GETTING STARTED WITH QT

JONO BACON

In this issue we will begin taking a look at Qt, and start to write some programmes using it. Before we get started, we should take a look at some of the concepts of writing Qt software.

On your marks...

To get started, lets actually look at what we need. The first thing is fairly obviously Qt itself. You will also need the various compilers, linkers and libraries to build your software. This all usually comes as standard with most modern Linux distributions. If you have any problems, please refer to your distribution manual.

The other thing you will need to be familiar with is C++. Qt software is natively written in C++, and although there are bindings for various other languages, I will be focusing on C++ in this series. If you do not know C++, or would like to brush up on it, do a search for c++ on *http://www.google.com/* and you should get plenty of documentation.

Although you do not need anything in particular to write the code, apart from a text editor of some description, I would suggest using an IDE for your development. For these purposes I will be using KDevelop as it is an excellent IDE for Qt and KDE development (as well as many other types of project also).

How Qt works

Qt is a powerful and flexible toolkit, and it is important to get to grips with some of the concepts of how Qt works. I will give a quick overview of the concepts here, but I will cover certain concepts in more detail as we continue the series.

Qt contains a number of classes. Each class does something specific and provides a lot of functionality. There are a number of classes that inherit features and functions from other classes, so you can build up a comprehensive set of functionality for higher level classes. Qt is primarily a graphical toolkit, and as such provides a number of on screen objects such as scroll bars, buttons, checkboxes etc, called widgets. These widgets are the primary objects you will use for user interaction and for the visual look and feel of your application. Although there are lots of graphical widgets, there are also a number of classes for dealing with behind the scenes processing, network access, data management etc. Qt provides a number of convenience classes written to make things such as stacks, linked lists, tree's and other such structures easier to use.

Qt also includes a clever and sophisticated system for giving your on and off screen objects functionality. This system is called the Signal and Slots system. I will be covering this in more detail later in the series. The basic functionality of the system is connecting desired functionality to your

Hello world!

objects when you interact with them. An example would be if you click on a button, a dialog box pops up.

Getting going

OK, let us get started on our Qt coding expedition and resurrect the traditional Hello World! program. Type in the following program into your editor or IDE and compile it. For a few details on compiling Qt code, see the Compiling Qt Programs box.

```
1      #include <qapplication.h>
2      #include <qlabel.h>
3
4      int main( int argc, char **argv )
5      {
6          QApplication a( argc, argv );
7
8          QLabel lab("Hello World!", 0,
"label", 0);
9
10         a.setMainWidget( &lab );
11         lab.show();
12
13         return a.exec();
14     }
```

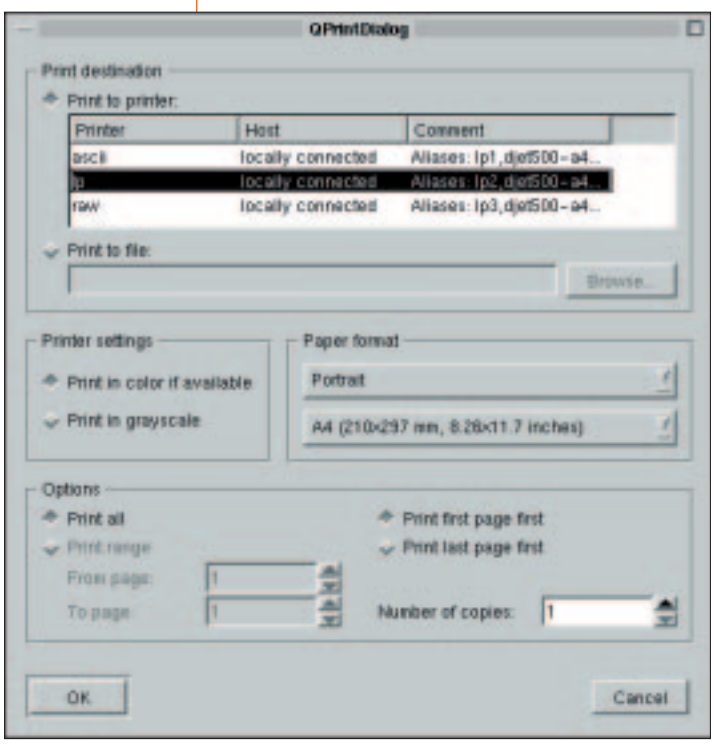
This simple program simply creates a window and puts Hello World! in it. Let's take a look at how this program works:

Lines 1 and 2 include the relevant header files for the Qt classes we will use. We are using QAppication and QLabel, so we therefore include these include files. On line 4 main() begins with the command line arguments we could process if we wanted to, we don't need to on this occasion though). On line 6 we then see the first part of our Qt program.

This then creates a QApplication called 'a', which accepts the command line arguments from main(). Each Qt application must have one QApplication object created. This class deals with application wide settings and garbage collection. Once we have created our QApplication object, we can then create our text.

Line 8 is where we create a QLabel object called 'lab', passing it some information that is useful when creating the object. The main two arguments to be familiar with on this line are the "Hello World!" argument which is the text that appears on the label, and the third argument, which is the parent of the label, details in a moment. As we have only one widget, we can set the parent argument (the third argument) to 0, and this puts the label in a new window.

The next part of this program is on line 10 where we use the setMainWidget() method to set the main widget of this application. Although this sounds obvious, it is quite important as when the main widget is killed or destroyed, the application exits. It is not essential to set the main widget, but most programs do. The next line in our program is line 11. This line shows the QLabel widget. It is important to remember that Qt widgets are not shown by default, and therefore you must run show() on them, alternatively other classes and methods automatically run show() for you. The final line on line 13 is where you let Qt take over interaction of the widgets and take control.



The QPrintDialog Widget

Parents and children

OK, so now we are playing the Qt game, lets discuss what this whole parent and child malarkey is all about. Parent/Child relationships are one of the key aspects to GUI programming, and a concept inherent in Qt. The idea is that you can have a widget that is a parent, and that there is another widget that sits on the parent widget called a child. An example of this would be a window with 4 buttons in it. The window would be the parent, and each button would be a child. This concept of a parent and child relationship is utilised in virtually every graphical widget in the Qt toolkit. At this point it is a good idea to point out that the Qt documentation is wonderful and discusses using the various classes and provides lots of useful information. You can find the documentation by opening up your web browser and looking at $QTDIR/doc/html/index.html.

Lets take a look at the various ways we can construct a QLabel like we did in our first program. The QLabel class documentation tells

us there are the following constructors available:

```
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 &,
QWidget * parent, const char * name=0,
WFlags f=0 )
```

As you can see from the code, I used the second line from this selection of constructors. This allows us set the text of the QLabel instead of using setText() to set it after we create the object. You can see that the second argument with the second constructor is the parent of the type QWidget *. A QWidget is a fundamental class in Qt that can act as a parent for other items. Typically the QWidget is used as an area of screen that can hold other widgets.

An example of using a QWidget as a parent would be:

```
1       #include <qapplication.h>
2       #include <qlabel.h>
3
4       class LabWidget : public QWidget
5       {
6       public:
7           LabWidget( QWidget *parent=0,
const char *name=0 );
8       };
9
10     LabWidget::LabWidget( QWidget
*parent, const char *name )
11       : QWidget( parent, name )
12     {
13         setMinimumSize( 200, 120 );
14         setMaximumSize( 200, 120 );
15
16         QLabel * lab = new QLabel("Hello
World", this, "label", 0);
17         lab->setGeometry( 80, 50, 75, 30
);
18     }
19
20     int main( int argc, char **argv )
21     {
22         QApplication a( argc, argv );
23
24         LabWidget myWidg;
25         myWidg.setGeometry( 200, 100,
200, 120 );
26         a.setMainWidget( &myWidg );
27         myWidg.show();
28         return a.exec();
29     }
```

In this example the application basically behaves pretty much the same, although I created a class called LabWidget. This LabWidget class inherits from QWidget on line 4, so therefore when we create this QLabel on line 16 and use 'this' we are using the QWidget as a parent. The only other difference is that I set the geometry of the QLabel on line 17, and I set the Geometry of the LabWidget object (which is QWidget derived) on line 26.

The QTextBrowser Widget displaying a HTML page

On we go next month...

Well that's all we have time for this month, but next month I will be looking at some of the other widgets and layout managers to make your interfaces more streamlined. Stay tuned folks...

Compiling Qt Programs

Compiling Qt programs is similar to compiling other software using libraries on Linux machines. It is suggested that you read the gcc manual and HOWTO's at *http://www.linuxdoc.org/*. Typically you need to make sure you link with -lqt and other XFree86 linker flags (these may include -lXext -lX11).