QT

GETTING STARTED WITH QT

Welcome to part three of our foray into the interesting world of Qt application development by Jono Bacon. This month we will take a long hard look at geometry classes for creating interfaces and examine how Qt deals with interaction with our widgets

Getting organised

To get us started this month we take a look at some of the layout classes Qt has available for helping create your interfaces. First, type in the following program and compile it:

```
1  #include <qapplication.h>
2  #include <qvbox.h>
3  #include <qpushbutton.h>
4
5  class MyClass : public QVBox
6  {
7  public:
8  MyClass();
9  ~MyClass();
10
11 private:
12 QPushButton * bobButt;
13 QPushButton * fredButt;
14 QPushButton * frankButt;
15 QPushButton * jimButt;
16 };
17
18 MyClass::MyClass()
19 {
20 bobButt = new QPushButton("Bob", this);
21 fredButt = new QPushButton("Fred", this);
22 frankButt = new QPushButton("Frank", this);
23 jimButt = new QPushButton("Jim", this);
24 }
25
26 MyClass::~MyClass()
27 {
28 }
29
30 int main( int argc, char **argv )
31 {
32 QApplication a( argc, argv );
33
34 MyClass w;
35 a.setMainWidget( &w );
36 w.show();
37 return a.exec();
38 }
```

In this snippet of code we create four QPushButton pointers on lines 12 – 15 (making sure to include qpushbutton.h on line three). The actual QPushButton objects are then created on lines 20 – 23. main() is much the same as in the previous code we have looked at. When you run the program you get something like in Figure 1. As you can see, the four buttons are lined up vertically in a nice neat fashion and they take up equal space in the window. Try editing out a button and recompiling, and you will see that the space is accommodated cleanly for each button again. So how does this magic work?

Figure 1: Four buttons equally spaced

Well, if you look at line five, you can see we inherit QVBox. QVBox is a class for arranging widgets in a vertical fashion and is very useful when you inherit from it as it will automatically arrange child widgets into a vertical layout. In this example we added push buttons as child widgets, but let's also look at combining QVBox with a QHBox (for horizontal layouts):

```
1  #include <qapplication.h>
2  #include <qhbox.h>
3  #include <qvbox.h>
4  #include <qpushbutton.h>
5
6  class MyClass : public QVBox
7  {
8  public:
9  MyClass();
10 ~MyClass();
11
12 private:
13 QHBox * hbox;
14 QPushButton * bobButt;
15 QPushButton * fredButt;
16 QPushButton * frankButt;
17 QPushButton * jimButt;
18 QPushButton * janButt;
19 QPushButton * aprilButt;
20 QPushButton * mayButt;
21
22 };
23
```

```
24 MyClass::MyClass()
25 {
26 hbox = new QHBox(this);
27 janButt = new QPushButton("Jan", hbox);
28 aprilButt = new QPushButton("April", hbox);
29 mayButt = new QPushButton("May", hbox);
30 bobButt = new QPushButton("Bob", this);
31 fredButt = new QPushButton("Fred", this);
32 frankButt = new QPushButton("Frank", this);
33 jimButt = new QPushButton("Jim", this);
34 }
35
36 MyClass::~MyClass()
37 {
38 }
39
40 int main( int argc, char **argv )
41 {
42 QApplication a( argc, argv );
43
44 MyClass w;
45 a.setMainWidget( &w );
46 w.show();
47 return a.exec();
48 }
```

In this example I firstly added hbox.h as an include file on line 2. I then created a pointer to a QHBox object on line 13, and created the object on line 26. On lines 27 – 29 I created three more QPushButton objects (their pointers being declared on lines 18 – 20), but instead of setting the parent to 'this', I set it to 'hbox' which is the name of the QHBox object. By setting the parent to 'hbox' of a widget, it is added to the layout manager specified in the parent and is organised. So when we create a QHBox object on line 26, it is then housing the new three push buttons horizontally at the top of the vertical manager. This can all be seen in Figure 2.

Figure 2: Now with three new push buttons

Layout management is something integral to Qt interface design. We will cover more on interface design in the next issue.

Connecting the pieces together

OK, so we've now come quite far. We have discussed widgets, layout, parent/child relationships and written a couple of small programs. This is all fine and dandy, but our programs don't actually do anything yet. For example when I click on a button, I want something to happen. To do this there is a comprehensive framework built right into Qt called the Signal/Slot framework. This is a system of connecting widgets to functions so that when you do something some functionality can be associated with it.

The way signals and slots work is that each widget (a graphical object on screen like a button) has a number of signals. A signal is a function that is emitted when you do something with the widget. For example, to see the signals that are available for QPushButton's, we need to look at the QButton documentation (as QPushButton is a type of QButton and inherits it). We can see the following signals:

- void pressed ()
- void released ()
- void clicked ()
- void toggled (bool)
- void stateChanged (int)

So when a user clicks on a QPushButton, the clicked () signal is emitted. We can then connect this signal to a slot. A slot is just a normal method that can do whatever needed in response to the signal being emitted. So how does this work, you ask? Well to explain, lets look at some code to get us started. You will need to use multiple files for this code. Type the following code in:

myclass.h:

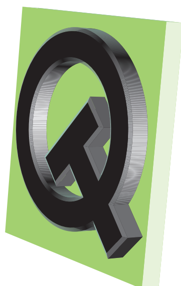
```
1  #include <qapplication.h>
2  #include <qhbox.h>
3  #include <qvbox.h>
4  #include <qpushbutton.h>
5
6  #ifndef MYCLASS_H
7  #define MYCLASS_H
8
9  class MyClass : public QVBox
10 {
11 Q_OBJECT
12
13 public:
14 MyClass();
15 ~MyClass();
16
17 public slots:
18 void slotJim();
19
20 private:
21 QHBox * hbox;
22 QPushButton * bobButt;
23 QPushButton * fredButt;
24 QPushButton * frankButt;
25 QPushButton * jimButt;
26 QPushButton * janButt;
27 QPushButton * aprilButt;
28 QPushButton * mayButt;
29
30 };
31
33 #endif
```

myclass.cpp:

```
1  #include <qmessagebox.h>
2  #include "myclass.h"
3
4  MyClass::MyClass()
```

> A slot is just a normal method that can do whatever needed in response to the signal being emitted

moc (Meta Object Compiler) is a little tool that converts some of the Qt signals and slots syntax into regular C++ code



```
5  {
6  hbox = new QHBox(this);
7  janButt = new QPushButton("Jan", hbox);
8  aprilButt = new QPushButton("April", hbox);
9  mayButt = new QPushButton("May", hbox);
10 bobButt = new QPushButton("Bob", this);
11 fredButt = new QPushButton("Fred", this);
12 frankButt = new QPushButton("Frank", this);
13 jimButt = new QPushButton("Jim", this);
14
15 connect( jimButt, SIGNAL( clicked() ), this,
SLOT( slotJim() ) );
16 }
17
18 MyClass::~MyClass()
19 {
20 }
21
22 void MyClass::slotJim()
23 {
24 QMessageBox::information( this, "Woohoo!",
"slotJim() has been called!\n", "Cancel" );
25 }
```

main.cpp:

```
1  #include <qapplication.h>
2  #include "myclass.h"
3
4  int main( int argc, char **argv )
5  {
6  QApplication a( argc, argv );
7
8  MyClass w;
9  a.setMainWidget( &w );
10 w.show();
11 return a.exec();
12 }
```

You will need to run the moc tool on the header file if you are building this by hand. See the Qt documentation for details on this. Before we look at the code, let's have a quick discussion of what moc actually is.

moc (Meta Object Compiler) is a little tool that converts some of the Qt signals and slots syntax into regular C++ code, and it also does some other nifty little things. You can see this code for example in the header file where you see 'Q_OBJECT' and public slots:. The 'Q_OBJECT' code indicates you are using the Qt object model (the signals/slots framework) in this header file. Always put this at the top of any class that uses signals and slots. The 'public slots:' part of the code indicates the following methods are slots that will be connected to signals. We have a single slot slotJim() which is a method like any other normal method.

Now let's take a look at line 15. This line is where the actual connection between the signal and slot occurs. It is in this format:

```
QObject::connect( object_that_emits_the_signal,
SLOT( signal() ), object_with_slot, SLOT(
slotname() ) );
```

We have the following code:

```
connect( jimButt, SIGNAL( clicked() ), this,
SLOT( slotJim() ) );
```

First of all we do not need the QObject:: prefix as we inherit QVBox which in turn inherits QObject down the line. We can see that the jimButt object (the button with "Jim" written on it) is the object we are connecting a slot to. We are dealing with the clicked() signal in this connection. We could of course use any of the other signals, but clicked() is a good one to start with. We then connect this signal to the slotJim() slot. We specify 'this' as the object whilst MyClass has the slot definition.

You may have seen some of the signals have a parameter such as toggled(bool). This signal is for when the button is a toggle button and you want to pass to the slot whether the button is toggled or not as the parameter. To use signals that pass a parameter, your slot MUST accept the same parameter type. This may sound like a limitation but in practice it really isn't: this feature is due to Qt being type safe which is a good thing. So for example you could have the following connection:

```
connect( toggleButt, SIGNAL( toggled( bool ) ),
this, SLOT( slotIsToggled( bool ) ) );
```

You could then use the slotIsToggled(bool) slot like this example:

```
MyClass::slotIsToggled( bool state)
{
if( state == TRUE )
{
// do something
}
else
{
// something else
}
}
```

Wrapping things up

Well, in this tutorial we have looked at layout managers, signals in widgets, signals and slots and a few others things. We are well on the way now to writing more comprehensive Qt applications. Next month we will build our first application based on this knowledge and use Qt Designer to develop our interfaces. Until then, I suggest you read through the Qt documentation and have a play with the different signals and methods available for widgets such as QPushButton, QLabel etc. Have fun!