Troll Tech AS

Qt 1.42

Tutorial and Reference

Preliminary Version, December 1, 1998

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

Introduction

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

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

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

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

Andrew Oram and Steve Talbott: Managing Projects with make, second edition
ISBN 0-937175-90-0
152 pages
http://www.oreilly.com/catalog/make2/

Steve Oualline: Practical C++ Programming
ISBN 1-56592-139-9
584 pages
http://www.oreilly.com/catalog/cplus/

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

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

1. Chapter 1: Hello, World!

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

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

1.1. Line by Line Walk-Through

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
}
```

1.2. Behavior

You should now try to compile and run this program.

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

1.3. Excercises

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

1.3.1. URL

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

2. Chapter 2: Calling it Quits

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

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

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

2.1. Line by Line Walk-Through

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

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

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

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

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

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

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

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

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

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

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

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

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

The Signals and Slots documentation describes this topic in detail.

2.2. Behavior

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

2.3. Excercises

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

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

2.3.1. URL

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

3. Chapter 3: Family Values

This example shows how to create mother and child widgets.

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

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

3.1. Line by Line Walk-Through

```
QWidget w;
```

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

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

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

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

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

A child is born.

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

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

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

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

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

3.2. Behavior

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

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

3.3. Excercises

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

For X11 users:

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

3.3.1. URL

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

4. Chapter 4: Let There Be Widgets

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

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

4.1. Line by Line Walk-Through

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

4.2. Behavior

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

4.3. Excercises

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

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

The background color can be changed.

4.3.1. URL

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

5. Chapter 5: Building Blocks

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

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

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

5.1. Line by Line Walk-Through

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

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

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

Nothing new compared to the previous chapter.

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

MyWidget now gains the ability to handle resize event.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

5.2. Behavior

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

5.3. Excercises

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

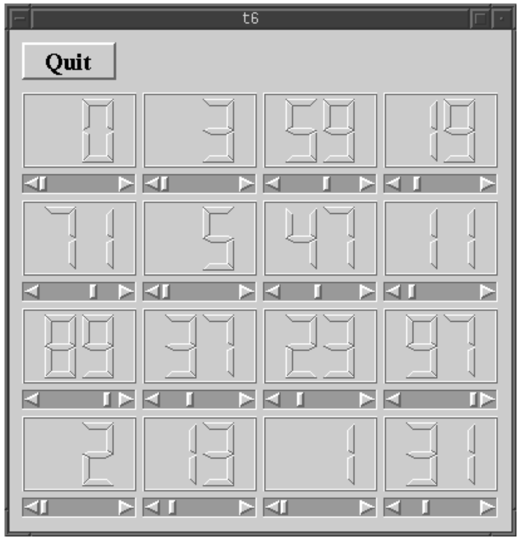
You can also change the scroll bar's range.

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

5.3.1. URL

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

6. Chapter 6: Building Blocks Galore!



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

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

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

6.1. Line by Line Walk-Through

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

6.2. Behavior

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

6.3. Excercises

Change MyWidget so each LCDRange is guaranteed to be square.

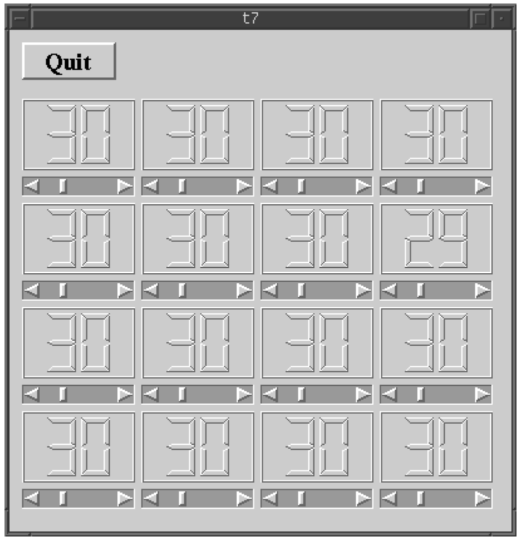
Initialize each scroll bar with a different/random value on startup.

6.3.1. URL

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

---

7. Chapter 7: One Thing Leads to Another



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

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

7.1. Line by Line Walk-Through

lcdrange.h

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

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

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

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

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

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

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

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

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

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

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

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

lcdrange.cpp

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

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

This code is from the LCDRange constructor.

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

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

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

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

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

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

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

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

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

main.cpp

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

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

7.2. Behavior

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

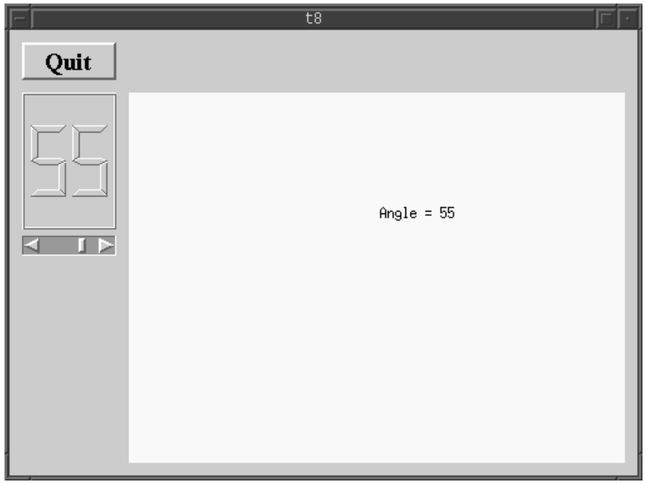
7.3. Excercises

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

7.3.1. URL

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

8. Chapter 8: Preparing for Battle



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

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

8.1. Line by Line Walk-Through

lcdrange.h

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

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

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

lcdrange.cpp

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

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

cannon.h

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

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

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

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

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

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

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

cannon.cpp

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

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

```
    ang = 45;
}
```

The constructor simply initializes the angle value to 45 degrees.

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

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

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

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

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

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

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

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

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

main.cpp

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

We include our new class.

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

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

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

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

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

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

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

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

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

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

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

This illustrates the power of component programming and proper encapsulation.

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

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

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

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

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

8.2. Behavior

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

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

8.3. Excercises

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

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

8.3.1. URL

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

9. Chapter 9: With Cannon You Can