clicking "up" will give 0. Use setWrapping() if you want circular behavior.

The displayed value can be prepended and appended with arbitrary strings indicating, for example, currency or the unit of measurement. See setPrefix() and setSuffix(). The text in the spin box is retrieved with text() (which includes any prefix() and suffix()), or with cleanText() (which has no prefix(), no suffix() and no leading or trailing whitespace). currentValueText() returns the spin box's current value as text.

Normally the spin box displays up and down arrows in the buttons. You can use setButtonSymbols() to change the display to show + and - symbols if this is clearer for your intended purpose. In either case the up and down arrow keys work as expected.

It is often desirable to give the user a special (often default) choice in addition to the range of numeric values. See setSpecialValueText() for how to do this with QSpinBox.

The default QWidget::focusPolicy() is StrongFocus.

If using prefix(), suffix() and specialValueText() don't provide enough control, you can ignore them and subclass QSpinBox instead.

QSpinBox can easily be subclassed to allow the user to input things other than an integer value as long as the allowed input can be mapped to a range of integers. This can be done by overriding the virtual functions mapValueToText() and mapTextToValue(), and setting another suitable validator using setValidator().

For example, these functions could be changed so that the user provided values from 0.0 to 10.0, or -1 to signify 'Auto', while the range of integers used inside the program would be -1 to 100:

```
class MySpinBox : public QSpinBox
{
    Q_OBJECT
public:
    ...

    QString mapValueToText( int value )
    {
        if ( value == -1 ) // Special case
            return QString( "Auto" );

        return QString( "%1.%2" ) // 0.0 to 10.0
               .arg( value / 10 ).arg( value % 10 );
    }

    int mapTextToValue( bool *ok )
    {
        if ( text() == "Auto" ) // Special case
            return -1;
```

```
            return (int) ( 10 * text().toFloat() ); // 0 to 100
        }
    };
```

See also QScrollBar [p. 252], QSlider [p. 285], GUI Design Handbook: Spin Box and Basic Widgets.

Member Type Documentation

QSpinBox::ButtonSymbols

This enum type determines what the buttons in a spin box show. The currently defined values are:

- `QSpinBox::UpDownArrows` - the buttons show little arrows in the classic style.
- `QSpinBox::PlusMinus` - the buttons show + and - symbols.

See also QSpinBox::buttonSymbols [p. 303].

Member Function Documentation

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

Constructs a spin box with the default QRangeControl range and step values. It has the parent *parent* and the name *name*.

See also minValue [p. 304], maxValue [p. 304], setRange() [p. 250], lineStep [p. 303] and setSteps() [p. 250].

QSpinBox::QSpinBox (int minValue, int maxValue, int step = 1, QWidget * parent = 0, const char * name = 0)

Constructs a spin box that allows values from *minValue* to *maxValue* inclusive, with step amount *step*. The value is initially set to *minValue*.

The widget's parent is *parent* and the spin box is called *name*.

See also minValue [p. 304], maxValue [p. 304], setRange() [p. 250], lineStep [p. 303] and setSteps() [p. 250].

QSpinBox::~QSpinBox ()

Destroys the spin box, freeing all memory and other resources.

ButtonSymbols QSpinBox::buttonSymbols () const

Returns the current button symbol mode. See the "buttonSymbols" [p. 303] property for details.

QString QSpinBox::cleanText () const [virtual]

Returns the text of the spin box with any prefix() or suffix() and with any whitespace at the start and end removed. See the "cleanText" [p. 303] property for details.

QString QSpinBox::currentValueText () [protected]

Returns the full text calculated from the current value, including any prefix and suffix. If there is special value text and the value is minValue() the specialValueText() is returned.

QRect QSpinBox::downRect () const

Returns the geometry of the "down" button.

QLineEdit * QSpinBox::editor () const [protected]

Returns a pointer to the embedded QLineEdit.

bool QSpinBox::eventFilter (QObject * obj, QEvent * ev) [virtual protected]

Intercepts and handles the events coming to the embedded QLineEdit that have special meaning for the QSpinBox. The object is passed as *obj* and the event is passed as *ev*.

Reimplemented from QObject [Additional Functionality with Qt].

void QSpinBox::interpretText () [virtual protected]

QSpinBox calls this after the user has manually edited the contents of the spin box (i.e. by typing in the embedded QLineEdit, rather than using the up/down buttons/keys).

The default implementation of this function interprets the new text using mapTextToValue(). If mapTextToValue() is successful, it changes the spin box's value; if not, the value is left unchanged.

See also editor() [p. 299].

int QSpinBox::lineStep () const

Returns the line step. See the "lineStep" [p. 303] property for details.

int QSpinBox::mapTextToValue (bool * ok) [virtual protected]

This virtual function is used by the spin box whenever it needs to interpret text entered by the user as a value. The text is available as text() and as cleanText(), and this function must parse it if possible, and set the bool **ok* to TRUE if successful and to FALSE otherwise.

Subclasses that need to display spin box values in a non-numeric way need to reimplement this function.

Note that Qt handles specialValueText() separately; this function is only concerned with the other values.

The default implementation tries to interpret the text() as an integer in the standard way and returns the integer value.

See also interpretText() [p. 299] and mapValueToText() [p. 299].

QString QSpinBox::mapValueToText (int v) [virtual protected]

This virtual function is used by the spin box whenever it needs to display value *v*. The default implementation returns a string containing *v* printed in the standard way. Reimplementations may return anything. (See the

example in the detailed description.)

Note that Qt does not call this function for specialValueText() and that neither prefix() nor suffix() are included in the return value.

If you reimplement this, you may also need to reimplement mapTextToValue().

See also updateDisplay() [p. 302] and mapTextToValue() [p. 299].

int QSpinBox::maxValue () const

Returns the maximum value of the spin box. See the "maxValue" [p. 304] property for details.

int QSpinBox::minValue () const

Returns the minimum value of the spin box. See the "minValue" [p. 304] property for details.

QString QSpinBox::prefix () const [virtual]

Returns the prefix of the spin box. See the "prefix" [p. 304] property for details.

void QSpinBox::rangeChange () [virtual protected]

This virtual function is called by QRangeControl whenever the range has changed. It adjusts the default validator and updates the display; if you need additional processing, you may reimplement this function.

Reimplemented from QRangeControl [p. 249].

void QSpinBox::selectAll () [virtual slot]

Selects all the text in the editor of the spinbox

void QSpinBox::setButtonSymbols (ButtonSymbols) [virtual]

Sets the current button symbol mode. See the "buttonSymbols" [p. 303] property for details.

void QSpinBox::setLineStep (int)

Sets the line step. See the "lineStep" [p. 303] property for details.

void QSpinBox::setMaxValue (int)

Sets the maximum value of the spin box. See the "maxValue" [p. 304] property for details.

void QSpinBox::setMinValue (int)

Sets the minimum value of the spin box. See the "minValue" [p. 304] property for details.

void QSpinBox::setPrefix (const QString & text) [virtual slot]

Sets the prefix of the spin box to *text*. See the "prefix" [p. 304] property for details.

void QSpinBox::setSpecialValueText (const QString & text) [virtual]

Sets the special-value text to *text*. See the "specialValueText" [p. 304] property for details.

void QSpinBox::setSuffix (const QString & text) [virtual slot]

Sets the suffix of the spin box to *text*. See the "suffix" [p. 305] property for details.

void QSpinBox::setValidator (const QValidator * v) [virtual]

Sets the validator to *v*. The validator controls what keyboard input is accepted when the user is editing in the value field. The default is to use a suitable QIntValidator.

Use setValidator(0) to turn off input validation (entered input will still be clamped to the range of the spinbox).

void QSpinBox::setValue (int value) [virtual slot]

Sets the value of the spin box to *value*. See the "value" [p. 305] property for details.

void QSpinBox::setWrapping (bool on) [virtual]

Sets whether it is possible to step the value from the highest value to the lowest value and vice versa to *on*. See the "wrapping" [p. 305] property for details.

QString QSpinBox::specialValueText () const

Returns the special-value text. See the "specialValueText" [p. 304] property for details.

void QSpinBox::stepDown () [virtual slot]

Decreases the spin box's value one lineStep(), wrapping as necessary. This is the same as clicking on the pointing-down button and can be used for keyboard accelerators, for example.

See also stepUp() [p. 301], subtractLine() [p. 250], lineStep [p. 303], setSteps() [p. 250], value [p. 305] and value [p. 305].

void QSpinBox::stepUp () [virtual slot]

Increases the spin box's value by one lineStep(), wrapping as necessary. This is the same as clicking on the pointing-up button and can be used for keyboard accelerators, for example.

See also stepDown() [p. 301], addLine() [p. 248], lineStep [p. 303], setSteps() [p. 250], value [p. 305] and value [p. 305].

QString QSpinBox::suffix () const [virtual]

Returns the suffix of the spin box. See the "suffix" [p. 305] property for details.

QString QSpinBox::text () const

Returns the text of the spin box, including any prefix() and suffix(). See the "text" [p. 305] property for details.

void QSpinBox::textChanged () [protected slot]

This slot is called whenever the user edits the text of the spin box.

QRect QSpinBox::upRect () const

Returns the geometry of the "up" button.

void QSpinBox::updateDisplay () [virtual protected]

Updates the contents of the embedded QLineEdit to reflect the current value using mapValueToText(). Also enables/disables the up/down push buttons accordingly.

See also mapValueToText() [p. 299].

const QValidator * QSpinBox::validator () const

Returns the validator that constrains editing for this spin box if there is any; otherwise returns 0.

See also setValidator() [p. 301] and QValidator [Additional Functionality with Qt].

int QSpinBox::value () const

Returns the value of the spin box. See the "value" [p. 305] property for details.

void QSpinBox::valueChange () [virtual protected]

This virtual function is called by QRangeControl whenever the value has changed. The QSpinBox reimplementation updates the display and emits the valueChanged() signals; if you need additional processing, either reimplement this or connect to one of the valueChanged() signals.

Reimplemented from QRangeControl [p. 251].

void QSpinBox::valueChanged (int value) [signal]

This signal is emitted every time the value of the spin box changes; the new value is passed in *value*. This signal will be emitted as a result of a call to setValue(), or because the user changed the value by using a keyboard accelerator or mouse click, etc.

Note that the valueChanged() signal is emitted *every* time, not just for the "final" step; i.e. if the user clicks "up" three times, this signal is emitted three times.

See also value [p. 305].

Examples: listbox/listbox.cpp, qfd/fontdisplayer.cpp and scribble/scribble.cpp.

void QSpinBox::valueChanged (const QString & valueText) [signal]

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

This signal is emitted whenever the valueChanged(int) signal is emitted, i.e. every time the value of the spin box changes (whatever the cause, e.g. by setValue(), by a keyboard accelerator, by mouse clicks, etc.).

The *valueText* parameter is the same string that is displayed in the edit field of the spin box.

See also value [p. 305], prefix [p. 304], suffix [p. 305] and specialValueText [p. 304].

bool QSpinBox::wrapping () const

Returns TRUE if it is possible to step the value from the highest value to the lowest value and vice versa; otherwise returns FALSE. See the "wrapping" [p. 305] property for details.

Property Documentation

ButtonSymbols buttonSymbols

This property holds the current button symbol mode.

The possible values can be either UpDownArrows or PlusMinus. The default is UpDownArrows.

See also ButtonSymbols [p. 298].

Set this property's value with setButtonSymbols() and get this property's value with buttonSymbols().

QString cleanText

This property holds the text of the spin box with any prefix() or suffix() and with any whitespace at the start and end removed.

Get this property's value with cleanText().

See also text [p. 305], prefix [p. 304] and suffix [p. 305].

int lineStep

This property holds the line step.

When the user uses the arrows to change the spin box's value the value will be incremented/decremented by the amount of the line step.

The setLineStep() function calls the virtual stepChange() function if the new line step is different from the previous setting.

See also QRangeControl::setSteps() [p. 250] and setRange() [p. 250].

Set this property's value with setLineStep() and get this property's value with lineStep().

int maxValue

This property holds the maximum value of the spin box.

When setting this property, the QSpinBox::minValue is adjusted so that the range remains valid if necessary.

See also setRange() [p. 250] and specialValueText [p. 304].

Set this property's value with setMaxValue() and get this property's value with maxValue().

int minValue

This property holds the minimum value of the spin box.

When setting this property, the QSpinBox::maxValue is adjusted so that the range remains valid if necessary.

See also setRange() [p. 250] and specialValueText [p. 304].

Set this property's value with setMinValue() and get this property's value with minValue().

QString prefix

This property holds the prefix of the spin box.

The prefix is prepended to the start of the displayed value. Typical use is to indicate the unit of measurement to the user. For example:

```
sb->setPrefix( "$" );
```

To turn off the prefix display, set this property to an empty string. The default is no prefix. The prefix is not displayed for the minValue() if specialValueText() is not empty.

If no prefix is set, prefix() returns a null string.

See also suffix [p. 305].

Set this property's value with setPrefix() and get this property's value with prefix().

QString specialValueText

This property holds the special-value text.

If set, the spin box will display this text instead of a numeric value whenever the current value is equal to minVal(). Typical use is to indicate that this choice has a special (default) meaning.

For example, if your spin box allows the user to choose the margin width in a print dialog and your application is able to automatically choose a good margin width, you can set up the spin box like this:

```
QSpinBox marginBox( -1, 20, 1, parent, "marginBox" );
marginBox->setSuffix( " mm" );
marginBox->setSpecialValueText( "Auto" );
```

The user will then be able to choose a margin width from 0-20 millimeters or select "Auto" to leave it to the application to choose. Your code must then interpret the spin box value of -1 as the user requesting automatic margin width.

All values are displayed with the prefix() and suffix() (if set), *except* for the special value, which only shows the special value text.

To turn off the special-value text display, call this function with an empty string. The default is no special-value text, i.e. the numeric value is shown as usual.

If no special-value text is set, specialValueText() returns a null string.

Set this property's value with setSpecialValueText() and get this property's value with specialValueText().

QString suffix

This property holds the suffix of the spin box.

The suffix is appended to the end of the displayed value. Typical use is to indicate the unit of measurement to the user. For example:

```
sb->setSuffix( " km" );
```

To turn off the suffix display, set this property to an empty string. The default is no suffix. The suffix is not displayed for the minValue() if specialValueText() is not empty.

If no suffix is set, suffix() returns a null string.

See also prefix [p. 304].

Set this property's value with setSuffix() and get this property's value with suffix().

QString text

This property holds the text of the spin box, including any prefix() and suffix().

There is no default text.

See also value [p. 305].

Get this property's value with text().

int value

This property holds the value of the spin box.

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

See also QRangeControl::setValue() [p. 250].

bool wrapping

This property holds whether it is possible to step the value from the highest value to the lowest value and vice versa.

By default, wrapping is turned off.

If you have a range of 0..100 and wrapping is off when the user reaches 100 and presses the Up Arrow nothing will happen; but if wrapping is on the value will change from 100 to 0, then to 1, etc. When wrapping is on, navigating past the highest value takes you to the lowest and vice versa.

See also minValue [p. 304], maxValue [p. 304] and setRange() [p. 250].

Set this property's value with setWrapping() and get this property's value with wrapping().

QSplitter Class Reference

The QSplitter class implements a splitter widget.

`#include <qsplitter.h>`

Inherits QFrame [p. 65].

Public Members

- enum **ResizeMode** { Stretch, KeepSize, FollowSizeHint }
- **QSplitter** (QWidget * parent = 0, const char * name = 0)
- **QSplitter** (Orientation o, QWidget * parent = 0, const char * name = 0)
- **~QSplitter** ()
- virtual void **setOrientation** (Orientation)
- Orientation **orientation** () const
- virtual void **setResizeMode** (QWidget * w, ResizeMode mode)
- virtual void **setOpaqueResize** (bool on = TRUE)
- bool **opaqueResize** () const
- void **moveToFirst** (QWidget * w)
- void **moveToLast** (QWidget * w)
- void **refresh** ()
- QValueList<int> **sizes** () const
- void **setSizes** (QValueList<int> list)

Properties

- Orientation **orientation** — the orientation of the splitter

Protected Members

- virtual void **childEvent** (QChildEvent * c)
- int **idAfter** (QWidget * w) const
- void **moveSplitter** (QCOORD p, int id)
- virtual void **drawSplitter** (QPainter * p, QCOORD x, QCOORD y, QCOORD w, QCOORD h)
- int **adjustPos** (int p, int id)
- virtual void **setRubberband** (int p)
- void **getRange** (int id, int * min, int * max)

Detailed Description

The QSplitter class implements a splitter widget.

A splitter lets the user control the size of child widgets by dragging the boundary between the children. Any number of widgets may be controlled.

To show a QListBox, a QListView and a QTextEdit side by side:

```
QSplitter *split = new QSplitter( parent );
QListBox *lb = new QListBox( split );
QListView *lv = new QListView( split );
QTextEdit *ed = new QTextEdit( split );
```

In QSplitter the boundary can be either horizontal or vertical. The default is horizontal (the children are side by side) but you can use setOrientation(QSplitter::Vertical) to set it to vertical.

By default, all widgets can be as large or as small as the user wishes, down to minimumSizeHint(). You can also use setMinimumSize() and setMaximumSize() on the children. Use setResizeMode() to specify that a widget should keep its size when the splitter is resized.

Although QSplitter normally resizes the children only at the end of a resize operation, if you call setOpaqueResize(TRUE) the widgets are resized as often as possible.

The initial distribution of size between the widgets is determined by the initial size of each widget. You can also use setSizes() to set the sizes of all the widgets. The function sizes() returns the sizes set by the user.

If you hide() a child its space will be distributed among the other children. It will be reinstated when you show() it again. It is also possible to reorder the widgets within the splitter using moveToFirst() and moveToLast().

See also QTabBar [p. 318] and Organizers.

Member Type Documentation

QSplitter::ResizeMode

This enum type describes how QSplitter will resize each of its child widgets. The currently defined values are:

- `QSplitter::Stretch` - the widget will be resized when the splitter itself is resized.
- `QSplitter::KeepSize` - QSplitter will try to keep this widget's size unchanged.
- `QSplitter::FollowSizeHint` - QSplitter will resize the widget when the widget's size hint changes.

Member Function Documentation

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

Constructs a horizontal splitter with the *parent* and *name* arguments being passed on to the QFrame constructor.

QSplitter::QSplitter (Orientation o, QWidget * parent = 0, const char * name = 0)

Constructs a splitter with orientation *o* with the *parent* and *name* arguments being passed on to the QFrame constructor.

QSplitter::~QSplitter ()

Destroys the splitter and any children.

int QSplitter::adjustPos (int p, int id) [protected]

Returns the closest legal position to *p* of the splitter with id *id*.

See also idAfter() [p. 308].

void QSplitter::childEvent (QChildEvent * c) [virtual protected]

Tells the splitter that a child widget has been inserted or removed. The event is passed in *c*.

Reimplemented from QObject [Additional Functionality with Qt].

void QSplitter::drawSplitter (QPainter * p, QCOORD x, QCOORD y, QCOORD w, QCOORD h) [virtual protected]

Draws the splitter handle in the rectangle described by *x*, *y*, *w*, *h* using painter *p*.

See also QStyle::drawPrimitive() [Events, Actions, Layouts and Styles with Qt].

void QSplitter::getRange (int id, int * min, int * max) [protected]

Returns the valid range of the splitter with id *id* in **min* and **max*.

See also idAfter() [p. 308].

int QSplitter::idAfter (QWidget * w) const [protected]

Returns the id of the splitter to the right of or below the widget *w*, or 0 if there is no such splitter (i.e. it is either not in this QSplitter or it is at the end).

void QSplitter::moveSplitter (QCOORD p, int id) [protected]

Moves the left/top edge of the splitter handle with id *id* as close as possible to position *p*, which is the distance from the left (or top) edge of the widget.

For Arabic and Hebrew the layout is reversed, and using this function to set the position of the splitter might lead to unexpected results, since in Arabic and Hebrew the position of splitter one is to the left of the position of splitter zero.

See also idAfter() [p. 308].

void QSplitter::moveToFirst (QWidget * w)

Moves widget *w* to the leftmost/top position.

Example: splitter/splitter.cpp.

void QSplitter::moveToLast (QWidget * w)

Moves widget *w* to the rightmost/bottom position.

bool QSplitter::opaqueResize () const

Returns TRUE if opaque resize is on; otherwise returns FALSE.

See also setOpaqueResize() [p. 309].

Orientation QSplitter::orientation () const

Returns the orientation of the splitter. See the "orientation" [p. 310] property for details.

void QSplitter::refresh ()

Updates the splitter's state. You should not need to call this function.

void QSplitter::setOpaqueResize (bool on = TRUE) [virtual]

If *on* is TRUE then opaque resizing is turned on; otherwise opaque resizing is turned off. Opaque resizing is initially turned off.

See also opaqueResize() [p. 309].

Examples: mainlyQt/editor.cpp and splitter/splitter.cpp.

void QSplitter::setOrientation (Orientation) [virtual]

Sets the orientation of the splitter. See the "orientation" [p. 310] property for details.

void QSplitter::setResizeMode (QWidget * w, ResizeMode mode) [virtual]

Sets resize mode of *w* to *mode*.

See also ResizeMode [p. 307].

Examples: fileiconview/mainwindow.cpp, listviews/listviews.cpp, network/ftpclient/ftpmainwindow.cpp and splitter/splitter.cpp.

void QSplitter::setRubberband (int p) [virtual protected]

Shows a rubber band at position *p*. If *p* is negative, the rubber band is removed.

void QSplitter::setSizes (QValueList<int> list)

Sets the size parameters to the values given in *list*. If the splitter is horizontal, the values set the sizes from left to right. If it is vertical, the sizes are applied from top to bottom. Extra values in *list* are ignored.

If *list* contains too few values, the result is undefined but the program will still be well-behaved.

See also sizes() [p. 310].

QValueList<int> QSplitter::sizes () const

Returns a list of the size parameters of all the widgets in this splitter.

Giving the values to another splitter's setSizes() function will produce a splitter with the same layout as this one.

See also setSizes() [p. 309].

Property Documentation

Orientation orientation

This property holds the orientation of the splitter.

By default the orientation is horizontal (the widgets are side by side). The possible orientations are Qt:Vertical and Qt::Horizontal (the default).

Set this property's value with setOrientation() and get this property's value with orientation().

QStatusBar Class Reference

The QStatusBar class provides a horizontal bar suitable for presenting status information.

`#include <qstatusbar.h>`

Inherits QWidget [p. 412].

Public Members

- **QStatusBar** (QWidget * parent = 0, const char * name = 0)
- virtual **~QStatusBar** ()
- virtual void **addWidget** (QWidget * widget, int stretch = 0, bool permanent = FALSE)
- virtual void **removeWidget** (QWidget * widget)
- void **setSizeGripEnabled** (bool)
- bool **isSizeGripEnabled** () const

Public Slots

- void **message** (const QString & message)
- void **message** (const QString & message, int ms)
- void **clear** ()

Properties

- bool **sizeGripEnabled** — whether the QSizeGrip in the bottom right of the status bar is enabled

Protected Members

- virtual void **paintEvent** (QPaintEvent *)
- void **reformat** ()
- void **hideOrShow** ()

Detailed Description

The QStatusBar class provides a horizontal bar suitable for presenting status information.

Each status indicator falls into one of three categories:

- *Temporary* - briefly occupies most of the status bar. Used to explain tool tip texts or menu entries, for example.
- *Normal* - occupies part of the status bar and may be hidden by temporary messages. Used to display the page and line number in a word processor, for example.
- *Permanent* - is never hidden. Used for important mode indications, for example, some applications put a Caps Lock indicator in the status bar.

QStatusBar lets you display all three types of indicators.

To display a *temporary* message, call message() (perhaps by connecting a suitable signal to it). To remove a temporary message, call clear(). There are two variants of message(): one that displays the message until the next clear() or mesage() and one that has a time limit:

```
connect( loader, SIGNAL(progressMessage(const QString&)),
         statusBar(), SLOT(message(const QString&)) );

statusBar()->message("Loading...");  // Initial message
loader.loadStuff();                  // Emits progress messages
statusBar()->message("Done.", 2000); // Final message for 2 seconds
```

Normal and *permanent* messages are displayed by creating a small widget and then adding it to the status bar with addWidget(). Widgets like QLabel, QProgressBar or even QToolButton are useful for adding to status bars. removeWidget() is used to remove widgets.

```
statusBar()->addWidget(new MyReadWriteIndication(statusBar()));
```

By default QStatusBar provides a QSizeGrip in the lower-right corner. You can disable it with setSizeGripEnabled(FALSE);

See also QToolBar [Dialogs and Windows with Qt], QMainWindow [Dialogs and Windows with Qt], QLabel [p. 117], GUI Design Handbook: Status Bar, Main Window and Related Classes and Help System.

Member Function Documentation

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

Constructs a status bar with the parent *parent* and the name *name* and with a size grip.

See also sizeGripEnabled [p. 314].

QStatusBar::~QStatusBar () [virtual]

Destroys the status bar and frees any allocated resources and child widgets.

void QStatusBar::addWidget (QWidget * widget, int stretch = 0, bool permanent = FALSE) [virtual]

Adds *widget* to this status bar.

widget is permanently visible if *permanent* is TRUE and may be obscured by temporary messages if *permanent* is FALSE. The default is FALSE.

If *permanent* is TRUE, *widget* is located at the far right of the status bar. If *permanent* is FALSE (the default), *widget* is located just to the left of the first permanent widget.

stretch is used to compute a suitable size for *widget* as the status bar grows and shrinks. The default of 0 uses a minimum of space.

This function may cause some flicker.

See also removeWidget() [p. 313].

void QStatusBar::clear () [slot]

Removes any temporary message being shown.

See also message() [p. 313].

void QStatusBar::hideOrShow () [protected]

Ensures that the right widgets are visible. Used by message() and clear().

bool QStatusBar::isSizeGripEnabled () const

Returns TRUE if the QSizeGrip in the bottom right of the status bar is enabled; otherwise returns FALSE. See the "sizeGripEnabled" [p. 314] property for details.

void QStatusBar::message (const QString & message) [slot]

Hides the normal status indicators and displays *message* until clear() or another message() is called.

See also clear() [p. 313].

void QStatusBar::message (const QString & message, int ms) [slot]

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

Hides the normal status indications and displays *message* for *ms* milli-seconds or until clear() or another message() is called, whichever occurs first.

void QStatusBar::paintEvent (QPaintEvent *) [virtual protected]

Shows the temporary message, if appropriate.

Reimplemented from QWidget [p. 441].

void QStatusBar::reformat () [protected]

Changes the status bar's appearance to account for item changes. Special subclasses may need this, but geometry management will usually take care of any necessary rearrangements.

void QStatusBar::removeWidget (QWidget * widget) [virtual]

Removes *widget* from the status bar.

This function may cause some flicker.

Note that *widget* is not deleted.

See also addWidget() [p. 312].

void QStatusBar::setSizeGripEnabled (bool)

Sets whether the QSizeGrip in the bottom right of the status bar is enabled. See the "sizeGripEnabled" [p. 314] property for details.

Property Documentation

bool sizeGripEnabled

This property holds whether the QSizeGrip in the bottom right of the status bar is enabled.

Enables or disables the QSizeGrip in the bottom right of the status bar. By default, the size grip is enabled.

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

QTab Class Reference

The QTab class provides the structures in a QTabBar.

`#include <qtabbar.h>`

Inherits Qt [Additional Functionality with Qt].

Public Members

- **QTab** ()
- virtual **~QTab** ()
- **QTab** (const QString & text)
- **QTab** (const QIconSet & icon, const QString & text = QString::null)
- void **setText** (const QString & text)
- QString **text** () const
- void **setIconSet** (const QIconSet & icon)
- QIconSet * **iconSet** () const
- void **setRect** (const QRect & rect)
- QRect **rect** () const
- void **setEnabled** (bool enable)
- bool **isEnabled** () const
- void **setIdentifier** (int i)
- int **identifier** () const

Detailed Description

The QTab class provides the structures in a QTabBar.

This class is used for custom QTabBar tab headings.

See also QTabBar [p. 318] and Advanced Widgets.

Member Function Documentation

QTab::QTab ()

Constructs an empty tab. All fields are set to empty.

QTab::QTab (const QString & text)

Constructs a tab with the text, *text*.

QTab::QTab (const QIconSet & icon, const QString & text = QString::null)

Constructs a tab with an *icon* and the text, *text*.

QTab::~QTab () [virtual]

Destroys the tab and frees up all allocated resources

QIconSet * QTab::iconSet () const

Return the QIconSet of the QTab.

int QTab::identifier () const

Return the identifier for the QTab.

bool QTab::isEnabled () const

Returns TRUE if the QTab is enabled, otherwise return FALSE.

QRect QTab::rect () const

Return the QRect for the QTab.

void QTab::setEnabled (bool enable)

If *enable* is TRUE enable the QTab, otherwise disable it.

void QTab::setIconSet (const QIconSet & icon)

Sets the tab iconset to *icon*

void QTab::setIdentifier (int i)

Set the identifier for the QTab to *i*. Each identifier for a QTabBar must be unique

void QTab::setRect (const QRect & rect)

Set the QTab QRect to *rect*.

void QTab::setText (const QString & text)

Sets the text of the tab to *text*.

QString QTab::text () const

Return the text of the QTab label.

QTabBar Class Reference

The QTabBar class provides a tab bar, e.g. for use in tabbed dialogs.

`#include <qtabbar.h>`

Inherits QWidget [p. 412].

Public Members

- **QTabBar** (QWidget * parent = 0, const char * name = 0)
- **~QTabBar** ()
- enum **Shape** { RoundedAbove, RoundedBelow, TriangularAbove, TriangularBelow }
- Shape **shape** () const
- virtual void **setShape** (Shape)
- virtual int **addTab** (QTab * newTab)
- virtual int **insertTab** (QTab * newTab, int index = -1)
- virtual void **removeTab** (QTab * t)
- virtual void **setTabEnabled** (int id, bool enabled)
- bool **isTabEnabled** (int id) const
- int **currentTab** () const
- int **keyboardFocusTab** () const
- QTab * **tab** (int id) const
- QTab * **tabAt** (int index) const
- int **indexOf** (int id) const
- int **count** () const
- virtual void **layoutTabs** ()
- virtual QTab * **selectTab** (const QPoint & p) const
- void **removeToolTip** (int index)
- void **setToolTip** (int index, const QString & tip)
- QString **toolTip** (int index) const

Public Slots

- virtual void **setCurrentTab** (int)
- virtual void **setCurrentTab** (QTab * tab)

Signals

- void **selected** (int id)

Properties

- int **count** — the number of tabs in the tab bar *(read only)*
- int **currentTab** — the id of the currently visible tab in the tab bar
- int **keyboardFocusTab** — the id of the tab that currently has the keyboard focus *(read only)*
- Shape **shape** — the shape of the tabs in the tab bar

Protected Members

- virtual void **paint** (QPainter * p, QTab * t, bool selected) const
- virtual void **paintLabel** (QPainter * p, const QRect & br, QTab * t, bool has_focus) const
- virtual void **paintEvent** (QPaintEvent * e)
- QPtrList<QTab> * **tabList** ()

Detailed Description

The QTabBar class provides a tab bar, e.g. for use in tabbed dialogs.

QTabBar is straightforward to use; it draws the tabs using one of the predefined shapes, and emits a signal when a tab is selected. It can be subclassed to tailor the look and feel.

The choice of tab shape is a matter of taste, although tab dialogs (preferences and the like) invariably use RoundedAbove, and nobody uses TriangularAbove. Tab controls in windows other than dialogs almost always use either RoundedBelow or TriangularBelow. Many spreadsheets and other tab controls in which all the pages are essentially similar use TriangularBelow, whereas RoundedBelow is used mostly when the pages are different (e.g. a multi-page tool palette).

The most important part of QTabBar's API is the signal selected(). This is emitted whenever the selected page changes (even at startup, when the selected page changes from 'none'). There is also a slot, setCurrentTab(), which can be used to select a page programmatically.

QTabBar creates automatic accelerator keys in the manner of QButton; e.g. if a tab's label is "&Graphics", Alt+G becomes an accelerator key for switching to that tab.

The following virtual functions may need to be reimplemented:

- paint() paints a single tab. paintEvent() calls paint() for each tab so that any overlap will look right.
- addTab() creates a new tab and adds it to the bar.
- selectTab() decides which tab, if any, the user selects with the mouse.

The index of the current tab is returned by currentTab(). The tab with a particular index is returned by tabAt(), the tab with a particular id is returned by tab(). The index of a tab is returned by indexOf(). The current tab can be set by index or tab pointer using one of the setCurrentTab() functions.

See also Advanced Widgets.

Member Type Documentation

QTabBar::Shape

This enum type lists the built-in shapes supported by QTabBar:

- `QTabBar::RoundedAbove` - the normal rounded look above the pages
- `QTabBar::RoundedBelow` - the normal rounded look below the pages
- `QTabBar::TriangularAbove` - triangular tabs above the pages (very unusual; included for completeness)
- `QTabBar::TriangularBelow` - triangular tabs similar to those used in the spreadsheet Excel, for example

Member Function Documentation

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

Constructs a new, empty tab bar; the *parent* and *name* arguments are passed on to the QWidget constructor.

QTabBar::~QTabBar ()

Destroys the tab control, freeing memory used.

int QTabBar::addTab (QTab * newTab) [virtual]

Adds the tab, *newTab*, to the tab control.

Sets *newTab*'s id to a new id and places the tab just to the right of the existing tabs. If the tab's label contains an ampersand, the letter following the ampersand is used as an accelerator for the tab, e.g. if the label is "Bro&wse" then Alt+W becomes an accelerator which will move the focus to this tab. Returns the id.

See also insertTab() [p. 320].

int QTabBar::count () const

Returns the number of tabs in the tab bar. See the "count" [p. 323] property for details.

int QTabBar::currentTab () const

Returns the id of the currently visible tab in the tab bar. See the "currentTab" [p. 323] property for details.

int QTabBar::indexOf (int id) const

Returns the position index of the tab with id *id*.

See also tabAt() [p. 322].

int QTabBar::insertTab (QTab * newTab, int index = -1) [virtual]

Inserts the tab, *newTab*, into the tab control.

If *index* is not specified, the tab is simply added. Otherwise it's inserted at the specified position.

Sets *newTab*'s id to a new id. If the tab's label contains an ampersand, the letter following the ampersand is used as an accelerator for the tab, e.g. if the label is "Bro&wse" then Alt+W becomes an accelerator which will move the focus to this tab. Returns the id.

See also addTab() [p. 320].

bool QTabBar::isTabEnabled (int id) const

Returns TRUE if the tab with id *id* is enabled, or FALSE if it is disabled or there is no such tab.

See also setTabEnabled() [p. 322].

int QTabBar::keyboardFocusTab () const

Returns the id of the tab that currently has the keyboard focus. See the "keyboardFocusTab" [p. 323] property for details.

void QTabBar::layoutTabs () [virtual]

Lays out all existing tabs according to their label and their iconset.

void QTabBar::paint (QPainter * p, QTab * t, bool selected) const [virtual protected]

Paints the tab *t* using painter *p*. If and only if *selected* is TRUE, *t* is drawn currently selected.

This virtual function may be reimplemented to change the look of QTabBar. If you decide to reimplement it, you may also need to reimplement sizeHint().

void QTabBar::paintEvent (QPaintEvent * e) [virtual protected]

Repaints the tab row. All the painting is done by paint(); paintEvent() only decides which tabs need painting and in what order. The event is passed in *e*.

See also paint() [p. 321].

Reimplemented from QWidget [p. 441].

void QTabBar::paintLabel (QPainter * p, const QRect & br, QTab * t, bool has_focus) const [virtual protected]

Paints the label of tab *t* centered in rectangle *br* using painter *p*. A focus indication is drawn if *has_focus* is TRUE.

void QTabBar::removeTab (QTab * t) [virtual]

Removes tab *t* from the tab control, and deletes the tab.

void QTabBar::removeToolTip (int index)

Removes the tool tip for the tab at index *index*.

QTab * QTabBar::selectTab (const QPoint & p) const [virtual]

This virtual function is called by the mouse event handlers to determine which tab is pressed. The default implementation returns a pointer to the tab whose bounding rectangle contains *p*, if exactly one tab's bounding rectangle contains *p*. Otherwise it returns 0.

See also mousePressEvent() [p. 440] and mouseReleaseEvent() [p. 440].

void QTabBar::selected (int id) [signal]

QTabBar emits this signal whenever any tab is selected, whether by the program or by the user. The argument *id* is the id of the tab as returned by addTab().

show() is guaranteed to emit this signal; you can display your page in a slot connected to this signal.

void QTabBar::setCurrentTab (int) [virtual slot]

Sets the id of the currently visible tab in the tab bar. See the "currentTab" [p. 323] property for details.

void QTabBar::setCurrentTab (QTab * tab) [virtual slot]

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

Raises *tab* and emits the selected() signal unless the tab was already current.

See also currentTab [p. 323] and selected() [p. 322].

void QTabBar::setShape (Shape) [virtual]

Sets the shape of the tabs in the tab bar. See the "shape" [p. 323] property for details.

void QTabBar::setTabEnabled (int id, bool enabled) [virtual]

Enables tab *id* if *enabled* is TRUE or disables it if *enabled* is FALSE. If *id* is currently selected, setTabEnabled(FALSE) makes another tab selected.

setTabEnabled() updates the display if this causes a change in *id*'s status.

See also update() [p. 456] and isTabEnabled() [p. 321].

void QTabBar::setToolTip (int index, const QString & tip)

Sets the tool tip for the tab at index *index* to *tip*.

Shape QTabBar::shape () const

Returns the shape of the tabs in the tab bar. See the "shape" [p. 323] property for details.

QTab * QTabBar::tab (int id) const

Returns a pointer to the tab with id *id* or 0 if there is no such tab.

See also count [p. 323].

QTab * QTabBar::tabAt (int index) const

Returns a pointer to the tab at the position *index*.

See also indexOf() [p. 320].

QPtrList<QTab> * QTabBar::tabList () [protected]

The list of QTab objects in the tab bar.

QString QTabBar::toolTip (int index) const

Returns the tool tip for the tab at index *index*.

Property Documentation

int count

This property holds the number of tabs in the tab bar.

Get this property's value with count().

See also tab() [p. 322].

int currentTab

This property holds the id of the currently visible tab in the tab bar.

If no tab page is currently visible, -1 will be the current value for this property. Even if the property value is not -1, you cannot assume that the user can see the relevant page, or that the tab is enabled. When you need to display something the value of this property represents the best page to display.

When this property is set to *id*, it will raise the tab with the id *id* and emit the selected() signal.

See also selected() [p. 322] and isTabEnabled() [p. 321].

Set this property's value with setCurrentTab() and get this property's value with currentTab().

int keyboardFocusTab

This property holds the id of the tab that currently has the keyboard focus.

This property contains the id of the tab that currently has the keyboard focus. If the tab bar does not have keyboard focus, the value of this property will be -1.

Get this property's value with keyboardFocusTab().

Shape shape

This property holds the shape of the tabs in the tab bar.

The value of this property can be one of the following: RoundedAbove (default), RoundedBelow, TriangularAbove or TriangularBelow.

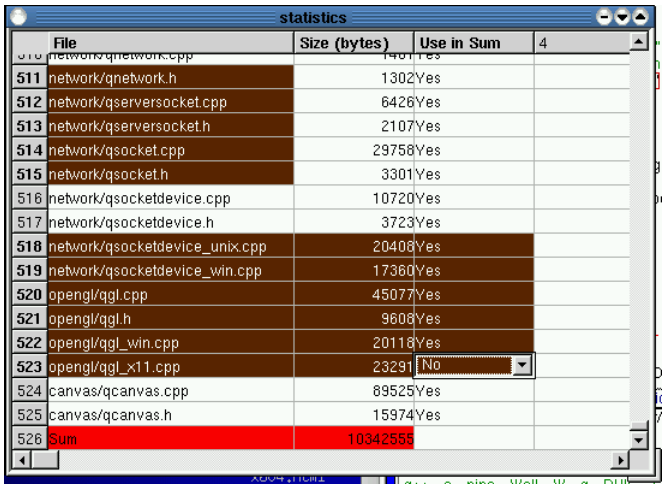
See also Shape [p. 319].

Set this property's value with setShape() and get this property's value with shape().

Table Module

This module is part of the Qt Enterprise Edition.

The table module provides a flexible and editable table widget, QTable. For many applications QTables can be used directly and simply, providing a grid of editable cells. QTable can also be subclassed in a straightforward way to provide very large sparse tables, e.g. one million by one million cells.



The table module provides the following classes:

- QTable itself is the abstract widget of choice whenever you need to provide your users with the ability to display and edit tabular data such as spreadsheet or database data.
- QTableItem objects are used to populate a QTable with data with each table item holding the contents of a cell.
- The QComboTableItem class provides memory-efficient combobox items for QTables.
- The QCheckTableItem class provides memory-efficient checkbox items for QTables.
- QTableSelection provides access to selections of cells in a QTable.
- QHeader provides access to the table's horizontal header (column headers) and vertical header (row headers).

Please see the appropriate class documentation for details and refer to the Qt table examples for practical demonstration.

QTable Class Reference

The QTable class provides a flexible editable table widget.

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

`#include <qtable.h>`

Inherits QScrollView [p. 259].

Inherited by QDataTable [Databases with Qt].

Public Members

- **QTable** (QWidget * parent = 0, const char * name = 0)
- **QTable** (int numRows, int numCols, QWidget * parent = 0, const char * name = 0)
- **~QTable** ()
- QHeader * **horizontalHeader** () const
- QHeader * **verticalHeader** () const
- enum **SelectionMode** { Single, Multi, SingleRow, MultiRow, NoSelection }
- virtual void **setSelectionMode** (SelectionMode mode)
- SelectionMode **selectionMode** () const
- virtual void **setItem** (int row, int col, QTableItem * item)
- virtual void **setText** (int row, int col, const QString & text)
- virtual void **setPixmap** (int row, int col, const QPixmap & pix)
- virtual QTableItem * **item** (int row, int col) const
- virtual QString **text** (int row, int col) const
- virtual QPixmap **pixmap** (int row, int col) const
- virtual void **clearCell** (int row, int col)
- virtual QRect **cellGeometry** (int row, int col) const
- virtual int **columnWidth** (int col) const
- virtual int **rowHeight** (int row) const
- virtual int **columnPos** (int col) const
- virtual int **rowPos** (int row) const
- virtual int **columnAt** (int x) const
- virtual int **rowAt** (int y) const
- virtual int **numRows** () const
- virtual int **numCols** () const
- void **updateCell** (int row, int col)
- int **currentRow** () const
- int **currentColumn** () const
- void **ensureCellVisible** (int row, int col)
- bool **isSelected** (int row, int col) const

- bool **isRowSelected** (int row, bool full = FALSE) const
- bool **isColumnSelected** (int col, bool full = FALSE) const
- int **numSelections** () const
- QTableSelection **selection** (int num) const
- virtual int **addSelection** (const QTableSelection & s)
- virtual void **removeSelection** (const QTableSelection & s)
- virtual void **removeSelection** (int num)
- virtual int **currentSelection** () const
- bool **showGrid** () const
- bool **columnMovingEnabled** () const
- bool **rowMovingEnabled** () const
- virtual void **sortColumn** (int col, bool ascending = TRUE, bool wholeRows = FALSE)
- bool **sorting** () const
- virtual void **takeItem** (QTableItem * i)
- virtual void **setCellWidget** (int row, int col, QWidget * e)
- virtual QWidget * **cellWidget** (int row, int col) const
- virtual void **clearCellWidget** (int row, int col)
- virtual QRect **cellRect** (int row, int col) const
- virtual void **paintCell** (QPainter * p, int row, int col, const QRect & cr, bool selected)
- virtual void **paintCell** (QPainter * p, int row, int col, const QRect & cr, bool selected, const QColorGroup & cg)
- virtual void **paintFocus** (QPainter * p, const QRect & cr)
- bool **isReadOnly** () const
- bool **isRowReadOnly** (int row) const
- bool **isColumnReadOnly** (int col) const
- void **repaintSelections** ()
- enum **FocusStyle** { FollowStyle, SpreadSheet }
- virtual void **setFocusStyle** (FocusStyle fs)
- FocusStyle **focusStyle** () const

Public Slots

- virtual void **setNumRows** (int r)
- virtual void **setNumCols** (int r)
- virtual void **setShowGrid** (bool b)
- virtual void **hideRow** (int row)
- virtual void **hideColumn** (int col)
- virtual void **showRow** (int row)
- virtual void **showColumn** (int col)
- virtual void **setColumnWidth** (int col, int w)
- virtual void **setRowHeight** (int row, int h)
- virtual void **adjustColumn** (int col)
- virtual void **adjustRow** (int row)
- virtual void **setColumnStretchable** (int col, bool stretch)
- virtual void **setRowStretchable** (int row, bool stretch)
- bool **isColumnStretchable** (int col) const
- bool **isRowStretchable** (int row) const
- virtual void **setSorting** (bool b)
- virtual void **swapRows** (int row1, int row2, bool swapHeader = FALSE)

- virtual void **swapColumns** (int col1, int col2, bool swapHeader = FALSE)
- virtual void **swapCells** (int row1, int col1, int row2, int col2)
- virtual void **setLeftMargin** (int m)
- virtual void **setTopMargin** (int m)
- virtual void **setCurrentCell** (int row, int col)
- void **clearSelection** (bool repaint = TRUE)
- virtual void **setColumnMovingEnabled** (bool b)
- virtual void **setRowMovingEnabled** (bool b)
- virtual void **setReadOnly** (bool b)
- virtual void **setRowReadOnly** (int row, bool ro)
- virtual void **setColumnReadOnly** (int col, bool ro)
- virtual void **setDragEnabled** (bool b)
- bool **dragEnabled** () const
- virtual void **insertRows** (int row, int count = 1)
- virtual void **insertColumns** (int col, int count = 1)
- virtual void **removeRow** (int row)
- virtual void **removeRows** (const QMemArray<int> & rows)
- virtual void **removeColumn** (int col)
- virtual void **removeColumns** (const QMemArray<int> & cols)
- virtual void **editCell** (int row, int col, bool replace = FALSE)

Signals

- void **currentChanged** (int row, int col)
- void **clicked** (int row, int col, int button, const QPoint & mousePos)
- void **doubleClicked** (int row, int col, int button, const QPoint & mousePos)
- void **pressed** (int row, int col, int button, const QPoint & mousePos)
- void **selectionChanged** ()
- void **valueChanged** (int row, int col)
- void **contextMenuRequested** (int row, int col, const QPoint & pos)
- void **dropped** (QDropEvent * e)

Properties

- bool **columnMovingEnabled** — whether columns can be moved by the user
- FocusStyle **focusStyle** — how the current (focus) cell is drawn
- int **numCols** — the number of columns in the table
- int **numRows** — the number of rows in the table
- bool **readOnly** — whether the table is read-only
- bool **rowMovingEnabled** — whether rows can be moved by the user
- SelectionMode **selectionMode** — the current selection mode
- bool **showGrid** — whether the table's grid is displayed
- bool **sorting** — whether a click on the header of a column sorts that column

Protected Members

- enum **EditMode** { NotEditing, Editing, Replacing }
- virtual void **drawContents** (QPainter * p, int cx, int cy, int cw, int ch)
- void **setEditMode** (EditMode mode, int row, int col)
- virtual void **contentsDragEnterEvent** (QDragEnterEvent * e)
- virtual void **contentsDragMoveEvent** (QDragMoveEvent * e)
- virtual void **contentsDragLeaveEvent** (QDragLeaveEvent * e)
- virtual void **contentsDropEvent** (QDropEvent * e)
- virtual QDragObject * **dragObject** ()
- virtual void **startDrag** ()
- virtual void **paintEmptyArea** (QPainter * p, int cx, int cy, int cw, int ch)
- virtual void **activateNextCell** ()
- virtual QWidget * **createEditor** (int row, int col, bool initFromCell) const
- virtual void **setCellContentFromEditor** (int row, int col)
- virtual QWidget * **beginEdit** (int row, int col, bool replace)
- virtual void **endEdit** (int row, int col, bool accept, bool replace)
- virtual void **resizeData** (int len)
- virtual void **insertWidget** (int row, int col, QWidget * w)
- int **indexOf** (int row, int col) const
- bool **isEditing** () const
- EditMode **editMode** () const
- int **currEditRow** () const
- int **currEditCol** () const

Protected Slots

- virtual void **columnWidthChanged** (int col)
- virtual void **rowHeightChanged** (int row)
- virtual void **columnIndexChanged** (int section, int fromIndex, int toIndex)
- virtual void **rowIndexChanged** (int section, int fromIndex, int toIndex)
- virtual void **columnClicked** (int col)

Detailed Description

The QTable class provides a flexible editable table widget.

QTable is easy to use, although it does have a large API because of the comprehensive functionality that it provides. QTable includes functions for manipulating headers, rows and columns, cells and selections. QTable also provides in-place editing and drag and drop, as well as a useful set of signals. QTable efficiently supports very large tables, for example, tables one million by one million cells are perfectly possible. QTable is economical with memory, using none for unused cells.

```
QTable *table = new QTable( 100, 250, this );
table->setPixmap( 3, 2, pix );
table->setText( 3, 2, "A pixmap" );
```

The first line constructs the table specifying its size in rows and columns. We then insert a pixmap and some text into the *same* cell, with the pixmap appearing to the left of the text. By default a vertical header appears at the left of the table showing row numbers and a horizontal header appears at the top of the table showing column numbers. (The numbers displayed start at 1, although row and column numbers within QTable begin at 0.)

If you want to use mouse tracking call setMouseTracking(TRUE) on the *viewport*; (see QScrollView).

Headers

QTable supports a header column, e.g. to display row numbers, and a header row, e.g to display column titles. To set row or column labels use QHeader::setLabel() on the pointers returned by verticalHeader() and horizontalHeader() respectively. The vertical header is displayed within the table's left margin whose width is set with setLeftMargin(). The horizontal header is displayed within the table's top margin whose height is set with setTopMargin(). The table's grid can be switched off with setShowGrid(). If you want to hide a vertical header call hide(), and call setTopMargin(0) so that the area the header would have occupied is reduced to zero size.

Header labels are indexed via their section numbers. Note that the default behavior of QHeader regarding section numbers is overriden for QTable. See the explanation below in Rows and Columns section in the discussion of moving columns and rows.

Rows and Columns

Row and column sizes are set with setRowHeight() and setColumnWidth(). If you want a row high enough to show the tallest item in its entirety, use adjustRow(). Similarly, to make a column wide enough to show the widest item use adjustColumn(). If you want the row height and column width to adjust automatically as the height and width of the table changes use setRowStretchable() and setColumnStretchable().

Rows and columns can be hidden and shown with hideRow(), hideColumn(), showRow() and showColumn(). New rows and columns are inserted using insertRows() and insertColumns(). Additional rows and columns are added at the bottom (rows) or right (columns) if you set setNumRows() or setNumCols() to be larger than numRows() or numCols(). Existing rows and columns are removed with removeRow() and removeColumn(). Multiple rows and columns can be removed with removeRows() and removeColumns().

Rows and columns can be set to be moveable, i.e. the user can drag them to reorder them, using rowMovingEnabled() and columnMovingEnabled(). For performance reasons, the default behavior of QHeader section numbers is overwritten by QTable. Currently in QTable, when a row or column is dragged and reordered, the section number is also changed to its new position. Therefore, there is no difference between the section and the index fields in QHeader. The QTable QHeader classes do not provide a mechanism for indexing independent of the user interface ordering.

The table can be sorted using sortColumn(). Users can sort a column by clicking its header if setSorting() is set to TRUE. Rows can be swapped with swapRows(), columns with swapColumns() and cells with swapCells().

For editable tables (see setReadOnly()) you can set the read-only property of individual rows and columns with setRowReadOnly() and setColumnReadOnly(). (Whether a cell is editable or read-only depends on these settings and the cell's QTableItem::EditType.)

The row and column which have the focus are returned by currentRow() and currentColumn() respectively.

Although many QTable functions operate in terms of rows and columns the indexOf() function returns a single integer identifying a particular cell.

Cells

All of a QTable's cells are empty when the table is constructed.

There are two approaches to populating the table's cells. The first and simplest approach is to use QTableItems or QTableItem subclasses. The second approach doesn't use QTableItems at all which is useful for very large sparse tables but requires you to reimplement a number of functions. We'll look at each approach in turn.

To put a string in a cell use setText(). This function will create a new QTableItem for the cell if one doesn't already exist, and displays the text in it. By default the table item's widget will be a QLineEdit. A pixmap may be put in a cell with setPixmap(), which also creates a table item if required. A cell may contain *both* a pixmap and text; the pixmap is displayed to the left of the text. Another approach is to construct a QTableItem or QTableItem subclass, set its properties, then insert it into a cell with setItem().

If you want cells which contain comboboxes use the QComboTableItem class. Similarly if you require cells con-

taining checkboxes use the QCheckTableItem class. These table items look and behave just like the combobox or checkbox widgets but consume far less memory.

```
for ( int j = 0; j < numRows; ++j )
    table.setItem( j, 1, new QCheckTableItem( &table, "Check me" ) );
```

In the example above we create a column of QCheckTableItems and insert them into the table using setItem().

QTable takes ownership of its QTableItems and will delete them when the table itself is destroyed. You can take ownership of a table item using takeItem() which you use to move a cell's contents from one cell to another, either within the same table, or from one table to another. (See also, swapCells()).

In-place editing of the text in QTableItems, and the values in QComboTableItems and QCheckTableItems works automatically. Cells may be editable or read-only, see QTableItem::EditType.

The contents of a cell can be retrieved as a QTableItem using item(), or as a string with text() or as a pixmap (if there is one) with pixmap(). A cell's bounding rectangle is given by cellGeometry(). Use updateCell() to repaint a cell, for example to clear away a cell's visual representation after it has been deleted with clearCell(). The table can be forced to scroll to show a particular cell with ensureCellVisible(). The isSelected() function indicates if a cell is selected.

It is possible to use your own widget as a cell's widget using setCellWidget(), but subclassing QTableItem might be a simpler approach. The cell's widget (if there is one) can be removed with clearCellWidget().

Large tables

For large, sparse, tables using QTableItems or other widgets is inefficient. The solution is to *draw* the cell as it should appear and to create and destroy cell editors on demand.

This approach requires that you reimplement various functions. Reimplement paintCell() to display your data, and createEditor() and setCellContentFromEditor() to facilitate in-place editing. It is very important to reimplement resizeData() to have no functionality, to prevent QTable from attempting to create a huge array. You will also need to reimplement item(), setItem(), clearCell(), and insertWidget(), cellWidget() and clearCellWidget(). If you wish to support sorting you should also reimplement swapRows(), swapCells() and possibly swapColumns().

If you represent active cells with a dictionary of QTableItems and QWidgets, i.e. only store references to cells that are actually used, most of the functions can be implemented with a single line of code. (See the table/bigtable/main.cpp example.)

For more information on cells see the QTableItem documenation.

Selections

QTable's support single selection, multi-selection (multiple cells) or no selection. The selection mode is set with setSelectionMode(). Use isSelected() to determine if a particular cell is selected, and isRowSelected() and isColumnSelected() to see if a row or column is selected.

QTable's support multiple selections. You can programmatically select cells with addSelection(). The number of selections is given by numSelections(). The current selection is returned by currentSelection(). You can remove a selection with removeSelection() and remove all selections with clearSelection(). Selections are QTableSelection objects.

Signals

When the user clicks a cell the currentChanged() signal is emitted. You can also connect to the lower level clicked(), doubleClicked() and pressed() signals. If the user changes the selection the selectionChanged() signal is emitted; similarly if the user changes a cell's value the valueChanged() signal is emitted. If the user right-clicks (or presses

the platform-specific key sequence) the contextMenuRequested() signal is emitted. If the user drops a drag and drop object the dropped() signal is emitted with the drop event.

See also Advanced Widgets.

Member Type Documentation

QTable::EditMode

- `QTable::NotEditing` - No cell is currently being edited.
- `QTable::Editing` - A cell is currently being edited. The editor was initialised with the cell's contents.
- `QTable::Replacing` - A cell is currently being edited. The editor was not initialised with the cell's contents.

QTable::FocusStyle

Specifies how the current cell (focus cell) is drawn.

- `QTable::FollowStyle` - The current cell is drawn according to the current style and the cell's background is also drawn selected, if the current cell is position within a selection
- `QTable::SpreadSheet` - The current cell is drawn as in a spread sheet. This means, it is indicated by a black rectangle around the cell, and the background of the current cell is always drawn with the widget's base color - even when selected.

QTable::SelectionMode

- `QTable::NoSelection` - No cell can be selected by the user.
- `QTable::Single` - The user may only select a single range of cells.
- `QTable::Multi` - The user may select multiple ranges of cells.
- `QTable::SingleRow` - The user may select one row at once (there is always the row of the current item selected)
- `QTable::MultiRow` - The user may select multiple rows

Member Function Documentation

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

Creates an empty table object called *name* as a child of *parent*.

Call setNumRows() and setNumCols() to set the table size before populating the table if you're using QTableItems.

See also QWidget::clearWFlags() [p. 424] and Qt::WidgetFlags [Additional Functionality with Qt].

QTable::QTable (int numRows, int numCols, QWidget * parent = 0, const char * name = 0)

Constructs an empty table called *name* with *numRows* rows and *numCols* columns. The table is a child of *parent*.

If you're using QTableItems to populate the table's cells, you can create QTableItem, QComboTableItem and QCheckTableItem items and insert them into the table using setItem(). (See the notes on large tables for an alternative to using QTableItems.)

See also QWidget::clearWFlags() [p. 424] and Qt::WidgetFlags [Additional Functionality with Qt].

QTable::~QTable ()

Destructor. Deletes all the resources used by a QTable object, including all QTableItems and their widgets.

void QTable::activateNextCell () [virtual protected]

This function is called to activate the next cell if in-place editing was finished by pressing the Return key.

The default behaviour is to move from top to bottom, i.e. move to the cell beneath the cell being edited. Reimplement this function if you want different behaviour, e.g. moving from left to right.

int QTable::addSelection (const QTableSelection & s) [virtual]

Adds a selection described by *s* to the table and returns its number or -1 if the selection is invalid.

Remember to call QTableSelection::init() and QTableSelection::expandTo() to make the selection valid (see also QTableSelection::isActive()).

See also numSelections() [p. 340], removeSelection() [p. 342] and clearSelection() [p. 333].

void QTable::adjustColumn (int col) [virtual slot]

Resizes column *col* so that the column width is wide enough to display the widest item the column contains.

See also adjustRow() [p. 332].

void QTable::adjustRow (int row) [virtual slot]

Resizes row *row* so that the row height is tall enough to display the tallest item the row contains.

See also adjustColumn() [p. 332].

QWidget * QTable::beginEdit (int row, int col, bool replace) [virtual protected]

This function is called to start in-place editing of the cell at *row*, *col*. Editing is achieved by creating an editor (createEditor() is called) and setting the cell's editor with setCellWidget() to the newly created editor. (After editing is complete endEdit() will be called to replace the cell's content with the editor's content.) If *replace* is TRUE the editor will be initialized with the cell's content (if any), i.e. the user will be modifying the original cell content; otherwise the user will be entering new data.

See also endEdit() [p. 337].

QRect QTable::cellGeometry (int row, int col) const [virtual]

Returns the bounding rectangle of the cell at *row*, *col* in content coordinates.

QRect QTable::cellRect (int row, int col) const [virtual]

Returns the geometry of cell *row*, *col* in the cell's coordinate system. This is a convenience function useful in paintCell(). It is equivalent to QRect(QPoint(0,0), cellGeometry(row, col).size());

See also cellGeometry() [p. 332].

QWidget * QTable::cellWidget (int row, int col) const [virtual]

Returns the widget that has been set for the cell at *row*, *col*, or 0 if no widget has been set.

If you don't use QTableItems you may need to reimplement this function: see the notes on large tables.

See also clearCellWidget() [p. 333] and setCellWidget() [p. 343].

void QTable::clearCell (int row, int col) [virtual]

Removes the QTableItem at *row*, *col*.

If you don't use QTableItems you may need to reimplement this function: see the notes on large tables.

void QTable::clearCellWidget (int row, int col) [virtual]

Removes the widget (if there is one) set for the cell at *row*, *col*.

If you don't use QTableItems you may need to reimplement this function: see the notes on large tables.

This function deletes the widget at *row*, *col*. Note that the widget is not deleted immediately but QObject::deleteLater() is called on the widget to avoid problems with timing issues.

See also cellWidget() [p. 333] and setCellWidget() [p. 343].

void QTable::clearSelection (bool repaint = TRUE) [slot]

Clears all selections and repaints the appropriate regions if *repaint* is TRUE.

See also removeSelection() [p. 342].

void QTable::clicked (int row, int col, int button, const QPoint & mousePos) [signal]

This signal is emitted when mouse button *button* is clicked. The cell where the event took place is at *row*, *col*, and the mouse's position is in *mousePos*.

int QTable::columnAt (int x) const [virtual]

Returns the number of the column at position *x*. *x* must be given in content coordinates.

See also columnPos() [p. 334] and rowAt() [p. 342].

void QTable::columnClicked (int col) [virtual protected slot]

This function is called when the column *col* has been clicked. The default implementation sorts this column if sorting() is TRUE.

void QTable::columnIndexChanged (int section, int fromIndex, int toIndex) [virtual protected slot]

This function is called when column order is to be changed, i.e. when the user moved the column header *section* from *fromIndex* to *toIndex*.

If you want to change the column order programmatically, call swapRows() or swapColumns();

See also QHeader::indexChange() [Additional Functionality with Qt] and rowIndexChanged() [p. 343].

bool QTable::columnMovingEnabled () const

Returns TRUE if columns can be moved by the user; otherwise returns FALSE. See the "columnMovingEnabled" [p. 349] property for details.

int QTable::columnPos (int col) const [virtual]

Returns the x-coordinate of the column *col* in content coordinates.

See also columnAt() [p. 333] and rowPos() [p. 343].

int QTable::columnWidth (int col) const [virtual]

Returns the width of column *col*.

See also setColumnWidth() [p. 344] and rowHeight() [p. 342].

void QTable::columnWidthChanged (int col) [virtual protected slot]

This function should be called whenever the column width of *col* has been changed. It updates the geometry of any affected columns and repaints the table to reflect the changes it has made.

void QTable::contentsDragEnterEvent (QDragEnterEvent * e) [virtual protected]

This event handler is called whenever a QTable object receives a QDragEnterEvent *e*, i.e. when the user pressed the mouse button to drag something.

The focus is moved to the cell where the QDragEnterEvent occurred.

Reimplemented from QScrollView [p. 267].

void QTable::contentsDragLeaveEvent (QDragLeaveEvent * e) [virtual protected]

This event handler is called when a drag activity leaves *this* QTable object with event *e*.

Reimplemented from QScrollView [p. 267].

void QTable::contentsDragMoveEvent (QDragMoveEvent * e) [virtual protected]

This event handler is called whenever a QTable object receives a QDragMoveEvent *e*, i.e. when the user actually drags the mouse.

The focus is moved to the cell where the QDragMoveEvent occurred.

Reimplemented from QScrollView [p. 267].

void QTable::contentsDropEvent (QDropEvent * e) [virtual protected]

This event handler is called when the user ends a drag and drop by dropping something onto *this* QTable and thus triggers the drop event, *e*.

Reimplemented from QScrollView [p. 267].

void QTable::contextMenuRequested (int row, int col, const QPoint & pos) [signal]

This signal is emitted when the user invokes a context menu with the right mouse button (or with a system-specific keyboard key). The cell where the event took place is at *row*, *col*. *pos* is the position where the context menu will appear in the global coordinate system.

QWidget * QTable::createEditor (int row, int col, bool initFromCell) const [virtual protected]

This function returns the widget which should be used as an editor for the contents of the cell at *row*, *col*.

If *initFromCell* is TRUE, the editor is used to edit the current contents of the cell (so the editor widget should be initialized with this content). If *initFromCell* is FALSE, the content of the cell is replaced with the new content which the user entered into the widget created by this function.

The default functionality is as follows: if *initFromCell* is TRUE or the cell has a QTableItem and the table item's QTableItem::isReplaceable() is FALSE then the cell is asked to create an appropriate editor (using QTableItem::createEditor()). Otherwise a QLineEdit is used as the editor.

If you want to create your own editor for certain cells, implement a custom QTableItem subclass and reimplement QTableItem::createEditor().

If you are not using QTableItems and you don't want to use a QLineEdit as the default editor, subclass QTable and reimplement this function with code like this:

```
QTableItem *i = item( row, col );
if ( initFromCell || ( i && !i->isReplaceable() ) )
    // If we had a QTableItem ask the base class to create the editor
    return QTable::createEditor( row, col, initFromCell );
else
    return ...(create your editor)
```

Ownership of the editor widget is transferred to the caller.

If you reimplement this function return 0 for read-only cells. You will need to reimplement setCellContentFromEditor() to retrieve the data the user entered.

See also QTableItem::createEditor() [p. 354].

int QTable::currEditCol () const [protected]

Returns the current edited column

int QTable::currEditRow () const [protected]

Returns the current edited row

void QTable::currentChanged (int row, int col) [signal]

This signal is emitted when the current cell has changed to *row*, *col*.

int QTable::currentColumn () const

Returns the current column.

See also currentRow() [p. 336].

int QTable::currentRow () const

Returns the current row.

See also currentColumn() [p. 336].

int QTable::currentSelection () const [virtual]

Returns the number of the current selection or -1 if there is no current selection.

See also numSelections() [p. 340].

void QTable::doubleClicked (int row, int col, int button, const QPoint & mousePos) [signal]

This signal is emitted when mouse button *button* is double-clicked. The cell where the event took place is at *row*, *col*, and the mouse's position is in *mousePos*.

bool QTable::dragEnabled () const [slot]

If this function returns TRUE, the table supports dragging.

See also setDragEnabled() [p. 344].

QDragObject * QTable::dragObject () [virtual protected]

If the user presses the mouse on a selected cell, starts moving (i.e. dragging), and dragEnabled() is TRUE, this function is called to obtain a drag object. A drag using this object begins immediately unless dragObject() returns 0.

By default this function returns 0. You might reimplement it and create a QDragObject depending on the selected items.

See also dropped() [p. 337].

void QTable::drawContents (QPainter * p, int cx, int cy, int cw, int ch) [virtual protected]

Draws the table contents on the painter *p*. This function is optimized so that it only draws the cells inside the *cw* pixels wide and *ch* pixels high clipping rectangle at position *cx*, *cy*.

Additionally, drawContents() highlights the current cell.

Reimplemented from QScrollView [p. 269].

void QTable::dropped (QDropEvent * e) [signal]

This signal is emitted when a drop event occurred on the table.

e contains information about the drop.

void QTable::editCell (int row, int col, bool replace = FALSE) [virtual slot]

Starts editing the cell at *row*, *col*.

If *replace* is TRUE the content of this cell will be replaced by the content of the editor when editing is finished, i.e. the user will be entering new data; otherwise the current content of the cell (if any) will be modified in the editor.

See also beginEdit() [p. 332].

EditMode QTable::editMode () const [protected]

Returns the current edit mode

void QTable::endEdit (int row, int col, bool accept, bool replace) [virtual protected]

This function is called when in-place editing of the cell at *row*, *col* is requested to stop.

If the cell is not being edited or *accept* is FALSE the function returns and the cell's contents are left unchanged.

If *accept* is TRUE the content of the editor must be transferred to the relevant cell. If *replace* is TRUE the current content of this cell should be replaced by the content of the editor (this means removing the current QTableItem of the cell and creating a new one for the cell). Otherwise (if possible) the content of the editor should just be set to the existing QTableItem of this cell.

If the cell contents should be replaced or if no QTableItem exists for the cell, setCellContentFromEditor() is called. Otherwise QTableItem::setContentFromEditor() is called on the QTableItem of the cell.

Finally clearCellWidget() is called to remove the editor widget.

See also setCellContentFromEditor() [p. 343] and beginEdit() [p. 332].

void QTable::ensureCellVisible (int row, int col)

Scrolls the table until the cell at *row*, *col* becomes visible.

FocusStyle QTable::focusStyle () const

Returns how the current (focus) cell is drawn. See the "focusStyle" [p. 349] property for details.

void QTable::hideColumn (int col) [virtual slot]

Hides column *col*.

See also showColumn() [p. 347] and hideRow() [p. 337].

void QTable::hideRow (int row) [virtual slot]

Hides row *row*.

See also showRow() [p. 347] and hideColumn() [p. 337].

QHeader * QTable::horizontalHeader () const

Returns the table's top QHeader.

This header contains the column labels.

To modify a column label use QHeader::setLabel(), e.g.

```
horizontalHeader()->setLabel( 0, tr( "File" ) );
```

See also verticalHeader() [p. 349], setTopMargin() [p. 346] and QHeader [Additional Functionality with Qt].

int QTable::indexOf (int row, int col) const [protected]

Returns a single integer which identifies a particular *row* and *col* by mapping the 2D table to a 1D array.

This is useful, for example, if you have a sparse table and want to use a QIntDict to map integers to the cells that are used.

void QTable::insertColumns (int col, int count = 1) [virtual slot]

Inserts *count* empty columns at column *col*.

See also insertRows() [p. 338] and removeColumn() [p. 341].

void QTable::insertRows (int row, int count = 1) [virtual slot]

Inserts *count* empty rows at row *row*.

See also insertColumns() [p. 338] and removeRow() [p. 341].

void QTable::insertWidget (int row, int col, QWidget * w) [virtual protected]

Inserts widget *w* at *row*, *col* into the internal datastructure. See the documentation of setCellWidget() for further details.

If you don't use QTableItems you may need to reimplement this function: see the notes on large tables.

bool QTable::isColumnReadOnly (int col) const

Returns whether column *col* is read-only.

Whether a cell in this column is editable or read-only depends on the cell's EditType, and this setting: see QTableItem::EditType.

See also setColumnReadOnly() [p. 344] and isRowReadOnly() [p. 339].

bool QTable::isColumnSelected (int col, bool full = FALSE) const

Returns TRUE if column *col* is selected; otherwise returns FALSE.

If *full* is FALSE (the default), 'col is selected' means that at least one cell in the column is selected. If *full* is TRUE, then 'col is selected' means every cell in the column is selected.

See also isRowSelected() [p. 339] and isSelected() [p. 339].

bool QTable::isColumnStretchable (int col) const [slot]

Returns TRUE if column *col* is stretchable; otherwise returns FALSE.

See also setColumnStretchable() [p. 344] and isRowStretchable() [p. 339].

bool QTable::isEditing () const [protected]

Returns TRUE if the EditMode is Editing or Replacing. Returns FALSE if the EditMode is NotEditing.

See also QTable::EditMode [p. 331].

bool QTable::isReadOnly () const

Returns TRUE if the table is read-only; otherwise returns FALSE. See the "readOnly" [p. 349] property for details.

bool QTable::isRowReadOnly (int row) const

Returns whether row *row* is read-only.

Whether a cell in this row is editable or read-only depends on the cell's EditType, and this setting: see QTableItem::EditType.

See also setRowReadOnly() [p. 346] and isColumnReadOnly() [p. 338].

bool QTable::isRowSelected (int row, bool full = FALSE) const

Returns TRUE if row *row* is selected; otherwise returns FALSE.

If *full* is FALSE (the default), 'row is selected' means that at least one cell in the row is selected. If *full* is TRUE, then 'row is selected' means every cell in the row is selected.

See also isColumnSelected() [p. 338] and isSelected() [p. 339].

bool QTable::isRowStretchable (int row) const [slot]

Returns TRUE if row *row* is stretchable; otherwise returns FALSE.

See also setRowStretchable() [p. 346] and isColumnStretchable() [p. 339].

bool QTable::isSelected (int row, int col) const

Returns TRUE if the cell at *row*, *col* is selected; otherwise returns FALSE.

See also isRowSelected() [p. 339] and isColumnSelected() [p. 338].

QTableItem * QTable::item (int row, int col) const [virtual]

Returns the QTableItem representing the contents of the cell at *row*, *col*.

If *row* or *col* are out of range or no content has been set for this cell, item() returns 0.

If you don't use QTableItems you may need to reimplement this function: see the notes on large tables.

See also setItem() [p. 345].

int QTable::numCols () const [virtual]

Returns the number of columns in the table. See the "numCols" [p. 349] property for details.

Reimplemented in QDataTable.

int QTable::numRows () const [virtual]

Returns the number of rows in the table. See the "numRows" [p. 349] property for details.

Reimplemented in QDataTable.

int QTable::numSelections () const

Returns the number of selections.

See also currentSelection() [p. 336].

void QTable::paintCell (QPainter * p, int row, int col, const QRect & cr, bool selected, const QColorGroup & cg) [virtual]

Paints the cell at *row*, *col* on the painter *p*. The painter has already been translated to the cell's origin. *cr* describes the cell coordinates in the content coordinate system.

If *selected* is TRUE the cell is highlighted.

cg is the colorgroup which should be used to draw the cell content.

If you want to draw custom cell content, for example right-aligned text, you must either reimplement paintCell(), or subclass QTableItem and reimplement QTableItem::paint() to do the custom drawing.

If you're using a QTableItem subclass, for example, to store a data structure, then reimplementing QTableItem::paint() may be the best approach. For data you want to draw immediately, e.g. data retrieved from a database, it is probably best to reimplement paintCell(). Note that if you reimplement paintCell, i.e. don't use QTableItems, you will have to reimplement other functions: see the notes on large tables.

Note that the painter is not clipped by default in order to get maximum efficiency. If you want clipping, use code like this:

```
p->setClipRect( cellRect(row, col), QPainter::ClipPainter );
//... your drawing code
p->setClipping( FALSE );
```

void QTable::paintCell (QPainter * p, int row, int col, const QRect & cr, bool selected) [virtual]

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

Use the other paintCell() function. This function is only included for backwards compatibilty.

void QTable::paintEmptyArea (QPainter * p, int cx, int cy, int cw, int ch) [virtual protected]

This function fills the *cw* pixels wide and *ch* pixels high rectangle starting at position *cx*, *cy* with the background color using the painter *p*.

paintEmptyArea() is invoked by drawContents() to erase or fill unused areas.

void QTable::paintFocus (QPainter * p, const QRect & cr) [virtual]

Draws the focus rectangle of the current cell (see currentRow(), currentColumn()).

The painter *p* is already translated to the cell's origin, while *cr* specifies the cell's geometry in content coordinates.

QPixmap QTable::pixmap (int row, int col) const [virtual]

Returns the pixmap set for the cell at *row*, *col*, or a null-pixmap if the cell contains no pixmap.

See also setPixmap() [p. 345].

void QTable::pressed (int row, int col, int button, const QPoint & mousePos) [signal]

This signal is emitted when mouse button *button* is pressed. The cell where the event took place is at *row*, *col*, and the mouse's position is in *mousePos*.

void QTable::removeColumn (int col) [virtual slot]

Removes column *col*, and deletes all its cells including any table items and widgets the cells may contain.

The array passed in has to contain only valid columns (in the range from 0 to numCols() - 1), no duplicates and must be sorted in ascending order.

See also removeColumns() [p. 341], hideColumn() [p. 337], insertColumns() [p. 338] and removeRow() [p. 341].

void QTable::removeColumns (const QMemArray<int> & cols) [virtual slot]

Removes the columns listed in the array *cols*, and deletes all their cells including any table items and widgets the cells may contain.

See also removeColumn() [p. 341], insertColumns() [p. 338] and removeRows() [p. 342].

void QTable::removeRow (int row) [virtual slot]

Removes row *row*, and deletes all its cells including any table items and widgets the cells may contain.

See also hideRow() [p. 337], insertRows() [p. 338], removeColumn() [p. 341] and removeRows() [p. 342].

void QTable::removeRows (const QMemArray<int> & rows) [virtual slot]

Removes the rows listed in the array *rows*, and deletes all their cells including any table items and widgets the cells may contain.

The array passed in has to contain only valid rows (in the range from 0 to numRows() - 1), no duplicates and must be sorted in ascending order.

See also removeRow() [p. 341], insertRows() [p. 338] and removeColumns() [p. 341].

void QTable::removeSelection (const QTableSelection & s) [virtual]

If the table has a selection, *s*, this selection is removed from the table.

See also addSelection() [p. 332] and numSelections() [p. 340].

void QTable::removeSelection (int num) [virtual]

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

Removes selection number *num* from the table.

See also numSelections() [p. 340], addSelection() [p. 332] and clearSelection() [p. 333].

void QTable::repaintSelections ()

Repaints all selections

void QTable::resizeData (int len) [virtual protected]

This is called when QTable's internal array needs to be resized to *len* elements.

If you don't use QTableItems you should reimplement this as an empty method to avoid wasting memory. See the notes on large tables [p. 330] for further details.

int QTable::rowAt (int y) const [virtual]

Returns the number of the row at position *y*. *y* must be given in content coordinates.

See also rowPos() [p. 343] and columnAt() [p. 333].

int QTable::rowHeight (int row) const [virtual]

Returns the height of row *row*.

See also setRowHeight() [p. 345] and columnWidth() [p. 334].

void QTable::rowHeightChanged (int row) [virtual protected slot]

This function should be called whenever the row height of *row* has been changed. It updates the geometry of any affected rows and repaints the table to reflect the changes it has made.

void QTable::rowIndexChanged (int section, int fromIndex, int toIndex) [virtual protected slot]

This function is called when the order of the rows is to be changed, i.e. the user moved the row header section *section* from *fromIndex* to *toIndex*.

If you want to change the order programmatically, call swapRows() or swapColumns();

See also QHeader::indexChange() [Additional Functionality with Qt] and columnIndexChanged() [p. 334].

bool QTable::rowMovingEnabled () const

Returns TRUE if rows can be moved by the user; otherwise returns FALSE. See the "rowMovingEnabled" [p. 350] property for details.

int QTable::rowPos (int row) const [virtual]

Returns the y-coordinate of the row *row* in content coordinates.

See also rowAt() [p. 342] and columnPos() [p. 334].

QTableSelection QTable::selection (int num) const

Returns selection number *num*, or an empty QTableSelection if *num* is out of range (see QTableSelection::isNull()).

void QTable::selectionChanged () [signal]

This signal is emitted whenever a selection changes.

See also QTableSelection [Additional Functionality with Qt].

SelectionMode QTable::selectionMode () const

Returns the current selection mode. See the "selectionMode" [p. 350] property for details.

void QTable::setCellContentFromEditor (int row, int col) [virtual protected]

This function is called to replace the contents of the cell at *row*, *col* with the contents of the cell's editor. If a QTableItem already exists for this cell, it is removed first (see clearCell()).

If for example, you want to create different QTableItems depending on the contents of the editor, you might reimplement this function.

If you want to work without QTableItems, you will need to reimplement this function to save the data the user entered into your data structure. (See the notes on large tables.)

See also QTableItem::setContentFromEditor() [p. 356] and createEditor() [p. 335].

void QTable::setCellWidget (int row, int col, QWidget * e) [virtual]

Sets the widget *e* to the cell at *row*, *col* and takes care of placing and resizing the widget when the cell geometry changes.

By default widgets are inserted into a vector with numRows() * numCols() elements. In very large tables you will probably want to store the widgets in a data structure that consumes less memory (see the notes on large tables). To support the use of your own data structure this function calls insertWidget() to add the widget to the internal data structure. To use your own data structure reimplement insertWidget(), cellWidget() and clearCellWidget().

void QTable::setColumnMovingEnabled (bool b) [virtual slot]

Sets whether columns can be moved by the user to *b*. See the "columnMovingEnabled" [p. 349] property for details.

void QTable::setColumnReadOnly (int col, bool ro) [virtual slot]

If *ro* is TRUE, column *col* is set to be read-only; otherwise the column is set to be editable.

Whether a cell in this column is editable or read-only depends on the cell's EditType, and this setting: see QTableItem::EditType.

See also isColumnReadOnly() [p. 338], setRowReadOnly() [p. 346] and readOnly [p. 349].

void QTable::setColumnStretchable (int col, bool stretch) [virtual slot]

If *stretch* is TRUE, column *col* is set to be stretchable; otherwise column *col* is set to be unstretchable.

If the table widget's width decreases or increases stretchable columns will grow narrower or wider to fit the space available as completely as possible. The user cannot manually resize stretchable columns.

See also isColumnStretchable() [p. 339], setRowStretchable() [p. 346] and adjustColumn() [p. 332].

void QTable::setColumnWidth (int col, int w) [virtual slot]

Resizes column *col* to be *w* pixels wide.

See also columnWidth() [p. 334] and setRowHeight() [p. 345].

void QTable::setCurrentCell (int row, int col) [virtual slot]

Moves the focus to the cell at *row*, *col*.

See also currentRow() [p. 336] and currentColumn() [p. 336].

void QTable::setDragEnabled (bool b) [virtual slot]

If *b* is TRUE, the table starts a drag (see dragObject()) when the user presses and moves the mouse on a selected cell.

void QTable::setEditMode (EditMode mode, int row, int col) [protected]

Sets the current edit mode to *mode*, the current edit row to *row* and the current edit column to *col*.

See also EditMode [p. 331].

void QTable::setFocusStyle (FocusStyle fs) [virtual]

Sets how the current (focus) cell is drawn to *fs*. See the "focusStyle" [p. 349] property for details.

void QTable::setItem (int row, int col, QTableItem * item) [virtual]

Inserts the table item *item* into the table at row *row*, column *col*, and repaints the cell. If a table item already exists in this cell it is deleted and replaced with *item*. The table takes ownership of the table item.

If you don't use QTableItems you may need to reimplement this function: see the notes on large tables.

See also item() [p. 340] and takeItem() [p. 348].

Example: table/small-table-demo/main.cpp.

void QTable::setLeftMargin (int m) [virtual slot]

Sets the left margin to be *m* pixels wide.

The verticalHeader(), which displays row labels, occupies this margin.

See also leftMargin() [p. 271], setTopMargin() [p. 346] and verticalHeader() [p. 349].

void QTable::setNumCols (int r) [virtual slot]

Sets the number of columns in the table to *r*. See the "numCols" [p. 349] property for details.

void QTable::setNumRows (int r) [virtual slot]

Sets the number of rows in the table to *r*. See the "numRows" [p. 349] property for details.

void QTable::setPixmap (int row, int col, const QPixmap & pix) [virtual]

Sets the pixmap in the cell at *row*, *col* to *pix*.

If the cell does not contain a table item a QTableItem is created with an EditType of `OnTyping`, otherwise the existing table item's pixmap (if any) is replaced with *pix*.

Note that QComboTableItems and QCheckTableItems don't show pixmaps.

See also pixmap() [p. 341], setText() [p. 346], setItem() [p. 345] and QTableItem::setPixmap() [p. 356].

void QTable::setReadOnly (bool b) [virtual slot]

Sets whether the table is read-only to *b*. See the "readOnly" [p. 349] property for details.

void QTable::setRowHeight (int row, int h) [virtual slot]

Resizes row *row* to be *h* pixels high.

See also rowHeight() [p. 342] and setColumnWidth() [p. 344].

void QTable::setRowMovingEnabled (bool b) [virtual slot]

Sets whether rows can be moved by the user to *b*. See the "rowMovingEnabled" [p. 350] property for details.

void QTable::setRowReadOnly (int row, bool ro) [virtual slot]

If *ro* is TRUE, row *row* is set to be read-only; otherwise the row is set to be editable.

Whether a cell in this row is editable or read-only depends on the cell's EditType, and this setting: see QTableItem::EditType.

See also isRowReadOnly() [p. 339], setColumnReadOnly() [p. 344] and readOnly [p. 349].

void QTable::setRowStretchable (int row, bool stretch) [virtual slot]

If *stretch* is TRUE, row *row* is set to be stretchable; otherwise row *row* is set to be unstretchable.

If the table widget's height decreases or increases stretchable rows will grow shorter or taller to fit the space available as completely as possible. The user cannot manually resize stretchable rows.

See also isRowStretchable() [p. 339] and setColumnStretchable() [p. 344].

void QTable::setSelectionMode (SelectionMode mode) [virtual]

Sets the current selection mode to *mode*. See the "selectionMode" [p. 350] property for details.

void QTable::setShowGrid (bool b) [virtual slot]

Sets whether the table's grid is displayed to *b*. See the "showGrid" [p. 350] property for details.

void QTable::setSorting (bool b) [virtual slot]

Sets whether a click on the header of a column sorts that column to *b*. See the "sorting" [p. 350] property for details.

void QTable::setText (int row, int col, const QString & text) [virtual]

Sets the text in the cell at *row*, *col* to *text*.

If the cell does not contain a table item a QTableItem is created with an EditType of `OnTyping`, otherwise the existing table item's text (if any) is replaced with *text*.

See also text() [p. 348], setPixmap() [p. 345], setItem() [p. 345] and QTableItem::setText() [p. 357].

void QTable::setTopMargin (int m) [virtual slot]

Sets the top margin to be *m* pixels high.

The horizontalHeader(), which displays column labels, occupies this margin.

See also topMargin() [p. 273] and setLeftMargin() [p. 345].

void QTable::showColumn (int col) [virtual slot]

Show column *col*.

See also hideColumn() [p. 337] and showRow() [p. 347].

bool QTable::showGrid () const

Returns TRUE if the table's grid is displayed; otherwise returns FALSE. See the "showGrid" [p. 350] property for details.

void QTable::showRow (int row) [virtual slot]

Show row *row*.

See also hideRow() [p. 337] and showColumn() [p. 347].

void QTable::sortColumn (int col, bool ascending = TRUE, bool wholeRows = FALSE) [virtual]

Sorts column *col*. If *ascending* is TRUE the sort is in ascending order, otherwise the sort is in descending order.

If *wholeRows* is TRUE, entire rows are sorted using swapRows(); otherwise only cells in the column are sorted using swapCells().

Note that if you are not using QTableItems you will need to reimplement swapRows() and swapCells(). (See the notes on large tables.)

See also swapRows() [p. 348].

Reimplemented in QDataTable.

bool QTable::sorting () const

Returns TRUE if a click on the header of a column sorts that column; otherwise returns FALSE. See the "sorting" [p. 350] property for details.

void QTable::startDrag () [virtual protected]

Starts a drag.

Usually you don't need to call or reimplement this function yourself.

See also dragObject() [p. 336].

void QTable::swapCells (int row1, int col1, int row2, int col2) [virtual slot]

Swaps the contents of the cell at *row1*, *col1* with the contents of the cell at *row2*, *col2*.

This function is also called when the table is sorted.

If you don't use QTableItems and want your users to be able to swap cells, you will need to reimplement this function. (See the notes on large tables.)

See also swapColumns() [p. 348] and swapRows() [p. 348].

void QTable::swapColumns (int col1, int col2, bool swapHeader = FALSE) [virtual slot]

Exchanges *col1* with *col2*.

This function is used to swap the positions of two columns. It is called when the user changes the order of columns (see setColumnMovingEnabled(), and when columns are sorted.

If you don't use QTableItems and want your users to be able to swap columns you will need to reimplement this function. (See the notes on large tables.)

If *swapHeader* is TRUE, also the header contents of the columns is swapped.

See also swapCells() [p. 347].

void QTable::swapRows (int row1, int row2, bool swapHeader = FALSE) [virtual slot]

Swaps data of *row1* and *row2*.

This function is used to swap the positions of two rows. It is called when the user changes the order of rows (see setRowMovingEnabled()), and when rows are sorted.

If you don't use QTableItems and want your users to be able to swap rows, e.g. for sorting, you will need to reimplement this function. (See the notes on large tables.)

If *swapHeader* is TRUE, also the header contents of the rows is swapped.

See also swapColumns() [p. 348] and swapCells() [p. 347].

void QTable::takeItem (QTableItem * i) [virtual]

Takes the table item *i* out of the table. This function does *not* delete the table item. You must either delete the table item yourself or put it into a table (using setItem()) which will then take ownership of it.

Use this function if you want to move an item from one cell in a table to another, or to move an item from one table to another, reinserting the item with setItem().

If you want to exchange two cells use swapCells().

QString QTable::text (int row, int col) const [virtual]

Returns the text in cell at *row*, *col*, or a null string if the relevant item does not exist or has no text.

See also setText() [p. 346] and setPixmap() [p. 345].

Reimplemented in QDataTable.

void QTable::updateCell (int row, int col)

Repaints the cell at *row*, *col*.

void QTable::valueChanged (int row, int col) [signal]

This signal is emitted when the user changed the value in the cell at *row*, *col*.

QHeader * QTable::verticalHeader () const

Returns the table's left QHeader.

This header contains the row labels.

See also horizontalHeader() [p. 338], setLeftMargin() [p. 345] and QHeader [Additional Functionality with Qt].

Property Documentation

bool columnMovingEnabled

This property holds whether columns can be moved by the user.

The default is FALSE.

See also rowMovingEnabled [p. 350].

Set this property's value with setColumnMovingEnabled() and get this property's value with columnMovingEnabled().

FocusStyle focusStyle

This property holds how the current (focus) cell is drawn.

The default style is SpreadSheet.

See also QTable::FocusStyle [p. 331].

Set this property's value with setFocusStyle() and get this property's value with focusStyle().

int numCols

This property holds the number of columns in the table.

Set this property's value with setNumCols() and get this property's value with numCols().

See also numRows [p. 349].

int numRows

This property holds the number of rows in the table.

Set this property's value with setNumRows() and get this property's value with numRows().

See also numCols [p. 349].

bool readOnly

This property holds whether the table is read-only.

Whether a cell in the table is editable or read-only depends on the cell's EditType, and this setting: see QTableItem::EditType.

See also QWidget::enabled [p. 461], setColumnReadOnly() [p. 344] and setRowReadOnly() [p. 346].

Set this property's value with setReadOnly() and get this property's value with isReadOnly().

bool rowMovingEnabled

This property holds whether rows can be moved by the user.

The default is FALSE.

See also columnMovingEnabled [p. 349].

Set this property's value with setRowMovingEnabled() and get this property's value with rowMovingEnabled().

SelectionMode selectionMode

This property holds the current selection mode.

The default mode is Multi which allows the user to select multiple ranges of cells.

See also SelectionMode [p. 331] and selectionMode [p. 350].

Set this property's value with setSelectionMode() and get this property's value with selectionMode().

bool showGrid

This property holds whether the table's grid is displayed.

The grid is shown by default.

Set this property's value with setShowGrid() and get this property's value with showGrid().

bool sorting

This property holds whether a click on the header of a column sorts that column.

Set this property's value with setSorting() and get this property's value with sorting().

See also sortColumn() [p. 347].

QTableItem Class Reference

The QTableItem class provides the cell content for QTable cells.

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

`#include <qtable.h>`

Inherits Qt [Additional Functionality with Qt].

Inherited by QComboTableItem [p. 44] and QCheckTableItem [p. 30].

Public Members

- enum **EditType** { Never, OnTyping, WhenCurrent, Always }
- **QTableItem** (QTable * table, EditType et, const QString & text)
- **QTableItem** (QTable * table, EditType et, const QString & text, const QPixmap & p)
- virtual **~QTableItem** ()
- virtual QPixmap **pixmap** () const
- virtual QString **text** () const
- virtual void **setPixmap** (const QPixmap & p)
- virtual void **setText** (const QString & str)
- QTable * **table** () const
- virtual int **alignment** () const
- virtual void **setWordWrap** (bool b)
- bool **wordWrap** () const
- EditType **editType** () const
- virtual QWidget * **createEditor** () const
- virtual void **setContentFromEditor** (QWidget * w)
- virtual void **setReplaceable** (bool b)
- bool **isReplaceable** () const
- virtual QString **key** () const
- virtual QSize **sizeHint** () const
- virtual void **setSpan** (int rs, int cs)
- int **rowSpan** () const
- int **colSpan** () const
- virtual void **setRow** (int r)
- virtual void **setCol** (int c)
- int **row** () const
- int **col** () const
- virtual void **paint** (QPainter * p, const QColorGroup & cg, const QRect & cr, bool selected)
- virtual void **setEnabled** (bool b)
- bool **isEnabled** () const
- virtual int **rtti** () const

Detailed Description

The QTableItem class provides the cell content for QTable cells.

For many applications QTableItems are ideal for presenting and editing the contents of table cells. In situations where you need to create very large tables you may prefer an alternative approach to using QTableItems: see the notes on large tables.

A QTableItem contains a cell's data, by default, a string and a pixmap. The table item also holds the cell's display size and how the data should be aligned. The table item specifies the cell's EditType and the editor used for in-place editing (by default a QLineEdit). If you want checkboxes use QCheckTableItem, and if you want comboboxes use QComboTableItem. The EditType (set in the constructor) determines whether the cell's contents may be edited; setReplaceable() sets whether the cell's contents may be replaced by another cell's contents.

If a pixmap is specified it is displayed to the left of any text. You can change the text or pixmap with setText() and setPixmap() respectively. For text you can use setWordWrap(). A table item's alignment is set in the constructor.

Reimplement createEditor() and setContentFromEditor() if you want to use your own widget instead of a QLineEdit for editing cell contents. Reimplement paint() if you want to display custom content. If you want a checkbox table item use QCheckTableItem, and if you want a combo table item use QComboTableItem.

When sorting table items the key() function is used; by default this returns the table item's text(). Reimplement key() to customize how your table items will sort.

Table items are inserted into a table using QTable::setItem(). If you insert an item into a cell that already contains a table item the original item will be deleted.

Example:

```
for ( int row = 0; row numRows(); row++ ) {
    for ( int col = 0; col numCols(); col++ ) {
        table->setItem( row, col,
            new QTableItem( table, WhenCurrent, QString::number( row * col ) ) );
    }
}
```

You can move a table item from one cell to another, in the same or a different table, using QTable::takeItem() and QTable::setItem() but see also QTable::swapCells().

Table items can be deleted with delete in the standard way; the table and cell will be updated accordingly.

See also Advanced Widgets.

Member Type Documentation

QTableItem::EditType

This enum is used to define whether a cell is editable or read-only (in conjunction with other settings), and how the cell should be displayed.

- `QTableItem::Always` - The cell always *looks* editable.

Using this EditType ensures that the editor created with createEditor() (by default a QLineEdit) is always visible. This has implications for the alignment of the content: the default editor aligns everything (even numbers) to the left whilst numerical values in the cell are by default aligned to the right.

If a cell with the edit type Always looks misaligned you could reimplement createEditor() for these items.

- `QTableItem::WhenCurrent` - The cell *looks* editable only when it has keyboard focus (see QTable::setCurrentCell()).

- `QTableItem::OnTyping` - The cell *looks* editable only when the user types in it or double-clicks it. It resembles the WhenCurrent functionality but is, perhaps, nicer.

The OnTyping edit type is the default when QTableItem objects are created by the convenience functions QTable::setText() and QTable::setPixmap().

- `QTableItem::Never` - The cell is not editable.

The cell is actually editable only if QTable::isRowReadOnly() is FALSE for its row, QTable::isColumnReadOnly() is FALSE for its column, and QTable::isReadOnly() is FALSE.

QComboTableItems have an isEditable() property. This property is used to indicate whether the user may enter their own text or are restricted to choosing one of the choices in the list. QComboTableItems may be interacted with only if they are editable in accordance with their EditType as described above.

Member Function Documentation

QTableItem::QTableItem (QTable * table, EditType et, const QString & text)

Creates a child item of table *table* with text *text*. The item has the EditType *et*.

The table item will use a QLineEdit for its editor, will not word-wrap and will occupy a single cell. Insert the table item into a table with QTable::setItem().

The table takes ownership of the table item, so a table item should not be inserted into more than one table at a time.

QTableItem::QTableItem (QTable * table, EditType et, const QString & text, const QPixmap & p)

Creates a child item of table *table* with text *text* and pixmap *p*. The item has the EditType *et*.

The table item will display the pixmap to the left of the text. It will use a QLineEdit for editing the text, will not word-wrap and will occupy a single cell. Insert the table item into a table with QTable::setItem().

The table takes ownership of the table item, so a table item should not be inserted in more than one table at a time.

QTableItem::~QTableItem () [virtual]

The destructor deletes this item and frees all allocated resources.

If the table item is in a table (i.e. was inserted with setItem()), it will be removed from the table and the cell it occupied.

int QTableItem::alignment () const [virtual]

The alignment function returns how the text contents of the cell are aligned when drawn. The default implementation aligns numbers to the right and any other text to the left.

See also Qt::AlignmentFlags [Additional Functionality with Qt].

int QTableItem::col () const

Returns the column where the table item is located. If the cell spans multiple columns, this function returns the left-most column.

See also row() [p. 355] and setCol() [p. 356].

int QTableItem::colSpan () const

Returns the column span of the table item, usually 1.

See also setSpan() [p. 357] and rowSpan() [p. 355].

QWidget * QTableItem::createEditor () const [virtual]

This virtual function creates an editor which the user can interact with to edit the cell's contents. The default implementation creates a QLineEdit.

If the function returns 0, the cell is read-only.

The returned widget should preferably be invisible, ideally with QTable::viewport() as parent.

If you reimplement this function you'll almost certainly need to reimplement setContentFromEditor(), and may need to reimplement sizeHint().

```
QWidget *ComboItem::createEditor() const
{
    // create an editor - a combobox in our case
    ( (ComboItem*)this )->cb = new QComboBox( table()->viewport() );
    cb->insertItem( "Yes" );
    cb->insertItem( "No" );
    // and initialize it
    cb->setCurrentItem( text() == "No" ? 1 : 0 );
    return cb;
```

See also QTable::createEditor() [p. 335], setContentFromEditor() [p. 356] and QTable::viewport() [p. 274].

Example: table/statistics/statistics.cpp.

EditType QTableItem::editType () const

Returns the table item's edit type.

This is set when the table item is constructed.

See also EditType [p. 352] and QTableItem() [p. 353].

bool QTableItem::isEnabled () const

Returns TRUE if the table item is enabled; otherwise returns FALSE.

See also setEnabled() [p. 356].

bool QTableItem::isReplaceable () const

This function returns whether the contents of the cell may be replaced with the contents of another table item. Regardless of this setting, table items that span more than one cell may not have their contents replaced by another table item.

(This differs from EditType because EditType is concerned with whether the *user* is able to change the contents of a cell.)

See also setReplaceable() [p. 357] and EditType [p. 352].

QString QTableItem::key () const [virtual]

This virtual function returns the key that should be used for sorting. The default implementation returns the text() of the relevant item.

See also QTable::sorting [p. 350].

void QTableItem::paint (QPainter * p, const QColorGroup & cg, const QRect & cr, bool selected) [virtual]

This virtual function is used to paint the contents of an item using the painter *p* in the rectangular area *cr* using the color group *cg*.

If *selected* is TRUE the cell is displayed in a way that indicates that it is highlighted.

You don't usually need to use this function but if you want to draw custom content in a cell you will need to reimplement it.

Note that the painter is not clipped by default in order to get maximum efficiency. If you want clipping, use

QPixmap QTableItem::pixmap () const [virtual]

Returns the table item's pixmap or a null pixmap if no pixmap has been set.

See also setPixmap() [p. 356] and text() [p. 358].

int QTableItem::row () const

Returns the row where the table item is located. If the cell spans multiple rows, this function returns the top-most row.

See also col() [p. 354] and setRow() [p. 357].

int QTableItem::rowSpan () const

Returns the row span of the table item, usually 1.

See also setSpan() [p. 357] and colSpan() [p. 354].

int QTableItem::rtti () const [virtual]

Returns the Run Time Type Identification value for this table item which for QTableItems is 0.

Although often frowned upon by purists, Run Time Type Identification is very useful for QTables as it allows for an efficient indexed storage mechanism.

When you create subclasses based on QTableItem make sure that each subclass returns a unique rtti() value. It is advisable to use values greater than 1000, preferably large random numbers, to allow for extensions to this class.

See also QCheckTableItem::rtti() [p. 31] and QComboTableItem::rtti() [p. 45].

Reimplemented in QComboTableItem and QCheckTableItem.

void QTableItem::setCol (int c) [virtual]

Sets column *c* as the table item's column. Usually you will not need to call this function.

If the cell spans multiple columns, this function sets the left-most column and retains the width of the multi-cell table item.

See also col() [p. 354], setRow() [p. 357] and colSpan() [p. 354].

void QTableItem::setContentFromEditor (QWidget * w) [virtual]

Whenever the content of a cell has been edited by the editor *w*, QTable calls this virtual function to copy the new values into the QTableItem.

If you reimplement createEditor() and return something that is not a QLineEdit you will almost certainly have to reimplement this function.

```
void ComboItem::setContentFromEditor( QWidget *w )
{
    // the user changed the value of the combobox, so synchronize the
    // value of the item (its text), with the value of the combobox
    if ( w->inherits( "QComboBox" ) )
        setText( ( (QComboBox*)w )->currentText() );
    else
        QTableItem::setContentFromEditor( w );
```

See also QTable::setCellContentFromEditor() [p. 343].

void QTableItem::setEnabled (bool b) [virtual]

If *b* is TRUE, the table item is enabled; if *b* is FALSE the table item is disabled.

A disabled item doesn't respond to user interaction.

See also isEnabled() [p. 354].

void QTableItem::setPixmap (const QPixmap & p) [virtual]

Sets pixmap *p* to be this item's pixmap.

Note that setPixmap() does not update the cell the table item belongs to. Use QTable::updateCell() to repaint the cell's contents.

For QComboTableItems and QCheckTableItems this function has no visible effect.

See also QTable::setPixmap() [p. 345], pixmap() [p. 355] and setText() [p. 357].

void QTableItem::setReplaceable (bool b) [virtual]

If *b* is TRUE it is acceptable to replace the contents of the cell with the contents of another QTableItem. If *b* is FALSE the contents of the cell may not be replaced by the contents of another table item. Table items that span more than one cell may not have their contents replaced by another table item.

(This differs from EditType because EditType is concerned with whether the *user* is able to change the contents of a cell.)

See also isReplaceable() [p. 355].

void QTableItem::setRow (int r) [virtual]

Sets row *r* as the table item's row. Usually you do not need to call this function.

If the cell spans multiple rows, this function sets the top row and retains the height of the multi-cell table item.

See also row() [p. 355], setCol() [p. 356] and rowSpan() [p. 355].

void QTableItem::setSpan (int rs, int cs) [virtual]

Changes the extent of the QTableItem so that it spans multiple cells covering *rs* rows and *cs* columns. The top left cell is the original cell.

Warning: This function only works, if the item has already been inserted into the table using e.g. QTable::setItem().

See also rowSpan() [p. 355] and colSpan() [p. 354].

void QTableItem::setText (const QString & str) [virtual]

Changes the text of the table item to *str*.

Note that setText() does not update the cell the table item belongs to. Use QTable::updateCell() to repaint the cell's contents.

See also QTable::setText() [p. 346], text() [p. 358], setPixmap() [p. 356] and QTable::updateCell() [p. 348].

void QTableItem::setWordWrap (bool b) [virtual]

If *b* is TRUE, the cell's text will be wrapped over multiple lines, when necessary, to fit the width of the cell; otherwise the text will be written as a single line.

See also wordWrap() [p. 358], QTable::adjustColumn() [p. 332] and QTable::setColumnStretchable() [p. 344].

QSize QTableItem::sizeHint () const [virtual]

This virtual function returns the size a cell needs to show its entire content.

If you subclass QTableItem you will often need to reimplement this function.

QTable * QTableItem::table () const

Returns the QTable the table item belongs to.

See also QTable::setItem() [p. 345] and QTableItem() [p. 353].

QString QTableItem::text () const [virtual]

Provides the text of the table item or a null string if there's no text.

See also setText() [p. 357] and pixmap() [p. 355].

bool QTableItem::wordWrap () const

Returns TRUE if word wrap is enabled for the cell; otherwise returns FALSE.

See also setWordWrap() [p. 357].

QTabWidget Class Reference

The QTabWidget class provides a stack of tabbed widgets.

`#include <qtabwidget.h>`

Inherits QWidget [p. 412].

Public Members

- **QTabWidget** (QWidget * parent = 0, const char * name = 0, WFlags f = 0)
- virtual void **addTab** (QWidget * child, const QString & label)
- virtual void **addTab** (QWidget * child, const QIconSet & iconset, const QString & label)
- virtual void **addTab** (QWidget * child, QTab * tab)
- virtual void **insertTab** (QWidget * child, const QString & label, int index = -1)
- virtual void **insertTab** (QWidget * child, const QIconSet & iconset, const QString & label, int index = -1)
- virtual void **insertTab** (QWidget * child, QTab * tab, int index = -1)
- void **changeTab** (QWidget * w, const QString & label)
- void **changeTab** (QWidget * w, const QIconSet & iconset, const QString & label)
- bool **isTabEnabled** (QWidget * w) const
- void **setTabEnabled** (QWidget * w, bool enable)
- QString **tabLabel** (QWidget * w) const
- void **setTabLabel** (QWidget * w, const QString & l)
- QIconSet **tabIconSet** (QWidget * w) const
- void **setTabIconSet** (QWidget * w, const QIconSet & iconset)
- void **removeTabToolTip** (QWidget * w)
- void **setTabToolTip** (QWidget * w, const QString & tip)
- QString **tabToolTip** (QWidget * w) const
- QWidget * **currentPage** () const
- QWidget * **page** (int index) const
- QString **label** (int index) const
- int **currentPageIndex** () const
- int **indexOf** (QWidget * w) const
- enum **TabPosition** { Top, Bottom }
- TabPosition **tabPosition** () const
- void **setTabPosition** (TabPosition)
- enum **TabShape** { Rounded, Triangular }
- TabShape **tabShape** () const
- void **setTabShape** (TabShape s)
- int **margin** () const
- void **setMargin** (int)
- int **count** () const

Public Slots

- void **setCurrentPage** (int)
- virtual void **showPage** (QWidget * w)
- virtual void **removePage** (QWidget * w)

Signals

- void **currentChanged** (QWidget *)

Properties

- bool **autoMask** — whether the tab widget is automatically masked *(read only)*
- int **count** — the number of tabs in the tab bar *(read only)*
- int **currentPage** — the index position of the current tab page
- int **margin** — the margin in this tab widget
- TabPosition **tabPosition** — the position of the tabs in this tab widget
- TabShape **tabShape** — the shape of the tabs in this tab widget

Protected Members

- void **setTabBar** (QTabBar * tb)
- QTabBar * **tabBar** () const

Detailed Description

The QTabWidget class provides a stack of tabbed widgets.

A tabbed widget is a widget that has a tab bar of tabs, and for each tab a "page" which is a widget. The user selects which page to see and use by clicking on its tab or by pressing the indicated Alt+*letter* key combination.

QTabWidget provides a single row of tabs along the top or bottom of the pages (see TabPosition).

The normal way to use QTabWidget is to do the following in the constructor:

1. Create a QTabWidget.
2. Create a QWidget for each of the pages in the tab dialog, insert children into it, set up geometry management for it and use addTab() (or insertTab()) to set up a tab and keyboard accelerator for it.
3. Connect to the signals and slots.

The position of the tabs is set with setTabPosition(), their shape with setTabShape(), and their margin with setMargin().

If you don't call addTab() the page you have created will not be visible. Don't confuse the object name you supply to the QWidget constructor and the tab label you supply to addTab(). addTab() takes a name which indicates an accelerator and is meaningful and descriptive to the user, whereas the widget name is used primarily for debugging.

The signal currentChanged() is emitted when the user selects a page.

The current page is available as an index position with currentPageIndex() or as a wiget pointer with currentPage(). You can retrieve a pointer to a page with a given index using page(), and can find the index position of a page

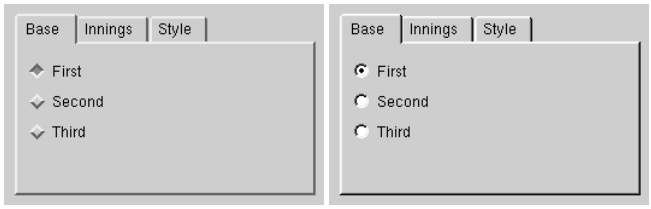
with indexOf(). Use setCurrentPage() to show a particular page by index, or showPage() to show a page by widget pointer.

You can change a tab's label and iconset using changeTab() or setTabLabel() and setTabIconSet(). A tab page can be removed with removePage().

Each tab is either enabled or disabled at any given time (see setTabEnabled()). If a tab is enabled, the tab text is drawn in black and the user can select that tab. If it is disabled, the tab is drawn in a different way and the user cannot select that tab. Note that even if a tab is disabled, the page can still be visible, for example if all of the tabs happen to be disabled.

Although tab widgets can be a very good way to split up a complex dialog, it's also very easy to get into a mess. See QTabDialog for some design hints.

Most of the functionality in QTabWidget is provided by a QTabBar (at the top, providing the tabs) and a QWidget-Stack (most of the area, organizing the individual pages).



See also QTabDialog [Dialogs and Windows with Qt], Advanced Widgets and Organizers.

Member Type Documentation

QTabWidget::TabPosition

This enum type defines where QTabWidget can draw the tab row:

- `QTabWidget::Top` - above the pages
- `QTabWidget::Bottom` - below the pages

QTabWidget::TabShape

This enum type defines the shape of the tabs:

- `QTabWidget::Rounded` - rounded look (normal)
- `QTabWidget::Triangular` - triangular look (very unusual, included for completeness)

Member Function Documentation

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

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

void QTabWidget::addTab (QWidget * child, const QString & label) [virtual]

Adds another tab and page to the tab view.

The new page is *child*; the tab's label is *label*. Note the difference between the widget name (which you supply to widget constructors and to setTabEnabled(), for example) and the tab label. The name is internal to the program and invariant, whereas the label is shown on-screen and may vary according to language and other factors.

If the tab's *label* contains an ampersand, the letter following the ampersand is used as an accelerator for the tab, e.g. if the label is "Bro&wse" then Alt+W becomes an accelerator which will move the focus to this tab.

If you call addTab() after show() the screen will flicker and the user may be confused.

See also insertTab() [p. 363].

Examples: addressbook/centralwidget.cpp and themes/themes.cpp.

void QTabWidget::addTab (QWidget * child, const QIconSet & iconset, const QString & label) [virtual]

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

Adds another tab and page to the tab view.

This function is the same as addTab(), but with an additional *iconset*.

void QTabWidget::addTab (QWidget * child, QTab * tab) [virtual]

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

This is a low-level function for adding tabs. It is useful if you are using setTabBar() to set a QTabBar subclass with an overridden QTabBar::paint() routine for a subclass of QTab. The *child* is the new page and *tab* is the tab to put the *child* on.

void QTabWidget::changeTab (QWidget * w, const QString & label)

Defines a new *label* for page *w*'s tab.

void QTabWidget::changeTab (QWidget * w, const QIconSet & iconset, const QString & label)

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

Defines a new *iconset* and a new *label* for page *w*'s tab.

int QTabWidget::count () const

Returns the number of tabs in the tab bar. See the "count" [p. 366] property for details.

void QTabWidget::currentChanged (QWidget *) [signal]

This signal is emitted whenever the current page changes. The parameter is the new current page.

See also currentPage() [p. 363], showPage() [p. 365] and tabLabel() [p. 365].

QWidget * QTabWidget::currentPage () const

Returns a pointer to the page currently being displayed by the tab dialog. The tab dialog does its best to make sure that this value is never 0 (but if you try hard enough, it can be).

int QTabWidget::currentPageIndex () const

Returns the index position of the current tab page. See the "currentPage" [p. 366] property for details.

int QTabWidget::indexOf (QWidget * w) const

Returns the index position of page *w*, or -1 if the widget cannot be found.

void QTabWidget::insertTab (QWidget * child, const QString & label, int index = -1) [virtual]

Inserts another tab and page to the tab view.

The new page is *child*; the tab's label is *label*. Note the difference between the widget name (which you supply to widget constructors and to setTabEnabled(), for example) and the tab label. The name is internal to the program and invariant, whereas the label is shown on-screen and may vary according to language and other factors.

If the tab's *label* contains an ampersand, the letter following the ampersand is used as an accelerator for the tab, e.g. if the label is "Bro&wse" then Alt+W becomes an accelerator which will move the focus to this tab.

If *index* is not specified, the tab is simply added. Otherwise it is inserted at the specified position.

If you call insertTab() after show(), the screen will flicker and the user may be confused.

See also addTab() [p. 361].

void QTabWidget::insertTab (QWidget * child, const QIconSet & iconset, const QString & label, int index = -1) [virtual]

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

Inserts another tab and page to the tab view.

This function is the same as insertTab(), but with an additional *iconset*.

void QTabWidget::insertTab (QWidget * child, QTab * tab, int index = -1) [virtual]

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

This is a lower-level method for inserting tabs, similar to the other insertTab() method. It is useful if you are using setTabBar() to set a QTabBar subclass with an overridden QTabBar::paint() routine for a subclass of QTab. The *child* is the new page, *tab* is the tab to put the *child* on and *index* is the position in the tab bar that this page should occupy.

bool QTabWidget::isTabEnabled (QWidget * w) const

Returns TRUE if the page *w* is enabled; otherwise returns FALSE.

See also setTabEnabled() [p. 364] and QWidget::enabled [p. 461].

QString QTabWidget::label (int index) const

Returns the label of the tab at index position *index*.

int QTabWidget::margin () const

Returns the margin in this tab widget. See the "margin" [p. 366] property for details.

QWidget * QTabWidget::page (int index) const

Returns the tab page at index position *index*.

void QTabWidget::removePage (QWidget * w) [virtual slot]

Removes page *w* from this stack of widgets. Does not delete *w*.

See also showPage() [p. 365] and QWidgetStack::removeWidget() [p. 480].

void QTabWidget::removeTabToolTip (QWidget * w)

Removes the tab tool tip for page *w*. If the page does not have a tip, nothing happens.

See also setTabToolTip() [p. 365] and tabToolTip() [p. 366].

void QTabWidget::setCurrentPage (int) [slot]

Sets the index position of the current tab page. See the "currentPage" [p. 366] property for details.

void QTabWidget::setMargin (int)

Sets the margin in this tab widget. See the "margin" [p. 366] property for details.

void QTabWidget::setTabBar (QTabBar * tb) [protected]

Replaces the QTabBar heading the dialog by the tab bar *tb*. Note that this must be called *before* any tabs have been added, or the behavior is undefined.

See also tabBar() [p. 365].

void QTabWidget::setTabEnabled (QWidget * w, bool enable)

If *enable* is TRUE, page *w* is enabled; otherwise page *w* is disabled. The page's tab is redrawn appropriately.

QTabWidget uses QWidget::setEnabled() internally, rather than keeping a separate flag.

Note that even a disabled tab/page may be visible. If the page is visible already, QTabWidget will not hide it; if all the pages are disabled, QTabWidget will show one of them.

See also isTabEnabled() [p. 363] and QWidget::enabled [p. 461].

void QTabWidget::setTabIconSet (QWidget * w, const QIconSet & iconset)

Sets the iconset for page *w* to *iconset*.

void QTabWidget::setTabLabel (QWidget * w, const QString & l)

Sets the tab label for page *w* to *l*

void QTabWidget::setTabPosition (TabPosition)

Sets the position of the tabs in this tab widget. See the "tabPosition" [p. 366] property for details.

void QTabWidget::setTabShape (TabShape s)

Sets the shape of the tabs in this tab widget to *s*. See the "tabShape" [p. 366] property for details.

void QTabWidget::setTabToolTip (QWidget * w, const QString & tip)

Sets the tab tool tip for page *w* to *tip*.

See also removeTabToolTip() [p. 364] and tabToolTip() [p. 366].

void QTabWidget::showPage (QWidget * w) [virtual slot]

Ensures that page *w* is shown. This is useful mainly for accelerators.

Warning: Used carelessly, this function can easily surprise or confuse the user.

See also QTabBar::currentTab [p. 323].

QTabBar * QTabWidget::tabBar () const [protected]

Returns the currently set QTabBar.

See also setTabBar() [p. 364].

QIconSet QTabWidget::tabIconSet (QWidget * w) const

Returns the iconset of page *w*.

QString QTabWidget::tabLabel (QWidget * w) const

Returns the label text for the tab on page *w*.

TabPosition QTabWidget::tabPosition () const

Returns the position of the tabs in this tab widget. See the "tabPosition" [p. 366] property for details.

TabShape QTabWidget::tabShape () const

Returns the shape of the tabs in this tab widget. See the "tabShape" [p. 366] property for details.

QString QTabWidget::tabToolTip (QWidget * w) const

Returns the tab tool tip for page *w*.

See also setTabToolTip() [p. 365] and removeTabToolTip() [p. 364].

Property Documentation

bool autoMask

This property holds whether the tab widget is automatically masked.

See also QWidget::autoMask [p. 459].

int count

This property holds the number of tabs in the tab bar.

Get this property's value with count().

int currentPage

This property holds the index position of the current tab page.

Set this property's value with setCurrentPage() and get this property's value with currentPageIndex().

See also QTabBar::currentTab [p. 323].

int margin

This property holds the margin in this tab widget.

The margin is the distance between the innermost pixel of the frame and the outermost pixel of the pages.

Set this property's value with setMargin() and get this property's value with margin().

TabPosition tabPosition

This property holds the position of the tabs in this tab widget.

Possible values for this property are QTabWidget::Top and QTabWidget::Bottom.

See also TabPosition [p. 361].

Set this property's value with setTabPosition() and get this property's value with tabPosition().

TabShape tabShape

This property holds the shape of the tabs in this tab widget.

Possible values for this property are QTabWidget::Rounded (default) or QTabWidget::Triangular.

See also TabShape [p. 361].

Set this property's value with setTabShape() and get this property's value with tabShape().

QTextBrowser Class Reference

The QTextBrowser class provides a rich text browser with hypertext navigation.

`#include <qtextbrowser.h>`

Inherits QTextEdit [p. 372].

Public Members

- **QTextBrowser** (QWidget * parent = 0, const char * name = 0)
- QString **source** () const

Public Slots

- virtual void **setSource** (const QString & name)
- virtual void **backward** ()
- virtual void **forward** ()
- virtual void **home** ()
- virtual void **reload** ()

Signals

- void **backwardAvailable** (bool available)
- void **forwardAvailable** (bool available)
- void **highlighted** (const QString & href)
- void **linkClicked** (const QString & link)

Properties

- QString **source** — the name of the currently displayed document

Protected Members

- virtual void **keyPressEvent** (QKeyEvent * e)

Detailed Description

The QTextBrowser class provides a rich text browser with hypertext navigation.

This class extends QTextEdit (in read-only mode), adding some navigation functionality so that users can follow links in hypertext documents. The contents of QTextEdit is set with setText(), but QTextBrowser has an additional function, setSource(), which makes it possible to set the text to a named document. The name is looked up in the text view's mime source factory. If a document name ends with an anchor (for example, "`#anchor`"), the text browser automatically scrolls to that position (using scrollToAnchor()). When the user clicks on a hyperlink, the browser will call setSource() itself, with the link's `href` value as argument.

QTextBrowser provides backward() and forward() slots which you can use to implement Back and Forward buttons. The home() slot sets the text to the very first document displayed. The linkClicked() signal is emitted when the user clicks a link.

By using QTextEdit::setMimeSourceFactory() you can provide your own subclass of QMimeSourceFactory. This makes it possible to access data from anywhere, for example from a network or from a database. See QMimeSourceFactory::data() for details.

If you intend using the mime factory to read the data directly from the file system, you may have to specify the encoding for the file extension you are using. For example:

```
mimeSourceFactory()->setExtensionType("qml", "text/utf8");
```

This is to ensure that the factory is able to resolve the document names.

If you want to provide your users with editable rich text use QTextEdit. If you want a text browser without hypertext navigation use QTextEdit, and use QTextEdit::setReadOnly() to disable editing. If you just need to display a small piece of rich text use QSimpleRichText or QLabel.

See also Advanced Widgets, Help System and Text Related Classes.

Member Function Documentation

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

Constructs an empty QTextBrowser with parent *parent* called *name*.

void QTextBrowser::backward () [virtual slot]

Changes the document displayed to the previous document in the list of documents built by navigating links. Does nothing if there is no previous document.

See also forward() [p. 370] and backwardAvailable() [p. 370].

Example: helpviewer/helpwindow.cpp.

void QTextBrowser::backwardAvailable (bool available) [signal]

This signal is emitted when the availability of the backward() changes. *available* is FALSE when the user is at home(); otherwise it is TRUE.

Example: helpviewer/helpwindow.cpp.

void QTextBrowser::forward () [virtual slot]

Changes the document displayed to the next document in the list of documents built by navigating links. Does nothing if there is no next document.

See also backward() [p. 369] and forwardAvailable() [p. 370].

Example: helpviewer/helpwindow.cpp.

void QTextBrowser::forwardAvailable (bool available) [signal]

This signal is emitted when the availability of the forward() changes. *available* is TRUE after the user navigates backward() and FALSE when the user navigates or goes forward().

Example: helpviewer/helpwindow.cpp.

void QTextBrowser::highlighted (const QString & href) [signal]

This signal is emitted when the user has selected but not activated a link in the document. *href* is the value of the href tag in the link.

Example: helpviewer/helpwindow.cpp.

void QTextBrowser::home () [virtual slot]

Changes the document displayed to be the first document the browser displayed.

Example: helpviewer/helpwindow.cpp.

void QTextBrowser::keyPressEvent (QKeyEvent * e) [virtual protected]

The event *e* is used to provide the following keyboard shortcuts:

- *Alt+Left Arrow* - backward()
- *Alt+Right Arrow* - forward()
- *Alt+Up Arrow* - home()

Reimplemented from QTextEdit [p. 386].

void QTextBrowser::linkClicked (const QString & link) [signal]

This signal is emitted when the user clicks a link. The *link* is the value of the `href` i.e. the name of the target document.

void QTextBrowser::reload () [virtual slot]

Reloads the current set source

void QTextBrowser::setSource (const QString & name) [virtual slot]

Sets the name of the currently displayed document to *name*. See the "source" [p. 371] property for details.

QString QTextBrowser::source () const

Returns the name of the currently displayed document. See the "source" [p. 371] property for details.

Property Documentation

QString source

This property holds the name of the currently displayed document.

This is a null string if no document is displayed or the source is unknown.

Setting this property uses the mimeSourceFactory to lookup the named document. It also checks for optional anchors and scrolls the document accordingly.

If the first tag in the document is `<qt type=detail>`, the document is displayed as a popup rather than as new document in the browser window itself. Otherwise, the document is displayed normally in the text browser with the text set to the contents of the named document with setText().

If you are using the filesystem access capabilities of the mime source factory, you must ensure that the factory knows about the encoding of specified files; otherwise no data will be available. The default factory handles a couple of common file extensions such as `*.html` and `*.txt` with reasonable defaults. See QMimeSourceFactory::data() for details.

Set this property's value with setSource() and get this property's value with source().

QTextEdit Class Reference

The QTextEdit widget provides a sophisticated single-page rich text editor.

`#include <qtextedit.h>`

Inherits QScrollView [p. 259].

Inherited by QMultiLineEdit [p. 217], QTextBrowser [p. 368] and QTextView.

Public Members

- enum **WordWrap** { NoWrap, WidgetWidth, FixedPixelWidth, FixedColumnWidth }
- enum **WrapPolicy** { AtWordBoundary, Anywhere, AtWhiteSpace = AtWordBoundary }
- enum **KeyboardAction** { ActionBackspace, ActionDelete, ActionReturn, ActionKill }
- enum **CursorAction** { MoveBackward, MoveForward, MoveWordBackward, MoveWordForward, MoveUp, MoveDown, MoveLineStart, MoveLineEnd, MoveHome, MoveEnd, MovePgUp, MovePgDown }
- enum **VerticalAlignment** { AlignNormal, AlignSuperScript, AlignSubScript }
- **QTextEdit** (const QString & text, const QString & context = QString::null, QWidget * parent = 0, const char * name = 0)
- **QTextEdit** (QWidget * parent = 0, const char * name = 0)
- QString **text** () const
- QString **text** (int para) const
- TextFormat **textFormat** () const
- QString **context** () const
- QString **documentTitle** () const
- void **getSelection** (int * paraFrom, int * indexFrom, int * paraTo, int * indexTo, int selNum = 0) const
- virtual bool **find** (const QString & expr, bool cs, bool wo, bool forward = TRUE, int * para = 0, int * index = 0)
- int **paragraphs** () const
- int **lines** () const
- int **linesOfParagraph** (int para) const
- int **lineOfChar** (int para, int index)
- int **length** () const
- QRect **paragraphRect** (int para) const
- int **paragraphAt** (const QPoint & pos) const
- int **charAt** (const QPoint & pos, int * para) const
- int **paragraphLength** (int para) const
- QStyleSheet * **styleSheet** () const
- QMimeSourceFactory * **mimeSourceFactory** () const
- QBrush **paper** () const
- bool **linkUnderline** () const
- virtual int **heightForWidth** (int w) const

- bool **hasSelectedText** () const
- QString **selectedText** () const
- bool **isUndoAvailable** () const
- bool **isRedoAvailable** () const
- WordWrap **wordWrap** () const
- int **wrapColumnOrWidth** () const
- WrapPolicy **wrapPolicy** () const
- int **tabStopWidth** () const
- QString **anchorAt** (const QPoint & pos)
- bool **isReadOnly** () const
- void **getCursorPosition** (int * para, int * index) const
- bool **isModified** () const
- bool **italic** () const
- bool **bold** () const
- bool **underline** () const
- QString **family** () const
- int **pointSize** () const
- QColor **color** () const
- QFont **font** () const
- int **alignment** () const
- int **undoDepth** () const
- virtual bool **getFormat** (int para, int index, QFont * font, QColor * color, VerticalAlignment * verticalAlignment)
- virtual bool **getParagraphFormat** (int para, QFont * font, QColor * color, VerticalAlignment * verticalAlignment, int * alignment, QStyleSheetItem::DisplayMode * displayMode, QStyleSheetItem::ListStyle * listStyle, int * listDepth)
- bool **isOverwriteMode** () const
- QColor **paragraphBackgroundColor** (int para) const
- bool **isUndoRedoEnabled** () const

Public Slots

- virtual void **setMimeSourceFactory** (QMimeSourceFactory * factory)
- virtual void **setStyleSheet** (QStyleSheet * styleSheet)
- virtual void **scrollToAnchor** (const QString & name)
- virtual void **setPaper** (const QBrush & pap)
- virtual void **setLinkUnderline** (bool)
- virtual void **setWordWrap** (WordWrap mode)
- virtual void **setWrapColumnOrWidth** (int)
- virtual void **setWrapPolicy** (WrapPolicy policy)
- virtual void **copy** ()
- virtual void **append** (const QString & text)
- void **setText** (const QString & txt)
- virtual void **setText** (const QString & text, const QString & context)
- virtual void **setTextFormat** (TextFormat f)
- virtual void **selectAll** (bool select = TRUE)
- virtual void **setTabStopWidth** (int ts)
- virtual void **zoomIn** (int range)
- virtual void **zoomIn** ()

- virtual void **zoomOut** (int range)
- virtual void **zoomOut** ()
- virtual void **zoomTo** (int size)
- virtual void **setReadOnly** (bool b)
- virtual void **undo** ()
- virtual void **redo** ()
- virtual void **cut** ()
- virtual void **paste** ()
- virtual void **pasteSubType** (const QCString & subtype)
- virtual void **clear** ()
- virtual void **del** ()
- virtual void **indent** ()
- virtual void **setItalic** (bool b)
- virtual void **setBold** (bool b)
- virtual void **setUnderline** (bool b)
- virtual void **setFamily** (const QString & fontFamily)
- virtual void **setPointSize** (int s)
- virtual void **setColor** (const QColor & c)
- virtual void **setVerticalAlignment** (VerticalAlignment a)
- virtual void **setAlignment** (int a)
- virtual void **setParagType** (QStyleSheetItem::DisplayMode dm, QStyleSheetItem::ListStyle listStyle)
- virtual void **setCursorPosition** (int para, int index)
- virtual void **setSelection** (int paraFrom, int indexFrom, int paraTo, int indexTo, int selNum = 0)
- virtual void **setSelectionAttributes** (int selNum, const QColor & back, bool invertText)
- virtual void **setModified** (bool m)
- virtual void **setUndoDepth** (int d)
- virtual void **ensureCursorVisible** ()
- virtual void **placeCursor** (const QPoint & pos, QTextCursor * c = 0)
- virtual void **moveCursor** (CursorAction action, bool select)
- virtual void **doKeyboardAction** (KeyboardAction action)
- virtual void **removeSelectedText** (int selNum = 0)
- virtual void **removeSelection** (int selNum = 0)
- virtual void **setCurrentFont** (const QFont & f)
- virtual void **setOverwriteMode** (bool b)
- virtual void **scrollToBottom** ()
- virtual void **insert** (const QString & text, bool indent = FALSE, bool checkNewLine = TRUE, bool removeSelected = TRUE)
- virtual void **insertAt** (const QString & text, int para, int index)
- virtual void **removeParagraph** (int para)
- virtual void **insertParagraph** (const QString & text, int para)
- virtual void **setParagraphBackgroundColor** (int para, const QColor & bg)
- virtual void **clearParagraphBackground** (int para)
- virtual void **setUndoRedoEnabled** (bool b)

Signals

- void **textChanged** ()
- void **selectionChanged** ()
- void **copyAvailable** (bool yes)
- void **undoAvailable** (bool yes)
- void **redoAvailable** (bool yes)
- void **currentFontChanged** (const QFont & f)
- void **currentColorChanged** (const QColor & c)
- void **currentAlignmentChanged** (int a)
- void **currentVerticalAlignmentChanged** (VerticalAlignment a)
- void **cursorPositionChanged** (QTextCursor * c)
- void **cursorPositionChanged** (int para, int pos)
- void **returnPressed** ()
- void **modificationChanged** (bool m)

Properties

- QString **documentTitle** — the title of the document parsed from the text *(read only)*
- bool **hasSelectedText** — whether some text is selected in selection 0 *(read only)*
- int **length** — the number of characters in the text *(read only)*
- bool **linkUnderline** — whether hypertext links will be underlined
- bool **modified** — whether the document has been modified by the user
- bool **overwriteMode** — the text edit's overwrite mode
- QBrush **paper** — the background (paper) brush
- bool **readOnly** — whether the text edit is read-only
- QString **selectedText** — the selected text (from selection 0) or an empty string if there is no currently selected text (in selection 0) *(read only)*
- QString **text** — the text edit's text
- TextFormat **textFormat** — the text format: rich text, plain text or auto text
- int **undoDepth** — the depth of the undo history
- bool **undoRedoEnabled** — whether undo/redo is enabled
- WordWrap **wordWrap** — the word wrap mode
- int **wrapColumnOrWidth** — the position (in pixels or columns depending on the wrap mode) where text will be wrapped
- WrapPolicy **wrapPolicy** — the word wrap policy, at whitespace or anywhere

Protected Members

- void **repaintChanged** ()
- void **updateStyles** ()
- virtual void **keyPressEvent** (QKeyEvent * e)
- virtual bool **focusNextPrevChild** (bool n)
- QTextCursor * **textCursor** () const
- virtual QPopupMenu * **createPopupMenu** (const QPoint & pos)
- virtual QPopupMenu * **createPopupMenu** ()

Detailed Description

The QTextEdit widget provides a sophisticated single-page rich text editor.

QTextEdit is an advanced WYSIWYG editor supporting rich text formatting. It is optimized to handle large documents and to respond quickly to user input.

Internally QTextEdit works on paragraphs and characters. A paragraph is a formatted string which is word-wrapped to fit into the width of the widget. The words in the paragraph are aligned in accordance with the paragraph's alignment(). Paragraphs are separated by hard line breaks. Each character within a paragraph has its own attributes, for example, font and color.

QTextEdit can display images (using QMimeSourceFactory), lists and tables. If the text is too large to view within the text edit's viewport, scrollbars will appear. The text edit can load both plain text and HTML files (a subset of HTML 3.2 and 4). The rendering style and the set of valid tags are defined by a styleSheet(). Change the style sheet with setStyleSheet(); see QStyleSheet for details. The images identified by image tags are displayed if they can be interpreted using the text edit's QMimeSourceFactory; see setMimeSourceFactory().

If you want a text browser with more navigation use QTextBrowser. If you just need to display a small piece of rich text use QLabel or QSimpleRichText.

If you create a new QTextEdit, and want to allow the user to edit rich text, call setTextFormat(Qt::RichText) to ensure that the text is treated as rich text. (Rich text uses HTML tags to set text formatting attributes. See QStyleSheet for information on the HTML tags that are supported.). If you don't call setTextFormat() explicitly the text edit will guess from the text itself whether it is rich text or plain text.

The text edit documentation uses the following concepts:

- *current format* — this is the format at the current cursor position, *and* it is the format of the selected text if any.
- *current paragraph* — the paragraph which contains the cursor.

The text is set or replaced using setText() which deletes any existing text and replaces it with the text passed in the setText() call. Text can be inserted with insert(), paste() and pasteSubType(). Text can also be cut(). The entire text is deleted with clear() and the selected text is deleted with removeSelectedText(). Selected (marked) text can also be deleted with del() (which will delete the character to the right of the cursor if no text is selected).

The current format's attributes are set with setItalic(), setBold(), setUnderline(), setFamily() (font family), setPointSize(), setColor() and setCurrentFont(). The current paragraph's style is set with setParagType() and its alignment is set with setAlignment().

Use setSelection() to select text. The setSelectionAttributes() function is used to indicate how selected text should be displayed. Use hasSelectedText() to find out if any text is selected. The currently selected text's position is available using getSelection() and the selected text itself is returned by selectedText(). The selection can be copied to the clipboard with copy(), or cut to the clipboard with cut(). It can be deleted with removeSelectedText(). The entire text can be selected (or unselected) using selectAll(). QTextEdit supports multiple selections. Most of the selection functions operate on the default selection, selection 0.

Set and get the position of the cursor with setCursorPosition() and getCursorPosition() respectively. When the cursor is moved, the signals currentFontChanged(), currentColorChanged() and currentAlignmentChanged() are emitted to reflect the font, color and alignment at the new cursor position.

If the text changes, the textChanged() signal is emitted, and if the user inserts a new line by pressing Return or Enter, returnPressed() is emitted. The isModified() function will return TRUE if the text has been modified.

QTextEdit provides command-based undo and redo. To set the depth of the command history use setUndoDepth() which defaults to 100 steps. To undo or redo the last operation call undo() or redo(). The signals undoAvailable() and redoAvailable() indicate whether the undo and redo operations can be executed.

The indent() function is used to reindent a paragraph. It is useful for code editors, for example in *Qt Designer*'s code editor *Ctrl+I* invokes the indent() function.

Loading and saving text is achieved using setText() and text(), for example:

```
QFile file( fileName ); // Read the text from a file
if ( file.open( IO_ReadOnly ) ) {
    QTextStream ts( &file );
    textEdit->setText( ts.read() );
}

QFile file( fileName ); // Write the text to a file
if ( file.open( IO_WriteOnly ) ) {
    QTextStream ts( &file );
    ts <text();
    textEdit->setModified( FALSE );
}
```

By default the text edit wraps words at whitespace to fit within the text edit widget. The setWordWrap() function is used to specify the kind of word wrap you want, or NoWrap if you don't want any wrapping. Call setWordWrap() to set a fixed pixel width FixedPixelWidth, or character column (e.g. 80 column) FixedColumnWidth with the pixels or columns specified with setWrapColumnOrWidth(). If you use word wrap to the widget's width WidgetWidth, you can specify whether to break on whitespace or anywhere with setWrapPolicy().

The background color is set differently than other widgets, using setPaper(). You specify a brush style which could be a plain color or a complex pixmap.

Hypertext links are automatically underlined; this can be changed with setLinkUnderline(). The tab stop width is set with setTabStopWidth().

The zoomIn() and zoomOut() functions can be used to resize the text by increasing (decreasing for zoomOut()) the point size used. Images are not affected by the zoom functions.

The lines() function returns the number of lines in the text and paragraphs() returns the number of paragraphs. The number of lines within a particular paragraph is returned by linesOfParagraph(). The length of the entire text in characters is returned by length().

You can scroll to an anchor in the text, e.g. `<a name="anchor">` with scrollToAnchor(). The find() function can be used to find and select a given string within the text.

The list of key-bindings which are implemented for editing:

- *Backspace* — Delete the character to the left of the cursor
- *Delete* — Delete the character to the right of the cursor
- *Ctrl+A* — Move the cursor to the beginning of the line
- *Ctrl+B* — Move the cursor one character left
- *Ctrl+C* — Copy the marked text to the clipboard (also *Ctrl+Insert* under Windows)
- *Ctrl+D* — Delete the character to the right of the cursor
- *Ctrl+E* — Move the cursor to the end of the line
- *Ctrl+F* — Move the cursor one character right
- *Ctrl+H* — Delete the character to the left of the cursor
- *Ctrl+K* — Delete to end of line
- *Ctrl+N* — Move the cursor one line down
- *Ctrl+P* — Move the cursor one line up
- *Ctrl+V* — Paste the clipboard text into line edit (also *Shift+Insert* under Windows)
- *Ctrl+X* — Cut the marked text, copy to clipboard (also *Shift+Delete* under Windows)
- *Ctrl+Z* — Undo the last operation
- *Ctrl+Y* — Redo the last operation
- *Left Arrow* — Move the cursor one character left

- *Ctrl+Left Arrow* — Move the cursor one word left
- *Right Arrow* — Move the cursor one character right
- *Ctrl+Right Arrow* — Move the cursor one word right
- *Up Arrow* — Move the cursor one line up
- *Ctrl+Up Arrow* — Move the cursor one word up
- *Down Arrow* — Move the cursor one line down
- *Ctrl+Down Arrow* — Move the cursor one word down
- *Page Up* — Move the cursor one page up
- *Page Down* — Move the cursor one page down
- *Home* — Move the cursor to the beginning of the line
- *Ctrl+Home Arrow* — Move the cursor to the beginning of the text
- *End* — Move the cursor to the end of the line
- *Ctrl+End Arrow* — Move the cursor to the end of the text
- *Shift+Wheel* — Scroll the page horizontally (the Wheel is the mouse wheel)
- *Ctrl+Wheel* — Zoom the text

To select (mark) text hold down the Shift key whilst pressing one of the movement keystrokes, for example, *Shift+Right Arrow* will select the character to the right, and *Shift+Ctrl+Right Arrow* will select the word to the right, etc.

By default the text edit widget operates in insert mode so all text that the user enters is inserted into the text edit and any text to the right of the cursor is moved out of the way. The mode can be changed to overwrite, where new text overwrites any text to the right of the cursor, using setOverwriteMode().

QTextEdit can also be used as read-only text viewer. Call setReadOnly(TRUE) to disable editing. A read-only QTextEdit provides the same functionality as the (obsolete) QTextView. (QTextView is still supplied for compatibility with old code.)

When QTextEdit is used read-only the key-bindings are limited to navigation, and text may only be selected with the mouse:

- *Up Arrow* — Move one line up
- *Down Arrow* — Move one line down
- *Left Arrow* — Move one character left
- *Right Arrow* — Move one character right
- *Page Up* — Move one (viewport) page up
- *Page Down* — Move one (viewport) page down
- *Home* — Move to the beginning of the text
- *End* — Move to the end of the text
- *Shift+Wheel* — Scroll the page horizontally (the Wheel is the mouse wheel)
- *Ctrl+Wheel* — Zoom the text

The text edit may be able to provide some meta-information. For example, the documentTitle() function will return the text from within HTML `<title>` tags.

The text displayed in a text edit has a *context*. The context is a path which the text edit's QMimeSourceFactory uses to resolve the locations of files and images. It is passed to the mimeSourceFactory() when quering data. (See QTextEdit() and context().)

Note that we do not intend to add a full-featured web browser widget to Qt (because that would easily double Qt's size and only a few applications would benefit from it). The rich text support in Qt is designed to provide a fast, portable and efficient way to add reasonable online help facilities to applications, and to provide a basis for rich text editors.

See also Basic Widgets and Text Related Classes.

Member Type Documentation

QTextEdit::CursorAction

This enum is used by moveCursor() to specify in which direction the cursor should be moved:

- `QTextEdit::MoveBackward` - Moves the cursor one character backward
- `QTextEdit::MoveWordBackward` - Moves the cursor one word backward
- `QTextEdit::MoveForward` - Moves the cursor one character forward
- `QTextEdit::MoveWordForward` - Moves the cursor one word forward
- `QTextEdit::MoveUp` - Moves the cursor up one line
- `QTextEdit::MoveDown` - Moves the cursor down one line
- `QTextEdit::MoveLineStart` - Moves the cursor to the beginning of the line
- `QTextEdit::MoveLineEnd` - Moves the cursor to the end of the line
- `QTextEdit::MoveHome` - Moves the cursor to the beginning of the document
- `QTextEdit::MoveEnd` - Moves the cursor to the end of the document
- `QTextEdit::MovePgUp` - Moves the cursor one page up
- `QTextEdit::MovePgDown` - Moves the cursor one page down

QTextEdit::KeyboardAction

This enum is used by doKeyboardAction() to specify which action should be executed:

- `QTextEdit::ActionBackspace` - Delete the character to the left of the cursor.
- `QTextEdit::ActionDelete` - Delete the character to the right of the cursor.
- `QTextEdit::ActionReturn` - Split the paragraph at the cursor position.
- `QTextEdit::ActionKill` - If the cursor is not at the end of the paragraph, delete the text from the cursor position until the end of the paragraph. If the cursor is at the end of the paragraph, delete the hard line break at the end of the paragraph - this will cause this paragraph to be joined with the following paragraph.

QTextEdit::VerticalAlignment

This enum is used to set the vertical alignment of the text.

- `QTextEdit::AlignNormal` - Normal alignment
- `QTextEdit::AlignSuperScript` - Superscript
- `QTextEdit::AlignSubScript` - Subscript

QTextEdit::WordWrap

This enum defines the QTextEdit's word wrap modes. The following values are valid:

- `QTextEdit::NoWrap` - Do not wrap the text.
- `QTextEdit::WidgetWidth` - Wrap the text at the current width of the widget (this is the default). Wrapping is at whitespace by default; this can be changed with setWrapPolicy().
- `QTextEdit::FixedPixelWidth` - Wrap the text at a fixed number of pixels from the widget's left side. The number of pixels is set with wrapColumnOrWidth().

- `QTextEdit::FixedColumnWidth` - Wrap the text at a fixed number of character columns from the widget's left side. The number of characters is set with wrapColumnOrWidth(). This is useful if you need formatted text that can also be displayed gracefully on devices with monospaced fonts, for example a standard VT100 terminal, where you might set wrapColumnOrWidth() to 80.

See also wordWrap [p. 398] and wordWrap [p. 398].

QTextEdit::WrapPolicy

This enum defines where text can be wrapped in word wrap mode.

The following values are valid:

- `QTextEdit::AtWhiteSpace` - Break lines at whitespace, e.g. spaces or newlines.
- `QTextEdit::Anywhere` - Break anywhere, including within words.
- `QTextEdit::AtWordBoundary` - Don't use this deprecated value (it is a synonym for AtWhiteSpace which you should use instead).

See also wrapPolicy [p. 399].

Member Function Documentation

QTextEdit::QTextEdit (const QString & text, const QString & context = QString::null, QWidget * parent = 0, const char * name = 0)

Constructs a QTextEdit with parent *parent* and name *name*. The text edit will display the text *text* using context *context*.

The *context* is a path which the text edit's QMimeSourceFactory uses to resolve the locations of files and images. It is passed to the mimeSourceFactory() when quering data.

For example if the text contains an image tag, `<img src="image.png">`, and the context is "path/to/look/in", the QMimeSourceFactory will try to load the image from "path/to/look/in/image.png". If the tag was `<img src="/image.png">`, the context will not be used (because QMimeSourceFactory recognizes that we have used an absolute path) and will try to load "/image.png". The context is applied in exactly the same way to *hrefs*, for example, `<a href="target.html">Target</a>`, would resolve to "path/to/look/in/target.html".

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

Constructs an empty QTextEdit with parent *parent* and name *name*.

int QTextEdit::alignment () const

Returns the alignment of the current paragraph.

See also setAlignment() [p. 390].

QString QTextEdit::anchorAt (const QPoint & pos)

If there is an anchor at position *pos* (in contents coordinates), its name is returned, otherwise an empty string is returned.

void QTextEdit::append (const QString & text) [virtual slot]

Appends the text *text* to the end of the text edit.

Examples: network/clientserver/client/client.cpp, network/clientserver/server/server.cpp, network/httpd/httpd.cpp and process/process.cpp.

bool QTextEdit::bold () const

Returns TRUE if the current format is bold; otherwise returns FALSE.

See also setBold() [p. 390].

int QTextEdit::charAt (const QPoint & pos, int * para) const

Returns the index of the character (relative to its paragraph) at position *pos* (in contents coordinates). If *para* is not null, *para* is set to this paragraph. If there is no character at *pos*, -1 is returned.

void QTextEdit::clear () [virtual slot]

Deletes all the text in the text edit.

See also cut() [p. 383], removeSelectedText() [p. 389] and text [p. 397].

void QTextEdit::clearParagraphBackground (int para) [virtual slot]

Clears the background color of the paragraph *para*, so that the default color is used again.

QColor QTextEdit::color () const

Returns the color of the current format.

See also setColor() [p. 390] and paper [p. 397].

QString QTextEdit::context () const

Returns the context of the edit. The context is a path which the text edit's QMimeSourceFactory uses to resolve the locations of files and images.

See also text [p. 397].

Examples: helpviewer/helpwindow.cpp and qdir/qdir.cpp.

void QTextEdit::copy () [virtual slot]

Copies any selected text (from selection 0) to the clipboard.

See also hasSelectedText [p. 396] and copyAvailable() [p. 382].

void QTextEdit::copyAvailable (bool yes) [signal]

This signal is emitted when text is selected or de-selected in the text edit.

When text is selected this signal will be emitted with *yes* set to TRUE. If no text has been selected or if the selected text is de-selected this signal is emitted with *yes* set to FALSE.

If *yes* is TRUE then copy() can be used to copy the selection to the clipboard. If *yes* is FALSE then copy() does nothing.

See also selectionChanged() [p. 390].

QPopupMenu * QTextEdit::createPopupMenu (const QPoint & pos) [virtual protected]

This function is called to create a right mouse button popup menu at the document position *pos*. If you want to create a custom popup menu, reimplement this function and return the created popup menu. Ownership of the popup menu is transferred to the caller.

QPopupMenu * QTextEdit::createPopupMenu () [virtual protected]

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

This function is called to create a right mouse button popup menu. If you want to create a custom popup menu, reimplement this function and return the created popup menu. Ownership of the popup menu is transferred to the caller.

void QTextEdit::currentAlignmentChanged (int a) [signal]

This signal is emitted if the alignment of the current paragraph has changed.

The new alignment is *a*.

See also setAlignment() [p. 390].

void QTextEdit::currentColorChanged (const QColor & c) [signal]

This signal is emitted if the color of the current format has changed.

The new color is *c*.

See also setColor() [p. 390].

void QTextEdit::currentFontChanged (const QFont & f) [signal]

This signal is emitted if the font of the current format has changed.

The new font is *f*.

See also setCurrentFont() [p. 390].

void QTextEdit::currentVerticalAlignmentChanged (VerticalAlignment a) [signal]

This signal is emitted if the vertical alignment of the current format has changed.

The new vertical alignment is *a*.

See also setVerticalAlignment() [p. 393].

void QTextEdit::cursorPositionChanged (QTextCursor * c) [signal]

This signal is emitted if the position of the cursor changed. *c* points to the text cursor object.

See also setCursorPosition() [p. 390].

void QTextEdit::cursorPositionChanged (int para, int pos) [signal]

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

This signal is emitted if the position of the cursor changed. *para* contains the paragraph index and *pos* contains the character position within the paragraph.

See also setCursorPosition() [p. 390].

void QTextEdit::cut () [virtual slot]

Copies the selected text (from selection 0) to the clipboard and deletes it from the text edit.

If there is no selected text (in selection 0) nothing happens.

See also QTextEdit::copy() [p. 381], paste() [p. 388] and pasteSubType() [p. 388].

void QTextEdit::del () [virtual slot]

If there is some selected text (in selection 0) it is deleted. If there is no selected text (in selection 0) the character to the right of the text cursor is deleted.

See also removeSelectedText() [p. 389] and cut() [p. 383].

void QTextEdit::doKeyboardAction (KeyboardAction action) [virtual slot]

Executes keyboard action *action*. This is normally called by a key event handler.

QString QTextEdit::documentTitle () const

Returns the title of the document parsed from the text. See the "documentTitle" [p. 396] property for details.

void QTextEdit::ensureCursorVisible () [virtual slot]

Ensures that the cursor is visible by scrolling the text edit if necessary.

See also setCursorPosition() [p. 390].

QString QTextEdit::family () const

Returns the font family of the current format.

See also setFamily() [p. 391], setCurrentFont() [p. 390] and setPointSize() [p. 392].

bool QTextEdit::find (const QString & expr, bool cs, bool wo, bool forward = TRUE, int * para = 0, int * index = 0) [virtual]

Finds the next occurrence of the string, *expr*, starting from character position **index* within paragraph **para*. Both *index* and *para* must be non-null int pointers.

If **para* and **index* are both 0 the search begins from the start of the text. If *cs* is TRUE the search is case sensitive, otherwise it is case insensitive. If *wo* is TRUE the search looks for whole word matches only; otherwise it searches for any matching text. If *forward* is TRUE (the default) the search works forward from the starting position to the end of the text, otherwise it works backwards to the beginning of the text.

If *expr* is found the function returns TRUE and sets **para* to the number of the paragraph in which the first character of the match was found and sets **index* to the index position of that character within the **para*.

If *expr* is not found the function returns FALSE and the contents of *index* and *para* are undefined.

bool QTextEdit::focusNextPrevChild (bool n) [virtual protected]

Reimplemented to allow tabbing through links. If *n* is TRUE the tab moves the focus to the next child; if *n* is FALSE the tab moves the focus to the previous child. Returns TRUE if the focus was moved; otherwise returns FALSE.

QFont QTextEdit::font () const

Returns the font of the current format.

See also setCurrentFont() [p. 390], setFamily() [p. 391] and setPointSize() [p. 392].

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

void QTextEdit::getCursorPosition (int * para, int * index) const

This function sets the **para* and **index* parameters to the current cursor position. *para* and *index* must be non-null int pointers.

See also setCursorPosition() [p. 390].

bool QTextEdit::getFormat (int para, int index, QFont * font, QColor * color, VerticalAlignment * verticalAlignment) [virtual]

This function gets the format of the character at position *index* in paragraph *para*. Sets *font* to the character's font, *color* to the character's color and *verticalAlignment* to the character's vertical alignment.

Returns FALSE if *para* or *index* is out of range otherwise returns TRUE.

bool QTextEdit::getParagraphFormat (int para, QFont * font, QColor * color, VerticalAlignment * verticalAlignment, int * alignment, QStyleSheetItem::DisplayMode * displayMode, QStyleSheetItem::ListStyle * listStyle, int * listDepth) [virtual]

This function gets the format of the paragraph *para*. Sets *font* to the paragraphs's font, *color* to the paragraph's color, *verticalAlignment* to the paragraph's vertical alignment, *alignment* to the paragraph's alignment, *displayMode* to the paragraph's display mode, *listStyle* to the paragraph's list style (if the display mode is QStyleSheetItem::DisplayListItem) and *listDepth* to the depth of the list (if the display mode is QStyleSheetItem::DisplayListItem).

Returns FALSE if *para* is out of range otherwise returns TRUE.

void QTextEdit::getSelection (int * paraFrom, int * indexFrom, int * paraTo, int * indexTo, int selNum = 0) const

If there is a selection, **paraFrom* is set to the number of the paragraph in which the selection begins and **paraTo* is set to the number of the paragraph in which the selection ends. (They could be the same.) **indexFrom* is set to the index at which the selection begins within **paraFrom*, and **indexTo* is set to the index at which the selection ends within **paraTo*.

If there is no selection, **paraFrom*, **indexFrom*, **paraTo* and **indexTo* are all set to -1.

paraFrom, *indexFrom*, *paraTo* and *indexTo* must be non-null int pointers.

The *selNum* is the number of the selection (multiple selections are supported). It defaults to 0 (the default selection).

See also setSelection() [p. 392] and selectedText [p. 397].

bool QTextEdit::hasSelectedText () const

Returns TRUE if some text is selected in selection 0; otherwise returns FALSE. See the "hasSelectedText" [p. 396] property for details.

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

Returns how many pixels high the text edit needs to be to display all the text if the text edit is *w* pixels wide.

Reimplemented from QWidget [p. 432].

void QTextEdit::indent () [virtual slot]

Re-indents the current paragraph.

void QTextEdit::insert (const QString & text, bool indent = FALSE, bool checkNewLine = TRUE, bool removeSelected = TRUE) [virtual slot]

Inserts *text* at the current cursor position. If *indent* is TRUE, the paragraph is re-indented. If *checkNewLine* is TRUE, newline characters in *text* result in hard line breaks (i.e. new paragraphs). If *checkNewLine* is FALSE the behaviour of the editor is undefined if the *text* contains newlines. If *removeSelected* is TRUE, any selected text (in selection 0) is removed before the text is inserted.

See also paste() [p. 388] and pasteSubType() [p. 388].

void QTextEdit::insertAt (const QString & text, int para, int index) [virtual slot]

Inserts *text* in the paragraph *para* and position *index*

void QTextEdit::insertParagraph (const QString & text, int para) [virtual slot]

Inserts *text* as the paragraph at position *para*. If *para* is -1, the text is appended.

bool QTextEdit::isModified () const

Returns TRUE if the document has been modified by the user; otherwise returns FALSE. See the "modified" [p. 397] property for details.

bool QTextEdit::isOverwriteMode () const

Returns the text edit's overwrite mode. See the "overwriteMode" [p. 397] property for details.

bool QTextEdit::isReadOnly () const

Returns TRUE if the text edit is read-only; otherwise returns FALSE. See the "readOnly" [p. 397] property for details.

bool QTextEdit::isRedoAvailable () const

Returns whether redo is available

bool QTextEdit::isUndoAvailable () const

Returns whether undo is available

bool QTextEdit::isUndoRedoEnabled () const

Returns TRUE if undo/redo is enabled; otherwise returns FALSE. See the "undoRedoEnabled" [p. 398] property for details.

bool QTextEdit::italic () const

Returns TRUE if the current format is italic; otherwise returns FALSE.

See also setItalic() [p. 391].

void QTextEdit::keyPressEvent (QKeyEvent * e) [virtual protected]

Processes the key event, *e*. By default key events are used to provide keyboard navigation and text editing.

Reimplemented from QWidget [p. 436].

Reimplemented in QTextBrowser.

int QTextEdit::length () const

Returns the number of characters in the text. See the "length" [p. 396] property for details.

int QTextEdit::lineOfChar (int para, int index)

Returns the line number of the line in paragraph *para* in which the character at position *index* appears. The *index* position is relative to the beginning of the paragraph. If there is no such paragraph or no such character at the *index* position (e.g. the index is out of range) -1 is returned.

int QTextEdit::lines () const

Returns the number of lines in the text edit.

Warning: This function may be slow. Lines change all the time during word wrapping, so this function has to iterate over all the paragraphs and get the number of lines from each one individually.

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

int QTextEdit::linesOfParagraph (int para) const

Returns the number of lines in paragraph *para*.

bool QTextEdit::linkUnderline () const

Returns TRUE if hypertext links will be underlined; otherwise returns FALSE. See the "linkUnderline" [p. 396] property for details.

QMimeSourceFactory * QTextEdit::mimeSourceFactory () const

Returns the QMimeSourceFactory which is currently used by this text edit.

See also setMimeSourceFactory() [p. 391].

Examples: helpviewer/helpwindow.cpp and qdir/qdir.cpp.

void QTextEdit::modificationChanged (bool m) [signal]

This signal is emitted when the modification of the document changed. If *m* is TRUE, the document was modified, otherwise the modification state has been reset to unmodified.

See also modified [p. 397].

void QTextEdit::moveCursor (CursorAction action, bool select) [virtual slot]

Moves the text cursor according to *action*. This is normally used by some key event handler. *select* specifies whether the text between the current cursor position and the new position should be selected.

QBrush QTextEdit::paper () const

Returns the background (paper) brush. See the "paper" [p. 397] property for details.

int QTextEdit::paragraphAt (const QPoint & pos) const

Returns the paragraph which is at position *pos* (in contents coordinates), or -1 if there is no paragraph at *pos*.

QColor QTextEdit::paragraphBackgroundColor (int para) const

Returns the background color of the paragraph *para* or an invalid color if *para* is out of range or the paragraph has no background set

int QTextEdit::paragraphLength (int para) const

Returns the length of the paragraph *para* (number of characters)

QRect QTextEdit::paragraphRect (int para) const

Returns the rectangle of the paragraph *para* in contents coordinates, or an invalid rectangle if *para* is out of range.

int QTextEdit::paragraphs () const

Returns the number of paragraphs in the text.

void QTextEdit::paste () [virtual slot]

Pastes the text from the clipboard into the text edit at the current cursor position. Only plain text is pasted.

If there is no text in the clipboard nothing happens.

See also pasteSubType() [p. 388], cut() [p. 383] and QTextEdit::copy() [p. 381].

void QTextEdit::pasteSubType (const QCString & subtype) [virtual slot]

Pastes the text with format *subtype* from the clipboard into the text edit at the current cursor position. The *subtype* can be "plain" or "html".

If there is no text with format *subtype* in the clipboard nothing happens.

See also paste() [p. 388], cut() [p. 383] and QTextEdit::copy() [p. 381].

void QTextEdit::placeCursor (const QPoint & pos, QTextCursor * c = 0) [virtual slot]

Places the cursor *c* at the character which is closest to position *pos* (in contents coordinates). If *c* is 0, the default text cursor is used.

See also setCursorPosition() [p. 390].

int QTextEdit::pointSize () const

Returns the point size of the font of the current format.

See also setFamily() [p. 391], setCurrentFont() [p. 390] and setPointSize() [p. 392].

void QTextEdit::redo () [virtual slot]

Redoes the last operation.

If there is no operation to redo, e.g. there is no redo step in the undo/redo history, nothing happens.

See also redoAvailable() [p. 389], undo() [p. 394] and undoDepth [p. 398].

void QTextEdit::redoAvailable (bool yes) [signal]

This signal is emitted when the availability of redo changes. If *yes* is TRUE, then redo() will work until redoAvailable(FALSE) is next emitted.

See also redo() [p. 388] and undoDepth [p. 398].

void QTextEdit::removeParagraph (int para) [virtual slot]

Removes the paragraph *para*

void QTextEdit::removeSelectedText (int selNum = 0) [virtual slot]

Deletes the selected text (i.e. the default selection's text) of the selection *selNum* (by default, 0). If there is no selected text nothing happens.

See also selectedText [p. 397] and removeSelection() [p. 389].

void QTextEdit::removeSelection (int selNum = 0) [virtual slot]

Removes the selection *selNum* (by default 0). This does not remove the selected text.

See also removeSelectedText() [p. 389].

void QTextEdit::repaintChanged () [protected]

Repaints any paragraphs that have changed.

Although used extensively internally you shouldn't need to call this yourself.

void QTextEdit::returnPressed () [signal]

This signal is emitted if the user pressed the Return or the Enter key.

void QTextEdit::scrollToAnchor (const QString & name) [virtual slot]

Scrolls the text edit to make the text at the anchor called *name* visible, if it can be found in the document. If the anchor isn't found no scrolling will occur. An anchor is defined using the HTML anchor tag, e.g. `<a name="target">`.

void QTextEdit::scrollToBottom () [virtual slot]

Scrolls to the bottom of the document and does formatting if required

void QTextEdit::selectAll (bool select = TRUE) [virtual slot]

If *select* is TRUE (the default), all the text is selected as selection 0. If *select* is FALSE any selected text is unselected, i.e., the default selection (selection 0) is cleared.

See also selectedText [p. 397].

QString QTextEdit::selectedText () const

Returns the selected text (from selection 0) or an empty string if there is no currently selected text (in selection 0). See the "selectedText" [p. 397] property for details.

void QTextEdit::selectionChanged () [signal]

This signal is emitted whenever the selection changes.

See also setSelection() [p. 392] and copyAvailable() [p. 382].

void QTextEdit::setAlignment (int a) [virtual slot]

Sets the alignment of the current paragraph to *a*. Valid alignments are Qt::AlignLeft, Qt::AlignRight, Qt::AlignJustify and Qt::AlignCenter (which centers horizontally).

See also setParagType() [p. 391].

Reimplemented in QMultiLineEdit.

void QTextEdit::setBold (bool b) [virtual slot]

If *b* is TRUE sets the current format to bold; otherwise sets the current format to non-bold.

See also bold() [p. 381].

void QTextEdit::setColor (const QColor & c) [virtual slot]

Sets the color of the current format, i.e. of the text, to *c*.

See also color() [p. 381] and paper [p. 397].

Example: action/actiongroup/editor.cpp.

void QTextEdit::setCurrentFont (const QFont & f) [virtual slot]

Sets the font of the current format to *f*.

See also font() [p. 384], setPointSize() [p. 392] and setFamily() [p. 391].

void QTextEdit::setCursorPosition (int para, int index) [virtual slot]

Sets the cursor to position *index* in paragraph *para*.

See also getCursorPosition() [p. 384].

void QTextEdit::setFamily (const QString & fontFamily) [virtual slot]

Sets the font family of the current format to *fontFamily*.

See also family() [p. 383] and setCurrentFont() [p. 390].

void QTextEdit::setItalic (bool b) [virtual slot]

If *b* is TRUE sets the current format to italic; otherwise sets the current format to non-italic.

See also italic() [p. 386].

void QTextEdit::setLinkUnderline (bool) [virtual slot]

Sets whether hypertext links will be underlined. See the "linkUnderline" [p. 396] property for details.

void QTextEdit::setMimeSourceFactory (QMimeSourceFactory * factory) [virtual slot]

Sets the text edit's mimesource factory to *factory*. See QMimeSourceFactory for further details.

See also mimeSourceFactory() [p. 387].

void QTextEdit::setModified (bool m) [virtual slot]

Sets whether the document has been modified by the user to *m*. See the "modified" [p. 397] property for details.

void QTextEdit::setOverwriteMode (bool b) [virtual slot]

Sets the text edit's overwrite mode to *b*. See the "overwriteMode" [p. 397] property for details.

void QTextEdit::setPaper (const QBrush & pap) [virtual slot]

Sets the background (paper) brush to *pap*. See the "paper" [p. 397] property for details.

void QTextEdit::setParagType (QStyleSheetItem::DisplayMode dm, QStyleSheetItem::ListStyle listStyle) [virtual slot]

Sets the paragraph style of the current paragraph to *dm*. If *dm* is QStyleSheetItem::DisplayListItem, the type of the list item is set to *listStyle*.

See also setAlignment() [p. 390].

void QTextEdit::setParagraphBackgroundColor (int para, const QColor & bg) [virtual slot]

Sets the background color of the paragraph *para* to *bg*

void QTextEdit::setPointSize (int s) [virtual slot]

Sets the point size of the current format to *s*.

Note that if *s* is zero or negative, the behaviour of this function is not defined.

See also pointSize() [p. 388], setCurrentFont() [p. 390] and setFamily() [p. 391].

void QTextEdit::setReadOnly (bool b) [virtual slot]

Sets whether the text edit is read-only to *b*. See the "readOnly" [p. 397] property for details.

void QTextEdit::setSelection (int paraFrom, int indexFrom, int paraTo, int indexTo, int selNum = 0) [virtual slot]

Sets a selection which starts at position *indexFrom* in paragraph *paraFrom* and ends at position *indexTo* in paragraph *paraTo*. Existing selections which have a different id (selNum) are not removed, existing selections which have the same id as *selNum* are removed.

Uses the selection settings of selection *selNum*. If *selNum* is 0, this is the default selection.

See also getSelection() [p. 385] and selectedText [p. 397].

void QTextEdit::setSelectionAttributes (int selNum, const QColor & back, bool invertText) [virtual slot]

Sets the background color of selection number *selNum* to *back* and specifies whether the text of this selection should be inverted with *invertText*.

This only works for \selNum > 0. The default selection (\selNum == 0) gets its attributes from the colorGroup() of this widget.

void QTextEdit::setStyleSheet (QStyleSheet * styleSheet) [virtual slot]

Sets the stylesheet to use with this text edit to *styleSheet*. Changes will only take effect for new text added with setText() or append().

See also styleSheet() [p. 394].

void QTextEdit::setTabStopWidth (int ts) [virtual slot]

Sets the tab width used by the text edit to *ts*.

See also tabStopWidth() [p. 394].

void QTextEdit::setText (const QString & txt) [slot]

Sets the text edit's text to *txt*. See the "text" [p. 397] property for details.

void QTextEdit::setText (const QString & text, const QString & context) [virtual slot]

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

Changes the text of the text edit to the string *text* and the context to *context*. Any previous text is removed.

text may be interpreted either as plain text or as rich text, depending on the textFormat(). The default setting is AutoText, i.e. the text edit autodetects the format from *text*.

The optional *context* is a path which the text edit's QMimeSourceFactory uses to resolve the locations of files and images. (See QTextEdit::QTextEdit().) It is passed to the text edit's QMimeSourceFactory when quering data.

See also text [p. 397] and textFormat [p. 398].

void QTextEdit::setTextFormat (TextFormat f) [virtual slot]

Sets the text format: rich text, plain text or auto text to *f*. See the "textFormat" [p. 398] property for details.

void QTextEdit::setUnderline (bool b) [virtual slot]

If *b* is TRUE sets the current format to underline; otherwise sets the current format to non-underline.

See also underline() [p. 394].

void QTextEdit::setUndoDepth (int d) [virtual slot]

Sets the depth of the undo history to *d*. See the "undoDepth" [p. 398] property for details.

void QTextEdit::setUndoRedoEnabled (bool b) [virtual slot]

Sets whether undo/redo is enabled to *b*. See the "undoRedoEnabled" [p. 398] property for details.

void QTextEdit::setVerticalAlignment (VerticalAlignment a) [virtual slot]

Sets the vertical alignment of the current format, i.e. of the text, to *a*.

See also color() [p. 381] and paper [p. 397].

void QTextEdit::setWordWrap (WordWrap mode) [virtual slot]

Sets the word wrap mode to *mode*. See the "wordWrap" [p. 398] property for details.

void QTextEdit::setWrapColumnOrWidth (int) [virtual slot]

Sets the position (in pixels or columns depending on the wrap mode) where text will be wrapped. See the "wrapColumnOrWidth" [p. 398] property for details.

void QTextEdit::setWrapPolicy (WrapPolicy policy) [virtual slot]

Sets the word wrap policy, at whitespace or anywhere to *policy*. See the "wrapPolicy" [p. 399] property for details.

QStyleSheet * QTextEdit::styleSheet () const

Returns the QStyleSheet which is currently used in this text edit.

See also setStyleSheet() [p. 392].

Example: helpviewer/helpwindow.cpp.

int QTextEdit::tabStopWidth () const

Returns the tab width used by the text edit.

See also setTabStopWidth() [p. 392].

QString QTextEdit::text () const

Returns the text edit's text. See the "text" [p. 397] property for details.

QString QTextEdit::text (int para) const

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

Returns the text of paragraph *para*.

If textFormat() is RichText the text will contain HTML formatting tags.

void QTextEdit::textChanged () [signal]

This signal is emitted whenever the text in the text edit changes.

See also text [p. 397] and append() [p. 381].

Examples: helpviewer/helpwindow.cpp, qwerty/qwerty.cpp and rot13/rot13.cpp.

QTextCursor * QTextEdit::textCursor () const [protected]

Returns the text edit's text cursor. QTextCursor is not in the public API, but in special circumstances you might wish to use it.

TextFormat QTextEdit::textFormat () const

Returns the text format: rich text, plain text or auto text. See the "textFormat" [p. 398] property for details.

bool QTextEdit::underline () const

Returns TRUE if the current format is underlined; otherwise returns FALSE.

See also setUnderline() [p. 393].

void QTextEdit::undo () [virtual slot]

Undoes the last operation.

If there is no operation to undo, e.g. there is no undo step in the undo/redo history, nothing happens.

See also undoAvailable() [p. 395], redo() [p. 388] and undoDepth [p. 398].

void QTextEdit::undoAvailable (bool yes) [signal]

This signal is emitted when the availability of undo changes. If *yes* is TRUE, then undo() will work until undoAvailable(FALSE) is next emitted.

See also undo() [p. 394] and undoDepth [p. 398].

int QTextEdit::undoDepth () const

Returns the depth of the undo history. See the "undoDepth" [p. 398] property for details.

void QTextEdit::updateStyles () [protected]

Updates all the rendering styles used to display the text. You will probably want to call this function after calling setStyleSheet().

WordWrap QTextEdit::wordWrap () const

Returns the word wrap mode. See the "wordWrap" [p. 398] property for details.

int QTextEdit::wrapColumnOrWidth () const

Returns the position (in pixels or columns depending on the wrap mode) where text will be wrapped. See the "wrapColumnOrWidth" [p. 398] property for details.

WrapPolicy QTextEdit::wrapPolicy () const

Returns the word wrap policy, at whitespace or anywhere. See the "wrapPolicy" [p. 399] property for details.

void QTextEdit::zoomIn (int range) [virtual slot]

Zooms in on the text by by making the base font size *range* points larger and recalculating all font sizes. This does not change the size of any images.

See also zoomOut() [p. 396].

void QTextEdit::zoomIn () [virtual slot]

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

Zooms in on the text by by making the base font size one point larger and recalculating all font sizes. This does not change the size of any images.

See also zoomOut() [p. 396].

void QTextEdit::zoomOut (int range) [virtual slot]

Zooms out on the text by making the base font size *range* points smaller and recalculating all font sizes. This does not change the size of any images.

See also zoomIn() [p. 395].

void QTextEdit::zoomOut () [virtual slot]

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

Zooms out on the text by by making the base font size one point smaller and recalculating all font sizes. This does not change the size of any images.

See also zoomIn() [p. 395].

void QTextEdit::zoomTo (int size) [virtual slot]

Zooms the text by making the base font size *size* points and recalculating all font sizes. This does not change the size of any images.

Property Documentation

QString documentTitle

This property holds the title of the document parsed from the text.

For PlainText the title will be an empty string. For RichText the title will be the text between the `<title>` tags, if present, otherwise an empty string.

Get this property's value with documentTitle().

bool hasSelectedText

This property holds whether some text is selected in selection 0.

Get this property's value with hasSelectedText().

int length

This property holds the number of characters in the text.

Get this property's value with length().

bool linkUnderline

This property holds whether hypertext links will be underlined.

If TRUE (the default) hypertext links will be displayed underlined. If FALSE links will not be displayed underlined.

Set this property's value with setLinkUnderline() and get this property's value with linkUnderline().

bool modified

This property holds whether the document has been modified by the user.

Set this property's value with setModified() and get this property's value with isModified().

bool overwriteMode

This property holds the text edit's overwrite mode.

If FALSE (the default) characters entered by the user are inserted with any characters to the right being moved out of the way. If TRUE, the editor is in overwrite mode, i.e. characters entered by the user overwrite any characters to the right of the cursor position.

Set this property's value with setOverwriteMode() and get this property's value with isOverwriteMode().

QBrush paper

This property holds the background (paper) brush.

The brush that is currently used to draw the background of the text edit. The initial setting is an empty brush.

Set this property's value with setPaper() and get this property's value with paper().

bool readOnly

This property holds whether the text edit is read-only.

In a read-only text edit the user can only navigate through the text and select text; modifying the text is not possible.

This property's default is FALSE.

Set this property's value with setReadOnly() and get this property's value with isReadOnly().

QString selectedText

This property holds the selected text (from selection 0) or an empty string if there is no currently selected text (in selection 0).

The text is always returned as PlainText regardless of the text format. In a future version of Qt an HTML subset *may* be returned depending on the text format.

See also hasSelectedText [p. 396].

Get this property's value with selectedText().

QString text

This property holds the text edit's text.

There is no default text.

On setting, any previous text is deleted.

The text may be interpreted either as plain text or as rich text, depending on the textFormat(). The default setting is AutoText, i.e. the text edit autodetects the format of the text.

See also textFormat [p. 398].

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

TextFormat textFormat

This property holds the text format: rich text, plain text or auto text.

The text format is one of the following:

- PlainText - all characters, except newlines, are displayed verbatim, including spaces. Whenever a newline appears in the text the text edit inserts a hard line break and begins a new paragraph.
- RichText - rich text rendering. The available styles are defined in the default stylesheet QStyleSheet::defaultSheet().
- AutoText - this is the default. The text edit autodetects which rendering style is best, PlainText or RichText. This is done by using the QStyleSheet::mightBeRichText() function.

Set this property's value with setTextFormat() and get this property's value with textFormat().

int undoDepth

This property holds the depth of the undo history.

The maximum number of steps in the undo/redo history. The default is 100.

See also undo() [p. 394] and redo() [p. 388].

Set this property's value with setUndoDepth() and get this property's value with undoDepth().

bool undoRedoEnabled

This property holds whether undo/redo is enabled.

The default is TRUE.

Set this property's value with setUndoRedoEnabled() and get this property's value with isUndoRedoEnabled().

WordWrap wordWrap

This property holds the word wrap mode.

The default mode is WidgetWidth which causes words to be wrapped at the right edge of the text edit. Wrapping occurs at whitespace, keeping whole words intact. If you want wrapping to occur within words use setWrapPolicy(). If you set a wrap mode of FixedPixelWidth or FixedColumnWidth you should also call setWrapColumnOrWidth() with the width you want.

See also WordWrap [p. 379], wrapColumnOrWidth [p. 398] and wrapPolicy [p. 399].

Set this property's value with setWordWrap() and get this property's value with wordWrap().

int wrapColumnOrWidth

This property holds the position (in pixels or columns depending on the wrap mode) where text will be wrapped.

If the wrap mode is FixedPixelWidth, the value is the number of pixels from the left edge of the text edit at which text should be wrapped. If the wrap mode is FixedColumnWidth, the value is the column number (in character columns) from the left edge of the text edit at which text should be wrapped.

See also wordWrap [p. 398].

Set this property's value with setWrapColumnOrWidth() and get this property's value with wrapColumnOrWidth().

WrapPolicy wrapPolicy

This property holds the word wrap policy, at whitespace or anywhere.

Defines where text can be wrapped when word wrap mode is not NoWrap. The choices are AtWhiteSpace (the default) and Anywhere.

See also wordWrap [p. 398].

Set this property's value with setWrapPolicy() and get this property's value with wrapPolicy().

QTimeEdit Class Reference

The QTimeEdit class provides a time editor.

`#include <qdatetimeedit.h>`

Public Members

- **QTimeEdit** (QWidget * parent = 0, const char * name = 0)
- **QTimeEdit** (const QTime & time, QWidget * parent = 0, const char * name = 0)
- **~QTimeEdit** ()
- virtual void **setTime** (const QTime & time)
- QTime **time** () const
- virtual void **setAutoAdvance** (bool advance)
- bool **autoAdvance** () const
- virtual void **setMinValue** (const QTime & d)
- QTime **minValue** () const
- virtual void **setMaxValue** (const QTime & d)
- QTime **maxValue** () const
- virtual void **setRange** (const QTime & min, const QTime & max)
- QString **separator** () const
- virtual void **setSeparator** (const QString & s)

Signals

- void **valueChanged** (const QTime & time)

Properties

- bool **autoAdvance** — whether the editor automatically advances to the next section
- QTime **maxValue** — the maximum time value
- QTime **minValue** — the minimum time value
- QTime **time** — the time value of the editor

Protected Members

- virtual QString **sectionFormattedText** (int sec)
- virtual void **setHour** (int h)
- virtual void **setMinute** (int m)
- virtual void **setSecond** (int s)

Protected Slots

- void **updateButtons** ()

Detailed Description

The QTimeEdit class provides a time editor.

QTimeEdit allows the user to edit times by using the keyboard or the arrow keys to increase/decrease time values. The arrow keys can be used to move from section to section within the QTimeEdit box. The user can automatically be moved to the next section once they complete a section using setAutoAdvance(). Times appear in hour, minute, second order. It is recommended that the QTimeEdit be initialised with a time, e.g.

```
QTime timeNow = QTime::currentTime();
QTimeEdit *timeEdit = new QTimeEdit( timeNow, this );
timeEdit->setRange( timeNow, timeNow.addSecs( 60 * 60 ) );
```

Here we've created a QTimeEdit widget set to the current time. We've also set the minimum value to the current time and the maximum time to one hour from now.

The maximum and minimum values for a time value in the time editor default to the maximum and minimum values for a QTime. You can change this by calling setMinValue(), setMaxValue() or setRange().

Terminology: A QTimeWidget consists of three sections, one each for the hour, minute and second. You can change the separator character using setSeparator(), by default the separator is read from the system's settings.

See also QTime [Additional Functionality with Qt], QDateEdit [p. 47], QDateTimeEdit [p. 53], Advanced Widgets and Time and Date.

Member Function Documentation

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

Constructs an empty time edit with parent *parent* and name *name*.

QTimeEdit::QTimeEdit (const QTime & time, QWidget * parent = 0, const char * name = 0)

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

Constructs a time edit with the initial time value, *time*, parent *parent* and name *name*.

QTimeEdit::~QTimeEdit ()

Destroys the object and frees any allocated resources.

bool QTimeEdit::autoAdvance () const

Returns TRUE if the editor automatically advances to the next section; otherwise returns FALSE. See the "autoAdvance" [p. 403] property for details.

QTime QTimeEdit::maxValue () const

Returns the maximum time value. See the "maxValue" [p. 403] property for details.

QTime QTimeEdit::minValue () const

Returns the minimum time value. See the "minValue" [p. 403] property for details.

QString QTimeEdit::sectionFormattedText (int sec) [virtual protected]

Returns the formatted number for section *sec*. This will correspond to either the hour, minute or second section, depending on *sec*.

QString QTimeEdit::separator () const

Returns the separator for the editor.

void QTimeEdit::setAutoAdvance (bool advance) [virtual]

Sets whether the editor automatically advances to the next section to *advance*. See the "autoAdvance" [p. 403] property for details.

void QTimeEdit::setHour (int h) [virtual protected]

Sets the hour to *h*, which must be a valid hour, i.e. in the range 0..24.

void QTimeEdit::setMaxValue (const QTime & d) [virtual]

Sets the maximum time value to *d*. See the "maxValue" [p. 403] property for details.

void QTimeEdit::setMinValue (const QTime & d) [virtual]

Sets the minimum time value to *d*. See the "minValue" [p. 403] property for details.

void QTimeEdit::setMinute (int m) [virtual protected]

Sets the minute to *m*, which must be a valid minute, i.e. in the range 0..59.

void QTimeEdit::setRange (const QTime & min, const QTime & max) [virtual]

Sets the valid input range for the editor to be from *min* to *max* inclusive. If *min* is invalid no minimum time is set. Similarly, if *max* is invalid no maximum time is set.

void QTimeEdit::setSecond (int s) [virtual protected]

Sets the second to *s*, which must be a valid second, i.e. in the range 0..59.

void QTimeEdit::setSeparator (const QString & s) [virtual]

Sets the separator to *s*. Note that currently only the first character of *s* is used.

void QTimeEdit::setTime (const QTime & time) [virtual]

Sets the time value of the editor to *time*. See the "time" [p. 404] property for details.

QTime QTimeEdit::time () const

Returns the time value of the editor. See the "time" [p. 404] property for details.

void QTimeEdit::updateButtons () [protected slot]

Enables/disables the push buttons according to the min/max time for this widget.

void QTimeEdit::valueChanged (const QTime & time) [signal]

This signal is emitted whenever the editor's value changes. The *time* parameter is the new value.

Property Documentation

bool autoAdvance

This property holds whether the editor automatically advances to the next section.

If autoAdvance is TRUE, the editor will automatically advance focus to the next time section if a user has completed a section. The default is FALSE.

Set this property's value with setAutoAdvance() and get this property's value with autoAdvance().

QTime maxValue

This property holds the maximum time value.

Setting the maximum time value is equivalent to calling QTimeEdit::setRange(minValue(), *t*), where *t* is the maximum time. The default maximum time is 23:59:59.

Set this property's value with setMaxValue() and get this property's value with maxValue().

QTime minValue

This property holds the minimum time value.

Setting the minimum time value is equivalent to calling QTimeEdit::setRange(*t*, maxValue()), where *t* is the minimum time. The default minimum time is 00:00:00.

Set this property's value with setMinValue() and get this property's value with minValue().

QTime time

This property holds the time value of the editor.

When changing the time property, if the time is less than minValue(), or is greater than maxValue(), nothing happens.

Set this property's value with setTime() and get this property's value with time().

QVBox Class Reference

The QVBox widget provides vertical geometry management on its children.

`#include <qvbox.h>`

Inherits QHBox [Events, Actions, Layouts and Styles with Qt].

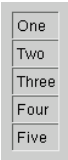
Public Members

- **QVBox** (QWidget * parent = 0, const char * name = 0, WFlags f = 0)

Detailed Description

The QVBox widget provides vertical geometry management on its children.

All its children will be placed vertically and sized according to their sizeHint()s.



See also QHBox [Events, Actions, Layouts and Styles with Qt], Widget Appearance and Style, Layout Management and Organizers.

Member Function Documentation

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

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

QVButtonGroup Class Reference

The QVButtonGroup widget organizes QButton widgets in a vertical column.

`#include <qvbuttongroup.h>`

Inherits QButtonGroup [p. 14].

Public Members

- **QVButtonGroup** (QWidget * parent = 0, const char * name = 0)
- **QVButtonGroup** (const QString & title, QWidget * parent = 0, const char * name = 0)
- **~QVButtonGroup** ()

Detailed Description

The QVButtonGroup widget organizes QButton widgets in a vertical column.

QVButtonGroup is a convenience class that offers a thin layer on top of QButtonGroup. Think of it as a QVBox that offers a frame with a title and is specifically designed for buttons.

See also Widget Appearance and Style, Layout Management and Organizers.

Member Function Documentation

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

Constructs a vertical button group with no title.

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

QVButtonGroup::QVButtonGroup (const QString & title, QWidget * parent = 0, const char * name = 0)

Constructs a vertical button group with the title *title*.

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

QVButtonGroup::~QVButtonGroup ()

Destroys the vertical button group, deleting its child widgets.

QVGroupBox Class Reference

The QVGroupBox widget organizes a group of widgets in a vertical column.

`#include <qvgroupbox.h>`

Inherits QGroupBox [p. 78].

Public Members

- **QVGroupBox** (QWidget * parent = 0, const char * name = 0)
- **QVGroupBox** (const QString & title, QWidget * parent = 0, const char * name = 0)
- **~QVGroupBox** ()

Detailed Description

The QVGroupBox widget organizes a group of widgets in a vertical column.

QVGroupBox is a convenience class that offers a thin layer on top of QGroupBox. Think of it as a QVBox that offers a frame with a title.

See also Widget Appearance and Style, Layout Management and Organizers.

Member Function Documentation

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

Constructs a vertical group box with no title.

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

QVGroupBox::QVGroupBox (const QString & title, QWidget * parent = 0, const char * name = 0)

Constructs a vertical group box with the title *title*.

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

QVGroupBox::~QVGroupBox ()

Destroys the vertical group box, deleting its child widgets.

QWhatsThis Class Reference

The QWhatsThis class provides a simple description of any widget, i.e. answering the question "What's this?".

`#include <qwhatsthis.h>`

Inherits Qt [Additional Functionality with Qt].

Public Members

- **QWhatsThis** (QWidget * widget)
- virtual **~QWhatsThis** ()
- virtual QString **text** (const QPoint &)
- virtual bool **clicked** (const QString & href)

Static Public Members

- void **add** (QWidget * widget, const QString & text)
- void **remove** (QWidget * widget)
- QString **textFor** (QWidget * w, const QPoint & pos = QPoint ( ), bool includeParents = FALSE)
- QToolButton * **whatsThisButton** (QWidget * parent)
- void **enterWhatsThisMode** ()
- bool **inWhatsThisMode** ()
- void **leaveWhatsThisMode** (const QString & text = QString::null, const QPoint & pos = QCursor::pos ( ), QWidget * w = 0)
- void **display** (const QString & text, const QPoint & pos = QCursor::pos ( ), QWidget * w = 0)

Detailed Description

The QWhatsThis class provides a simple description of any widget, i.e. answering the question "What's this?".

"What's this?" help is part of an application's online help system that provides users with information about functionality, usage, background etc. in various levels of detail from short tool tips to full text browsing help windows.

QWhatsThis provides a single window with an explanatory text which pops up when the user asks "What's this?". The default way to do this is to focus the relevant widget and press Shift+F1. The help text appears immediately; it goes away as soon as the user does something else.

(Note that if there is an accelerator for Shift+F1, this mechanism will not work.)

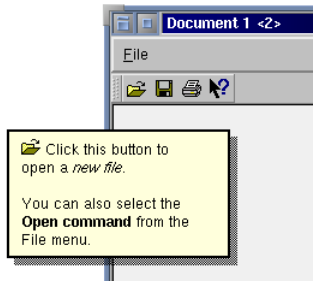
To add "What's this?" text to a widget you simply call QWhatsThis::add() for the widget. For example, to assign text to a menu item, call QMenuData::setWhatsThis(); for a global accelerator key, call QAccel::setWhatsThis() and If you're using actions, use QAction::setWhatsThis().

The text can be either rich text or plain text. If you specify a rich text formatted string, it will be rendered using the default stylesheet. This makes it possible to embed images. See QStyleSheet::defaultSheet() for details.

```
const char * fileOpenText = " "
                "Click this button to open a new file. "
                "You can also select the Open command "
                "from the File menu.";
QMimeSourceFactory::defaultFactory()->setPixmap( "fileopen",
                     fileOpenAction->iconSet().pixmap() );
fileOpenAction->setWhatsThis( fileOpenText );
```

(For further explanation of the above code refer to the Simple Application Walkthrough featuring QAction.)

An alternative way to enter "What's this?" mode is to use the ready-made tool bar tool button from QWhatsThis::whatsThisButton(). By invoking this context help button (in the picture below the first one from the right) the user switches into "What's this?" mode. If they now click on a widget the appropriate help text is shown. The mode is left when help is given or when the user presses Esc.



If you are using QMainWindow you can also use the QMainWindow::whatsThis() slot to invoke the mode from a menu item.

For more control you can create a dedicated QWhatsThis object for a special widget. By subclassing and reimplementing QWhatsThis::text() it is possible to have different help texts, depending on the position of the mouse click. By reimplementing QWhatsThis::clicked() it is possible to have hyperlinks inside the help texts.

If you wish to control the "What's this?" behavior of a widget manually see QWidget::customWhatsThis().

The What's This object can be removed using QWhatsThis::remove(), although this is rarely necessary because it is automatically removed when the widget is destroyed.

See also QToolTip [Dialogs and Windows with Qt] and Help System.

Member Function Documentation

QWhatsThis::QWhatsThis (QWidget * widget)

Constructs a dynamic "What's this?" object for *widget*. The object is deleted when the passed widget is destroyed.

When the widget is queried by the user the text() function of this QWhatsThis will be called to provide the appropriate text, rather than using the text assigned by add().

QWhatsThis::~QWhatsThis () [virtual]

Destroys the object and frees any allocated resources.

void QWhatsThis::add (QWidget * widget, const QString & text) [static]

Adds *text* as "What's this" help for *widget*. If the text is rich text formatted (i.e. it contains markup) it will be rendered with the default stylesheet QStyleSheet::defaultSheet().

The text is destroyed if the widget is later destroyed, so it need not be explicitly removed.

See also remove() [p. 411].

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

bool QWhatsThis::clicked (const QString & href) [virtual]

This virtual function is called when the user clicks inside the "What's this?" window. *href* is the link the user clicked on, or QString::null if there was no link.

If the function returns TRUE (the default), the "What's this?" window is closed, otherwise it remains visible.

The default implementation ignores *href* and returns TRUE.

void QWhatsThis::display (const QString & text, const QPoint & pos = QCursor::pos (), QWidget * w = 0) [static]

Display *text* in a help window at the global screen position *pos*.

If widget *w* is not null and has its own dedicated QWhatsThis object, this object will receive clicked() messages when the user clicks on hyperlinks inside the help text.

See also QWhatsThis::clicked() [p. 410].

void QWhatsThis::enterWhatsThisMode () [static]

Enters "What's this?" mode and returns immediately.

Qt will install a special cursor and take over mouse input until the user clicks somewhere. It then shows any help available and switches out of "What's this?" mode. Finally, Qt removes the special cursor and help window and then restores ordinary event processing, at which point the left mouse button is no longer pressed.

The user can also use the Esc key to leave "What's this?" mode.

See also inWhatsThisMode() [p. 410] and leaveWhatsThisMode() [p. 410].

bool QWhatsThis::inWhatsThisMode () [static]

Returns TRUE if the application is in "What's this?" mode; otherwise returns FALSE.

See also enterWhatsThisMode() [p. 410] and leaveWhatsThisMode() [p. 410].

void QWhatsThis::leaveWhatsThisMode (const QString & text = QString::null, const QPoint & pos = QCursor::pos (), QWidget * w = 0) [static]

Leaves "What's this?" question mode.

This function is used internally by widgets that support QWidget::customWhatsThis(); applications do not usually call it. An example of such a widget is QPopupMenu: menus still work normally in "What's this?" mode but also provide help texts for individual menu items.

If *text* is not a null string, a "What's this?" help window is displayed at the global screen position *pos*. If widget *w* is not null and has its own dedicated QWhatsThis object, this object will receive clicked() messages when the user clicks on hyperlinks inside the help text.

See also inWhatsThisMode() [p. 410], enterWhatsThisMode() [p. 410] and QWhatsThis::clicked() [p. 410].

void QWhatsThis::remove (QWidget * widget) [static]

Removes the "What's this?" help associated with the *widget*. This happens automatically if the widget is destroyed.

See also add() [p. 410].

QString QWhatsThis::text (const QPoint &) [virtual]

This virtual function returns the text for position *p* in the widget that this "What's this?" object documents. If there is no "What's this?" text for a position, QString::null is returned.

The default implementation returns QString::null.

QString QWhatsThis::textFor (QWidget * w, const QPoint & pos = QPoint ( ), bool includeParents = FALSE) [static]

Returns the what's this text for widget *w* or a null string if there is no "What's this?" help for the widget. *pos* contains the mouse position; this is useful, for example, if you've subclassed to make the text that is displayed position dependent.

If *includeParents* is TRUE, parent widgets are taken into consideration as well.

See also add() [p. 410].

QToolButton * QWhatsThis::whatsThisButton (QWidget * parent) [static]

Creates a QToolButton preconfigured to enter "What's this?" mode when clicked. You will often use this with a tool bar as *parent*:

```
(void) QWhatsThis::whatsThisButton( my_help_tool_bar );
```

QWidget Class Reference

The QWidget class is the base class of all user interface objects.

`#include <qwidget.h>`

Inherits QObject [Additional Functionality with Qt] and QPaintDevice [Graphics with Qt].

Inherited by QButton [p. 4], QFrame [p. 65], QDialog [Dialogs and Windows with Qt], QComboBox [p. 32], QDataBrowser [Databases with Qt], QDataView [Databases with Qt], QDateTimeEdit [p. 53], QDesktopWidget [Dialogs and Windows with Qt], QDial [p. 56], QDockArea [Dialogs and Windows with Qt], QGLWidget [Graphics with Qt], QHeader [Additional Functionality with Qt], QMainWindow [Dialogs and Windows with Qt], QNPWidget, QScrollBar [p. 252], QSizeGrip [p. 278], QSlider [p. 285], QSpinBox [p. 295], QStatusBar [p. 311], QTabBar [p. 318], QTabWidget [p. 359], QWorkspace [Dialogs and Windows with Qt] and QXtWidget [p. 484].

Public Members

- **QWidget** (QWidget * parent = 0, const char * name = 0, WFlags f = 0)
- **~QWidget** ()
- WId **winId** () const
- QStyle & **style** () const
- void **setStyle** (QStyle * style)
- QStyle * **setStyle** (const QString & style)
- bool **isTopLevel** () const
- bool **isDialog** () const
- bool **isPopup** () const
- bool **isDesktop** () const
- bool **isModal** () const
- bool **isEnabled** () const
- bool **isEnabledTo** (QWidget * ancestor) const
- bool isEnabledToTLW () const *(obsolete)*
- QRect **frameGeometry** () const
- const QRect & **geometry** () const
- int **x** () const
- int **y** () const
- QPoint **pos** () const
- QSize **frameSize** () const
- QSize **size** () const
- int **width** () const
- int **height** () const
- QRect **rect** () const
- QRect **childrenRect** () const
- QRegion **childrenRegion** () const

- QSize **minimumSize** () const
- QSize **maximumSize** () const
- int **minimumWidth** () const
- int **minimumHeight** () const
- int **maximumWidth** () const
- int **maximumHeight** () const
- void **setMinimumSize** (const QSize &)
- virtual void **setMinimumSize** (int minw, int minh)
- void **setMaximumSize** (const QSize &)
- virtual void **setMaximumSize** (int maxw, int maxh)
- void **setMinimumWidth** (int minw)
- void **setMinimumHeight** (int minh)
- void **setMaximumWidth** (int maxw)
- void **setMaximumHeight** (int maxh)
- QSize **sizeIncrement** () const
- void **setSizeIncrement** (const QSize &)
- virtual void **setSizeIncrement** (int w, int h)
- QSize **baseSize** () const
- void **setBaseSize** (const QSize &)
- void **setBaseSize** (int basew, int baseh)
- void **setFixedSize** (const QSize & s)
- void **setFixedSize** (int w, int h)
- void **setFixedWidth** (int w)
- void **setFixedHeight** (int h)
- QPoint **mapToGlobal** (const QPoint & pos) const
- QPoint **mapFromGlobal** (const QPoint & pos) const
- QPoint **mapToParent** (const QPoint & pos) const
- QPoint **mapFromParent** (const QPoint & pos) const
- QPoint **mapTo** (QWidget * parent, const QPoint & pos) const
- QPoint **mapFrom** (QWidget * parent, const QPoint & pos) const
- QWidget * **topLevelWidget** () const
- BackgroundMode **backgroundMode** () const
- virtual void **setBackgroundMode** (BackgroundMode)
- void **setBackgroundMode** (BackgroundMode m, BackgroundMode visual)
- const QColor & **foregroundColor** () const
- const QColor & **eraseColor** () const
- virtual void **setEraseColor** (const QColor & color)
- const QPixmap * **erasePixmap** () const
- virtual void **setErasePixmap** (const QPixmap & pixmap)
- const QColorGroup & **colorGroup** () const
- const QPalette & **palette** () const
- bool **ownPalette** () const
- virtual void **setPalette** (const QPalette &)
- void **unsetPalette** ()
- const QColor & **paletteForegroundColor** () const
- void **setPaletteForegroundColor** (const QColor &)
- const QColor & **paletteBackgroundColor** () const
- virtual void **setPaletteBackgroundColor** (const QColor &)
- const QPixmap * **paletteBackgroundPixmap** () const
- virtual void **setPaletteBackgroundPixmap** (const QPixmap &)

- const QBrush & **backgroundBrush** () const
- QFont **font** () const
- bool **ownFont** () const
- virtual void **setFont** (const QFont &)
- void **unsetFont** ()
- QFontMetrics **fontMetrics** () const
- QFontInfo **fontInfo** () const
- const QCursor & **cursor** () const
- bool **ownCursor** () const
- virtual void **setCursor** (const QCursor &)
- virtual void **unsetCursor** ()
- QString **caption** () const
- const QPixmap * **icon** () const
- QString **iconText** () const
- bool **hasMouseTracking** () const
- bool **hasMouse** () const
- virtual void **setMask** (const QBitmap & bitmap)
- virtual void **setMask** (const QRegion & region)
- void **clearMask** ()
- const QColor & backgroundColor () const *(obsolete)*
- virtual void setBackgroundColor (const QColor & c) *(obsolete)*
- const QPixmap * backgroundPixmap () const *(obsolete)*
- virtual void setBackgroundPixmap (const QPixmap & pm) *(obsolete)*
- enum **FocusPolicy** { NoFocus = 0, TabFocus = 0x1, ClickFocus = 0x2, StrongFocus = 0x3, WheelFocus = 0x7 }
- bool **isActiveWindow** () const
- virtual void **setActiveWindow** ()
- bool **isFocusEnabled** () const
- FocusPolicy **focusPolicy** () const
- virtual void **setFocusPolicy** (FocusPolicy)
- bool **hasFocus** () const
- virtual void **setFocusProxy** (QWidget * w)
- QWidget * **focusProxy** () const
- void **grabMouse** ()
- void **grabMouse** (const QCursor & cursor)
- void **releaseMouse** ()
- void **grabKeyboard** ()
- void **releaseKeyboard** ()
- bool **isUpdatesEnabled** () const
- virtual bool **close** (bool alsoDelete)
- bool **isVisible** () const
- bool **isVisibleTo** (QWidget * ancestor) const
- bool isVisibleToTLW () const *(obsolete)*
- QRect **visibleRect** () const
- bool **isHidden** () const
- bool **isMinimized** () const
- bool **isMaximized** () const
- virtual QSize **sizeHint** () const
- virtual QSize **minimumSizeHint** () const
- virtual QSizePolicy **sizePolicy** () const

- virtual void **setSizePolicy** (QSizePolicy)
- virtual int **heightForWidth** (int w) const
- virtual void **adjustSize** ()
- QLayout * **layout** () const
- void **updateGeometry** ()
- virtual void **reparent** (QWidget * parent, WFlags f, const QPoint & p, bool showIt = FALSE)
- void **reparent** (QWidget * parent, const QPoint & p, bool showIt = FALSE)
- void recreate (QWidget * parent, WFlags f, const QPoint & p, bool showIt = FALSE) *(obsolete)*
- void **erase** ()
- void **erase** (int x, int y, int w, int h)
- void **erase** (const QRect & r)
- void **erase** (const QRegion & reg)
- void **scroll** (int dx, int dy)
- void **scroll** (int dx, int dy, const QRect & r)
- void **drawText** (int x, int y, const QString & str)
- void **drawText** (const QPoint & pos, const QString & str)
- QWidget * **focusWidget** () const
- QRect **microFocusHint** () const
- bool **acceptDrops** () const
- virtual void **setAcceptDrops** (bool on)
- virtual void **setAutoMask** (bool)
- bool **autoMask** () const
- enum **BackgroundOrigin** { WidgetOrigin, ParentOrigin, WindowOrigin }
- virtual void **setBackgroundOrigin** (BackgroundOrigin)
- BackgroundOrigin **backgroundOrigin** () const
- virtual bool **customWhatsThis** () const
- QWidget * **parentWidget** (bool sameWindow = FALSE) const
- WFlags **testWFlags** (WFlags f) const
- QWidget * **childAt** (int x, int y, bool includeThis = FALSE) const
- QWidget * **childAt** (const QPoint & p, bool includeThis = FALSE) const
- void setPalette (const QPalette & p, bool) *(obsolete)*
- void setFont (const QFont & f, bool) *(obsolete)*

Public Slots

- virtual void **setEnabled** (bool)
- void **setDisabled** (bool disable)
- virtual void **setCaption** (const QString &)
- virtual void **setIcon** (const QPixmap &)
- virtual void **setIconText** (const QString &)
- virtual void **setMouseTracking** (bool enable)
- virtual void **setFocus** ()
- void **clearFocus** ()
- virtual void **setUpdatesEnabled** (bool enable)
- void **update** ()
- void **update** (int x, int y, int w, int h)
- void **update** (const QRect & r)
- void **repaint** ()
- void **repaint** (bool erase)

- void **repaint** (int x, int y, int w, int h, bool erase = TRUE)
- void **repaint** (const QRect & r, bool erase = TRUE)
- void **repaint** (const QRegion & reg, bool erase = TRUE)
- virtual void **show** ()
- virtual void **hide** ()
- void iconify () *(obsolete)*
- virtual void **showMinimized** ()
- virtual void **showMaximized** ()
- void **showFullScreen** ()
- virtual void **showNormal** ()
- virtual void **polish** ()
- void **constPolish** () const
- bool **close** ()
- void **raise** ()
- void **lower** ()
- void **stackUnder** (QWidget * w)
- virtual void **move** (int x, int y)
- void **move** (const QPoint &)
- virtual void **resize** (int w, int h)
- void **resize** (const QSize &)
- virtual void **setGeometry** (int x, int y, int w, int h)
- virtual void **setGeometry** (const QRect &)

Static Public Members

- void **setTabOrder** (QWidget * first, QWidget * second)
- QWidget * **mouseGrabber** ()
- QWidget * **keyboardGrabber** ()
- QWidget * **find** (WId id)

Properties

- bool **acceptDrops** — whether drop events are enabled for this widget
- bool **autoMask** — whether the auto mask feature is enabled for the widget
- QBrush **backgroundBrush** — the widget's background brush *(read only)*
- BackgroundMode **backgroundMode** — the color role used for painting the background of the widget
- BackgroundOrigin **backgroundOrigin** — the origin of the widget's background
- QSize **baseSize** — the base size of the widget
- QString **caption** — the window caption (title)
- QRect **childrenRect** — the bounding rectangle of the widget's children *(read only)*
- QRegion **childrenRegion** — the combined region occupied by the widget's children *(read only)*
- QColorGroup **colorGroup** — the current color group of the widget palette *(read only)*
- QCursor **cursor** — the cursor shape for this widget
- bool **customWhatsThis** — whether the widget wants to handle What's This help manually *(read only)*
- bool **enabled** — whether the widget is enabled
- bool **focus** — whether this widget (or its focus proxy) has the keyboard input focus *(read only)*
- bool **focusEnabled** — whether the widget accepts keyboard focus *(read only)*
- FocusPolicy **focusPolicy** — the way the widget accepts keyboard focus

- QFont **font** — the font currently set for the widget
- QRect **frameGeometry** — geometry of the widget relative to its parent including any window frame *(read only)*
- QSize **frameSize** — the size of the widget including any window frame *(read only)*
- QRect **geometry** — the geometry of the widget relative to its parent and excluding the window frame
- int **height** — the height of the widget excluding any window frame *(read only)*
- bool **hidden** — whether the widget is explicitly hidden *(read only)*
- QPixmap **icon** — the widget icon pixmap
- QString **iconText** — the widget icon text
- bool **isActiveWindow** — whether this widget is the active window or a child of it *(read only)*
- bool **isDesktop** — whether the widget is a desktop widget *(read only)*
- bool **isDialog** — whether the widget is a dialog widget *(read only)*
- bool **isModal** — whether the widget is a modal widget *(read only)*
- bool **isPopup** — whether the widget is a popup widget *(read only)*
- bool **isTopLevel** — whether the widget is a top-level widget *(read only)*
- int **maximumHeight** — the widget's maximum height
- QSize **maximumSize** — the widget's maximum size
- int **maximumWidth** — the widget's maximum width
- QRect **microFocusHint** — the currently set micro focus hint for this widget *(read only)*
- bool **minimized** — whether this widget is minimized (iconified) *(read only)*
- int **minimumHeight** — the widget's minimum height
- QSize **minimumSize** — the widget's minimum size
- QSize **minimumSizeHint** — the recommended minimum size for the widget *(read only)*
- int **minimumWidth** — the widget's minimum width
- bool **mouseTracking** — whether mouse tracking is enabled for this widget
- bool **ownCursor** — whether the widget uses its own cursor *(read only)*
- bool **ownFont** — whether the widget uses its own font *(read only)*
- bool **ownPalette** — whether the widget uses its own palette *(read only)*
- QPalette **palette** — the widget's palette
- QColor **paletteBackgroundColor** — the background color of the widget
- QPixmap **paletteBackgroundPixmap** — the background pixmap of the widget
- QColor **paletteForegroundColor** — the foreground color of the widget
- QPoint **pos** — the position of the widget in its parent widget
- QRect **rect** — the internal geometry of the widget excluding any window frame *(read only)*
- QSize **size** — the size of the widget excluding any window frame
- QSize **sizeHint** — the recommended size for the widget *(read only)*
- QSize **sizeIncrement** — the size increment of the widget
- QSizePolicy **sizePolicy** — the default layout behavior of the widget
- bool **underMouse** — whether the widget is under the mouse cursor *(read only)*
- bool **updatesEnabled** — whether updates are enabled
- bool **visible** — whether the widget is visible *(read only)*
- QRect **visibleRect** — the currently visible rectangle of the widget *(read only)*
- int **width** — the width of the widget excluding any window frame *(read only)*
- int **x** — the x coordinate of the widget relative to its parent including any window frame *(read only)*
- int **y** — the y coordinate of the widget relative to its parent and including any window frame *(read only)*

Protected Members

- virtual bool **event** (QEvent * e)
- virtual void **mousePressEvent** (QMouseEvent * e)
- virtual void **mouseReleaseEvent** (QMouseEvent * e)
- virtual void **mouseDoubleClickEvent** (QMouseEvent * e)
- virtual void **mouseMoveEvent** (QMouseEvent * e)
- virtual void **wheelEvent** (QWheelEvent * e)
- virtual void **keyPressEvent** (QKeyEvent * e)
- virtual void **keyReleaseEvent** (QKeyEvent * e)
- virtual void **focusInEvent** (QFocusEvent *)
- virtual void **focusOutEvent** (QFocusEvent *)
- virtual void **enterEvent** (QEvent *)
- virtual void **leaveEvent** (QEvent *)
- virtual void **paintEvent** (QPaintEvent *)
- virtual void **moveEvent** (QMoveEvent *)
- virtual void **resizeEvent** (QResizeEvent *)
- virtual void **closeEvent** (QCloseEvent * e)
- virtual void **contextMenuEvent** (QContextMenuEvent * e)
- virtual void **imStartEvent** (QIMEvent * e)
- virtual void **imComposeEvent** (QIMEvent * e)
- virtual void **imEndEvent** (QIMEvent * e)
- virtual void **tabletEvent** (QTabletEvent * e)
- virtual void **dragEnterEvent** (QDragEnterEvent *)
- virtual void **dragMoveEvent** (QDragMoveEvent *)
- virtual void **dragLeaveEvent** (QDragLeaveEvent *)
- virtual void **dropEvent** (QDropEvent *)
- virtual void **showEvent** (QShowEvent *)
- virtual void **hideEvent** (QHideEvent *)
- virtual bool **winEvent** (MSG *)
- virtual bool **x11Event** (XEvent *)
- virtual void **updateMask** ()
- virtual void **styleChange** (QStyle & oldStyle)
- virtual void **enabledChange** (bool oldEnabled)
- virtual void **paletteChange** (const QPalette & oldPalette)
- virtual void **fontChange** (const QFont & oldFont)
- virtual void **windowActivationChange** (bool oldActive)
- virtual int **metric** (int m) const
- void **resetInputContext** ()
- virtual void **create** (WId window = 0, bool initializeWindow = TRUE, bool destroyOldWindow = TRUE)
- virtual void **destroy** (bool destroyWindow = TRUE, bool destroySubWindows = TRUE)
- WFlags **getWFlags** () const
- virtual void **setWFlags** (WFlags f)
- void **clearWFlags** (WFlags f)
- virtual bool **focusNextPrevChild** (bool next)
- QFocusData * **focusData** ()
- virtual void **setKeyCompression** (bool compress)
- virtual void **setMicroFocusHint** (int x, int y, int width, int height, bool text = TRUE, QFont * f = 0)

Detailed Description

The QWidget class is the base class of all user interface objects.

The widget is the atom of the user interface: it receives mouse, keyboard and other events from the window system, and paints a representation of itself on the screen. Every widget is rectangular, and they are sorted in a Z-order. A widget is clipped by its parent and by the widgets in front of it.

A widget that isn't embedded in a parent widget is called a top-level widget. Usually, top-level widgets are windows with a frame and a title bar (though it is also possible to create top level widgets without such decoration by the use of widget flags). In Qt, QMainWindow and the various subclasses of QDialog are the most common top-level windows.

A widget without a parent widget is always a top-level widget.

Non-top-level widgets are child widgets. These are child windows in their parent widgets. You usually cannot distinguish a child widget from its parent visually. Most other widgets in Qt are useful only as child widgets. (You can make a e.g. button into a top-level widget, but most people prefer to put their buttons in e.g. dialogs.)

QWidget has many member functions, but some of them have little direct functionality: for example, QWidget it has a font property, but never uses this itself. There are many subclasses which provide real functionality, as diverse as QPushButton, QListBox and QTabDialog.

Groups of functions:

Window functions: show(), hide(), raise(), lower(), close().

Top level windows: caption(), setCaption(), icon(), setIcon(), iconText(), setIconText(), isActiveWindow(), setActiveWindow(), showMinimized(). showMaximized(), showFullScreen(), showNormal().

Window contents: update(), repaint(), erase(), scroll(), updateMask().

Geometry: pos(), size(), rect(), x(), y(), width(), height(), sizePolicy(), setSizePolicy(), sizeHint(), updateGeometry(), layout(), move(), resize(), setGeometry(), frameGeometry(), geometry(), childrenRect(), adjustSize(), mapFromGlobal(), mapFromParent() mapToGlobal(), mapToParent(), maximumSize(), minimumSize(), sizeIncrement(), setMaximumSize(), setMinimumSize(), setSizeIncrement(), setBaseSize(), setFixedSize()

Mode: isVisible(), isVisibleTo(), visibleRect(), isMinimized(), isDesktop(), isEnabled(), isEnabledTo(), isModal(), isPopup(), isTopLevel(), setEnabled(), hasMouseTracking(), setMouseTracking(), isUpdatesEnabled(), setUpdatesEnabled(),

Look and feel: style(), setStyle(), cursor(), setCursor() font(), setFont(), palette(), setPalette(), backgroundMode(), setBackgroundMode(), colorGroup(), fontMetrics(), fontInfo().

Keyboard focus functions: isFocusEnabled(), setFocusPolicy(), focusPolicy(), hasFocus(), setFocus(), clearFocus(), setTabOrder(), setFocusProxy().

Mouse and keyboard grabbing: grabMouse(), releaseMouse(), grabKeyboard(), releaseKeyboard(), mouseGrabber(), keyboardGrabber().

Event handlers: event(), mousePressEvent(), mouseReleaseEvent(), mouseDoubleClickEvent(), mouseMoveEvent(), keyPressEvent(), keyReleaseEvent(), focusInEvent(), focusOutEvent(), wheelEvent(), enterEvent(), leaveEvent(), paintEvent(), moveEvent(), resizeEvent(), closeEvent(), dragEnterEvent(), dragMoveEvent(), dragLeaveEvent(), dropEvent(), childEvent(), showEvent(), hideEvent(), customEvent().

Change handlers: enabledChange(), fontChange(), paletteChange(), styleChange(), windowActivationChange().

System functions: parentWidget(), topLevelWidget(), reparent(), polish(), winId(), find(), metric().

Internal kernel functions: focusNextPrevChild(), wmapper(), clearWFlags(), getWFlags(), setWFlags(), testWFlags().

What's this help: customWhatsThis()

Every widget's constructor accepts two or three standard arguments:

- `QWidget *parent = 0` is the parent of the new widget. If it is 0 (the default), the new widget will be a top-level window. If not, it will be a child of *parent*, and be constrained by *parent*'s geometry (Unless you specify WType_TopLevel as widget flag).
- `const char *name = 0` is the widget name of the new widget. You can access it using name(). The widget name is little used by programmers but is quite useful with GUI builders such as *Qt Designer* (you can name a widget in the builder, and connect() to it using the name in your code). The dumpObjectTree() debugging function also uses it.
- `WFlags f = 0` (where available) sets the widget flags; the default is good for almost all widgets, but to get e.g. top-level widgets without a window system frame, you must use special flags.

The tictac/tictac.cpp example program is good example of a simple widget. It contains a few event handlers (as all widgets must), a few custom routines that are peculiar to it (as all useful widgets do), and has a few children and connections. Everything it does is done in response to an event: this is by far the most common way to design GUI applications.

You will need to supply the content for your widgets yourself, but here is a brief run-down of the events, starting with the most common ones:

- paintEvent() - called whenever the widget needs to be repainted. Every widget which displays output must implement it, and it is wise *not* to paint on the screen outside paintEvent().
- resizeEvent() - called when the widget has been resized.
- mousePressEvent() - called when a mouse button is pressed. There are six mouse-related events, but mouse press and mouse release events are by far the most important. A widget receives mouse press events when the widget is inside it, or when it has grabbed the mouse using grabMouse().
- mouseReleaseEvent() - called when a mouse button is released. A widget receives mouse release events when it has received the corresponding mouse press event. This means that if the user presses the mouse inside *your* widget, then drags the mouse to somewhere else, then releases, *your* widget receives the release event. There is one exception, however: if a popup menu appears while the mouse button is held down, that popup steals the mouse events at once.
- mouseDoubleClickEvent() - not quite as obvious as it might seem. If the user double-clicks, the widget receives a mouse press event (perhaps a mouse move event or two if they don't hold the mouse quite steady), a mouse release event and finally this event. It is *not possible* to distinguish a click from a double click until you've seen whether the second click arrives. (This is one reason why most GUI books recommend that double clicks be an extension of single clicks, rather than trigger a different action.)

If your widget only contains child widgets, you probably do not need to implement any event handlers.

Widgets that accept keyboard input need to reimplement a few more event handlers:

- keyPressEvent() - called whenever a key is pressed, and again when a key has been held down long enough for it to auto-repeat. Note that the Tab and Shift+Tab keys are only passed to the widget if they are not used by the focus-change mechanisms. To force those keys to be processed by your widget, you must reimplement QWidget::event().
- focusInEvent() - called when the widget gains keyboard focus (assuming you have called setFocusPolicy(), of course). Well written widgets indicate that they own the keyboard focus in a clear but discreet way.
- focusOutEvent() - called when the widget loses keyboard focus.

Some widgets will need to reimplement some more obscure event handlers, too:

- mouseMoveEvent() - called whenever the mouse moves while a button is held down. This is useful for e.g. dragging. If you call setMouseTracking(TRUE), you get mouse move events even when no buttons are held down. (Note that applications which make use of mouse tracking are often not very useful on low-bandwidth X connections.) (See also the drag and drop information.)
- keyReleaseEvent() - called whenever a key is released, and also while it is held down if the key is auto-repeating. In that case the widget receives a key release event and immediately a key press event for every repeat. Note that the Tab and Shift+Tab keys are only passed to the widget if they are not used by the focus-change mechanisms. To force those keys to be processed by your widget, you must reimplement QWidget::event().
- wheelEvent() — called whenever the user turns the mouse wheel while the widget has the focus.
- enterEvent() - called when the mouse enters the widget's screen space. (This excludes screen space owned by any children of the widget.)
- leaveEvent() - called when the mouse leaves the widget's screen space.
- moveEvent() - called when the widget has been moved relative to its parent.
- closeEvent() - called when the user closes the widget (or when close() is called).

There are also some *really* obscure events. They are listed in qevent.h and you need to reimplement event() to handle them. The default implementation of event() handles Tab and Shift+Tab (to move the keyboard focus), and passes on most other events to one of the more specialized handlers above.

When implementing a widget, there are a few more things to look out for.

- In the constructor, be sure to set up your member variables early on, before there's any chance that you might receive an event.
- It is almost always useful to reimplement sizeHint() and to set the correct size policy with setSizePolicy(), so users of your class can set up layout management more easily. A size policy lets you supply good defaults for the layout management handling, so that other widgets can contain and manage yours easily. sizeHint() indicates a "good" size for the widget.
- If your widget is a top-level window, setCaption() and setIcon() set the title bar and icon respectively.

See also QEvent [Events, Actions, Layouts and Styles with Qt], QPainter [Graphics with Qt], QGridLayout [Events, Actions, Layouts and Styles with Qt], QBoxLayout [Events, Actions, Layouts and Styles with Qt] and Abstract Widget Classes.

Member Type Documentation

QWidget::BackgroundOrigin

This enum defines the origin used to draw a widget's background pixmap.

The pixmap is drawn using the:

- `QWidget::WidgetOrigin` - widget's coordinate system.
- `QWidget::ParentOrigin` - parent's coordinate system.
- `QWidget::WindowOrigin` - toplevel window's coordinate system.

QWidget::FocusPolicy

This enum type defines the various policies a widget can have with respect to acquiring keyboard focus.

The *policy* can be:

- `QWidget::TabFocus` - the widget accepts focus by tabbing.
- `QWidget::ClickFocus` - the widget accepts focus by clicking.
- `QWidget::StrongFocus` - the widget accepts focus by both tabbing and clicking.
- `QWidget::WheelFocus` - like StrongFocus plus the widget accepts focus by using the mouse wheel.
- `QWidget::NoFocus` - the widget does not accept focus.

Member Function Documentation

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

Constructs a widget which is a child of *parent*, with the name *name* and widget flags set to *f*.

If *parent* is 0, the new widget becomes a top-level window. If *parent* is another widget, this widget becomes a child window inside *parent*. The new widget is deleted when its *parent* is deleted.

The *name* is sent to the QObject constructor.

The widget flags argument, *f*, is normally 0, but it can be set to customize the window frame of a top-level widget (i.e. *parent* must be 0). To customize the frame, set the WStyle_Customize flag OR'ed with any of the Qt::WidgetFlags.

If you add a child widget to an already visible widget you must explicitly show the child to make it visible.

Note that the X11 version of Qt may not be able to deliver all combinations of style flags on all systems. This is because on X11, Qt can only ask the window manager, and the window manager can override the application's settings. On Windows, Qt can set whatever flags you want.

Example:

```
QLabel *spashScreen = new QLabel( 0, "mySplashScreen",
                                  WStyle_Customize | WStyle_NoBorder |
                                  WStyle_Tool );
```

QWidget::~QWidget ()

Destroys the widget.

All this widget's children are deleted first. The application exits if this widget is the main widget.

bool QWidget::acceptDrops () const

Returns TRUE if drop events are enabled for this widget; otherwise returns FALSE. See the "acceptDrops" [p. 458] property for details.

void QWidget::adjustSize () [virtual]

Adjusts the size of the widget to fit the contents.

Uses sizeHint() if valid (i.e if the size hint's width and height are equal to or greater than 0), otherwise sets the size to the children rectangle (the union of all child widget geometries).

See also sizeHint [p. 470] and childrenRect [p. 460].

Example: xform/xform.cpp.

Reimplemented in QMessageBox.

bool QWidget::autoMask () const

Returns TRUE if the auto mask feature is enabled for the widget; otherwise returns FALSE. See the "autoMask" [p. 459] property for details.

const QBrush & QWidget::backgroundBrush () const

Returns the widget's background brush. See the "backgroundBrush" [p. 459] property for details.

const QColor & QWidget::backgroundColor () const

This function is obsolete. It is provided to keep old source working. We strongly advise against using it in new code. Use paletteBackgroundColor() or eraseColor() instead.

BackgroundMode QWidget::backgroundMode () const

Returns the color role used for painting the background of the widget. See the "backgroundMode" [p. 459] property for details.

BackgroundOrigin QWidget::backgroundOrigin () const

Returns the origin of the widget's background. See the "backgroundOrigin" [p. 460] property for details.

const QPixmap * QWidget::backgroundPixmap () const

This function is obsolete. It is provided to keep old source working. We strongly advise against using it in new code. Use paletteBackgroundPixmap() or erasePixmap() instead.

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

QSize QWidget::baseSize () const

Returns the base size of the widget. See the "baseSize" [p. 460] property for details.

QString QWidget::caption () const

Returns the window caption (title). See the "caption" [p. 460] property for details.

QWidget * QWidget::childAt (int x, int y, bool includeThis = FALSE) const

Returns the visible child widget at pixel position *(x, y)* in the widget's own coordinate system.

If *includeThis* is TRUE, and there is no child visible at *(x, y)*, the widget itself is returned.

QWidget * QWidget::childAt (const QPoint & p, bool includeThis = FALSE) const

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

Returns the visible child widget at point *p* in the widget's own coordinate system.

If *includeThis* is TRUE, and there is no child visible at *p*, the widget itself is returned.

QRect QWidget::childrenRect () const

Returns the bounding rectangle of the widget's children. See the "childrenRect" [p. 460] property for details.

QRegion QWidget::childrenRegion () const

Returns the combined region occupied by the widget's children. See the "childrenRegion" [p. 460] property for details.

void QWidget::clearFocus () [slot]

Takes keyboard input focus from the widget.

If the widget has active focus, a focus out event is sent to this widget to tell it that it is about to lose the focus.

This widget must enable focus setting in order to get the keyboard input focus, i.e. it must call setFocusPolicy().

See also focus [p. 462], setFocus() [p. 448], focusInEvent() [p. 429], focusOutEvent() [p. 430], focusPolicy [p. 462] and QApplication::focusWidget() [Additional Functionality with Qt].

void QWidget::clearMask ()

Removes any mask set by setMask().

See also setMask() [p. 450].

void QWidget::clearWFlags (WFlags f) [protected]

Clears the widget flags *f*.

Widget flags are a combination of Qt::WidgetFlags.

See also testWFlags() [p. 455], getWFlags() [p. 431] and setWFlags() [p. 453].

bool QWidget::close () [slot]

Closes this widget. Returns TRUE if the widget was closed; otherwise returns FALSE.

First it sends the widget a QCloseEvent. The widget is hidden if it accepts the close event. The default implementation of QWidget::closeEvent() accepts the close event.

The QApplication::lastWindowClosed() signal is emitted when the last visible top level widget is closed.

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

bool QWidget::close (bool alsoDelete) [virtual]

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

Closes this widget. Returns TRUE if the widget was closed; otherwise returns FALSE.

If *alsoDelete* is TRUE or the widget has the WDestructiveClose widget flag, the widget is also deleted. The widget can prevent itself from being closed by rejecting the QCloseEvent it gets.

The QApplication::lastWindowClosed() signal is emitted when the last visible top level widget is closed.

Note that closing the QApplication::mainWidget() terminates the application.

See also closeEvent() [p. 425], QCloseEvent [Events, Actions, Layouts and Styles with Qt], hide() [p. 433], QApplication::quit() [Additional Functionality with Qt], QApplication::setMainWidget() [Additional Functionality with Qt] and QApplication::lastWindowClosed() [Additional Functionality with Qt].

void QWidget::closeEvent (QCloseEvent * e) [virtual protected]

This event handler, for event *e*, can be reimplemented in a subclass to receive widget close events.

The default implementation calls e->accept(), which hides this widget. See the QCloseEvent [Events, Actions, Layouts and Styles with Qt] documentation for more details.

See also event() [p. 428], hide() [p. 433], close() [p. 424] and QCloseEvent [Events, Actions, Layouts and Styles with Qt].

Examples: action/application.cpp, application/application.cpp, i18n/mywidget.cpp, popup/popup.cpp and qwerty/qwerty.cpp.

const QColorGroup & QWidget::colorGroup () const

Returns the current color group of the widget palette. See the "colorGroup" [p. 460] property for details.

void QWidget::constPolish () const [slot]

Ensures that the widget is properly initialized by calling polish().

Call constPolish() from functions like sizeHint() that depends on the widget being initialized, and that may be called before show().

Warning: Do not call constPolish() on a widget from inside that widget's constructor.

See also polish() [p. 442].

void QWidget::contextMenuEvent (QContextMenuEvent * e) [virtual protected]

This event handler, for event *e*, can be reimplemented in a subclass to receive widget context menu events.

The default implementation calls e->ignore(), which rejects the context event. See the QContextMenuEvent [Events, Actions, Layouts and Styles with Qt] documentation for more details.

See also event() [p. 428] and QContextMenuEvent [Events, Actions, Layouts and Styles with Qt].

void QWidget::create (WId window = 0, bool initializeWindow = TRUE, bool destroyOldWindow = TRUE) [virtual protected]

Creates a new widget window if *window* is null, otherwise sets the widget's window to *window*.

Initializes the window (sets the geometry etc.) if *initializeWindow* is TRUE. If *initializeWindow* is FALSE, no initialization is performed. This parameter makes only sense if *window* is a valid window.

Destroys the old window if *destroyOldWindow* is TRUE. If *destroyOldWindow* is FALSE, you are responsible for destroying the window yourself (using platform native code).

The QWidget constructor calls create(0,TRUE,TRUE) to create a window for this widget.

const QCursor & QWidget::cursor () const

Returns the cursor shape for this widget. See the "cursor" [p. 461] property for details.

bool QWidget::customWhatsThis () const [virtual]

Returns TRUE if the widget wants to handle What's This help manually; otherwise returns FALSE. See the "customWhatsThis" [p. 461] property for details.

void QWidget::destroy (bool destroyWindow = TRUE, bool destroySubWindows = TRUE) [virtual protected]

Frees up window system resources. Destroys the widget window if *destroyWindow* is TRUE.

destroy() calls itself recursively for all the child widgets, passing *destroySubWindows* for the *destroyWindow* parameter. To have more control over destruction of subwidgets, destroy subwidgets selectively first.

This function is usually called from the QWidget destructor.

void QWidget::dragEnterEvent (QDragEnterEvent *) [virtual protected]

This event handler is called when a drag is in progress and the mouse enters this widget.

See the Drag-and-drop documentation [Programming with Qt] for an overview of how to provide drag-and-drop in your application.

See also QTextDrag [Events, Actions, Layouts and Styles with Qt], QImageDrag [Events, Actions, Layouts and Styles with Qt] and QDragEnterEvent [Events, Actions, Layouts and Styles with Qt].

void QWidget::dragLeaveEvent (QDragLeaveEvent *) [virtual protected]

This event handler is called when a drag is in progress and the mouse leaves this widget.

See the Drag-and-drop documentation [Programming with Qt] for an overview of how to provide drag-and-drop in your application.

See also QTextDrag [Events, Actions, Layouts and Styles with Qt], QImageDrag [Events, Actions, Layouts and Styles with Qt] and QDragLeaveEvent [Events, Actions, Layouts and Styles with Qt].

void QWidget::dragMoveEvent (QDragMoveEvent *) [virtual protected]

This event handler is called when a drag is in progress and the mouse enters this widget, and whenever it moves within the widget.

See the Drag-and-drop documentation [Programming with Qt] for an overview of how to provide drag-and-drop in your application.

See also QTextDrag [Events, Actions, Layouts and Styles with Qt], QImageDrag [Events, Actions, Layouts and Styles with Qt] and QDragMoveEvent [Events, Actions, Layouts and Styles with Qt].

void QWidget::drawText (int x, int y, const QString & str)

Draws the string *str* at position *(x, y)*.

The *y* position is the base line position of the text. The text is drawn using the default font and the default foreground color.

This function is provided for convenience. You will generally get more flexible results and often higher speed by using a a painter instead.

See also font [p. 462], foregroundColor() [p. 431] and QPainter::drawText() [Graphics with Qt].

void QWidget::drawText (const QPoint & pos, const QString & str)

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

Draws the string *str* at position *pos*.

void QWidget::dropEvent (QDropEvent *) [virtual protected]

This event handler is called when the drag is dropped on this widget.

See the Drag-and-drop documentation [Programming with Qt] for an overview of how to provide drag-and-drop in your application.

See also QTextDrag [Events, Actions, Layouts and Styles with Qt], QImageDrag [Events, Actions, Layouts and Styles with Qt] and QDropEvent [Events, Actions, Layouts and Styles with Qt].

void QWidget::enabledChange (bool oldEnabled) [virtual protected]

This virtual function is called from setEnabled(). *oldEnabled* is the previous setting; you can get the new setting from isEnabled().

Reimplement this function if your widget needs to know when it becomes enabled or disabled. You will almost certainly need to update the widget using update().

The default implementation repaints the visible part of the widget.

See also enabled [p. 461], enabled [p. 461], repaint() [p. 443], update() [p. 456] and visibleRect [p. 471].

void QWidget::enterEvent (QEvent *) [virtual protected]

This event handler can be reimplemented in a subclass to receive widget enter events.

An event is sent to the widget when the mouse cursor enters the widget.

See also leaveEvent() [p. 437], mouseMoveEvent() [p. 439] and event() [p. 428].

void QWidget::erase (int x, int y, int w, int h)

Erases the specified area *(x, y, w, h)* in the widget without generating a paint event.

If *w* is negative, it is replaced with `width() - x`. If *h* is negative, it is replaced width `height() - y`.

Child widgets are not affected.

See also repaint() [p. 443].

void QWidget::erase ()

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

This version erases the entire widget.

void QWidget::erase (const QRect & r)

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

Erases the specified area *r* in the widget without generating a paint event.

void QWidget::erase (const QRegion & reg)

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

Erases the area defined by *reg*, without generating a paint event.

Child widgets are not affected.

const QColor & QWidget::eraseColor () const

Returns the erase color of the widget.

See also setEraseColor() [p. 447], setErasePixmap() [p. 447] and backgroundColor() [p. 423].

const QPixmap * QWidget::erasePixmap () const

Returns the widget's erase pixmap.

See also setErasePixmap() [p. 447] and eraseColor() [p. 428].

bool QWidget::event (QEvent * e) [virtual protected]

This is the main event handler; it handles event *e*. You may reimplement this function in a subclass, but we recommend using one of the specialized event handlers instead.

The main event handler first passes an event through all event filters that have been installed. If none of the filters intercept the event, it calls one of the specialized event handlers.

Key press and release events are treated differently from other events. event() checks for Tab and Shift+Tab and tries to move the focus appropriately. If there is no widget to move the focus to (or the key press is not Tab or Shift+Tab), event() calls keyPressEvent().

This function returns TRUE if it is able to pass the event over to someone, or FALSE if nobody wanted the event.

See also closeEvent() [p. 425], focusInEvent() [p. 429], focusOutEvent() [p. 430], enterEvent() [p. 427], keyPressEvent() [p. 436], keyReleaseEvent() [p. 436], leaveEvent() [p. 437], mouseDoubleClickEvent() [p. 439], mouseMoveEvent() [p. 439], mousePressEvent() [p. 440], mouseReleaseEvent() [p. 440], moveEvent() [p. 440], paintEvent() [p. 441], resizeEvent() [p. 445], QObject::event() [Additional Functionality with Qt] and QObject::timerEvent() [Additional Functionality with Qt].

Reimplemented from QObject [Additional Functionality with Qt].

QWidget * QWidget::find (WId id) [static]

Returns a pointer to the widget with window identifer/handle *id*.

The window identifier type depends on the underlying window system, see qwindowdefs.h for the actual definition. If there is no widget with this identifier, a null pointer is returned.

QFocusData * QWidget::focusData () [protected]

Returns a pointer to the focus data for this widget's top-level widget.

Focus data always belongs to the top-level widget. The focus data list contains all the widgets in this top-level widget that can accept focus, in tab order. An iterator points to the current focus widget (focusWidget() returns a pointer to this widget).

This information is useful for implementing advanced versions of focusNextPrevChild().

void QWidget::focusInEvent (QFocusEvent *) [virtual protected]

This event handler can be reimplemented in a subclass to receive keyboard focus events (focus received) for the widget.

A widget normally must setFocusPolicy() to something other than NoFocus in order to receive focus events. (Note that the application programmer can call setFocus() on any widget, even those that do not normally accept focus.)

The default implementation updates the widget if it accepts focus (see focusPolicy()). It also calls setMicroFocusHint(), hinting any system-specific input tools about the focus of the user's attention.

See also focusOutEvent() [p. 430], focusPolicy [p. 462], keyPressEvent() [p. 436], keyReleaseEvent() [p. 436], event() [p. 428] and QFocusEvent [Events, Actions, Layouts and Styles with Qt].

Reimplemented in QtMultiLineEdit.

bool QWidget::focusNextPrevChild (bool next) [virtual protected]

Finds a new widget to give the keyboard focus to, as appropriate for Tab and Shift+Tab, and returns TRUE if is can find a new widget and FALSE if it can't,

If *next* is TRUE, this function searches "forwards", if *next* is FALSE, it searches "backwards".

Sometimes, you will want to reimplement this function. For example, a web browser might reimplement it to move its "current active link" forwards or backwards, and call QWidget::focusNextPrevChild() only when it reaches the last or first link on the "page".

Child widgets call focusNextPrevChild() on their parent widgets, but only the top-level widget decides where to redirect focus. By overriding this method for an object, you thus gain control of focus traversal for all child widgets.

See also focusData() [p. 429].

void QWidget::focusOutEvent (QFocusEvent *) [virtual protected]

This event handler can be reimplemented in a subclass to receive keyboard focus events (focus lost) for the widget.

A widget normally must setFocusPolicy() to something other than NoFocus in order to receive focus events. (Note that the application programmer can call setFocus() on any widget, even those that do not normally accept focus.)

The default implementation calls repaint() since the widget's colorGroup() changes from active to normal, so the widget probably needs repainting. It also calls setMicroFocusHint(), hinting any system-specific input tools about the focus of the user's attention.

See also focusInEvent() [p. 429], focusPolicy [p. 462], keyPressEvent() [p. 436], keyReleaseEvent() [p. 436], event() [p. 428] and QFocusEvent [Events, Actions, Layouts and Styles with Qt].

Example: qmag/qmag.cpp.

FocusPolicy QWidget::focusPolicy () const

Returns the way the widget accepts keyboard focus. See the "focusPolicy" [p. 462] property for details.

QWidget * QWidget::focusProxy () const

Returns a pointer to the focus proxy, or 0 if there is no focus proxy.

See also setFocusProxy() [p. 448].

QWidget * QWidget::focusWidget () const

Returns the focus widget in this widget's window. This is not the same as QApplication::focusWidget(), which returns the focus widget in the currently active window.

QFont QWidget::font () const

Returns the font currently set for the widget. See the "font" [p. 462] property for details.

void QWidget::fontChange (const QFont & oldFont) [virtual protected]

This virtual function is called from setFont(). *oldFont* is the previous font; you can get the new font from font().

Reimplement this function if your widget needs to know when its font changes. You will almost certainly need to update the widget using update().

The default implementation updates the widget including its geometry.

See also font [p. 462], font [p. 462], update() [p. 456] and updateGeometry() [p. 457].

QFontInfo QWidget::fontInfo () const

Returns the font info for the widget's current font. Equivalent to QFontInto(widget->font()).

See also font [p. 462], fontMetrics() [p. 431] and font [p. 462].

QFontMetrics QWidget::fontMetrics () const

Returns the font metrics for the widget's current font. Equivalent to QFontMetrics(widget->font()).

See also font [p. 462], fontInfo() [p. 430] and font [p. 462].

Examples: drawdemo/drawdemo.cpp and qmag/qmag.cpp.

const QColor & QWidget::foregroundColor () const

Same as paletteForegroundColor()

QRect QWidget::frameGeometry () const

Returns geometry of the widget relative to its parent including any window frame. See the "frameGeometry" [p. 463] property for details.

QSize QWidget::frameSize () const

Returns the size of the widget including any window frame. See the "frameSize" [p. 463] property for details.

const QRect & QWidget::geometry () const

Returns the geometry of the widget relative to its parent and excluding the window frame. See the "geometry" [p. 463] property for details.

WFlags QWidget::getWFlags () const [protected]

Returns the widget flags for this this widget.

Widget flags are a combination of Qt::WidgetFlags.

See also testWFlags() [p. 455], setWFlags() [p. 453] and clearWFlags() [p. 424].

void QWidget::grabKeyboard ()

Grabs the keyboard input.

This widget reveives all keyboard events and other widgets none until releaseKeyboard() is called. Mouse events are not affected. Use grabMouse() if you want to grab that.

The focus widget is not affected, except that it doesn't receive any keyboard events. setFocus() moves the focus as usual, but the new focus widget receives keyboard events only after releaseKeyboard() is called.

If a different widget is currently grabbing keyboard input, that widget's grab is released first.

See also releaseKeyboard() [p. 443], grabMouse() [p. 431], releaseMouse() [p. 443] and focusWidget() [p. 430].

void QWidget::grabMouse ()

Grabs the mouse input.

This widget receives all mouse events and other widgets none until releaseMouse() is called. Keyboard events are not affected. Use grabKeyboard() if you want to grab that.

Warning: Bugs in mouse-grabbing applications very often lock the terminal. Use this function with extreme caution, and consider using the -nograb command line option while debugging.

It is almost never necessary to grab the mouse when using Qt, as Qt grabs and releases it sensibly. In particular, Qt grabs the mouse when a mouse button is pressed and keeps it until the last button is released.

Note that only visible widgets can grab mouse input. If isVisible() returns FALSE for a widget, that widget can not call grabMouse().

See also releaseMouse() [p. 443], grabKeyboard() [p. 431], releaseKeyboard() [p. 443], grabKeyboard() [p. 431] and focusWidget() [p. 430].

void QWidget::grabMouse (const QCursor & cursor)

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

Grabs the mouse input and changes the cursor shape.

The cursor will assume shape *cursor* (for as long as the mouse focus is grabbed) and this widget will be the only one to receive mouse events until releaseMouse() is called().

Warning: Grabbing the mouse might lock the terminal.

See also releaseMouse() [p. 443], grabKeyboard() [p. 431], releaseKeyboard() [p. 443] and cursor [p. 461].

bool QWidget::hasFocus () const

Returns TRUE if this widget (or its focus proxy) has the keyboard input focus; otherwise returns FALSE. See the "focus" [p. 462] property for details.

bool QWidget::hasMouse () const

Returns TRUE if the widget is under the mouse cursor; otherwise returns FALSE. See the "underMouse" [p. 470] property for details.

bool QWidget::hasMouseTracking () const

Returns TRUE if mouse tracking is enabled for this widget; otherwise returns FALSE. See the "mouseTracking" [p. 467] property for details.

int QWidget::height () const

Returns the height of the widget excluding any window frame. See the "height" [p. 463] property for details.

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

Returns the preferred height for this widget, given the width *w*. The default implementation returns 0, indicating that the preferred height does not depend on the width.

Warning: Does not look at the widget's layout.

Reimplemented in QMenuBar and QTextEdit.

void QWidget::hide () [virtual slot]

Hides the widget.

You almost never have to reimplement this function. If you need to do something after a widget is hidden, use hideEvent() instead.

See also hideEvent() [p. 433], hidden [p. 463], show() [p. 453], showMinimized() [p. 454], visible [p. 471] and close() [p. 424].

Examples: mdi/application.cpp, network/ftpclient/ftpmainwindow.cpp, popup/popup.cpp, progress/progress.cpp, scrollview/scrollview.cpp and xform/xform.cpp.

Reimplemented in QMenuBar.

void QWidget::hideEvent (QHideEvent *) [virtual protected]

This event handler can be reimplemented in a subclass to receive widget hide events.

Hide events are sent to widgets immediately after they have been hidden.

See also event() [p. 428] and QHideEvent [Events, Actions, Layouts and Styles with Qt].

const QPixmap * QWidget::icon () const

Returns the widget icon pixmap. See the "icon" [p. 464] property for details.

QString QWidget::iconText () const

Returns the widget icon text. See the "iconText" [p. 464] property for details.

void QWidget::iconify () [slot]

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

void QWidget::imComposeEvent (QIMEvent * e) [virtual protected]

This event handler, for event *e*, can be reimplemented in a subclass to receive Input Method composition events. This handler is called when the user has entered some text via an Input Method.

The default implementation calls e->ignore(), which rejects the Input Method event. See the QIMEvent [Accessibility and Internationalization with Qt] documentation for more details.

See also event() [p. 428] and QIMEvent [Accessibility and Internationalization with Qt].

void QWidget::imEndEvent (QIMEvent * e) [virtual protected]

This event handler, for event *e*, can be reimplemented in a subclass to receive Input Method composition events. This handler is called when the user has finished inputting text via an Input Method.

The default implementation calls e->ignore(), which rejects the Input Method event. See the QIMEvent [Accessibility and Internationalization with Qt] documentation for more details.

See also event() [p. 428] and QIMEvent [Accessibility and Internationalization with Qt].

void QWidget::imStartEvent (QIMEvent * e) [virtual protected]

This event handler, for event *e*, can be reimplemented in a subclass to receive Input Method composition events. This handler is called when the user begins inputting text via an Input Method.

The default implementation calls e->ignore(), which rejects the Input Method event. See the QIMEvent [Accessibility and Internationalization with Qt] documentation for more details.

See also event() [p. 428] and QIMEvent [Accessibility and Internationalization with Qt].

bool QWidget::isActiveWindow () const

Returns TRUE if this widget is the active window or a child of it; otherwise returns FALSE. See the "isActiveWindow" [p. 464] property for details.

bool QWidget::isDesktop () const

Returns TRUE if the widget is a desktop widget; otherwise returns FALSE. See the "isDesktop" [p. 464] property for details.

bool QWidget::isDialog () const

Returns TRUE if the widget is a dialog widget; otherwise returns FALSE. See the "isDialog" [p. 464] property for details.

bool QWidget::isEnabled () const

Returns TRUE if the widget is enabled; otherwise returns FALSE. See the "enabled" [p. 461] property for details.

bool QWidget::isEnabledTo (QWidget * ancestor) const

Returns TRUE if this widget would become enabled if *ancestor* is enabled; otherwise returns FALSE.

This is the case if neither the widget itself nor every parent up to but excluding *ancestor* has been explicitly disabled.

isEnabledTo(0) is equivalent to isEnabled().

See also enabled [p. 461] and enabled [p. 461].

bool QWidget::isEnabledToTLW () const

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

This function is deprecated. It is equivalent to isEnabled()

bool QWidget::isFocusEnabled () const

Returns TRUE if the widget accepts keyboard focus; otherwise returns FALSE. See the "focusEnabled" [p. 462] property for details.

bool QWidget::isHidden () const

Returns TRUE if the widget is explicitly hidden; otherwise returns FALSE. See the "hidden" [p. 463] property for details.

bool QWidget::isMaximized () const

Returns TRUE if this widget is a top-level widget that is maximized; otherwise returns FALSE.

Note that due to limitations in some window-systems, this does not always report the expected results (e.g. if the user on X11 maximizes the window via the window manager, Qt has no way of distinguishing this from any other resize). This is expected to improve as window manager protocols advance.

See also showMaximized() [p. 454].

bool QWidget::isMinimized () const

Returns TRUE if this widget is minimized (iconified); otherwise returns FALSE. See the "minimized" [p. 466] property for details.

bool QWidget::isModal () const

Returns TRUE if the widget is a modal widget; otherwise returns FALSE. See the "isModal" [p. 464] property for details.

bool QWidget::isPopup () const

Returns TRUE if the widget is a popup widget; otherwise returns FALSE. See the "isPopup" [p. 465] property for details.

bool QWidget::isTopLevel () const

Returns TRUE if the widget is a top-level widget; otherwise returns FALSE. See the "isTopLevel" [p. 465] property for details.

bool QWidget::isUpdatesEnabled () const

Returns TRUE if updates are enabled; otherwise returns FALSE. See the "updatesEnabled" [p. 471] property for details.

bool QWidget::isVisible () const

Returns TRUE if the widget is visible; otherwise returns FALSE. See the "visible" [p. 471] property for details.

bool QWidget::isVisibleTo (QWidget * ancestor) const

Returns TRUE if this widget would become visible if *ancestor* is shown; otherwise returns FALSE.

The TRUE case occurs if neither the widget itself nor any parent up to but excluding *ancestor* has been explicitly hidden.

This function will still return TRUE if the widget it is obscured by other windows on the screen, but could be physically visible if it or they were to be moved.

isVisibleTo(0) is very similar to isVisible(), with the exception that it does not cover the iconfied-case or the situation where the window exists on another virtual desktop.

See also show() [p. 453], hide() [p. 433] and visible [p. 471].

bool QWidget::isVisibleToTLW () const

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

This function is deprecated. It is equivalent to isVisible()

void QWidget::keyPressEvent (QKeyEvent * e) [virtual protected]

This event handler, for event *e*, can be reimplemented in a subclass to receive key press events for the widget.

A widget must call setFocusPolicy() to accept focus initially and have focus in order to receive a key press event.

If you reimplement this handler, it is very important that you ignore() the event if you do not understand it, so that the widget's parent can interpret it.

The default implementation closes popup widgets if the user presses Esc. Otherwise the event is ignored.

See also keyReleaseEvent() [p. 436], QKeyEvent::ignore() [Events, Actions, Layouts and Styles with Qt], focusPolicy [p. 462], focusInEvent() [p. 429], focusOutEvent() [p. 430], event() [p. 428] and QKeyEvent [Events, Actions, Layouts and Styles with Qt].

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

Reimplemented in QLineEdit, QTextEdit and QtMultiLineEdit.

void QWidget::keyReleaseEvent (QKeyEvent * e) [virtual protected]

This event handler, for event *e*, can be reimplemented in a subclass to receive key release events for the widget.

A widget must accept focus initially and have focus in order to receive a key release event.

If you reimplement this handler, it is very important that you ignore() the release if you do not understand it, so that the widget's parent can interpret it.

The default implementation ignores the event.

See also keyPressEvent() [p. 436], QKeyEvent::ignore() [Events, Actions, Layouts and Styles with Qt], focusPolicy [p. 462], focusInEvent() [p. 429], focusOutEvent() [p. 430], event() [p. 428] and QKeyEvent [Events, Actions, Layouts and Styles with Qt].

QWidget * QWidget::keyboardGrabber () [static]

Returns a pointer to the widget that is currently grabbing the keyboard input.

If no widget in this application is currently grabbing the keyboard, 0 is returned.

See also grabMouse() [p. 431] and mouseGrabber() [p. 439].

QLayout * QWidget::layout () const

Returns a pointer to the layout engine that manages the geometry of this widget's children.

If the widget does not have a layout, layout() returns a null pointer.

See also sizePolicy [p. 470].

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

void QWidget::leaveEvent (QEvent *) [virtual protected]

This event handler can be reimplemented in a subclass to receive widget leave events.

A leave event is sent to the widget when the mouse cursor leaves the widget.

See also enterEvent() [p. 427], mouseMoveEvent() [p. 439] and event() [p. 428].

void QWidget::lower () [slot]

Lowers the widget to the bottom of the parent widget's stack.

If there are siblings of this widget that overlap it on the screen, this widget will be obscured by its siblings afterwards.

See also raise() [p. 442] and stackUnder() [p. 455].

QPoint QWidget::mapFrom (QWidget * parent, const QPoint & pos) const

Translates the widget coordinate *pos* from the coordinate system of *parent* to this widget's coordinate system, which must be non-null and be a parent of this widget.

See also mapTo() [p. 437], mapFromParent() [p. 437] and mapFromGlobal() [p. 437].

QPoint QWidget::mapFromGlobal (const QPoint & pos) const

Translates the global screen coordinate *pos* to widget coordinates.

See also mapToGlobal() [p. 438], mapFrom() [p. 437] and mapFromParent() [p. 437].

QPoint QWidget::mapFromParent (const QPoint & pos) const

Translates the parent widget coordinate *pos* to widget coordinates.

Same as mapFromGlobal() if the widget has no parent.

See also mapToParent() [p. 438], mapFrom() [p. 437] and mapFromGlobal() [p. 437].

QPoint QWidget::mapTo (QWidget * parent, const QPoint & pos) const

Translates the widget coordinate *pos* to the coordinate system of *parent*, which must be non-null and be a parent of this widget.

See also mapFrom() [p. 437], mapToParent() [p. 438] and mapToGlobal() [p. 438].

QPoint QWidget::mapToGlobal (const QPoint & pos) const

Translates the widget coordinate *pos* to global screen coordinates. For example,

```
mapToGlobal(QPoint(0,0))
```

would give the global coordinates of the top-left pixel of the widget.

See also mapFromGlobal() [p. 437], mapTo() [p. 437] and mapToParent() [p. 438].

Example: scribble/scribble.cpp.

QPoint QWidget::mapToParent (const QPoint & pos) const

Translates the widget coordinate *pos* to a coordinate in the parent widget.

Same as mapToGlobal() if the widget has no parent.

See also mapFromParent() [p. 437], mapTo() [p. 437] and mapToGlobal() [p. 438].

int QWidget::maximumHeight () const

Returns the widget's maximum height. See the "maximumHeight" [p. 465] property for details.

QSize QWidget::maximumSize () const

Returns the widget's maximum size. See the "maximumSize" [p. 465] property for details.

int QWidget::maximumWidth () const

Returns the widget's maximum width. See the "maximumWidth" [p. 466] property for details.

int QWidget::metric (int m) const [virtual protected]

Internal implementation of the virtual QPaintDevice::metric() function.

Use the QPaintDeviceMetrics class instead.

m is the metric to get.

Reimplemented from QPaintDevice [Graphics with Qt].

QRect QWidget::microFocusHint () const

Returns the currently set micro focus hint for this widget. See the "microFocusHint" [p. 466] property for details.

int QWidget::minimumHeight () const

Returns the widget's minimum height. See the "minimumHeight" [p. 466] property for details.

QSize QWidget::minimumSize () const

Returns the widget's minimum size. See the "minimumSize" [p. 466] property for details.

QSize QWidget::minimumSizeHint () const [virtual]

Returns the recommended minimum size for the widget. See the "minimumSizeHint" [p. 466] property for details.

Reimplemented in QLineEdit and QtMultiLineEdit.

int QWidget::minimumWidth () const

Returns the widget's minimum width. See the "minimumWidth" [p. 467] property for details.

void QWidget::mouseDoubleClickEvent (QMouseEvent * e) [virtual protected]

This event handler, for event *e*, can be reimplemented in a subclass to receive mouse double click events for the widget.

The default implementation generates a normal mouse press event.

Note that the widgets gets a mousePressEvent() and a mouseReleaseEvent() before the mouseDoubleClickEvent().

See also mousePressEvent() [p. 440], mouseReleaseEvent() [p. 440], mouseMoveEvent() [p. 439], event() [p. 428] and QMouseEvent [Events, Actions, Layouts and Styles with Qt].

QWidget * QWidget::mouseGrabber () [static]

Returns a pointer to the widget that is currently grabbing the mouse input.

If no widget in this application is currently grabbing the mouse, 0 is returned.

See also grabMouse() [p. 431] and keyboardGrabber() [p. 436].

void QWidget::mouseMoveEvent (QMouseEvent * e) [virtual protected]

This event handler, for event *e*, can be reimplemented in a subclass to receive mouse move events for the widget.

If mouse tracking is switched off, mouse move events only occur if a mouse button is down while the mouse is being moved. If mouse tracking is switched on, mouse move events occur even if no mouse button is down.

QMouseEvent::pos() reports the position of the mouse cursor, relative to this widget. For press and release events, the position is usually the same as the position of the last mouse move event, but it might be different if the user moves and clicks the mouse fast. This is a feature of the underlying window system, not Qt.

See also mouseTracking [p. 467], mousePressEvent() [p. 440], mouseReleaseEvent() [p. 440], mouseDoubleClickEvent() [p. 439], event() [p. 428] and QMouseEvent [Events, Actions, Layouts and Styles with Qt].

Examples: aclock/aclock.cpp, drawlines/connect.cpp, life/life.cpp, popup/popup.cpp, qmag/qmag.cpp, scribble/scribble.cpp and showimg/showimg.cpp.

Reimplemented in QSizeGrip.

void QWidget::mousePressEvent (QMouseEvent * e) [virtual protected]

This event handler, for event *e*, can be reimplemented in a subclass to receive mouse press events for the widget.

If you create new widgets in the mousePressEvent() the mouseReleaseEvent() may not end up where you expect, depending on the underlying window system (or X11 window manager), the widgets' location and maybe more.

The default implementation implements the closing of popup widgets when you click outside the window. For other widget types it does nothing.

See also mouseReleaseEvent() [p. 440], mouseDoubleClickEvent() [p. 439], mouseMoveEvent() [p. 439], event() [p. 428] and QMouseEvent [Events, Actions, Layouts and Styles with Qt].

Examples: aclock/aclock.cpp, biff/biff.cpp, drawlines/connect.cpp, life/life.cpp, qmag/qmag.cpp, scribble/scribble.cpp and tooltip/tooltip.cpp.

Reimplemented in QSizeGrip.

void QWidget::mouseReleaseEvent (QMouseEvent * e) [virtual protected]

This event handler, for event *e*, can be reimplemented in a subclass to receive mouse release events for the widget.

See also mouseDoubleClickEvent() [p. 439], mouseMoveEvent() [p. 439], event() [p. 428] and QMouseEvent [Events, Actions, Layouts and Styles with Qt].

Examples: drawlines/connect.cpp, hello/hello.cpp, popup/popup.cpp, qmag/qmag.cpp, scribble/scribble.cpp, showimg/showimg.cpp and t14/cannon.cpp.

void QWidget::move (const QPoint &) [slot]

Sets the position of the widget in its parent widget. See the "pos" [p. 469] property for details.

void QWidget::move (int x, int y) [virtual slot]

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

This corresponds to move(QSize(*x*, *y*)).

void QWidget::moveEvent (QMoveEvent *) [virtual protected]

This event handler can be reimplemented in a subclass to receive widget move events. When the widget receives this event, it is already at the new position.

The old position is accessible through QMoveEvent::oldPos().

See also resizeEvent() [p. 445], event() [p. 428], pos [p. 469] and QMoveEvent [Events, Actions, Layouts and Styles with Qt].

bool QWidget::ownCursor () const

Returns TRUE if the widget uses its own cursor; otherwise returns FALSE. See the "ownCursor" [p. 467] property for details.

bool QWidget::ownFont () const

Returns TRUE if the widget uses its own font; otherwise returns FALSE. See the "ownFont" [p. 467] property for details.

bool QWidget::ownPalette () const

Returns TRUE if the widget uses its own palette; otherwise returns FALSE. See the "ownPalette" [p. 467] property for details.

void QWidget::paintEvent (QPaintEvent *) [virtual protected]

This event handler can be reimplemented in a subclass to receive paint events.

A paint event is a request to repaint all or part of the widget. It can happen as a result of repaint() or update(), or because the widget was obscured and has now been uncovered, or for many other reasons.

Many widgets can simply repaint their entire surface when asked to, but some slow widgets need to optimize by painting only the requested region: QPaintEvent::region(). This speed optimization does not change the result, as painting is clipped to that region during event processing. QListView and QCanvas do this, for example.

Qt also tries to speed up painting by merging multiple paint events into one. When update() is called several times or the window system sends several paint events, Qt merges these events into one event with a larger region (see QRegion::unite()). repaint() does not permit this optimization, so we suggest using update() when possible.

When the paint event occurs, the update region normally has been erased, so that you're painting on the widget's background. There are a couple of exceptions and QPaintEvent::erased() tells you whether the widget has been erased or not.

The background can be set using setBackgroundMode(), setPaletteBackgroundColor() or setBackgroundPixmap(). The documentation for setBackgroundMode() elaborates on the background; we recommend reading it.

See also event() [p. 428], repaint() [p. 443], update() [p. 456], QPainter [Graphics with Qt], QPixmap [Graphics with Qt] and QPaintEvent [Events, Actions, Layouts and Styles with Qt].

Examples: drawdemo/drawdemo.cpp, drawlines/connect.cpp, qmag/qmag.cpp, scribble/scribble.cpp, splitter/splitter.cpp, t8/cannon.cpp and t9/cannon.cpp.

Reimplemented in QButton, QFrame, QGLWidget, QSizeGrip, QStatusBar and QTabBar.

const QPalette & QWidget::palette () const

Returns the widget's palette. See the "palette" [p. 468] property for details.

const QColor & QWidget::paletteBackgroundColor () const

Returns the background color of the widget. See the "paletteBackgroundColor" [p. 468] property for details.

const QPixmap * QWidget::paletteBackgroundPixmap () const

Returns the background pixmap of the widget. See the "paletteBackgroundPixmap" [p. 468] property for details.

void QWidget::paletteChange (const QPalette & oldPalette) [virtual protected]

This virtual function is called from setPalette(). *oldPalette* is the previous palette; you can get the new palette from palette().

Reimplement this function if your widget needs to know when its palette changes.

See also palette [p. 468] and palette [p. 468].

const QColor & QWidget::paletteForegroundColor () const

Returns the foreground color of the widget. See the "paletteForegroundColor" [p. 468] property for details.

QWidget * QWidget::parentWidget (bool sameWindow = FALSE) const

Returns a pointer to the parent of this widget, or a null pointer if it does not have any parent widget. If *sameWindow* is TRUE and the widget is top level returns 0; otherwise returns the widget's parent.

void QWidget::polish () [virtual slot]

Delayed initialization of a widget.

This function will be called *after* a widget has been fully created and *before* it is shown the very first time.

Polishing is useful for final initialization which depends on having an instantiated widget. This is something a constructor cannot guarantee since the initialization of the subclasses might not be finished.

After this function, the widget has a proper font and palette and QApplication::polish() has been called.

Remember to call QWidget's implementation when reimplementing this function.

See also constPolish() [p. 425] and QApplication::polish() [Additional Functionality with Qt].

Example: menu/menu.cpp.

QPoint QWidget::pos () const

Returns the position of the widget in its parent widget. See the "pos" [p. 469] property for details.

void QWidget::raise () [slot]

Raises this widget to the top of the parent widget's stack.

If there are any siblings of this widget that overlap it on the screen, this widget will be visually in front of its siblings afterwards.

See also lower() [p. 437] and stackUnder() [p. 455].

Example: showimg/showimg.cpp.

void QWidget::recreate (QWidget * parent, WFlags f, const QPoint & p, bool showIt = FALSE)

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

This method is provided to aid porting from Qt 1.0 to 2.0. It has been renamed reparent() in Qt 2.0.

QRect QWidget::rect () const

Returns the internal geometry of the widget excluding any window frame. See the "rect" [p. 469] property for details.

void QWidget::releaseKeyboard ()

Releases the keyboard grab.

See also grabKeyboard() [p. 431], grabMouse() [p. 431] and releaseMouse() [p. 443].

void QWidget::releaseMouse ()

Releases the mouse grab.

See also grabMouse() [p. 431], grabKeyboard() [p. 431] and releaseKeyboard() [p. 443].

void QWidget::repaint (int x, int y, int w, int h, bool erase = TRUE) [slot]

Repaints the widget directly by calling paintEvent() immediately, unless updates are disabled or the widget is hidden.

If *erase* is TRUE, Qt erases the area *(x,y,w,h)* before the paintEvent() call.

If *w* is negative, it is replaced with `width() - x`, and if *h* is negative, it is replaced width `height() - y`.

We suggest using repaint() if you need an immediate repaint, for example during animation. In almost all circumstances update() is better, as it permits Qt to optimize for speed and against flicker.

Warning: If you call repaint() in a function which may itself be called from paintEvent(), you may see infinite recursion. The update() function never generates recursion.

See also update() [p. 456], paintEvent() [p. 441], updatesEnabled [p. 471] and erase() [p. 428].

Example: qwerty/qwerty.cpp.

void QWidget::repaint () [slot]

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

This version erases and repaints the entire widget.

void QWidget::repaint (bool erase) [slot]

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

This version repaints the entire widget.

void QWidget::repaint (const QRect & r, bool erase = TRUE) [slot]

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

Repaints the widget directly by calling paintEvent() directly, unless updates are disabled or the widget is hidden.

Erases the widget region *r* if *erase* is TRUE.

void QWidget::repaint (const QRegion & reg, bool erase = TRUE) [slot]

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

Repaints the widget directly by calling paintEvent() directly, unless updates are disabled or the widget is hidden.

Erases the widget region *reg* if *erase* is TRUE.

Use repaint if your widget needs to be repainted immediately, for example when doing some animation. In all other cases, update() is to be preferred. Calling update() many times in a row will generate a single paint event.

Warning: If you call repaint() in a function which may itself be called from paintEvent(), you may see infinite recursion. The update() function never generates recursion.

See also update() [p. 456], paintEvent() [p. 441], updatesEnabled [p. 471] and erase() [p. 428].

void QWidget::reparent (QWidget * parent, WFlags f, const QPoint & p, bool showIt = FALSE) [virtual]

Reparents the widget. The widget gets a new *parent*, new widget flags (*f*, but as usual, use 0) at a new position in its new parent (*p*).

If *showIt* is TRUE, show() is called once the widget has been reparented.

If the new parent widget is in a different top-level widget, the reparented widget and its children are appended to the end of the tab chain of the new parent widget, in the same internal order as before. If one of the moved widgets had keyboard focus, reparent() calls clearFocus() for that widget.

If the new parent widget is in the same top-level widget as the old parent, reparent doesn't change the tab order or keyboard focus.

Warning: It is extremely unlikely that you will ever need this function. If you have a widget that changes its content dynamically, it is far easier to use QWidgetStack or QWizard.

See also getWFlags() [p. 431].

void QWidget::reparent (QWidget * parent, const QPoint & p, bool showIt = FALSE)

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

A convenience version of reparent that does not take widget flags as argument.

Calls reparent(*parent*, getWFlags() & ~WType_Mask, *p*, *showIt*).

void QWidget::resetInputContext () [protected]

This function is called when the user finishes input composition, e.g. changes focus to another widget, moves the cursor, etc.

void QWidget::resize (const QSize &) [slot]

Sets the size of the widget excluding any window frame. See the "size" [p. 469] property for details.

void QWidget::resize (int w, int h) [virtual slot]

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

This corresponds to resize(QSize(*w*, *h*)).

void QWidget::resizeEvent (QResizeEvent *) [virtual protected]

This event handler can be reimplemented in a subclass to receive widget resize events. When resizeEvent() is called, the widget already has its new geometry. The old size is accessible through QResizeEvent::oldSize(), though.

The widget will be erased and receive a paint event immediately after processing the resize event. No drawing need be (or should be) done inside this handler.

Widgets that have been created with the WResizeNoErase flag will not be erased. Nevertheless, they will receive a paint event for their entire area afterwards. Again, no drawing needs to be done inside this handler.

The default implementation calls updateMask() if the widget has automatic masking enabled.

See also moveEvent() [p. 440], event() [p. 428], size [p. 469], QResizeEvent [Events, Actions, Layouts and Styles with Qt] and paintEvent() [p. 441].

Examples: drawdemo/drawdemo.cpp, mainlyQt/editor.cpp, mainlyXt/editor.cpp, menu/menu.cpp, qmag/qmag.cpp, scribble/scribble.cpp and tooltip/tooltip.cpp.

Reimplemented in QFrame and QGLWidget.

void QWidget::scroll (int dx, int dy)

Scrolls the widget including its children *dx* pixels to the right and *dy* downwards. Both *dx* and *dy* may be negative.

After scrolling, scroll() sends a paint event for the the part that is read but not written. For example, when scrolling 10 pixels rightwards, the leftmost ten pixels of the widget need repainting. The paint event may be delivered immediately or later, depending on some heuristics.

See also QScrollView [p. 259], erase() [p. 428] and bitBlt() [Graphics with Qt].

void QWidget::scroll (int dx, int dy, const QRect & r)

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

This version only scrolls *r* and does not move the children of the widget.

If *r* is empty or invalid, the result is undefined.

See also QScrollView [p. 259], erase() [p. 428] and bitBlt() [Graphics with Qt].

void QWidget::setAcceptDrops (bool on) [virtual]

Sets whether drop events are enabled for this widget to *on*. See the "acceptDrops" [p. 458] property for details.

void QWidget::setActiveWindow () [virtual]

Sets the top-level widget containing this widget to be the active window.

An active window is a visible top-level window that has the keyboard input focus.

This function performs the same operation as clicking the mouse on the title bar of a top-level window. On X11, the result depends on the Window Manager. If you want to ensure that the window is stacked on top as well, call raise() in addition. Note that the window has be to visible, otherwise setActiveWindow() has no effect.

On Windows, if you are calling this when the application is not currently the active one then it will not make it the active window. It will flash the task bar entry blue to indicate that the window has done something. This is due to Microsoft not allowing an application to interrupt what the user is currently doing in another application.

See also isActiveWindow [p. 464], topLevelWidget() [p. 456] and show() [p. 453].

Reimplemented in QXtWidget.

void QWidget::setAutoMask (bool) [virtual]

Sets whether the auto mask feature is enabled for the widget. See the "autoMask" [p. 459] property for details.

void QWidget::setBackgroundColor (const QColor & c) [virtual]

This function is obsolete. It is provided to keep old source working. We strongly advise against using it in new code. Use setPaletteBackgroundColor() or setEraseColor() instead.

Examples: customlayout/main.cpp, desktop/desktop.cpp, hello/main.cpp and splitter/splitter.cpp.

void QWidget::setBackgroundMode (BackgroundMode) [virtual]

Sets the color role used for painting the background of the widget. See the "backgroundMode" [p. 459] property for details.

void QWidget::setBackgroundMode (BackgroundMode m, BackgroundMode visual)

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

Sets the widget's own background mode to *m* and the visual background mode to *visual*. The visual background mode is used with the designable properties backgroundColor, foregroundColor and backgroundPixmap.

For complex controls, the logical background mode sometimes differs from a widget's own background mode. A spinbox for example has PaletteBackground as background mode (typically dark grey), while it's embedded lineedit control uses PaletteBase (typically white). Since the lineedit covers most of the visual area of a spinbox, it defines PaletteBase to be its *visual* background mode. Changing the backgroundColor property thus changes the lineedit control's background, which is exactly what the user expects in *Qt Designer*.

void QWidget::setBackgroundOrigin (BackgroundOrigin) [virtual]

Sets the origin of the widget's background. See the "backgroundOrigin" [p. 460] property for details.

void QWidget::setBackgroundPixmap (const QPixmap & pm) [virtual]

This function is obsolete. It is provided to keep old source working. We strongly advise against using it in new code. Use setPaletteBackgroundPixmap() or setErasePixmap() instead.

Example: desktop/desktop.cpp.

void QWidget::setBaseSize (const QSize &)

Sets the base size of the widget. See the "baseSize" [p. 460] property for details.

void QWidget::setBaseSize (int basew, int baseh)

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

This corresponds to setBaseSize(QSize(*basew*, *baseh*)). Sets the widgets base size to width *basew* and height *baseh*.

void QWidget::setCaption (const QString &) [virtual slot]

Sets the window caption (title). See the "caption" [p. 460] property for details.

void QWidget::setCursor (const QCursor &) [virtual]

Sets the cursor shape for this widget. See the "cursor" [p. 461] property for details.

void QWidget::setDisabled (bool disable) [slot]

Disables widget input events if *disable* is TRUE; otherwise enables input events.

See the enabled [p. 461] documentation for more information.

See also isEnabledTo() [p. 434], QKeyEvent [Events, Actions, Layouts and Styles with Qt], QMouseEvent [Events, Actions, Layouts and Styles with Qt] and enabledChange() [p. 427].

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

Sets whether the widget is enabled. See the "enabled" [p. 461] property for details.

void QWidget::setEraseColor (const QColor & color) [virtual]

Sets the erase color of the widget to *color*.

The erase color is the color the widget is to be cleared to before paintEvent() is called. If there is an erase pixmap (set using setErasePixmap()), then this property has an indeterminate value.

See also erasePixmap() [p. 428], backgroundColor() [p. 423], backgroundMode [p. 459] and palette [p. 468].

void QWidget::setErasePixmap (const QPixmap & pixmap) [virtual]

Sets the widget's erase pixmap to *pixmap*.

This pixmap is used to clear the widget before paintEvent() is called.

void QWidget::setFixedHeight (int h)

Sets both the minimum and maximum heights of the widget to *h* without changing the widths. Provided for convenience.

See also sizeHint [p. 470], minimumSize [p. 466], maximumSize [p. 465] and setFixedSize() [p. 448].

Examples: fonts/simple-qfont-demo/viewer.cpp, layout/layout.cpp, qdir/qdir.cpp and showimg/showimg.cpp.

void QWidget::setFixedSize (const QSize & s)

Sets both the minimum and maximum sizes of the widget to *s*, thereby preventing it from ever growing or shrinking.

See also maximumSize [p. 465] and minimumSize [p. 466].

void QWidget::setFixedSize (int w, int h)

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

Sets the width of the widget to *w* and the height to *h*.

void QWidget::setFixedWidth (int w)

Sets both the minimum and maximum width of the widget to *w* without changing the heights. Provided for convenience.

See also sizeHint [p. 470], minimumSize [p. 466], maximumSize [p. 465] and setFixedSize() [p. 448].

Examples: network/ftpclient/ftpmainwindow.cpp, progressbar/progressbar.cpp and qdir/qdir.cpp.

void QWidget::setFocus () [virtual slot]

Gives the keyboard input focus to this widget (or its focus proxy).

First, a focus out event is sent to the focus widget (if any) to tell it that it is about to lose the focus. Then a focus in event is sent to this widget to tell it that it just received the focus. (If the focus in and focus out widgets are the same nothing happens.)

setFocus() gives focus to a widget regardless of its focus policy, but does not clear any keyboard grab (see grabKeyboard()).

Warning: If you call setFocus() in a function which may itself be called from focusOutEvent() or focusInEvent(), you may experience infinite recursion.

See also focus [p. 462], clearFocus() [p. 424], focusInEvent() [p. 429], focusOutEvent() [p. 430], focusPolicy [p. 462], QApplication::focusWidget() [Additional Functionality with Qt], grabKeyboard() [p. 431] and grabMouse() [p. 431].

Examples: addressbook/centralwidget.cpp, lineedits/lineedits.cpp, popup/popup.cpp, rot13/rot13.cpp, t8/main.cpp, wizard/wizard.cpp and xform/xform.cpp.

void QWidget::setFocusPolicy (FocusPolicy) [virtual]

Sets the way the widget accepts keyboard focus. See the "focusPolicy" [p. 462] property for details.

void QWidget::setFocusProxy (QWidget * w) [virtual]

Sets this widget's focus proxy to widget *w*. If *w* is 0, this function resets this widget to have no focus proxy.

Some widgets, such as QComboBox, can "have focus", but create a child widget to actually handle the focus. QComboBox, for example, creates a QLineEdit which handles the focus.

setFocusProxy() sets the widget which will actually get focus when "this widget" gets it. If there is a focus proxy, focusPolicy(), setFocusPolicy(), setFocus() and hasFocus() all operate on the focus proxy.

See also focusProxy() [p. 430].

void QWidget::setFont (const QFont &) [virtual]

Sets the font currently set for the widget. See the "font" [p. 462] property for details.

Reimplemented in QComboBox, QLabel and QTabDialog.

void QWidget::setFont (const QFont & f, bool)

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

Use setFont(const QFont& font) instead.

void QWidget::setGeometry (const QRect &) [virtual slot]

Sets the geometry of the widget relative to its parent and excluding the window frame. See the "geometry" [p. 463] property for details.

void QWidget::setGeometry (int x, int y, int w, int h) [virtual slot]

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

This corresponds to setGeometry(QRect(*x*, *y*, *w*, *h*)).

void QWidget::setIcon (const QPixmap &) [virtual slot]

Sets the widget icon pixmap. See the "icon" [p. 464] property for details.

void QWidget::setIconText (const QString &) [virtual slot]

Sets the widget icon text. See the "iconText" [p. 464] property for details.

void QWidget::setKeyCompression (bool compress) [virtual protected]

Enables key event compression, if *compress* is TRUE, and disables it if *compress* is FALSE.

By default key compression is off, so widgets receive one key press event for each key press (or more, since autorepeat is usually on). If you turn it on and your program doesn't keep up with key input, Qt tries to compress key events so that more than one character can be processed in each event.

For example, a word processor widget might receive 2, 3 or more characters in each QKeyEvent::text(), if the layout recalculation takes too long for the CPU.

If a widget supports multiple character unicode input, it is always safe to turn the compression on.

See also QKeyEvent::text() [Events, Actions, Layouts and Styles with Qt].

void QWidget::setMask (const QBitmap & bitmap) [virtual]

Causes only the pixels of the widget for which *bitmap* has a corresponding 1 bit to be visible. If the region includes pixels outside the rect() of the widget, window system controls in that area may or may not be visible, depending on the platform.

Note that this effect can be slow if the region is particularly complex.

See also clearMask() [p. 424].

void QWidget::setMask (const QRegion & region) [virtual]

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

Causes only the parts of the widget which overlap *region* to be visible. If the region includes pixels outside the rect() of the widget, window system controls in that area may or may not be visible, depending on the platform.

Note that this effect can be slow if the region is particularly complex.

See also clearMask() [p. 424].

void QWidget::setMaximumHeight (int maxh)

Sets the widget's maximum height to *maxh*. See the "maximumHeight" [p. 465] property for details.

void QWidget::setMaximumSize (const QSize &)

Sets the widget's maximum size. See the "maximumSize" [p. 465] property for details.

void QWidget::setMaximumSize (int maxw, int maxh) [virtual]

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

This function corresponds to setMaximumSize(QSize(*maxw*, *maxh*)). Sets the maximum width to *maxw* and the maximum height to *maxh*.

void QWidget::setMaximumWidth (int maxw)

Sets the widget's maximum width to *maxw*. See the "maximumWidth" [p. 466] property for details.

void QWidget::setMicroFocusHint (int x, int y, int width, int height, bool text = TRUE, QFont * f = 0) [virtual protected]

When a widget gets focus, it should call setMicroFocusHint for some appropriate position and size - *x*, *y* and *width* by *height*. This has no *visual* effect, it just provides hints to any system-specific input handling tools.

The *text* argument should be TRUE if this is a position for text input.

In the Windows version of Qt, this method sets the system caret, which is used for user Accessibility focus handling. If *text* is TRUE, it also sets the IME composition window in Far East Asian language input systems.

In the X11 version of Qt, if *text* is TRUE, this method sets the XIM "spot" point for complex language input handling.

The font parameter, *f*, is currently unused.

See also microFocusHint [p. 466].

void QWidget::setMinimumHeight (int minh)

Sets the widget's minimum height to *minh*. See the "minimumHeight" [p. 466] property for details.

void QWidget::setMinimumSize (const QSize &)

Sets the widget's minimum size. See the "minimumSize" [p. 466] property for details.

void QWidget::setMinimumSize (int minw, int minh) [virtual]

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

This function corresponds to setMinimumSize(QSize(minw, minh)). Sets the minimum width to *minw* and the minimum height to *minh*.

void QWidget::setMinimumWidth (int minw)

Sets the widget's minimum width to *minw*. See the "minimumWidth" [p. 467] property for details.

void QWidget::setMouseTracking (bool enable) [virtual slot]

Sets whether mouse tracking is enabled for this widget to *enable*. See the "mouseTracking" [p. 467] property for details.

void QWidget::setPalette (const QPalette &) [virtual]

Sets the widget's palette. See the "palette" [p. 468] property for details.

Reimplemented in QComboBox, QScrollBar and QSlider.

void QWidget::setPalette (const QPalette & p, bool)

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

Use setPalette(const QPalette& p) instead.

void QWidget::setPaletteBackgroundColor (const QColor &) [virtual]

Sets the background color of the widget. See the "paletteBackgroundColor" [p. 468] property for details.

void QWidget::setPaletteBackgroundPixmap (const QPixmap &) [virtual]

Sets the background pixmap of the widget. See the "paletteBackgroundPixmap" [p. 468] property for details.

void QWidget::setPaletteForegroundColor (const QColor &)

Sets the foreground color of the widget. See the "paletteForegroundColor" [p. 468] property for details.

void QWidget::setSizeIncrement (const QSize &)

Sets the size increment of the widget. See the "sizeIncrement" [p. 470] property for details.

void QWidget::setSizeIncrement (int w, int h) [virtual]

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

Sets the x (width) size increment to *w* and the y (height) size increment to *h*.

void QWidget::setSizePolicy (QSizePolicy) [virtual]

Sets the default layout behavior of the widget. See the "sizePolicy" [p. 470] property for details.

void QWidget::setStyle (QStyle * style)

Sets the widget's GUI style to *style*. Ownership of the style object is not transferred.

If no style is set, the widget uses the application's style QApplication::style() instead.

Setting a widget's style has no effect on existing or future child widgets.

Warning: This function is particularly useful for demonstration purposes, where you want to show Qt's styling capabilities. Real applications should avoid it and use one consistent GUI style instead.

See also style() [p. 455], QStyle [Events, Actions, Layouts and Styles with Qt], QApplication::style() [Additional Functionality with Qt] and QApplication::setStyle() [Additional Functionality with Qt].

Examples: grapher/grapher.cpp and progressbar/progressbar.cpp.

QStyle * QWidget::setStyle (const QString & style)

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

Sets the widget's GUI style to *style* using the QStyleFactory.

void QWidget::setTabOrder (QWidget * first, QWidget * second) [static]

Moves the *second* widget around the ring of focus widgets so that keyboard focus moves from *first* widget to *second* widget when Tab is pressed.

Note that since the tab order of the *second* widget is changed, you should order a chain like this:

```
setTabOrder( a, b ); // a to b
setTabOrder( b, c ); // a to b to c
setTabOrder( c, d ); // a to b to c to d
```

not like this:

```
setTabOrder( c, d ); // c to d   WRONG
setTabOrder( a, b ); // a to b AND c to d
setTabOrder( b, c ); // a to b to c, but not c to d
```

If either *first* or *second* has a focus proxy, setTabOrder() substitutes its/their proxies.

See also focusPolicy [p. 462] and setFocusProxy() [p. 448].

Example: customlayout/main.cpp.

void QWidget::setUpdatesEnabled (bool enable) [virtual slot]

Sets whether updates are enabled to *enable*. See the "updatesEnabled" [p. 471] property for details.

void QWidget::setWFlags (WFlags f) [virtual protected]

Sets the widget flags *f*.

Widget flags are a combination of Qt::WidgetFlags.

See also testWFlags() [p. 455], getWFlags() [p. 431] and clearWFlags() [p. 424].

void QWidget::show () [virtual slot]

Shows the widget and its child widgets.

If its size or position has changed, Qt guarantees that a widget gets move and resize events just before it is shown.

You almost never have to reimplement this function. If you need to change some settings before a widget is shown, use showEvent() instead. If you need to do some delayed initialization use polish().

See also showEvent() [p. 453], hide() [p. 433], showMinimized() [p. 454], showMaximized() [p. 454], showNormal() [p. 454], visible [p. 471] and polish() [p. 442].

Examples: fonts/simple-qfont-demo/simple-qfont-demo.cpp, life/main.cpp, network/ftpclient/ftpmainwindow.cpp, popup/popup.cpp, t1/main.cpp, t3/main.cpp and t4/main.cpp.

Reimplemented in QDialog and QMenuBar.

void QWidget::showEvent (QShowEvent *) [virtual protected]

This event handler can be reimplemented in a subclass to receive widget show events.

Non-spontaneous show events are sent to widgets immediately before they are shown. Spontaneous show events of top level widgets are delivered afterwards.

See also event() [p. 428] and QShowEvent [Events, Actions, Layouts and Styles with Qt].

Example: qdir/qdir.cpp.

void QWidget::showFullScreen () [slot]

Shows the widget in full-screen mode.

Calling this function only effects top-level widgets.

To return from full-screen mode, call showNormal().

Full-screen mode works fine under Windows, but has certain problems under X. These problems are due to limitations of the ICCCM protocol that specifies the communication between X11 clients and the window manager. ICCCM simply does not understand the concept of non-decorated full-screen windows. Therefore, the best we can do is to request a borderless window and place and resize it to fill the entire screen. Depending on the window

manager, this may or may not work. The borderless window is requested using MOTIF hints, which are at least partially supported by virtually all modern window managers.

An alternative would be to bypass the window manager entirely and create a window with the WX11BypassWM flag. This has other severe problems though, like totally broken keyboard focus and very strange effects on desktop changes or when the user raises other windows.

Future X11 window managers that follow modern post-ICCCM specifications may support full-screen mode properly.

See also showNormal() [p. 454], showMaximized() [p. 454], show() [p. 453], hide() [p. 433] and visible [p. 471].

void QWidget::showMaximized () [virtual slot]

Shows the widget maximized.

Calling this function has no effect for other than top-level widgets.

On X11, this function may not work properly with certain window managers. See the Window Geometry documentation for details on why.

See also showNormal() [p. 454], showMinimized() [p. 454], show() [p. 453], hide() [p. 433] and visible [p. 471].

Examples: helpviewer/main.cpp, mdi/application.cpp, qwerty/main.cpp, qwerty/qwerty.cpp and scribble/main.cpp.

void QWidget::showMinimized () [virtual slot]

Shows the widget minimized, as an icon.

Calling this function has no effect for other than top-level widgets.

See also showNormal() [p. 454], showMaximized() [p. 454], show() [p. 453], hide() [p. 433], visible [p. 471] and minimized [p. 466].

void QWidget::showNormal () [virtual slot]

Restores the widget after it has been maximized or minimized.

Calling this function has no effect for other than top-level widgets.

See also showMinimized() [p. 454], showMaximized() [p. 454], show() [p. 453], hide() [p. 433] and visible [p. 471].

Example: mdi/application.cpp.

QSize QWidget::size () const

Returns the size of the widget excluding any window frame. See the "size" [p. 469] property for details.

QSize QWidget::sizeHint () const [virtual]

Returns the recommended size for the widget. See the "sizeHint" [p. 470] property for details.

Reimplemented in QSizeGrip.

QSize QWidget::sizeIncrement () const

Returns the size increment of the widget. See the "sizeIncrement" [p. 470] property for details.

QSizePolicy QWidget::sizePolicy () const [virtual]

Returns the default layout behavior of the widget. See the "sizePolicy" [p. 470] property for details.

void QWidget::stackUnder (QWidget * w) [slot]

Places the widget under *w* in the parent widget's stack.

To make this work, the widget itself and *w* have to be siblings.

See also raise() [p. 442] and lower() [p. 437].

QStyle & QWidget::style () const

Returns the GUI style for this widget

See also QWidget::setStyle() [p. 452], QApplication::setStyle() [Additional Functionality with Qt] and QApplication::style() [Additional Functionality with Qt].

void QWidget::styleChange (QStyle & oldStyle) [virtual protected]

This virtual function is called when the style of the widgets. changes. *oldStyle* is the previous GUI style; you can get the new style from style().

Reimplement this function if your widget needs to know when its GUI style changes. You will almost certainly need to update the widget using update().

The default implementation updates the widget including its geometry.

See also QApplication::setStyle() [Additional Functionality with Qt], style() [p. 455], update() [p. 456] and updateGeometry() [p. 457].

void QWidget::tabletEvent (QTabletEvent * e) [virtual protected]

This event handler, for event *e*, can be reimplemented in a subclass to receive tablet events for the widget.

If you reimplement this handler, it is very important that you ignore() the event if you do not handle it, so that the widget's parent can interpret it.

The default implementation ignores the event.

See also QTabletEvent::ignore(), QTabletEvent::accept(), event() [p. 428] and QTabletEvent.

WFlags QWidget::testWFlags (WFlags f) const

Returns the bitwise AND of the widget flags and *f*.

Widget flags are a combination of Qt::WidgetFlags.

See also getWFlags() [p. 431], setWFlags() [p. 453] and clearWFlags() [p. 424].

QWidget * QWidget::topLevelWidget () const

Returns the top-level widget for this widget, i.e. the next ancestor widget that has (or may have) a window-system frame.

If the widget is a top-level, the widget itself is returned.

Typical usage is changing the window caption:

```
aWidget->topLevelWidget()->setCaption( "New Caption" );
```

See also isTopLevel [p. 465].

void QWidget::unsetCursor () [virtual]

Resets the cursor shape for this widget. See the "cursor" [p. 461] property for details.

void QWidget::unsetFont ()

Resets the font currently set for the widget. See the "font" [p. 462] property for details.

void QWidget::unsetPalette ()

Resets the widget's palette. See the "palette" [p. 468] property for details.

void QWidget::update () [slot]

Updates the widget unless updates are disabled or the widget is hidden.

This function does not cause an immediate repaint - rather, it schedules a paint event for processing when Qt returns to the main event loop. This permits Qt to optimize for more speed and less flicker and a call to repaint() does.

Calling update() several times normally results in just one paintEvent() call.

Qt normally erases the widget's area before the paintEvent() call. Only if the WRepaintNoErase widget flag is set, the widget itself is responsible for painting all its pixels.

See also repaint() [p. 443], paintEvent() [p. 441], updatesEnabled [p. 471], erase() [p. 428] and setWFlags() [p. 453].

Examples: desktop/desktop.cpp and scrollview/scrollview.cpp.

void QWidget::update (int x, int y, int w, int h) [slot]

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

Updates a rectangle (x, y, w, h) inside the widget unless updates are disabled or the widget is hidden.

This function does not cause an immediate repaint - rather, it schedules a paint event for processing when Qt returns to the main event loop. This permits Qt to optimize for more speed and less flicker and a call to repaint() does.

Calling update() several times normally results in just one paintEvent() call.

If w is negative, it is replaced with `width() - x`. If h is negative, it is replaced width `height() - y`.

Qt normally erases the specified area before the paintEvent() call. Only if the WRepaintNoErase widget flag is set, the widget itself is responsible for painting all its pixels.

See also repaint() [p. 443], paintEvent() [p. 441], updatesEnabled [p. 471] and erase() [p. 428].

void QWidget::update (const QRect & r) [slot]

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

Updates a rectangle *r* inside the widget unless updates are disabled or the widget is hidden.

This function does not cause an immediate repaint - rather, it schedules a paint event for processing when Qt returns to the main event loop. This permits Qt to optimize for more speed and less flicker and a call to repaint() does.

Calling update() several times normally results in just one paintEvent() call.

void QWidget::updateGeometry ()

Notifies the layout system that this widget has changed and may need to change geometry.

Call this function if the sizeHint() or sizePolicy() have changed.

For explicitly hidden widgets, updateGeometry() is a no-op. The layout system will be notified as soon as the widget is shown.

void QWidget::updateMask () [virtual protected]

This function can be reimplemented in a subclass to support transparent widgets. It is supposed to be called whenever a widget changes state in a way that the shape mask has to be recalculated.

See also autoMask [p. 459], setMask() [p. 450] and clearMask() [p. 424].

QRect QWidget::visibleRect () const

Returns the currently visible rectangle of the widget. See the "visibleRect" [p. 471] property for details.

void QWidget::wheelEvent (QWheelEvent * e) [virtual protected]

This event handler, for event *e*, can be reimplemented in a subclass to receive wheel events for the widget.

If you reimplement this handler, it is very important that you ignore() the event if you do not handle it, so that the widget's parent can interpret it.

The default implementation ignores the event.

See also QWheelEvent::ignore() [Events, Actions, Layouts and Styles with Qt], QWheelEvent::accept() [Events, Actions, Layouts and Styles with Qt], event() [p. 428] and QWheelEvent [Events, Actions, Layouts and Styles with Qt].

int QWidget::width () const

Returns the width of the widget excluding any window frame. See the "width" [p. 472] property for details.

bool QWidget::winEvent (MSG *) [virtual protected]

This special event handler can be reimplemented in a subclass to receive native Windows events.

In your reimplementation of this function, if you want to stop the event being handled by Qt, return TRUE. If you return FALSE, this native event is passed back to Qt, which translates the event into a Qt event and sends it to the widget.

Warning: This function is not portable.

See also QApplication::winEventFilter() [Additional Functionality with Qt].

WId QWidget::winId () const

Returns the window system identifier of the widget.

Portable in principle, but if you use it you are probably about to do something non-portable. Be careful.

See also find() [p. 429].

Example: mainlyXt/editor.cpp.

void QWidget::windowActivationChange (bool oldActive) [virtual protected]

This virtual function is called for a widget when its window is activated or deactivated by the window system. *oldActive* is the previous state; you can get the new setting from isActiveWindow().

Reimplement this function if your widget needs to know when its window becomes activated or deactivated.

The default implementation updates the visible part of the widget if the inactive and the active colorgroup are different for colors other than the highlight and link colors.

See also setActiveWindow() [p. 445], isActiveWindow [p. 464], update() [p. 456] and palette [p. 468].

int QWidget::x () const

Returns the x coordinate of the widget relative to its parent including any window frame. See the "x" [p. 472] property for details.

int QWidget::y () const

Returns the y coordinate of the widget relative to its parent and including any window frame. See the "y" [p. 472] property for details.

Property Documentation

bool acceptDrops

This property holds whether drop events are enabled for this widget.

Setting this property to TRUE announces to the system that this widget *may* be able to accept drop events.

If the widgets is the desktop (QWidget::isDesktop()), this may fail if another application is using the desktop; you can call acceptDrops() to test if this occurs.

Set this property's value with setAcceptDrops() and get this property's value with acceptDrops().

bool autoMask

This property holds whether the auto mask feature is enabled for the widget.

Transparent widgets use a mask to define their visible region. QWidget has some built-in support to make the task of recalculating the mask easier. When setting auto mask to TRUE, updateMask() will be called whenever the widget is resized or changes its focus state. Note that you must reimplement updateMask() (which should include a call to setMask()) or nothing will happen.

Note: when you re-implement resizeEvent(), focusInEvent() or focusOutEvent() in your custom widgets and still want to ensure that the auto mask calculation works, you should add:

```
if ( autoMask() )
    updateMask();
```

at the end of your event handlers. This is true for all member functions that change the appearance of the widget in a way that requires a recalculation of the mask.

While being a technically appealing concept, masks have a big drawback: when using complex masks that cannot be expressed easily with relatively simple regions, they can be very slow on some window systems. The classic example is a transparent label. The complex shape of its contents makes it necessary to represent its mask by a bitmap, which consumes both memory and time. If all you want is to blend the background of several neighboring widgets together seamlessly, you will probably want to use setBackgroundOrigin() rather than a mask.

See also autoMask [p. 459], updateMask() [p. 457], setMask() [p. 450], clearMask() [p. 424] and backgroundOrigin [p. 460].

Set this property's value with setAutoMask() and get this property's value with autoMask().

QBrush backgroundBrush

This property holds the widget's background brush.

The background brush depends on a widget's palette and its background mode.

See also backgroundColor() [p. 423], backgroundPixmap() [p. 423], eraseColor() [p. 428], palette [p. 468] and QApplication::setPalette() [Additional Functionality with Qt].

Get this property's value with backgroundBrush().

BackgroundMode backgroundMode

This property holds the color role used for painting the background of the widget.

setPaletteBackgroundColor() reads this property to determine which entry of the palette to set.

For most widgets the default suffices (PaletteBackground, typically gray), but some need to use PaletteBase (the background color for text output, typically white) or another role.

QListBox, which is "sunken" and uses the base color to contrast with its environment, does this in its constructor:

```
setBackgroundMode( PaletteBase );
```

You will never need to set the background mode of a built-in widget in Qt, but you might consider setting it in your custom widgets, so that setPaletteBackgroundColor() works as expected.

Note that two of the BackgroundMode values make no sense for setBackgroundMode(), namely FixedPixmap and FixedColor. You have to call setBackgroundPixmap() and setPaletteBackgroundColor() instead.

Set this property's value with setBackgroundMode() and get this property's value with backgroundMode().

BackgroundOrigin backgroundOrigin

This property holds the origin of the widget's background.

The origin is either WidgetOrigin (the default), ParentOrigin or WindowOrigin.

This makes a difference only if the widget has a background pixmap, in which case positioning matters. Using WindowOrigin for several neighboring widgets makes the background blend together seamlessly.

See also backgroundPixmap() [p. 423] and backgroundMode [p. 459].

Set this property's value with setBackgroundOrigin() and get this property's value with backgroundOrigin().

QSize baseSize

This property holds the base size of the widget.

The base size is used to calculate a proper widget size in case the widget defines sizeIncrement().

See also sizeIncrement [p. 470].

Set this property's value with setBaseSize() and get this property's value with baseSize().

QString caption

This property holds the window caption (title).

This property only makes sense for top-level widgets. If no caption has been set, the caption is QString::null.

See also icon [p. 464] and iconText [p. 464].

Set this property's value with setCaption() and get this property's value with caption().

QRect childrenRect

This property holds the bounding rectangle of the widget's children.

Hidden children are excluded.

See also childrenRegion [p. 460] and geometry [p. 463].

Get this property's value with childrenRect().

QRegion childrenRegion

This property holds the combined region occupied by the widget's children.

Hidden children are excluded.

See also childrenRect [p. 460] and geometry [p. 463].

Get this property's value with childrenRegion().

QColorGroup colorGroup

This property holds the current color group of the widget palette.

The color group is determined by the state of the widget. A disabled widget has the QPalette::disabled() color group, a widget with keyboard focus has the QPalette::active() color group, and an inactive widget has the QPalette::inactive() color group.

See also palette [p. 468].

Get this property's value with colorGroup().

QCursor cursor

This property holds the cursor shape for this widget.

The mouse cursor will assume this shape when it's over this widget. See the list of predefined cursor objects [Additional Functionality with Qt] for a range of useful shapes.

An editor widget would for example use an I-beam cursor:

```
setCursor( IbeamCursor );
```

If no cursor has been set, or after a call to unsetCursor(), the parent's cursor is used. The function unsetCursor() has no effect on top-level widgets.

See also QApplication::setOverrideCursor() [Additional Functionality with Qt].

Set this property's value with setCursor(), get this property's value with cursor() and reset this property's value with unsetCursor().

bool customWhatsThis

This property holds whether the widget wants to handle What's This help manually.

The default implementation of customWhatsThis() returns FALSE, which means the widget will not receive any events in Whats This mode.

The widget may leave What's This mode by calling QWhatsThis::leaveWhatsThisMode(), with or without actually displaying any help text.

You can also reimplement customWhatsThis() if your widget is a "passive interactor" supposed to work under all circumstances. Simply don't call QWhatsThis::leaveWhatsThisMode() in that case.

See also QWhatsThis::inWhatsThisMode() [p. 410] and QWhatsThis::leaveWhatsThisMode() [p. 410].

Get this property's value with customWhatsThis().

bool enabled

This property holds whether the widget is enabled.

An enabled widget receives keyboard and mouse events; a disabled widget does not. In fact, an enabled widget only receives keyboard events when it is in focus.

Some widgets display themselves differently when they are disabled. For example a button might draw its label grayed out. If your widget needs to know when it becomes enabled or disabled, you can reimplement the enabledChange() function.

Disabling a widget implicitly disables all its children. Enabling respectively enables all child widgets unless they have been explicitly disabled.

See also enabled [p. 461], isEnabledTo() [p. 434], QKeyEvent [Events, Actions, Layouts and Styles with Qt], QMouseEvent [Events, Actions, Layouts and Styles with Qt] and enabledChange() [p. 427].

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

bool focus

This property holds whether this widget (or its focus proxy) has the keyboard input focus.

Effectively equivalent to `qApp->focusWidget() == this`.

See also setFocus() [p. 448], clearFocus() [p. 424], focusPolicy [p. 462] and QApplication::focusWidget() [Additional Functionality with Qt].

Get this property's value with hasFocus().

bool focusEnabled

This property holds whether the widget accepts keyboard focus.

Keyboard focus is initially disabled (i.e. focusPolicy() == QWidget::NoFocus).

You must enable keyboard focus for a widget if it processes keyboard events. This is normally done from the widget's constructor. For instance, the QLineEdit constructor calls setFocusPolicy(QWidget::StrongFocus).

See also focusPolicy [p. 462], focusInEvent() [p. 429], focusOutEvent() [p. 430], keyPressEvent() [p. 436], keyReleaseEvent() [p. 436] and enabled [p. 461].

Get this property's value with isFocusEnabled().

FocusPolicy focusPolicy

This property holds the way the widget accepts keyboard focus.

The policy is QWidget::TabFocus if the widget accepts keyboard focus by tabbing, QWidget::ClickFocus if the widget accepts focus by clicking, QWidget::StrongFocus if it accepts both and QWidget::NoFocus if it does not accept focus at all (the default for QWidget).

You must enable keyboard focus for a widget if it processes keyboard events. This is normally done from the widget's constructor. For instance, the QLineEdit constructor calls setFocusPolicy(QWidget::StrongFocus).

See also focusEnabled [p. 462], focusInEvent() [p. 429], focusOutEvent() [p. 430], keyPressEvent() [p. 436], keyReleaseEvent() [p. 436] and enabled [p. 461].

Set this property's value with setFocusPolicy() and get this property's value with focusPolicy().

QFont font

This property holds the font currently set for the widget.

The fontInfo() function reports the actual font that is being used by the widget.

As long as no special font has been set, or after unsetFont() is called, this is either a special font for the widget class, the parent's font or (if this widget is a top level widget) the default application font.

This code fragment sets a 12 point helvetica bold font:

```
QFont f( "Helvetica", 12, QFont::Bold );
setFont( f );
```

Apart from setting the font, setFont() informs all children about the change.

See also fontChange() [p. 430], fontInfo() [p. 430], fontMetrics() [p. 431] and ownFont [p. 467].

Set this property's value with setFont(), get this property's value with font() and reset this property's value with unsetFont().

QRect frameGeometry

This property holds geometry of the widget relative to its parent including any window frame.

See the Window Geometry documentation [Graphics with Qt] for an overview of geometry issues with top-level widgets.

See also geometry [p. 463], x [p. 472], y [p. 472] and pos [p. 469].

Get this property's value with frameGeometry().

QSize frameSize

This property holds the size of the widget including any window frame.

Get this property's value with frameSize().

QRect geometry

This property holds the geometry of the widget relative to its parent and excluding the window frame.

When changing the geometry, the widget, if visible, receives a move event (moveEvent()) and/or a resize event (resizeEvent()) immediately. If the widget is not currently visible, it is guaranteed to receive appropriate events before it is shown.

The size component is adjusted if it lies outside the range defined by minimumSize() and maximumSize().

setGeometry() is virtual, and all other overloaded setGeometry() implementations in Qt call it.

Warning: If you call setGeometry() from resizeEvent() or moveEvent(), you may experience infinite recursion.

See the Window Geometry documentation [Graphics with Qt] for an overview of geometry issues with top-level widgets.

See also frameGeometry [p. 463], rect [p. 469], pos [p. 469], size [p. 469], moveEvent() [p. 440], resizeEvent() [p. 445], minimumSize [p. 466] and maximumSize [p. 465].

Set this property's value with setGeometry() and get this property's value with geometry().

int height

This property holds the height of the widget excluding any window frame.

See the Window Geometry documentation [Graphics with Qt] for an overview of geometry issues with top-level widgets.

See also geometry [p. 463], width [p. 472] and size [p. 469].

Get this property's value with height().

bool hidden

This property holds whether the widget is explicitly hidden.

If FALSE, the widget is visible or would become visible if all its ancestors became visible.

See also hide() [p. 433], show() [p. 453], visible [p. 471] and isVisibleTo() [p. 435].

Get this property's value with isHidden().

QPixmap icon

This property holds the widget icon pixmap.

This property makes sense only for top-level widgets. If no icon has been set, icon() returns 0.

See also iconText [p. 464], caption [p. 460] and Setting the Application Icon [Programming with Qt].

Set this property's value with setIcon() and get this property's value with icon().

QString iconText

This property holds the widget icon text.

This property makes sense only for top-level widgets. If no icon text has been set, this functions returns QString::null.

See also icon [p. 464] and caption [p. 460].

Set this property's value with setIconText() and get this property's value with iconText().

bool isActiveWindow

This property holds whether this widget is the active window or a child of it.

The active window is the window that has keyboard focus.

When popup windows are visible, this property is TRUE for both the active window *and* for the popup.

See also setActiveWindow() [p. 445] and QApplication::activeWindow() [Additional Functionality with Qt].

Get this property's value with isActiveWindow().

bool isDesktop

This property holds whether the widget is a desktop widget.

A desktop widget is also a top-level widget.

See also isTopLevel [p. 465] and QApplication::desktop() [Additional Functionality with Qt].

Get this property's value with isDesktop().

bool isDialog

This property holds whether the widget is a dialog widget.

A dialog widget is a secondary top-level widget.

See also isTopLevel [p. 465] and QDialog [Dialogs and Windows with Qt].

Get this property's value with isDialog().

bool isModal

This property holds whether the widget is a modal widget.

This property only makes sense for top-level widgets. A modal widget prevents widgets in all other top-level widgets from getting any input.

See also isTopLevel [p. 465], isDialog [p. 464] and QDialog [Dialogs and Windows with Qt].

Get this property's value with isModal().

bool isPopup

This property holds whether the widget is a popup widget.

A popup widget is created by specifying the widget flag WType_Popup to the widget constructor. A popup widget is also a top-level widget.

See also isTopLevel [p. 465].

Get this property's value with isPopup().

bool isTopLevel

This property holds whether the widget is a top-level widget.

A top-level widget is a widget which usually has a frame and a caption (title). Popup and desktop widgets are also top-level widgets.

A top-level widget can have a parent widget. It will then be grouped with its parent: deleted when the parent is deleted, minimized when the parent is minimized etc. If supported by the window manager, it will also have a common taskbar entry with its parent.

QDialog and QMainWindow widgets are by default top-level, even if a parent widget is specified in the constructor. This behavior is specified by the WType_TopLevel widget flag.

Child widgets are the opposite of top-level widgets.

See also topLevelWidget() [p. 456], isDialog [p. 464], isModal [p. 464], isPopup [p. 465], isDesktop [p. 464] and parentWidget() [p. 442].

Get this property's value with isTopLevel().

int maximumHeight

This property holds the widget's maximum height.

This property corresponds to maximumSize().height().

See also maximumSize [p. 465] and maximumWidth [p. 466].

Set this property's value with setMaximumHeight() and get this property's value with maximumHeight().

QSize maximumSize

This property holds the widget's maximum size.

The widget cannot be resized to a larger size than the maximum widget size.

See also maximumWidth [p. 466], maximumHeight [p. 465], maximumSize [p. 465], minimumSize [p. 466] and sizeIncrement [p. 470].

Set this property's value with setMaximumSize() and get this property's value with maximumSize().

int maximumWidth

This property holds the widget's maximum width.

This property corresponds to maximumSize().width().

See also maximumSize [p. 465] and maximumHeight [p. 465].

Set this property's value with setMaximumWidth() and get this property's value with maximumWidth().

QRect microFocusHint

This property holds the currently set micro focus hint for this widget.

See the documentation of setMicroFocusHint() [p. 450] for more information.

Get this property's value with microFocusHint().

bool minimized

This property holds whether this widget is minimized (iconified).

This property is relevant only for top-level widgets.

See also showMinimized() [p. 454], visible [p. 471], show() [p. 453], hide() [p. 433] and showNormal() [p. 454].

Get this property's value with isMinimized().

int minimumHeight

This property holds the widget's minimum height.

This property corresponds to minimumSize().height().

See also minimumSize [p. 466] and minimumWidth [p. 467].

Set this property's value with setMinimumHeight() and get this property's value with minimumHeight().

QSize minimumSize

This property holds the widget's minimum size.

The widget cannot be resized to a smaller size than the minimum widget size. The widget's size is forced to the minimum size if the current size is smaller.

If you use a layout inside the widget, the minimum size will be set by the layout and not by setMinimumSize(), unless you set the layout's resize mode to QLayout::FreeResize.

See also minimumWidth [p. 467], minimumHeight [p. 466], maximumSize [p. 465], sizeIncrement [p. 470] and QLayout::resizeMode [Events, Actions, Layouts and Styles with Qt].

Set this property's value with setMinimumSize() and get this property's value with minimumSize().

QSize minimumSizeHint

This property holds the recommended minimum size for the widget.

If the value of this property is an invalid size, no minimum size is recommended.

The default implementation of minimumSizeHint() returns an invalid size if there is no layout for this widget, and returns the layout's minimum size otherwise. Most built-in widgets reimplement minimumSizeHint().

QLayout will never resize a widget to a size smaller than minimumSizeHint.

See also QSize::isValid() [Graphics with Qt], size [p. 469], minimumSize [p. 466] and sizePolicy [p. 470].

Get this property's value with minimumSizeHint().

int minimumWidth

This property holds the widget's minimum width.

This property corresponds to minimumSize().width().

See also minimumSize [p. 466] and minimumHeight [p. 466].

Set this property's value with setMinimumWidth() and get this property's value with minimumWidth().

bool mouseTracking

This property holds whether mouse tracking is enabled for this widget.

If mouse tracking is disabled (the default), this widget only receives mouse move events when at least one mouse button is pressed down while the mouse is being moved.

If mouse tracking is enabled, this widget receives mouse move events even if no buttons are pressed down.

See also mouseMoveEvent() [p. 439] and QApplication::setGlobalMouseTracking() [Additional Functionality with Qt].

Set this property's value with setMouseTracking() and get this property's value with hasMouseTracking().

bool ownCursor

This property holds whether the widget uses its own cursor.

If FALSE, the widget uses its parent widget's cursor.

See also cursor [p. 461].

Get this property's value with ownCursor().

bool ownFont

This property holds whether the widget uses its own font.

If FALSE, the widget uses its parent widget's font.

See also font [p. 462].

Get this property's value with ownFont().

bool ownPalette

This property holds whether the widget uses its own palette.

If FALSE, the widget uses its parent widget's palette.

See also palette [p. 468].

Get this property's value with ownPalette().

QPalette palette

This property holds the widget's palette.

As long as no special palette has been set, or after unsetPalette() has been called, this is either a special palette for the widget class, the parent's palette or (if this widget is a top level widget) the default application palette.

Instead of defining an entirely new palette, you can also use the paletteBackgroundColor, paletteBackgroundPixmap and paletteForegroundColor convenience properties to change a widget's background and foreground appearance only.

See also ownPalette [p. 467], colorGroup [p. 460] and QApplication::palette() [Additional Functionality with Qt].

Set this property's value with setPalette(), get this property's value with palette() and reset this property's value with unsetPalette().

QColor paletteBackgroundColor

This property holds the background color of the widget.

The palette background color is usually set implicitly by setBackgroundMode(), although it can also be set explicitly by setPaletteBackgroundColor(). setPaletteBackgroundColor() is a convenience function that creates and sets a modified QPalette with setPalette(). The palette is modified according to the widget's background mode. For example, if the background mode is PaletteButton the color used for the palette's QColorGroup::Button color entry is set.

If there is a background pixmap (set using setPaletteBackgroundPixmap()), then the return value of this function is indeterminate.

See also paletteBackgroundPixmap [p. 468], paletteForegroundColor [p. 468], palette [p. 468] and colorGroup [p. 460].

Set this property's value with setPaletteBackgroundColor(), get this property's value with paletteBackgroundColor() and reset this property's value with unsetPalette().

QPixmap paletteBackgroundPixmap

This property holds the background pixmap of the widget.

The palette background pixmap is usually set implicitly by setBackgroundMode(), although it can also be set explicitly by setPaletteBackgroundPixmap(). setPaletteBackgroundPixmap() is a convenience function that creates and sets a modified QPalette with setPalette(). The palette is modified according to the widget's background mode. For example, if the background mode is PaletteButton the pixmap used for the palette's QColorGroup::Button color entry is set.

If there is a plain background color (set using setPaletteBackgroundColor()), then this function returns 0.

See also paletteBackgroundColor [p. 468], paletteForegroundColor [p. 468], palette [p. 468] and colorGroup [p. 460].

Set this property's value with setPaletteBackgroundPixmap(), get this property's value with paletteBackgroundPixmap() and reset this property's value with unsetPalette().

QColor paletteForegroundColor

This property holds the foreground color of the widget.

setPaletteForegroundColor() is a convenience function that creates and sets a modified QPalette with setPalette(). The palette is modified according to the widget's *background mode*. For example, if the background mode is PaletteButton the palette entry QColorGroup::ButtonText is set to color.

See also palette [p. 468], QApplication::setPalette() [Additional Functionality with Qt], backgroundMode [p. 459], foregroundColor() [p. 431], backgroundMode [p. 459] and setEraseColor() [p. 447].

Set this property's value with setPaletteForegroundColor(), get this property's value with paletteForegroundColor() and reset this property's value with unsetPalette().

QPoint pos

This property holds the position of the widget in its parent widget.

If the widget is a top-level widget, the position is that of the widget on the desktop, including the frame.

When changing the position, the widget, if visible, receives a move event (moveEvent()) immediately. If the widget is not currently visible, it is guaranteed to receive an event before it is shown.

move() is virtual, and all other overloaded move() implementations in Qt call it.

Warning: If you call move() or setGeometry() from moveEvent(), you may experience infinite recursion.

See the Window Geometry documentation [Graphics with Qt] for an overview of geometry issues with top-level widgets.

See also frameGeometry [p. 463], size [p. 469], x [p. 472] and y [p. 472].

Set this property's value with move() and get this property's value with pos().

QRect rect

This property holds the internal geometry of the widget excluding any window frame.

The rect property equals QRect(0, 0, width(), height()).

See the Window Geometry documentation [Graphics with Qt] for an overview of geometry issues with top-level widgets.

See also size [p. 469].

Get this property's value with rect().

QSize size

This property holds the size of the widget excluding any window frame.

When resizing, the widget, if visible, receives a resize event (resizeEvent()) immediately. If the widget is not currently visible, it is guaranteed to receive an event before it is shown.

The size is adjusted if it lies outside the range defined by minimumSize() and maximumSize(). Furthermore, the size is always at least QSize(1, 1).

resize() is virtual, and all other overloaded resize() implementations in Qt call it.

Warning: If you call resize() or setGeometry() from resizeEvent(), you may experience infinite recursion.

See also pos [p. 469], geometry [p. 463], minimumSize [p. 466], maximumSize [p. 465] and resizeEvent() [p. 445].

Set this property's value with resize() and get this property's value with size().

QSize sizeHint

This property holds the recommended size for the widget.

If the value of this property is an invalid size, no size is recommended.

The default implementation of sizeHint() returns an invalid size if there is no layout for this widget, and returns the layout's preferred size otherwise.

See also QSize::isValid() [Graphics with Qt], minimumSizeHint [p. 466], sizePolicy [p. 470], minimumSize [p. 466] and updateGeometry() [p. 457].

Get this property's value with sizeHint().

QSize sizeIncrement

This property holds the size increment of the widget.

When the user resizes the window, the size will move in steps of sizeIncrement().width() pixels horizontally and sizeIncrement.height() pixels vertically, with baseSize() as basis. Preferred widget sizes are for nonnegative integers *i* and *j*:

```
width = baseSize().width() + i * sizeIncrement().width();
height = baseSize().height() + j * sizeIncrement().height();
```

Note that while you can set the size increment for all widgets, it only effects top-level widgets.

Warning: The size increment has no effect under Windows, and may be disregarded by the window manager on X.

See also size [p. 469], minimumSize [p. 466] and maximumSize [p. 465].

Set this property's value with setSizeIncrement() and get this property's value with sizeIncrement().

QSizePolicy sizePolicy

This property holds the default layout behavior of the widget.

If there is a QLayout that manages this widget's children, the size policy specified by that layout is used. If there is no such QLayout, the result of this function is used.

The default policy is Preferred/Preferred, which means that the widget can be freely resized, but prefers to be the size sizeHint() returns. Button-like widgets set the size policy to specify that they may stretch horizontally, but are fixed vertically. The same applies to lineedit controls (such as QLineEdit, QSpinBox or an editable QComboBox) and other horizontally orientated widgets (such as QProgressBar). QToolButton's are normally square, so they allow growth in both directions. Widgets that support different directions (such as QSlider, QScrollBar or QHeader) specify stretching in the respective direction only. Widgets that can provide scrollbars (usually subclasses of QScrollView) tend to specify that they can use additional space, and that they can survive on less than sizeHint().

See also sizeHint [p. 470], QLayout [Events, Actions, Layouts and Styles with Qt], QSizePolicy [p. 280] and updateGeometry() [p. 457].

Set this property's value with setSizePolicy() and get this property's value with sizePolicy().

bool underMouse

This property holds whether the widget is under the mouse cursor.

Get this property's value with hasMouse().

See also QEvent::Enter [Events, Actions, Layouts and Styles with Qt] and QEvent::Leave [Events, Actions, Layouts and Styles with Qt].

bool updatesEnabled

This property holds whether updates are enabled.

Calling update() and repaint() has no effect if updates are disabled. Paint events from the window system are processed normally even if updates are disabled.

setUpdatesEnabled() is normally used to disable updates for a short period of time, for instance to avoid screen flicker during large changes.

Example:

```
setUpdatesEnabled( FALSE );
bigVisualChanges();
setUpdatesEnabled( TRUE );
repaint();
```

See also update() [p. 456], repaint() [p. 443] and paintEvent() [p. 441].

Set this property's value with setUpdatesEnabled() and get this property's value with isUpdatesEnabled().

bool visible

This property holds whether the widget is visible.

Calling show() sets the widget to visible status if all its parent widgets up to the top-level widget are visible. If an ancestor is not visible, the widget won't become visible until all its ancestors are shown.

Calling hide() hides a widget explicitly. An explicitly hidden widget will never become visible, even if all its ancestors become visible, unless you show it.

Iconified top-level widgets also have hidden status, as well as having isMinimized() return TRUE. Windows that exist on another virtual desktop (on platforms that support this concept) also have hidden status.

A widget that happens to be obscured by other windows on the screen is considered to be visible.

A widget receives show and hide events when its visibility status changes. Between a hide and a show event, there is no need in wasting any CPU on preparing or displaying information to the user. A video application, for example, might simply stop generating new frames.

See also show() [p. 453], hide() [p. 433], hidden [p. 463], isVisibleTo() [p. 435], minimized [p. 466], showEvent() [p. 453] and hideEvent() [p. 433].

Get this property's value with isVisible().

QRect visibleRect

This property holds the currently visible rectangle of the widget.

This property is useful to optimize immediate repainting of a widget. Typical usage is:

```
repaint( w->visibleRect() );
```

or

```
repaint( w->visibleRect(), FALSE );
```

If nothing is visible, the rectangle returned is empty.

Get this property's value with visibleRect().

int width

This property holds the width of the widget excluding any window frame.

See the Window Geometry documentation [Graphics with Qt] for an overview of geometry issues with top-level widgets.

See also geometry [p. 463], height [p. 463] and size [p. 469].

Get this property's value with width().

int x

This property holds the x coordinate of the widget relative to its parent including any window frame.

See the Window Geometry documentation [Graphics with Qt] for an overview of geometry issues with top-level widgets.

See also frameGeometry [p. 463], y [p. 472] and pos [p. 469].

Get this property's value with x().

int y

This property holds the y coordinate of the widget relative to its parent and including any window frame.

See the Window Geometry documentation [Graphics with Qt] for an overview of geometry issues with top-level widgets.

See also frameGeometry [p. 463], x [p. 472] and pos [p. 469].

Get this property's value with y().

QWidgetFactory Class Reference

The QWidgetFactory class provides for the dynamic creation of widgets from Qt Designer .ui files.

`#include <qwidgetfactory.h>`

Public Members

- **QWidgetFactory** ()
- virtual **~QWidgetFactory** ()
- virtual QWidget * **createWidget** (const QString & className, QWidget * parent, const char * name) const

Static Public Members

- QWidget * **create** (const QString & uiFile, QObject * connector = 0, QWidget * parent = 0, const char * name = 0)
- QWidget * **create** (QIODevice * dev, QObject * connector = 0, QWidget * parent = 0, const char * name = 0)
- void **addWidgetFactory** (QWidgetFactory * factory)
- void **loadImages** (const QString & dir)

Detailed Description

The QWidgetFactory class provides for the dynamic creation of widgets from Qt Designer .ui files.

This class basically offers two things:

- Dynamically creating widgets from *Qt Designer* user interface description files. You can do this using the static function QWidgetFactory::create(). This function also performs signal and slot connections, tab ordering, etc., as defined in the .ui file, and returns the top-level widget in the .ui file. After creating the widget you can use QObject::child() and QObject::queryList() to access child widgets of this returned widget.
- Adding additional widget factories to be able to create custom widgets. See createWidget() for details.

This class is not included in the Qt library itself. To use it you must link against `libqui.so` (Unix) or `qui.lib` (Windows), which is built into `$(QTDIR)/lib` if you built *Qt Designer*.

See the Creating Dynamic Dialogs from .ui Files section of the Qt Designer manual for an example.

Member Function Documentation

QWidgetFactory::QWidgetFactory ()

Constructs a QWidgetFactory.

QWidgetFactory::~QWidgetFactory () [virtual]

Destructor.

void QWidgetFactory::addWidgetFactory (QWidgetFactory * factory) [static]

Installs a widget factory *factory*, which normally contains additional widgets that can then be created using a QWidgetFactory. See createWidget() for further details.

QWidget * QWidgetFactory::create (const QString & uiFile, QObject * connector = 0, QWidget * parent = 0, const char * name = 0) [static]

Loads the *Qt Designer* user interface description file *uiFile* and returns the top-level widget in that description. *parent* and *name* are passed to the constructor of the top-level widget.

This function also performs signal and slot connections, tab ordering, etc., as described in the .ui file. In *Qt Designer* it is possible to add custom slots to a form and connect to them. If you want these connections to be made, you must create a class derived from QObject, which implements all these slots. Then pass an instance of the object as *connector* to this function. If you do this, the connections to the custom slots will be done using the *connector* as slot.

If something fails, 0 is returned.

The ownership of the returned widget is passed to the caller.

QWidget * QWidgetFactory::create (QIODevice * dev, QObject * connector = 0, QWidget * parent = 0, const char * name = 0) [static]

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

Loads the user interface description from device *dev*.

QWidget * QWidgetFactory::createWidget (const QString & className, QWidget * parent, const char * name) const [virtual]

Creates a widget of the type `className` passing *parent* and *name* to its constructor.

If *className* is a widget in the Qt library, it is directly created by this function. If the widget isn't in the Qt library, each of the installed widget plugins is asked, in turn, to create the widget. As soon as a plugin says it can create the widget it is asked to do so. It may occur that none of the plugins can create the widget, in which case each installed widget factory is asked to create the widget (see addWidgetFactory()). If the widget cannot be created by any of these means, 0 is returned.

If you have a custom widget, and want it to be created using the widget factory, there are two approaches you can use:

1. Write a widget plugin. This allows you to use the widget in *Qt Designer* and in this QWidgetFactory. See the widget plugin documentation for further details. (See the Creating Custom Widgets with Plugins section of the Qt Designer manual for an example.
2. Subclass QWidgetFactory. Then reimplement this function to create and return an instance of your custom widget if *className* equals the name of your widget, otherwise return 0. Then at the beginning of your program where you want to use the widget factory to create widgets do a:

   ```
   QWidgetFactory::addWidgetFactory( new MyWidgetFactory );
   ```

   where MyWidgetFactory is your QWidgetFactory subclass.

void QWidgetFactory::loadImages (const QString & dir) [static]

If you use a pixmap collection (which is the default for new projects) rather than saving the pixmaps within the .ui XML file, you must load the pixmap collection. QWidgetFactory looks in the default QMimeSourceFactory for the pixmaps. Either add it there manually, or call this function and specify the directory where the images can be found, as *dir*. This is normally the directory called `images` in the project's directory.

QWidgetItem Class Reference

The QWidgetItem class is a layout item that represents a widget.

`#include <qlayout.h>`

Inherits QLayoutItem [Events, Actions, Layouts and Styles with Qt].

Public Members

- **QWidgetItem** (QWidget * w)
- virtual QSize **sizeHint** () const
- virtual QSize **minimumSize** () const
- virtual QSize **maximumSize** () const
- virtual QSizePolicy::ExpandData **expanding** () const
- virtual bool **isEmpty** () const
- virtual void **setGeometry** (const QRect & r)
- virtual QWidget * **widget** ()
- const QSize & **widgetSizeHint** () const

Detailed Description

The QWidgetItem class is a layout item that represents a widget.

This is used by custom layouts.

See also QLayout [Events, Actions, Layouts and Styles with Qt], Widget Appearance and Style and Layout Management.

Member Function Documentation

QWidgetItem::QWidgetItem (QWidget * w)

Creates an item containing widget *w*.

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

Returns TRUE if this item's widget is expanding; otherwise returns FALSE.

Reimplemented from QLayoutItem [Events, Actions, Layouts and Styles with Qt].

bool QWidgetItem::isEmpty () const [virtual]

Returns TRUE if the widget has been hidden; otherwise returns FALSE.

Reimplemented from QLayoutItem [Events, Actions, Layouts and Styles with Qt].

QSize QWidgetItem::maximumSize () const [virtual]

Returns the maximum size of this item.

Reimplemented from QLayoutItem [Events, Actions, Layouts and Styles with Qt].

QSize QWidgetItem::minimumSize () const [virtual]

Returns the minimum size of this item.

Reimplemented from QLayoutItem [Events, Actions, Layouts and Styles with Qt].

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

Sets the geometry of this item's widget to be contained within rect *r*, taking alignment and maximum size into account.

Reimplemented from QLayoutItem [Events, Actions, Layouts and Styles with Qt].

QSize QWidgetItem::sizeHint () const [virtual]

Returns the preferred size of this item.

Reimplemented from QLayoutItem [Events, Actions, Layouts and Styles with Qt].

QWidget * QWidgetItem::widget () [virtual]

Returns the widget managed by this item.

Reimplemented from QLayoutItem [Events, Actions, Layouts and Styles with Qt].

const QSize & QWidgetItem::widgetSizeHint () const

Implemented in subclasses to return a reference to the preferred size of this item.

QWidgetStack Class Reference

The QWidgetStack class provides a stack of widgets of which only the top widget is user-visible.

`#include <qwidgetstack.h>`

Inherits QFrame [p. 65].

Public Members

- **QWidgetStack** (QWidget * parent = 0, const char * name = 0)
- **~QWidgetStack** ()
- int **addWidget** (QWidget * w, int id = -1)
- void **removeWidget** (QWidget * w)
- QWidget * **widget** (int id) const
- int **id** (QWidget * widget) const
- QWidget * **visibleWidget** () const

Public Slots

- void **raiseWidget** (int id)
- void **raiseWidget** (QWidget * w)

Signals

- void **aboutToShow** (int)
- void **aboutToShow** (QWidget *)

Protected Members

- virtual void **setChildGeometries** ()

Detailed Description

The QWidgetStack class provides a stack of widgets of which only the top widget is user-visible.

The application programmer can move any widget to the top of the stack at any time using raiseWidget(), and add or remove widgets using addWidget() and removeWidget().

visibleWidget() is the *get* equivalent of raiseWidget(); it returns a pointer to the widget that is currently at the top of the stack.

QWidgetStack also provides the ability to manipulate widgets through application-specified integer ids. You can also translate from widget pointers to ids using id() and from ids to widget pointers using widget(). These numeric ids are unique (per QWidgetStack, not globally), but QWidgetStack does not attach any additional meaning to them.

The default widget stack is frameless, but you can use the usual QFrame functions (such as setFrameStyle()) to add a frame.

QWidgetStack provides a signal, aboutToShow(), which is emitted just before a managed widget is shown.

See also QTabDialog [Dialogs and Windows with Qt], QTabBar [p. 318], QFrame [p. 65] and Organizers.

Member Function Documentation

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

Constructs an empty widget stack with parent *parent* and called *name*.

QWidgetStack::~QWidgetStack ()

Destroys the object and frees any allocated resources.

void QWidgetStack::aboutToShow (int) [signal]

This signal is emitted just before a managed widget is shown if that managed widget has an id != -1. The argument is the numeric id of the widget.

void QWidgetStack::aboutToShow (QWidget *) [signal]

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

This signal is emitted just before a managed widget is shown. The argument is a pointer to the widget.

int QWidgetStack::addWidget (QWidget * w, int id = -1)

Adds widget *w* to this stack of widgets, with id *id*.

If you pass an id >= 0 this id is used. If you pass an *id* of -1 (the default), the widgets will be numbered automatically. If you pass -2 a unique negative integer will be generated. No widget has an id of -1.

If *w* is not a child of this QWidgetStack moves it using reparent().

Example: xform/xform.cpp.

int QWidgetStack::id (QWidget * widget) const

Returns the id of the *widget*. If *widget* is 0 or is not being managed by this widget stack, this function returns -1.

See also widget() [p. 480] and addWidget() [p. 479].

void QWidgetStack::raiseWidget (int id) [slot]

Raises the widget with id, *id*, to the top of the widget stack.

See also visibleWidget() [p. 480].

Example: xform/xform.cpp.

void QWidgetStack::raiseWidget (QWidget * w) [slot]

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

Raises widget *w* to the top of the widget stack.

void QWidgetStack::removeWidget (QWidget * w)

Removes widget *w* from this stack of widgets. Does not delete *w*. If *w* is the currently visible widget, no other widget is substituted. pareent

See also visibleWidget() [p. 480] and raiseWidget() [p. 480].

void QWidgetStack::setChildGeometries () [virtual protected]

Fixes up the children's geometries.

QWidget * QWidgetStack::visibleWidget () const

Returns a pointer to the currently visible widget (the one at the top of the stack), or 0 if nothing is currently being shown.

See also aboutToShow() [p. 479], id() [p. 479] and raiseWidget() [p. 480].

QWidget * QWidgetStack::widget (int id) const

Returns a pointer to the widget with id *id*. If this widget stack does not manage a widget with id *id*, this function returns 0.

See also id() [p. 479] and addWidget() [p. 479].

Qt Xt/Motif Support Extension

The Qt Xt/Motif Support Extension assists the migration of old Xt and Motif based code to the more comfortable Qt toolkit.

Information

The Qt Xt/Motif Extension consists of the following classes:

- QXtApplication allows mixing of Xt/Motif and Qt widgets.
- QXtWidget is the base widget for Xt/Motif and Qt combination.

How-to

1. Ensure you are using X11R6.
2. Ensure the programs you wish to port are compilable by a C++ compiler, or at least that this is the case for the parts which need migration.
3. Decide whether you are writing a new Qt application reusing old Xt widgets, or introducing new Qt widgets into an existing Xt/Motif application. See the corresponding example programs.

Known Bugs

This extension is under development. The following problems exist:

- The different focus models of Xt, Motif and Qt interfere.
- Clipboard handling between Xt/Motif and Qt does not work properly.

QXtApplication Class Reference

The QXtApplication class facilitates the mixing of Xt/Motif and Qt widgets.

This class is part of the **Qt Xt/Motif Extension**.

`#include <qxt.h>`

Inherits QApplication [Additional Functionality with Qt].

Public Members

- **QXtApplication** (int & argc, char ** argv, const char * appclass = 0, XrmOptionDescRec * options = 0, int num_options = 0, const char ** resources = 0)
- **QXtApplication** (Display * display, HANDLE visual = 0, HANDLE colormap = 0)
- **QXtApplication** (Display * display, int argc, char ** argv, HANDLE visual = 0, HANDLE colormap = 0)
- **~QXtApplication** ()

Detailed Description

This class is defined in the **Qt Xt/Motif Extension**, which can be found in the `qt/extensions` directory. It is not included in the main Qt API.

The QXtApplication class facilitates the mixing of Xt/Motif and Qt widgets.

The QXtApplication and QXtWidget classes allow old Xt or Motif widgets to be used in new Qt applications. They also allow Qt widgets to be used in primarily Xt/Motif applications. The facility is intended to aid migration from Xt/Motif to the more comfortable Qt system.

Member Function Documentation

QXtApplication::QXtApplication (int & argc, char ** argv, const char * appclass = 0, XrmOptionDescRec * options = 0, int num_options = 0, const char ** resources = 0)

Constructs a QApplication and initializes the Xt toolkit. The *appclass*, *options*, *num_options*, and *resources* arguments are passed on to XtAppSetFallbackResources and XtDisplayInitialize.

Use this constructor when writing a new Qt application which needs to use some existing Xt/Motif widgets.

The *argc* and *argv* arguments are passed to the QApplication constructor.

QXtApplication::QXtApplication (Display * display, HANDLE visual = 0, HANDLE colormap = 0)

Constructs a QApplication from the *display* of an already-initialized Xt application. If *visual* and *colormap* are non-zero, the application will use those as the default Visual and Colormap contexts.

Use this constructor when introducing Qt widgets into an existing Xt/Motif application.

QXtApplication::QXtApplication (Display * display, int argc, char ** argv, HANDLE visual = 0, HANDLE colormap = 0)

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

Constructs a QApplication from the *display* of an already-initialized Xt application. If *visual* and *colormap* are non-zero, the application will use those as the default Visual and Colormap contexts.

Use this constructor when introducing Qt widgets into an existing Xt/Motif application.

The *argc* and *argv* arguments are passed to the QApplication constructor.

QXtApplication::~QXtApplication ()

Destructs the application. Does not close the Xt toolkit.

QXtWidget Class Reference

The QXtWidget class allows mixing of Xt/Motif and Qt widgets.

This class is part of the **Qt Xt/Motif Extension**.

`#include <qxt.h>`

Inherits QWidget [p. 412].

Public Members

- **QXtWidget** (const char * name, Widget parent, bool managed = FALSE)
- **QXtWidget** (const char * name, WidgetClass widget_class, QWidget * parent = 0, ArgList args = 0, Cardinal num_args = 0, bool managed = FALSE)
- **~QXtWidget** ()
- Widget **xtWidget** () const
- bool **isActiveWindow** () const
- virtual void **setActiveWindow** ()

Protected Members

- virtual bool **x11Event** (XEvent * e)

Detailed Description

This class is defined in the **Qt Xt/Motif Extension**, which can be found in the `qt/extensions` directory. It is not included in the main Qt API.

The QXtWidget class allows mixing of Xt/Motif and Qt widgets.

QXtWidget acts as a bridge between Xt and Qt. For utilizing old Xt widgets, it can be a QWidget based on a Xt widget class. For including Qt widgets in an existing Xt/Motif application, it can be a special Xt widget class that is a QWidget. See the constructors for the different behaviors.

Member Function Documentation

QXtWidget::QXtWidget (const char * name, Widget parent, bool managed = FALSE)

Constructs a QXtWidget of the special Xt widget class known as "QWidget" to the resource manager.

Use this constructor to utilize Qt widgets in an Xt/Motif application. The QXtWidget is a QWidget, so you can create subwidgets, layouts, etc. using Qt functionality.

The *name* is the object name passed to the QWidget constructor. The widget's parent is *parent*.

If the *managed* parameter is TRUE and *parent* in not null, XtManageChild it used to manage the child.

QXtWidget::QXtWidget (const char * name, WidgetClass widget_class, QWidget * parent = 0, ArgList args = 0, Cardinal num_args = 0, bool managed = FALSE)

Constructs a QXtWidget of the given *widget_class* called *name*.

Use this constructor to utilize Xt or Motif widgets in a Qt application. The QXtWidget looks and behaves like the Xt class, but can be used like any QWidget.

Note that Xt requires that the most top level Xt widget is a shell. This means, if *parent* is a QXtWidget, the *widget_class* can be of any kind. If there isn't a parent or the parent is just a normal QWidget, *widget_class* should be something like `topLevelShellWidgetClass`.

The arguments, *args*, *num_args* are passed on to XtCreateWidget.

If the *managed* parameter is TRUE and *parent* in not null, XtManageChild it used to manage the child.

QXtWidget::~QXtWidget ()

Destructs the QXtWidget.

bool QXtWidget::isActiveWindow () const

Different from QWidget::isActiveWindow()

void QXtWidget::setActiveWindow () [virtual]

Implement a degree of focus handling for Xt widgets.

Reimplemented from QWidget [p. 445].

bool QXtWidget::x11Event (XEvent * e) [virtual protected]

Reimplemented to produce the Xt effect of getting focus when the mouse enters the widget. The event is passed in *e*.

This function is under development and is subject to change.

Widget QXtWidget::xtWidget () const

Returns the Xt widget equivalent for the Qt widget.

Examples: mainlyMotif/editor.cpp and mainlyXt/editor.cpp.

Index