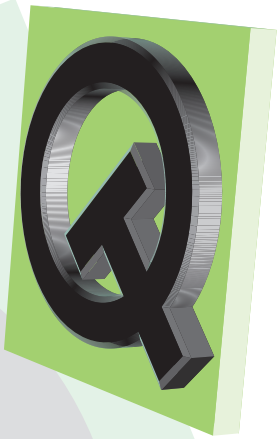
QT GETTING STARTED WITH QT



We've come a long way since our tentative steps in the first article of this series. For our latest instalment, Jono Bacon shows us how create interfaces with Qt Designer

Last issue we covered some of the ways we can add controls to our programs, and how we can have automatic layout managers. In this issue we will begin using an interface-building tool, called Qt Designer, to create our interfaces quickly and easily, and look into ways that we can embed these interfaces into our code. Before we get started though, lets look into some theory of how all this fits together.

Today we are going to first create a simple form in Qt Designer and then merging it with our code so it will be displayed. This involves a few processes:

- 1. Create the form
- 2. Create the code
- 3. Add the form to the build system
- 4. Compile

The first step is to create the form. This is done using the graphical Qt Designer. Qt Designer gives us the opportunity to select what kind of form we will be creating. There are options for a Widget, Dialog, Configuration Dialog, Wizard and more. Each of these templates gives us a form, which we can then draw on more widgets that we can use.

When we save this form, Qt Designer stores it as an XML formatted file, which ends in .ui. This *.ui file contains a number of formatting commands that specify to another program (uic) how our form is organised. The purpose of uic is to take our *.ui file and to create a header and source code file with the C++ code needed to create the form. This code can then be utilised in your project to create the form.

That's the basics of how we will create the form and embed it in our code. We will now go ahead and look at each step in detail, and I will guide you through fitting the pieces together.

Getting going with Qt Designer

In this article I do not have the space to give a complete step-by-step tutorial on using Qt Designer, but rest assured, it comes with a plentiful supply of documentation, which can be accessed via the help menu. I will however go through the main points to get started and cover the (few) pitfalls you may encounter. Qt Designer has been well designed by Trolltech, so I am sure you will find it fairly straightforward.

First of all, load up Qt Designer by selecting it from your GUI menu or by typing designer in a terminal. The main Qt Designer window will load up, and you will see a number of icons, toolbars and some status displays. The first thing we need to do is to create a new form from one of the templates. This can be done by clicking File/ New on the menu bar. You can then select the type of form to use. Click on the Widget option and you will then be presented with a form inside the Qt Designer window. This form is where you can add your buttons, text, input boxes and more.

To get us started, we will be build the interface, as shown in Figure 1. In this form we have a frame, a button, a text entry box (called a line edit) and a label. To create these items, we use the toolbar buttons to select the relevant tool and then draw on the form. You can then use the Property Editor (activated with the Window menu) to set the widget's properties and fine tune it. For the time being we don't really need to set any properties, as we will discuss this later.

Although we don't really need to set any widget properties, there is one property you should set on EVERY form you make, and this is the name property for the form. This property is displayed when you click on the form background and look at the Property Editor. The reason why this is so important is that this name is the name that will be used for the class in the generated code uic makes.

When you have created your form, select File/ Save and save it to the directory where your project is. The general naming convention for Qt Designer files is to call it the same name as the class name (the text you entered in the form's name property, a little while back). Make sure you put .ui at the end of the filename.

Adding the form to your project

To add the form to your project, you can do it either manually or via your IDE. If you are using KDevelop, you can use the Add File To Project dialog to add the

The input box built

.ui file (ensure you deselect the checkbox for adding header information). To add it to a Makefile.am (the configuration file for Automake), check near the top of the file and add it to the sources list. For example:

```
qtprog_SOURCES = myclass.cpp main.cpp myform.ui
```

The clever thing about this is that when you add your .ui file to a Makefile.am, uic is automatically run for you to convert your .ui file into code. If you're not using Makefiles at all, you can manually convert the code as such:

```
uic -o myform.h MyForm.ui
uic -i myform.h -o myform.cpp MyForm.ui
```

Now this has all been generated, the last step is to implement it into our program. This is where a little bit of clever C++ comes into play, and I will explain why.

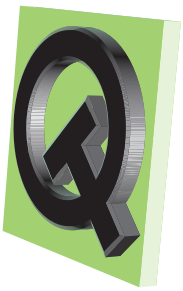
Late binding and Qt Designer

When using Qt Designer to create our interfaces we have a problem on our hands. The problem is that when you modify your *.ui file in Qt Designer, the XML *.ui is updated and therefore uic must be rerun on it to regenerate your C++ code. The problem with this is that you cannot then edit your generated *.cpp and *.h source files each time you update the interface. The main problem with this is how to add your slot methods to the class when they will be destroyed when the interface is updated.

To resolve this problem you need to use a C++ technique called late binding. The idea behind this is that you create another class called an implementation class, which inherits your generated interface class. When you call the slot from the interface class it really calls the slot from the implementation class. This is best explained with an example, and we will use MyForm.ui interface that we developed earlier. Add the following code to your project:

File: myclass.h:

```
1  #include "myform.h"
2
3  #ifndef MYCLASS_H
4  #define MYCLASS_H
5
6  class MyClass : public MyForm
7  {
8       public:
9               MyClass( QWidget* parent = 0,
const char* name = 0, WFlags fl = 0 );
10              ~MyClass();
11 };
12
13 #endif
```



File: myclass.cpp:

```
1  #include "myclass.h"
2
3  MyClass::MyClass( QWidget* parent = 0, const
char* name = 0, WFlags fl = 0 )
4  : MyForm( parent, name, fl )
5  {
6  }
7
8  MyClass::~MyClass()
9  {
10 }
```

When implemented into your build system, these files will show your interface, but do not have any slots. Let's first discuss how it shows our interface. First, in myclass.h on line 6 we inherit MyForm, which is the name of the Form that we set in the Properties Editor in Qt Designer. As we inherit from this class, we need to line our constructors up so they pass the right arguments across. This is done on line 9 in myform.h and we can see the constructor in myclass.cpp passes the arguments from the MyForm constructor. If you check in your generated code for the form you can see it is built in the constructor, so by us inheriting MyForm, the MyForm constructor is called first and builds the interface. If this is confusing you, I suggest you get a decent book on C++; a good one is C++ FAQs 2nd Edition.

OK, so now the interface is up, we need to add our slots. We can do this in Qt Designer. Let's assume that we wish to have the button on our form call a slot called slotMySlot(). First click on the button and set the name in the Properties Editor. Next we need to actually create a connection between a signal and a slot. Luckily this is very simple in Qt Designer. Click on the Connections icon or select Edit/ Connections, and your mouse pointer will change to a crosshair. This crosshair can then be used to click on the button and drag the mouse so the red outline covers the form. This outline indicates which signals/slots will be used. You will then be presented with the Edit Conenctions dialog box. On the left-hand side we can see the signals available for the button, and on the right is the selection of slots we can use. At the moment we have not created any slots so the list is empty. We can add a slot by clicking on the Edit Slots button. Click on New Slot and add slotMySlot() in the text box. Click on OK to finish. You can then select the clicked() signal on the slotMySlot() slot and the connection is made automatically. Click OK to finish. You can now save your *.ui file.

We then modify the above files so we can add the slots. Are files are now:

File: myclass.h

```
1  #include "myform.h"
2
3  #ifndef MYCLASS_H
```

```
4  #define MYCLASS_H
5
6  class MyClass : public MyForm
7  {
8       Q_OBJECT
9
10      public:
11              MyClass( QWidget* parent = 0,
const char* name = 0, WFlags fl = 0 );
12              ~MyClass();
13
14      public slots:
15              virtual void slotMySlot();
16 };
17
18 #endif
```

File: myclass.cpp

```
1  #include <qmessagebox.h>
2  #include "myclass.h"
3
4  MyClass::MyClass( QWidget* parent = 0, const
char* name = 0, WFlags fl = 0 )
5  : MyForm( parent, name, fl )
6  {
7  }
8
9  MyClass::~MyClass()
10 {
11 }
12
13 void MyClass::slotMySlot()
14 {
16      QMessageBox::information( this,
"Notice", "Woohoo! slotMySlot() has been
called!");
17 }
```

The modifications here are in the addition of the Q_OBJECT and public slots to the myclass.h, and on line 15 we can see the slot. Note the addition of the keyword virtual. This keyword says that this slot will be called using the class of the object that calls it. So if, for example, we have an object called aObj which is created from class A and bObj which is created from class B, both classes can have a slotMySlot() but aObj/ slotMySlot() will call A::slotMySlot() and bObj/ slotMySlot() will call B::slotMySlot. This is why when we create an instance of MyClass in main.cpp, the MyClass version of slotMySlot() is called.

Wrapping things up

Today you have learned some complex concepts in C++ and how to apply these concepts to building interfaces in a rapid fashion with Qt Designer. Again, I suggest you supplement this information by using resources on the Internet, and maybe visiting #qt on *irc.openproject.net*.

We will be finishing this series in the next issue where I will be looking at some of the other remaining features of Qt that may be useful to you. Have fun!