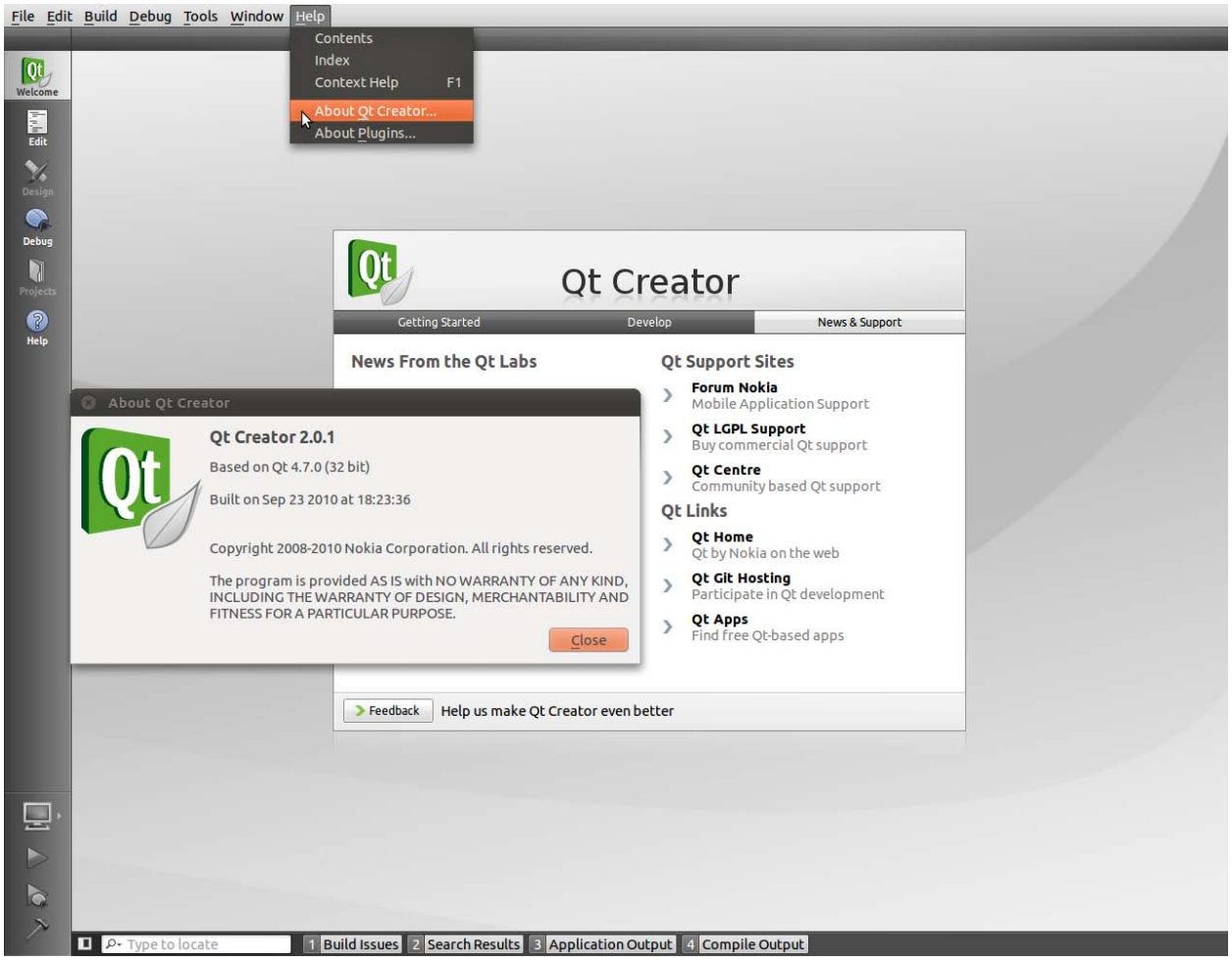
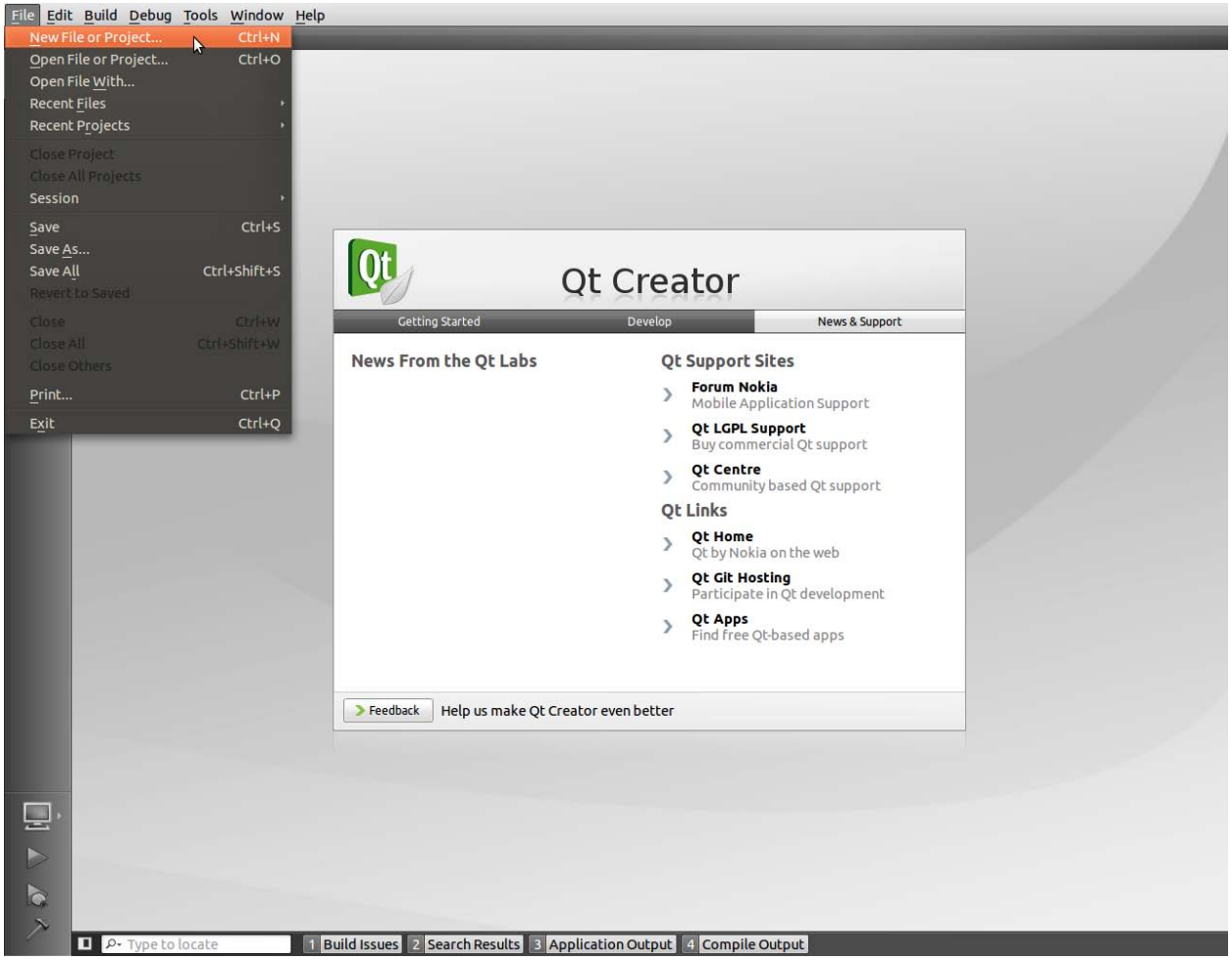
Qt Tutorial

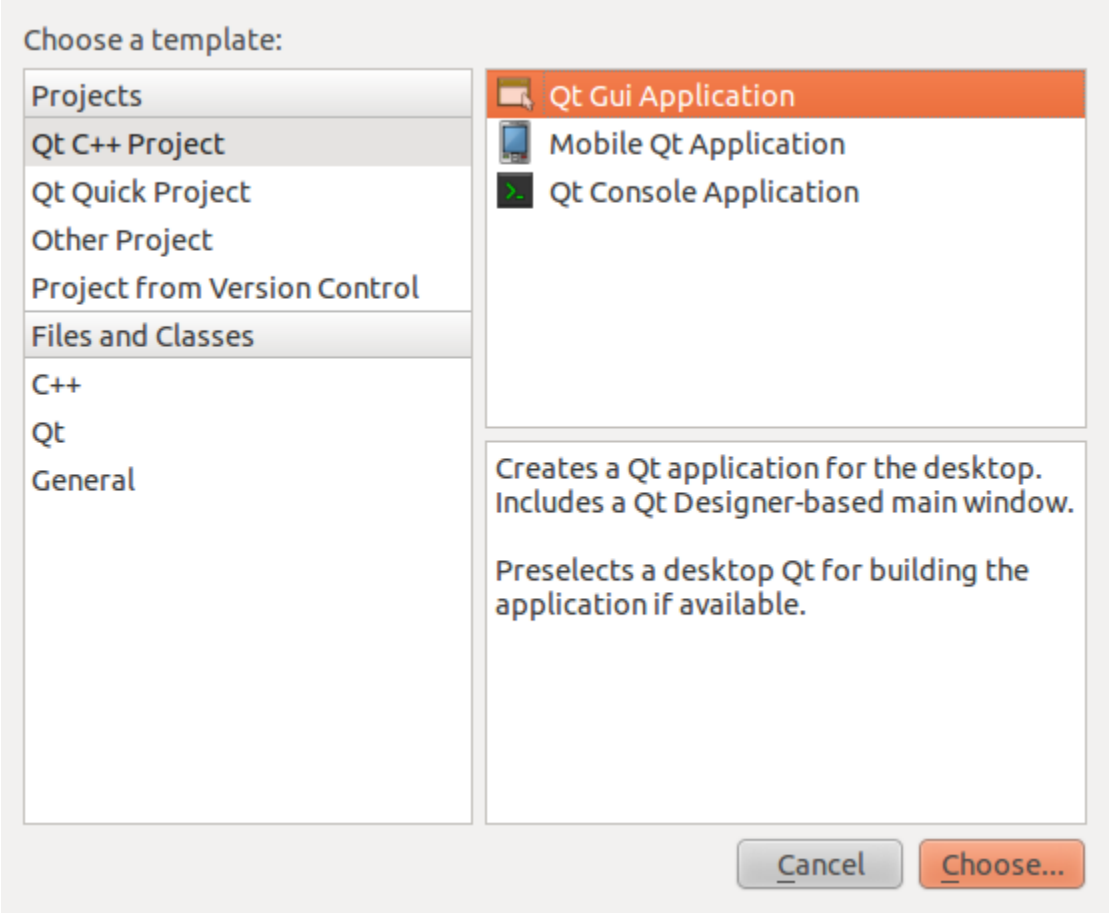
This tutorial is based on Qt Creator version 2.01 running on Ubuntu 10.10. Current versions of Qt Creator have many more features, but the basic operation should be about the same. You can install Qt Creator on your Ubuntu install by searching for “QtCreator” through the *Ubuntu Software Center* or the *Synaptic Package Manager*.



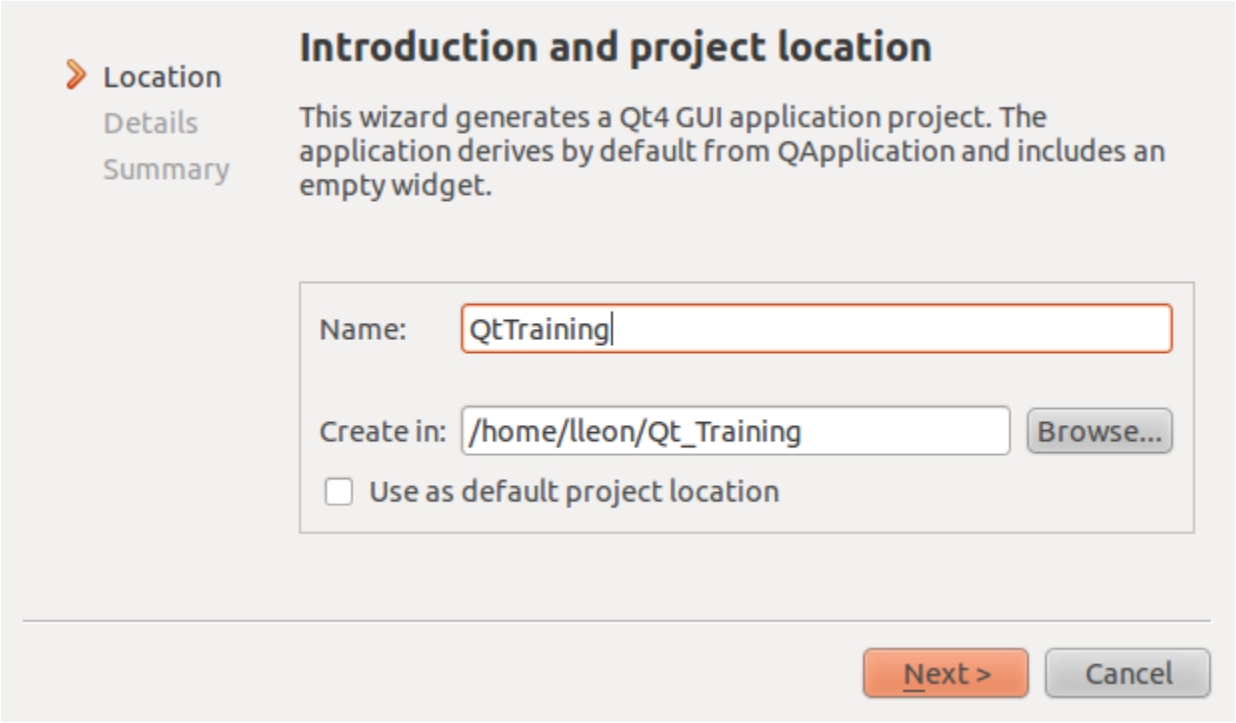
After you load Qt Creator, create a new project. It will load up a project manager wizard to step you through the options for creating a Qt project.



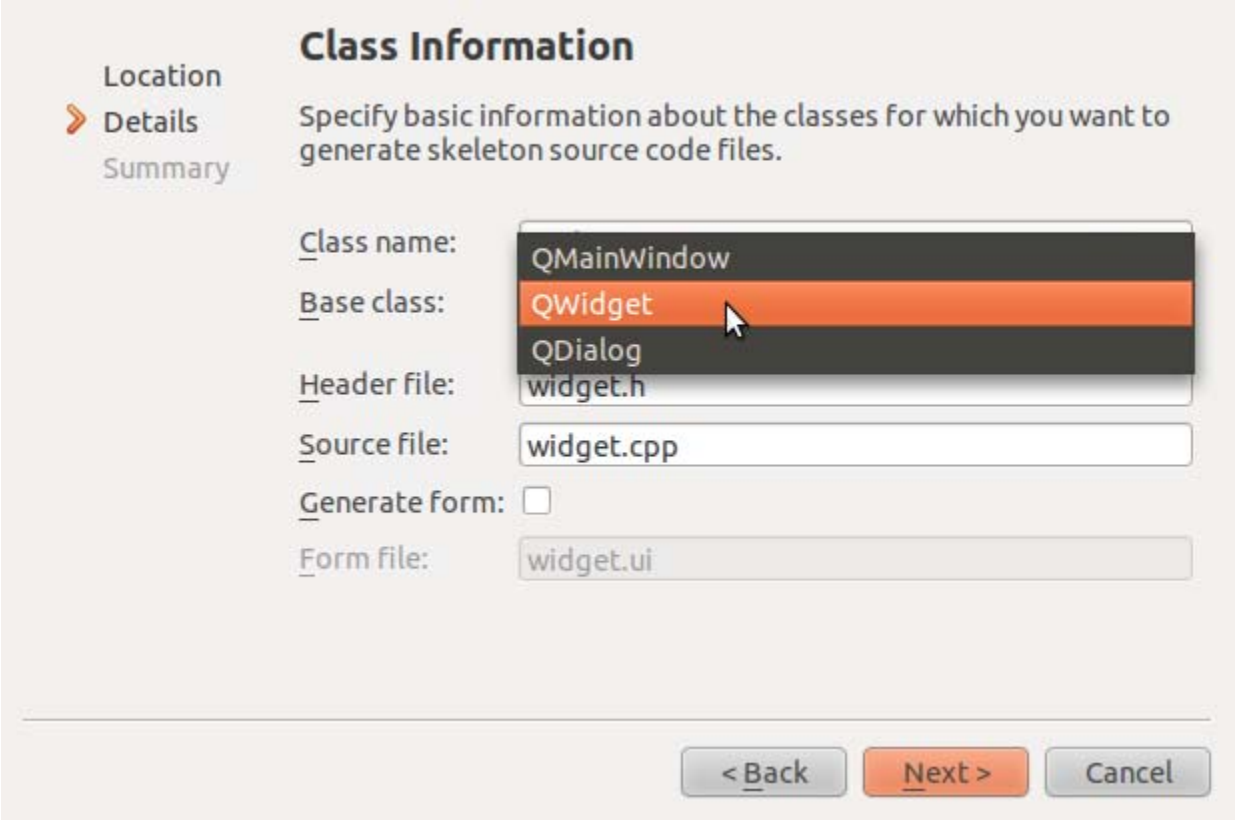
The following dialog appears. Select the project template for “Qt Gui Application”.



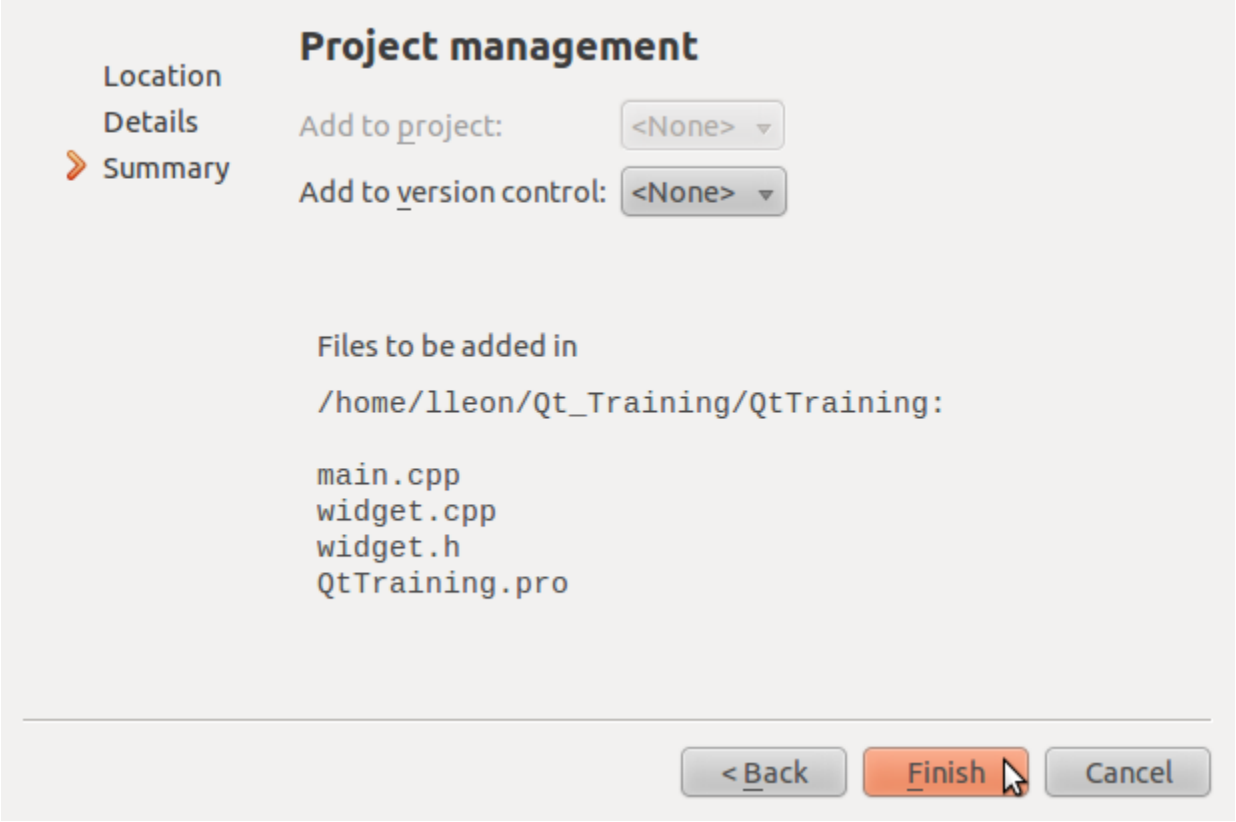
Enter information about your project. In this tutorial, the settings below will create the folder “QtTraining” in the “/home/lleon/Qt_Training” folder. Once you have entered your project information, select “next”.



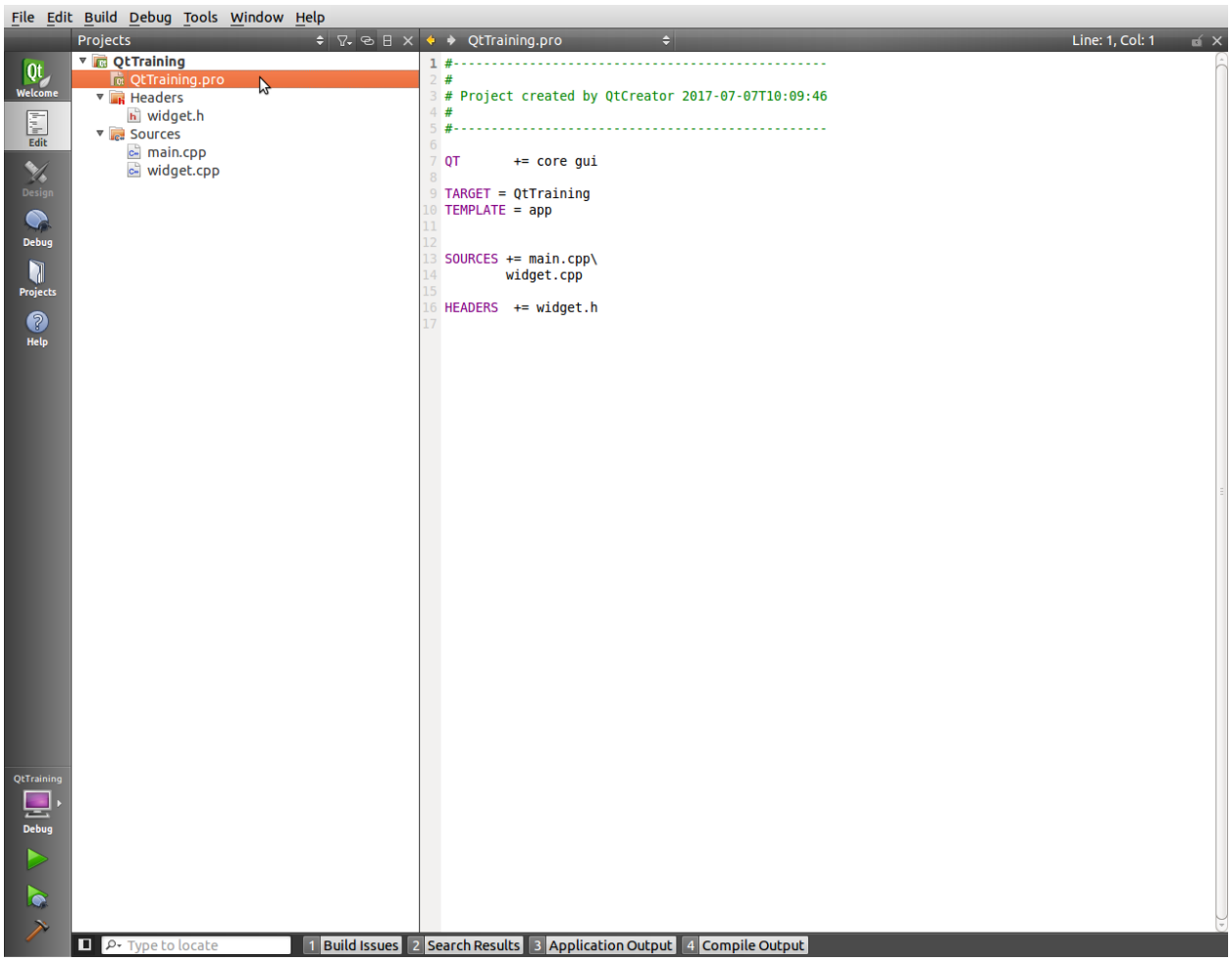
Select QWidget for the base class. Also, you will need to enter the name of the class. In this example, “Widget” is the name of the class (which you don’t see), and will be defined by “widget.h” and “widget.cpp”. Lastly, this tutorial assumes that GUI development will not be done through a Qt form, which can be generated through Qt Designer. Because of this, make sure the “Generate form:” is not selected. If you are into UI development, then you should consdier using Qt Designer. In my case, I did not care about the look and feel of the UI. I just wanted a basic UI with basic widgets (checkboxes, pushbuttons, etc.) that I can connect to specific pieces of code. Select “next” once you are done.



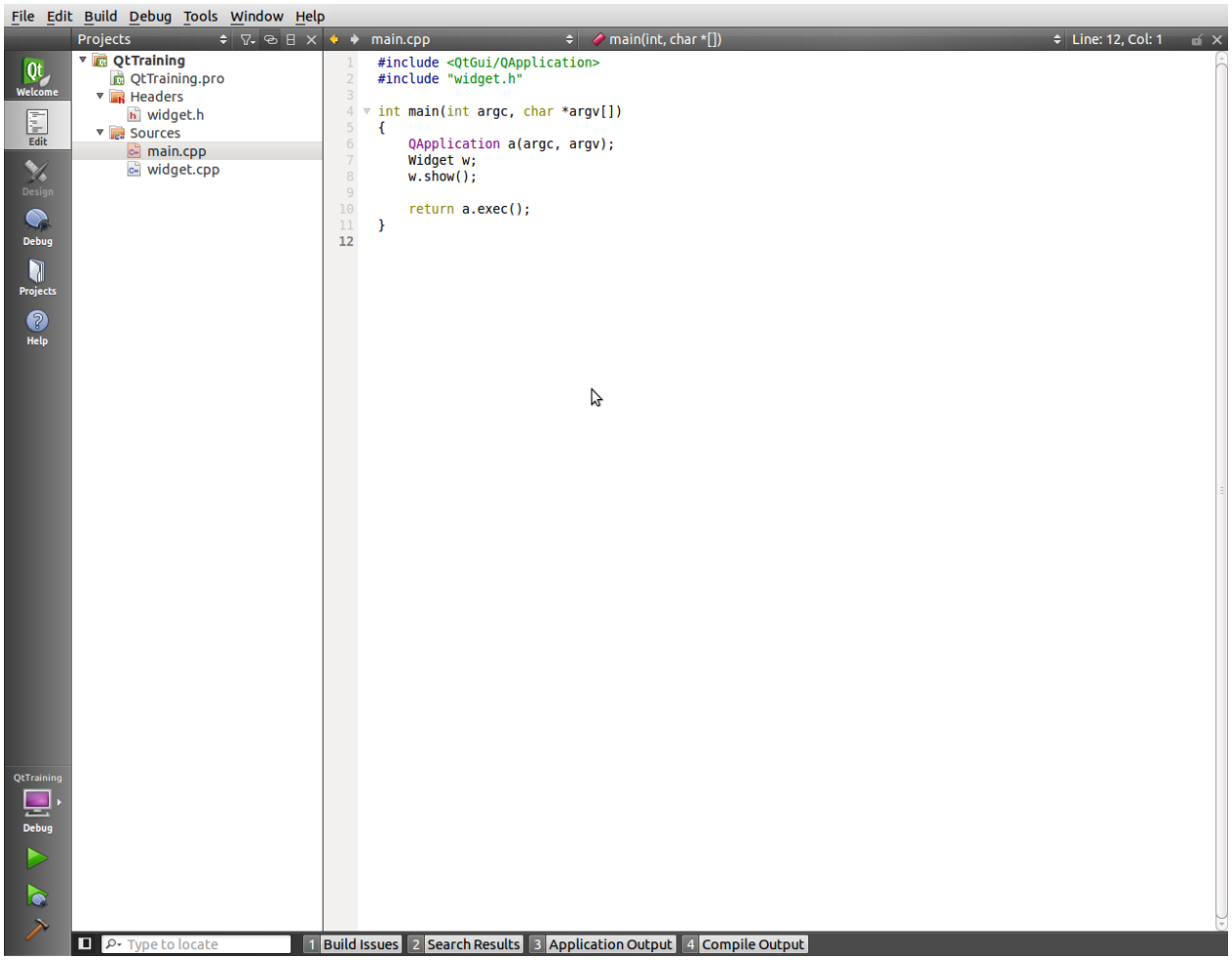
This is the final dialog of the project wizard. You can add version control management in this dialog. I highly recommend using some form of version control, but for the purpose of this tutorial, I will skip this. Select “Finish”.



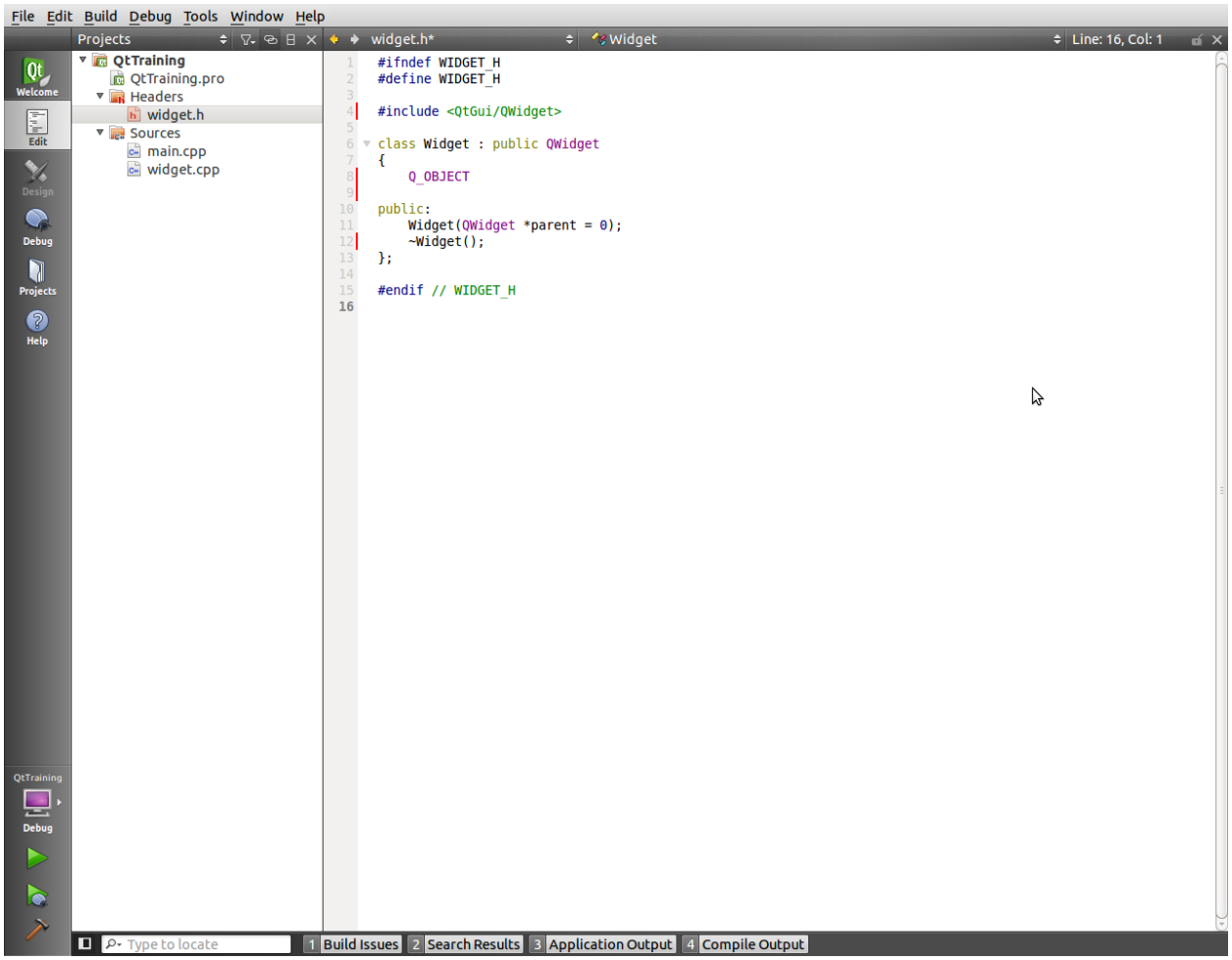
The following screen will appear. The project is completely defined by the *.pro file, which is shown below. Qt uses this *.pro file to determine which code is included for building the project. In addition, the “TARGET” tag indicates the name of the executable that will be compiled. In this example, the executable will be called “QtTraining”. You can manually add sources and headers to this *.pro. However, if files are added through Qt Creator, this file will be automatically be updated.



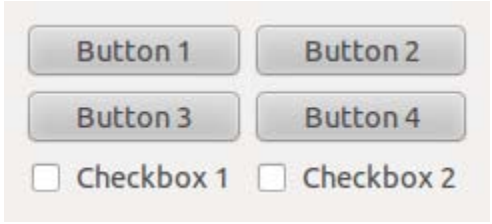
The wizard also creates a default “main.cpp” file. For now, don’t change it. This is what the auto generated file looks like.



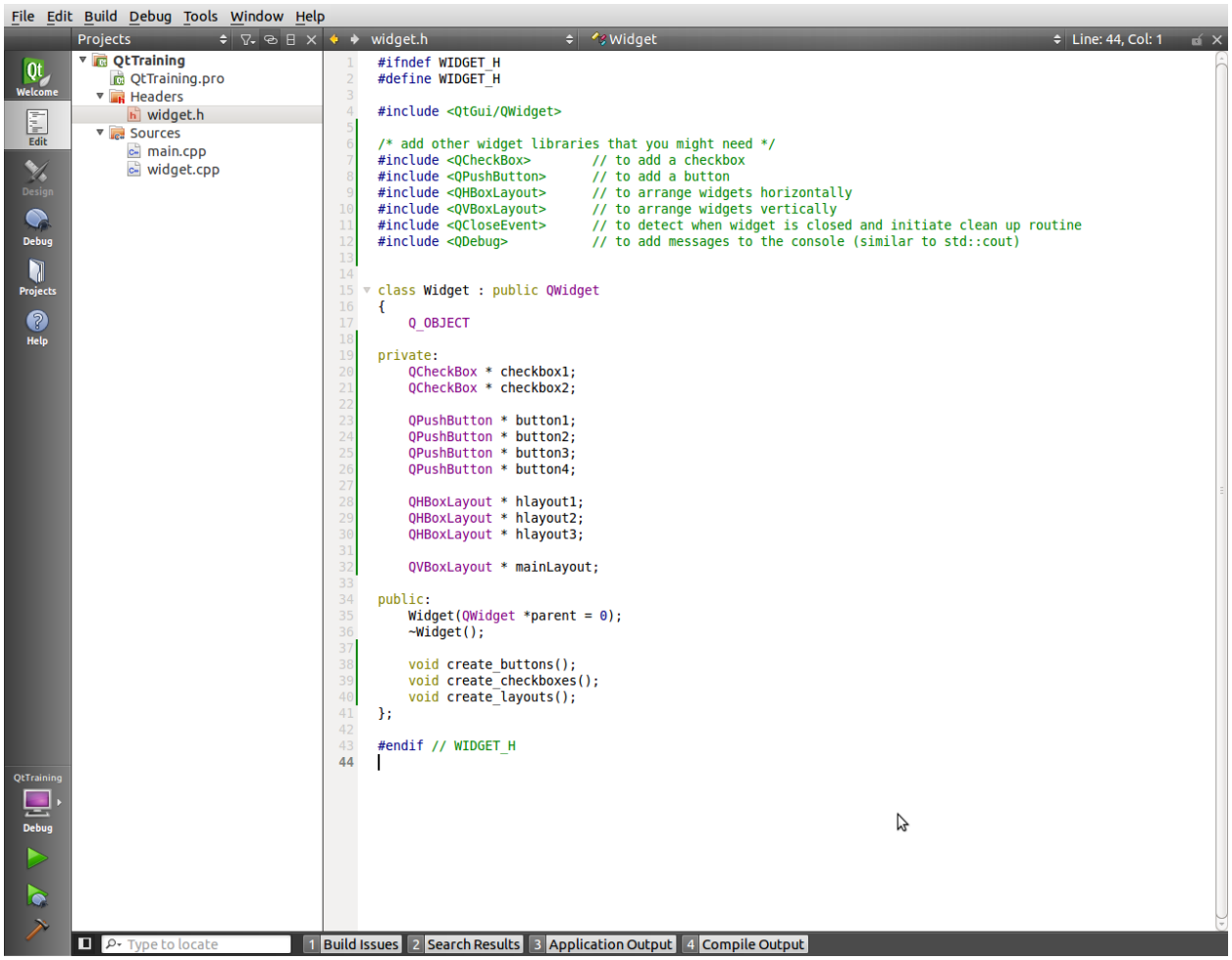
The wizard also creates a default header file titled “class.h”. In this case, since the class is named “Widget”, the header file that is auto-generated is “widget.h”. We will edit this file first.



The goal of this tutorial is to create a simple widget and connect the UI features to C++ code, all within a class structure. The widget will look like this. It has four buttons and two checkboxes. Prior to creating the widget, it is generally a good idea to determine how many UI features you want, and how you want them arranged. I decided to group pairs of buttons in a horizontal grid. Buttons 1 and 2 were grouped horizontally together. Buttons 3 and 4 were grouped horizontally together. Checkboxes 1 and 2 were grouped horizontally together. Then all three horizontal groupings were finally grouped in a vertical grid.

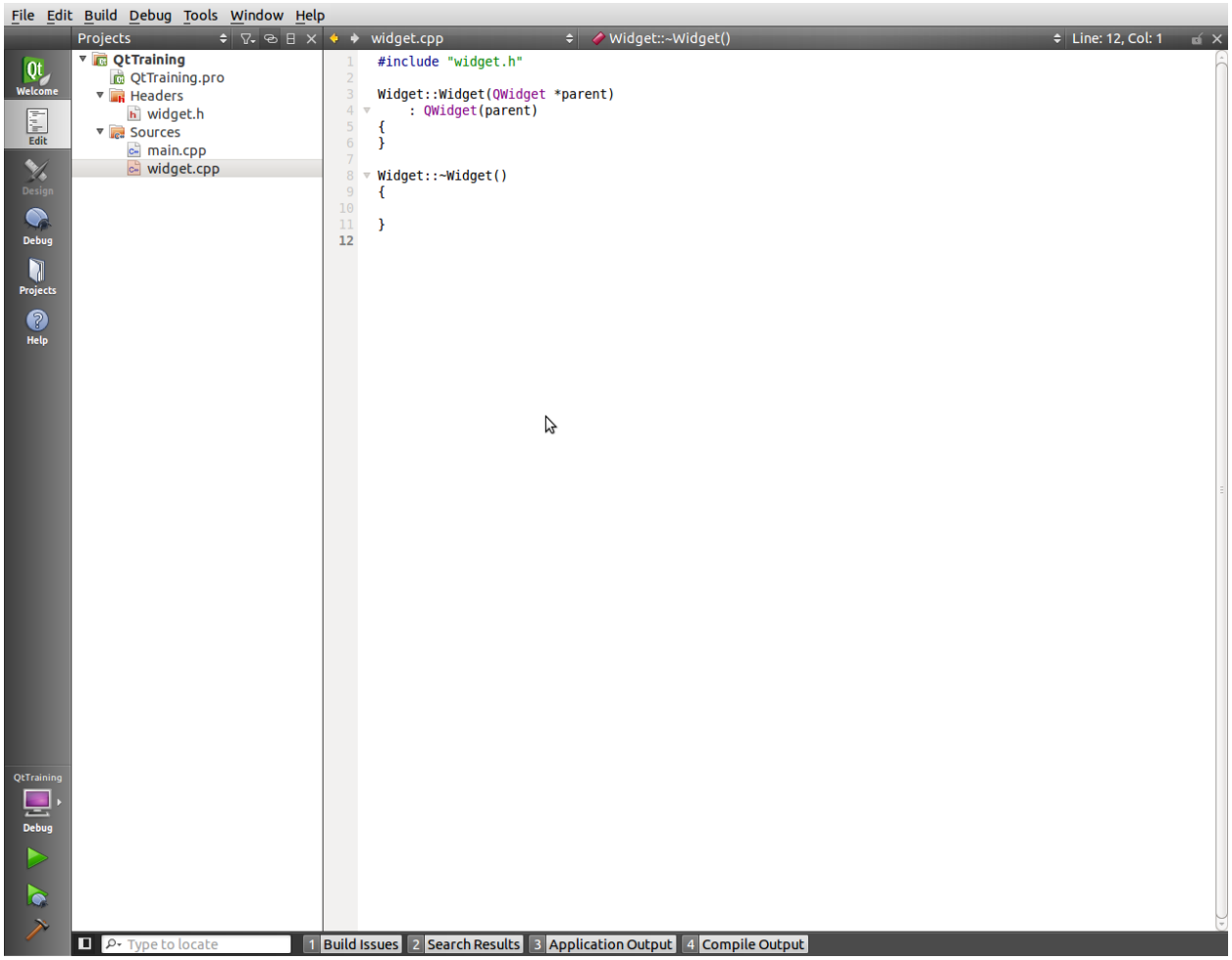


Based on this design, we will need four pushbuttons, 2 checkboxes, three horizontal layouts, and 1 vertical layout. First, you will need to modify the header file to include the libraries that define these UI features. This is shown by the “#include” statements near the top of the header file. Next, create pointers to member variables that are of the UI type. For example, I need two checkboxes. So I define one pointer to a QCheckBox type named “checkbox1”. Then I define a second pointer to a QCheckBox type named “checkbox2”. Do this for all the UI features you want in your widget. Finally, for organization, I create member functions where the code to create these UI features reside. So in this case, all the code I need to create my QCheckBox variables will be defined by the member function “void create_checkboxes()”



The code is provided in the file “10_Qt_header.h” which you can copy and paste into your Qt Creator session.

Next, edit the cpp file associated with the header file. So in this case, that file is “widget.cpp”. There should already be a “widget.cpp” file automatically generated by the project wizard. It is shown below.



Edit the cpp file so that there is code associated with the definitions found in the header file.

Let’s start with defining the code to create the buttons. Copy and paste the following code.

```
void Widget::create_buttons()
{
    this->button1 = new QPushButton();
    this->button1->setText("Button 1");
    this->button2 = new QPushButton();
    this->button2->setText("Button 2");
    this->button3 = new QPushButton();
```

```
    this->button3->setText("Button 3");
    this->button4 = new QPushButton();
    this->button4->setText("Button 4");
}
```

Essentially, two things are done for each pushbutton. An instance of QPushButton is assigned to the pointer using “new QPushButton()”, and a text string is assigned to the pushbutton.

Next, define the code to create the checkboxes. Copy and paste the following code.

```
void Widget::create_checkboxes()
{
    this->checkbox1 = new QCheckBox();
    this->checkbox1->setText("Checkbox 1");

    this->checkbox2 = new QCheckBox();
    this->checkbox2->setText("Checkbox 2");
}
```

Again, two things are done for each checkbox, similar to the pushbuttons. An instance of QCheckBox is assigned to the pointer, and a test string is assigned to the checkbox.

Next, define the layouts and add the UI features to the layouts. Layouts are an automated way to arrange the features in horizontal or vertical grids. The pattern is slightly different here than in the previous functions. First, the pointer is assigned to a new instance of the Layout. Then existing UI features that have already been defined (also called widgets) are added to the layout.

```
void Widget::create_layouts()
{
    // combine buttons 1 and 2 into a horizontal grid
    this->hlayout1 = new QHBoxLayout();
    this->hlayout1->addWidget(this->button1);
    this->hlayout1->addWidget(this->button2);

    // combine buttons 3 and 4 into a horizontal grid
    this->hlayout2 = new QHBoxLayout();
    this->hlayout2->addWidget(this->button3);
```

```
    this->hlayout2->addWidget(this->button4);

    // combine checkboxes 1 and 2 into a horizontal grid
    this->hlayout3 = new QHBoxLayout();
    this->hlayout3->addWidget(this->checkbox1);
    this->hlayout3->addWidget(this->checkbox2);

    // stack all horizontal layouts into one vertical layout
    this->mainLayout = new QVBoxLayout();
    this->mainLayout->addLayout(this->hlayout1);
    this->mainLayout->addLayout(this->hlayout2);
    this->mainLayout->addLayout(this->hlayout3);

    // set the main layout
    this->setLayout(this->mainLayout);
}
```

The final line sets the layout of the main, top-level widget, to the mainLayout which was defined as a vertical layout.

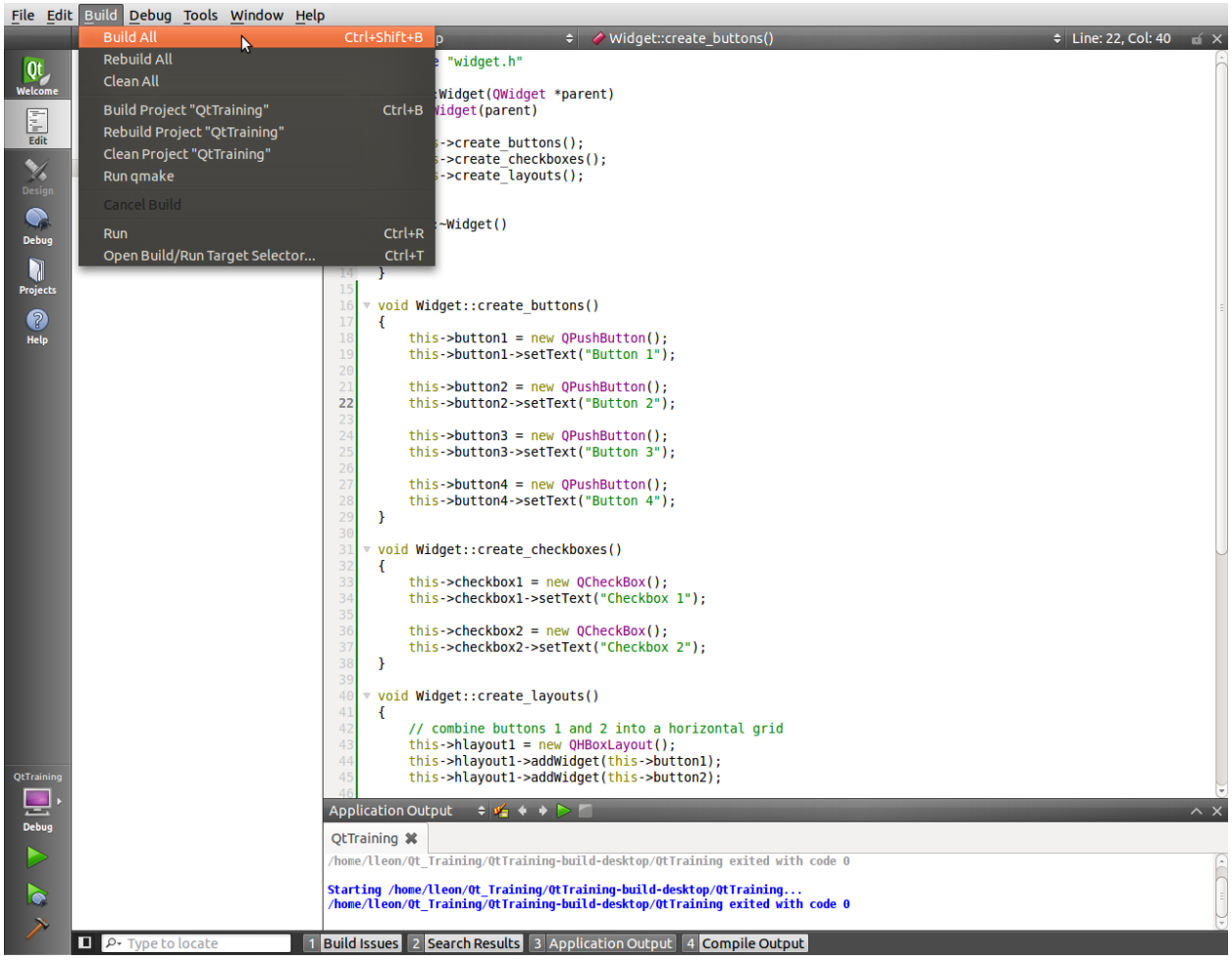
Finally, in the constructor to the class, these functions need to be called.

```
Widget::Widget(QWidget *parent)
    : QWidget(parent)
{
    this->create_buttons();
    this->create_checkboxes();
    this->create_layouts();
}
```

The order is important. Because the layouts assume UI features that have already been defined, they must exist. Otherwise, there will be an error during run time. So “create_buttons()” and “create_checkboxes()” must be called prior to calling “create_layout()”.

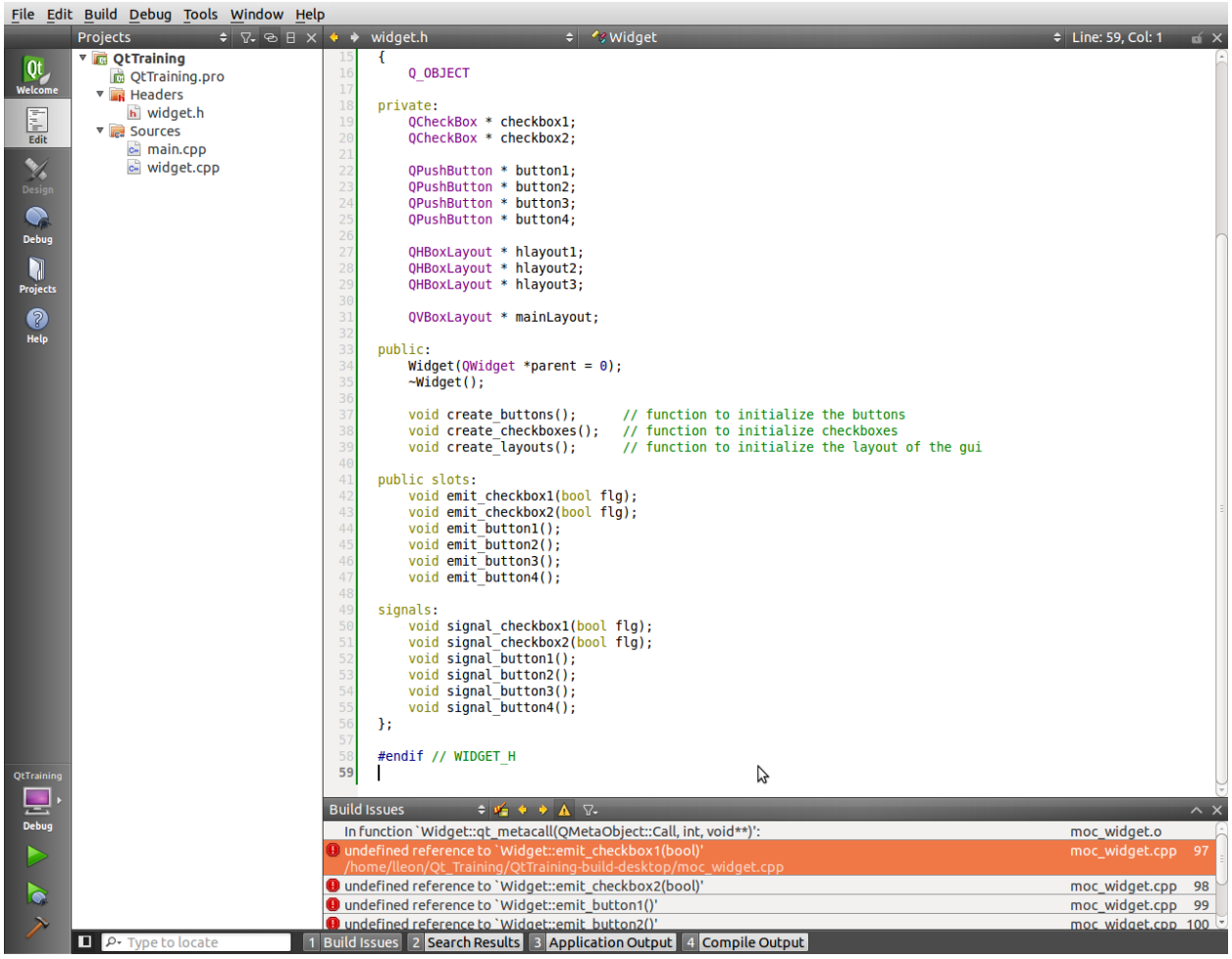
The code is provided in the file “12_Qt_cpp.cpp” if you want to copy from there.

Next, build the project and run the executable. You can also run the executable by pressing the play button on the lower left side of the task bar. If done correctly, you should see the widget shown on page 9.



Now that the widget has been designed, you will need to hook up the UI features to code. Qt's model for doing this is called signals and slots. Basically, the UI feature (i.e., widget) generates a "signal", and you must connect that signal to a function called a "slot".

Let's go back to the header file and include all the lines that define "public slots" and "signals". Make sure you include these lines before the final ending brace "}" that ends the class declaration. What these lines essentially do is prototype the function so that the compiler knows to look somewhere else for the actual code definition. That somewhere else is the *.cpp file associated with this header.



In this example, every time I click on the checkboxes, I also send a flag that indicates whether the checkbox is checked or not. This enables me to keep track of the checkbox mode. For the pushbuttons, I don't send anything that would indicate the state of the pushbutton.

Now, you can edit the *.cpp file to add the code for the “public slots”. So open up the “widget.cpp” file and copy and paste the following code.

```
/* code for slots */
void Widget::emit_checkbox1(bool flg)
{
    emit this->signal_checkbox1(flg);
    qDebug() << "checkbox 1 set to " << flg;
}
void Widget::emit_checkbox2(bool flg)
{
    emit this->signal_checkbox1(flg);
    qDebug() << "checkbox 2 set to " << flg;
}
void Widget::emit_button1()
{
    emit this->signal_button1();
    qDebug() << "button 1 clicked";
}
void Widget::emit_button2()
{
    emit this->signal_button2();
    qDebug() << "button 2 clicked";
}
void Widget::emit_button3()
{
    emit this->signal_button3();
    qDebug() << "button 3 clicked";
}
void Widget::emit_button4()
{
    emit this->signal_button4();
    qDebug() << "button 4 clicked";
}
```

You now have defined the slots. For good measure, I included console statements so that when these slots are called, you will see some output to the console. For example, you should see on the console “button 1 clicked” every time emit_button1() is called.

So we now have defined what to do (the slot) once the signal has been generated. The problem is that we have not connected the slot to the signal yet. This is what we will do next. Copy and paste the following to the “widget.cpp” file

```
/* connect the slots to the signals */
void Widget::connect_slots()
{
    /* connect the checkboxes to slots */
    connect(this->checkbox1, SIGNAL(clicked(bool)), this, SLOT(emit_checkbox1(bool)));
    connect(this->checkbox2, SIGNAL(clicked(bool)), this, SLOT(emit_checkbox2(bool)));

    /* connect the pushbuttons to the slots */
    connect(this->button1, SIGNAL(clicked()), this, SLOT(emit_button1()));
    connect(this->button2, SIGNAL(clicked()), this, SLOT(emit_button2()));
    connect(this->button3, SIGNAL(clicked()), this, SLOT(emit_button3()));
    connect(this->button4, SIGNAL(clicked()), this, SLOT(emit_button4()));
}
```

Notice that this function “connects” specific UI actions to specific slots. For example, clicking button1 will generate a signal that will call emit_button1(). This is how the signals get connected to the slots and is the unique feature, in my opinion, of Qt UI development. We will see the power of this later in this tutorial.

Lastly, add the line

```
this->connect_slots();
```

before immediately before the line `this->create_layouts();`

We want the connection to take place before the layouts are assigned.

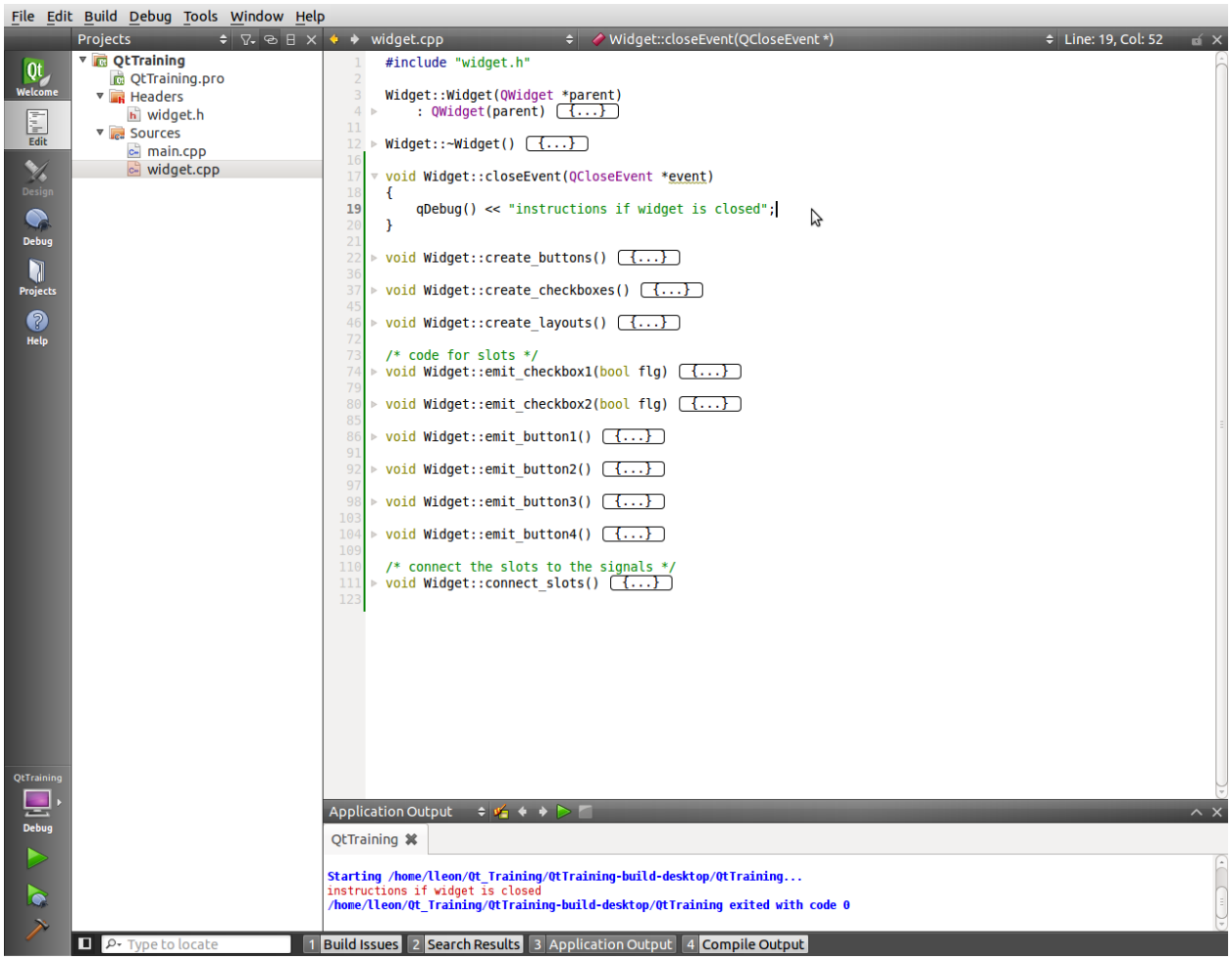
The full code for the cpp file is found in “16_Qt_signal_slots.cpp”.

We did forget one thing. The “connect_slots” function was never prototyped in the header file. This will cause a syntax error when attempting to build the project. So place the following line at the end of the private section, before the public section of the header file

```
void connect_slots();
```

If you build and run the project, you should see outputs on the console when the widget is used.

One thing that I recommend with Qt widgets is to use a function that detects if the widget is being closed. This allows you to do some clean up, if necessary, before the widget disappears completely. In this instance, if the widget is about the close, the function closeEvent(QCloseEvent *event) will be called, and the lines of code in that function will be executed. When running this widget, the statement "instructions if widget is closed" should appear in the console window when you close the widget.



Remember to prototype this function as well. Copy the following line

```
void closeEvent(QCloseEvent *event); // function is called when widget is closed
```

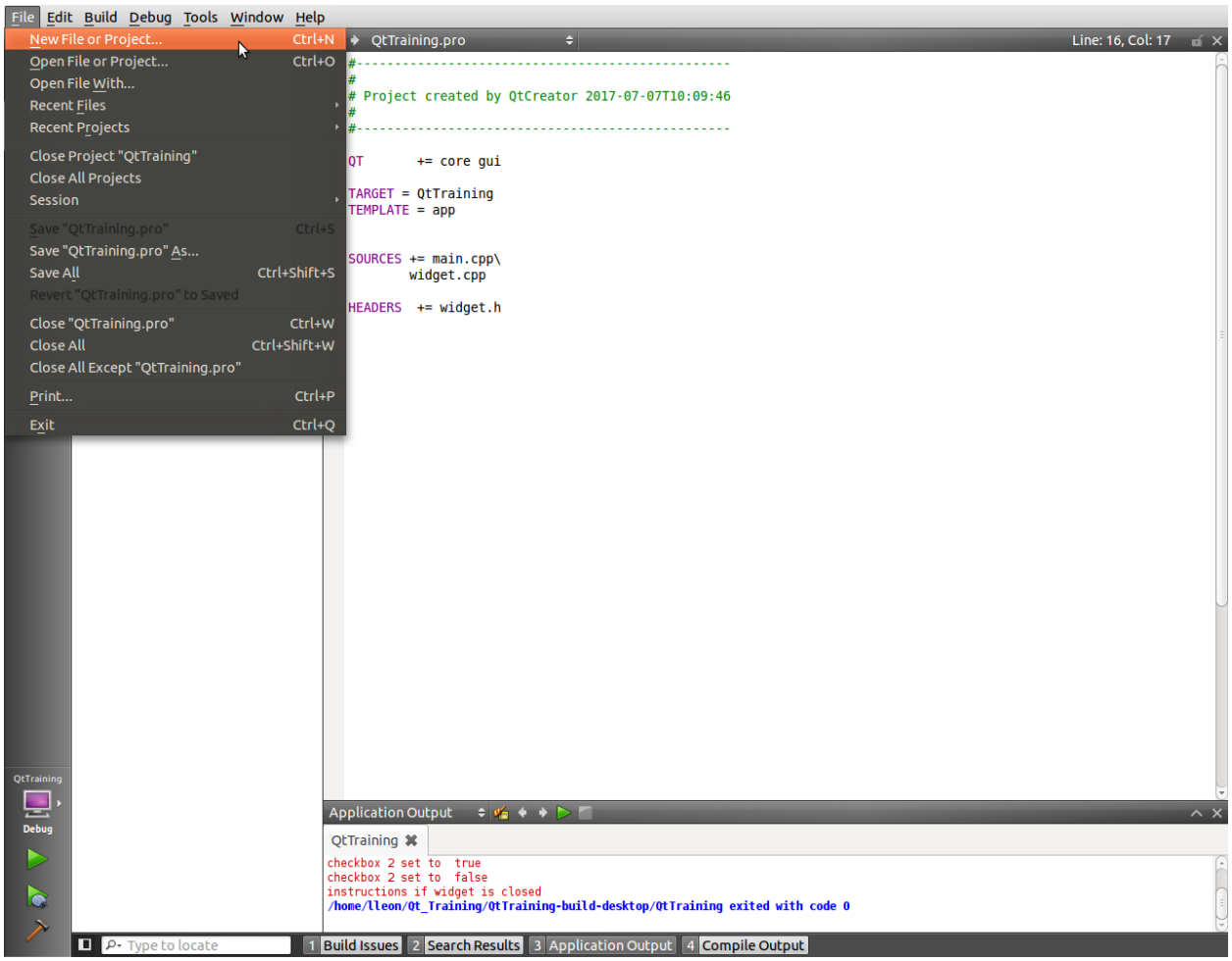
and place it after the default destructor ~Widget();

The full header file is found in "17_Qt_close_event.h" for your convenience.

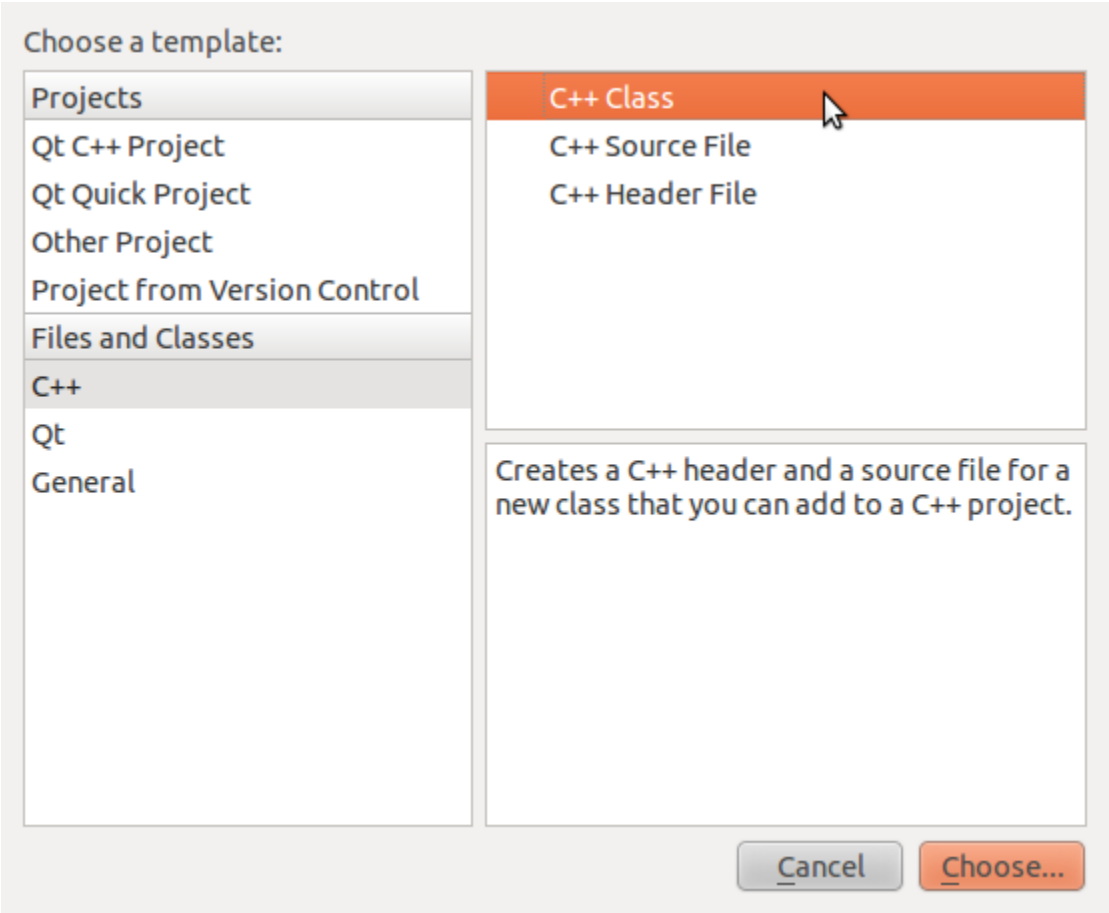
So far we've just created a widget that does stuff when you activate the UI features. In this final section, we will connect the signals generated by the widget to code that is defined in a different class altogether. The power of this feature is that the signal generated by the widget (and keep in mind the signal is generated by the UI actions since we connected them) can be connected to any slot from any class using Qt connect command (which we have already used).

One thing that is nice about Qt is that you can design the UI completely independent of any other code that you intend to connect the UI controls to. Then once you are satisfied with the look, feel, and functionality of the UI, you can connect it to any code you want to execute.

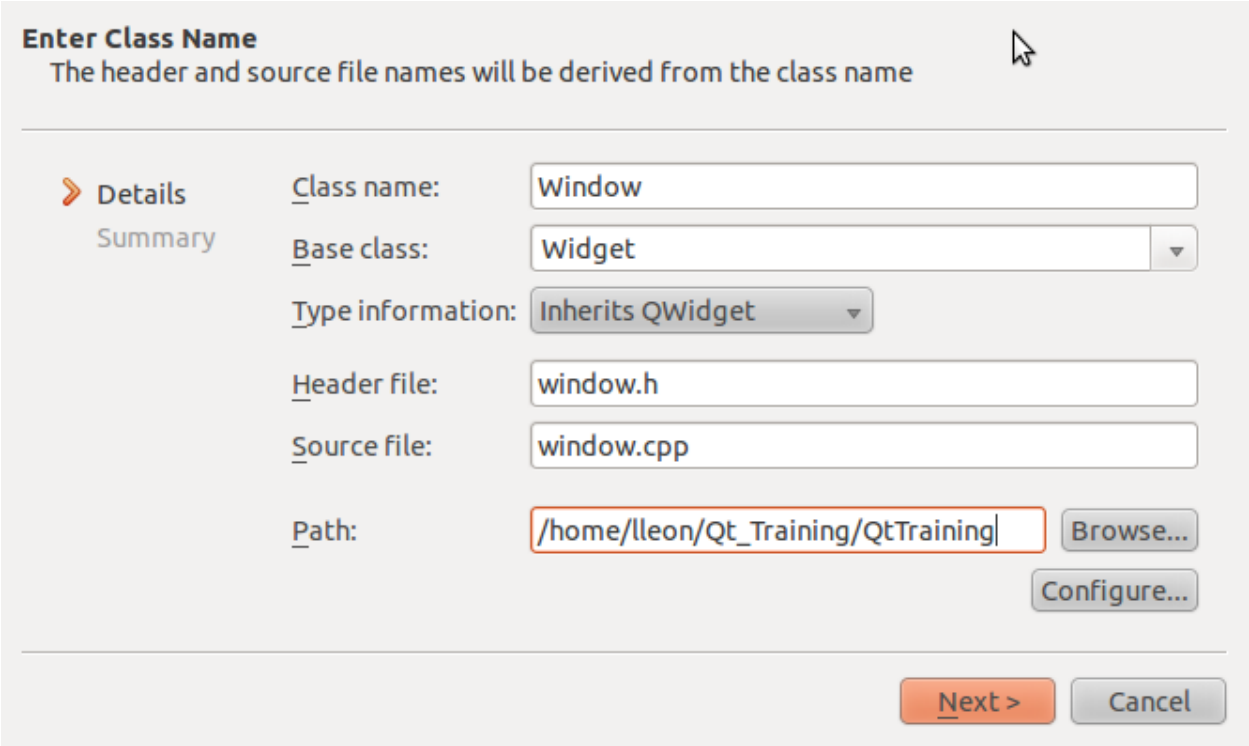
First, let's define a new class that inherits from the Widget class just created.



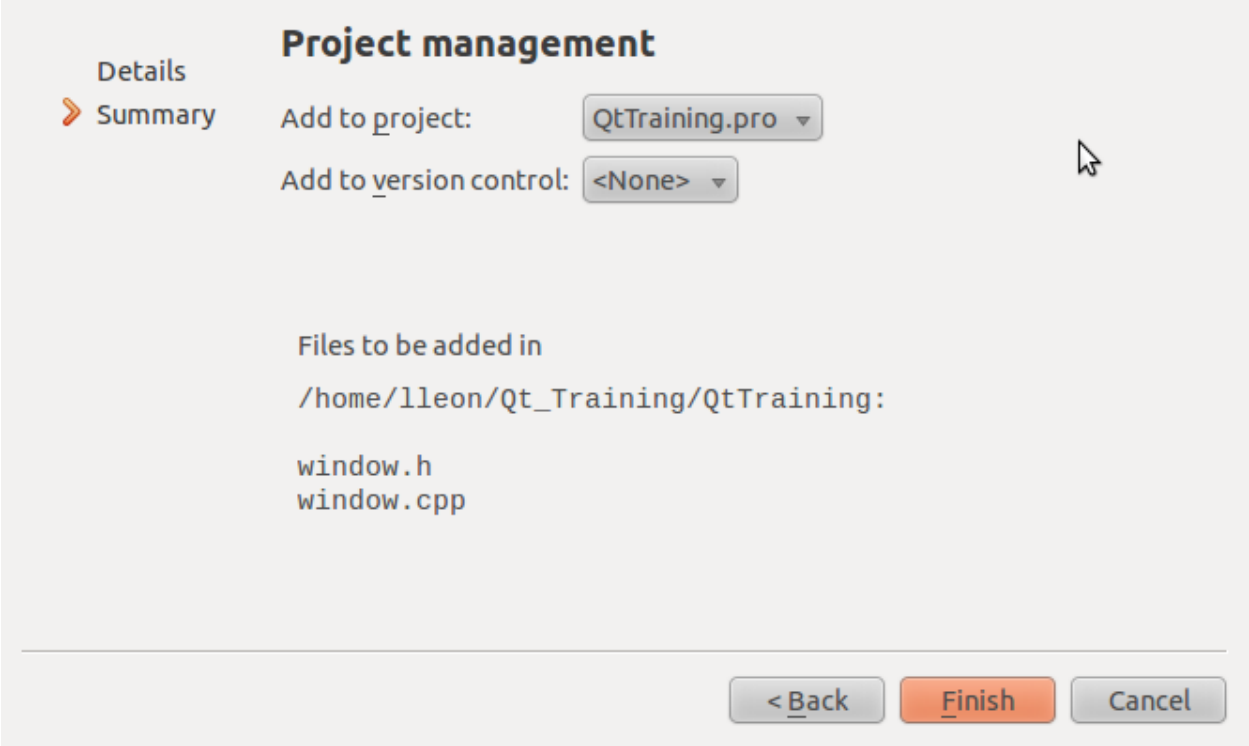
Select C++ Class. We are not creating a new project. We are simply creating a new class in the same project.



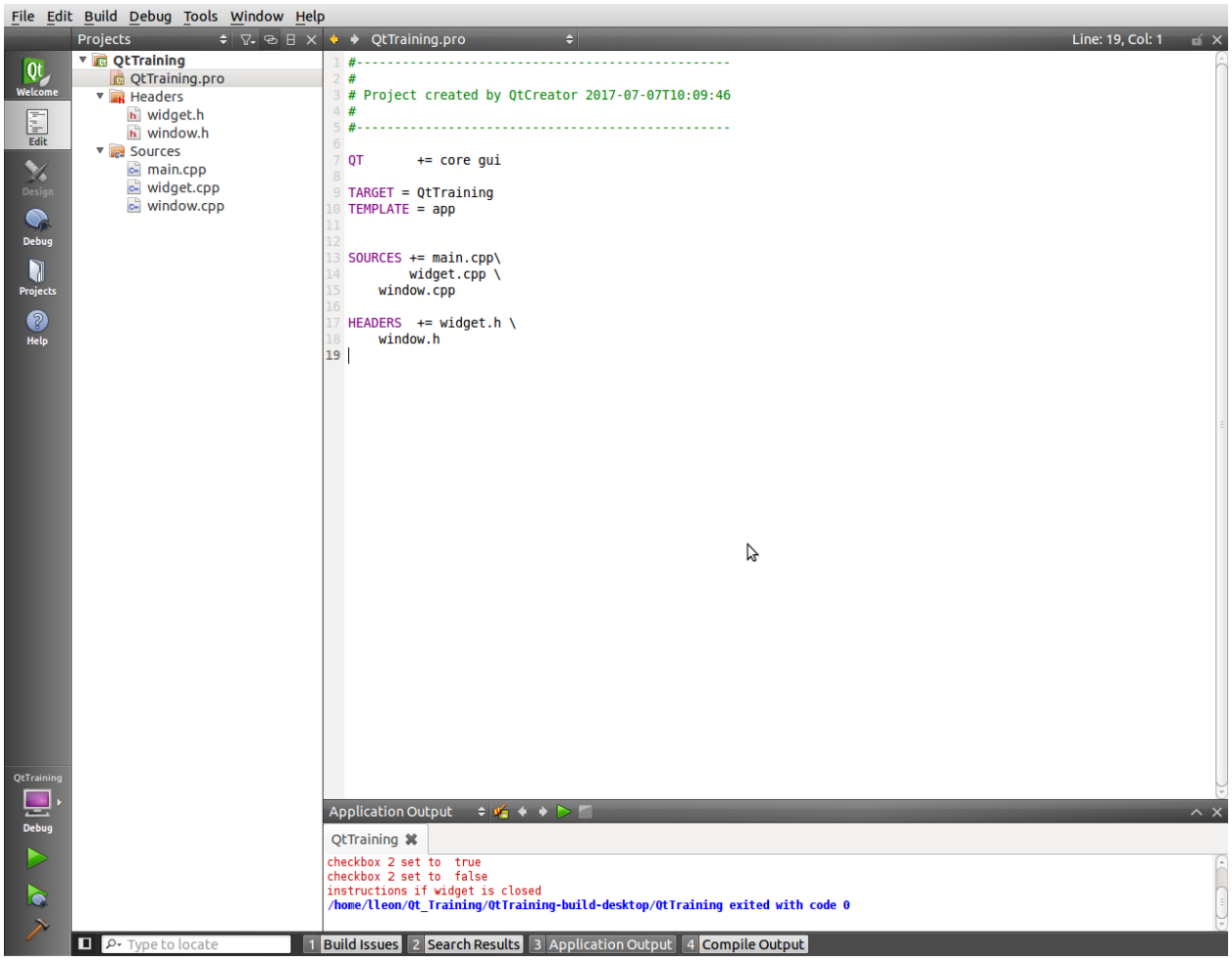
Next, define the properties of this new class. In this example, the new class is called “Window”, the base class is Widget (which we just defined), inheriting QWidget. These settings will automatically generate a new Window class as defined by “window.h” and “window.cpp”. Select “Next”.



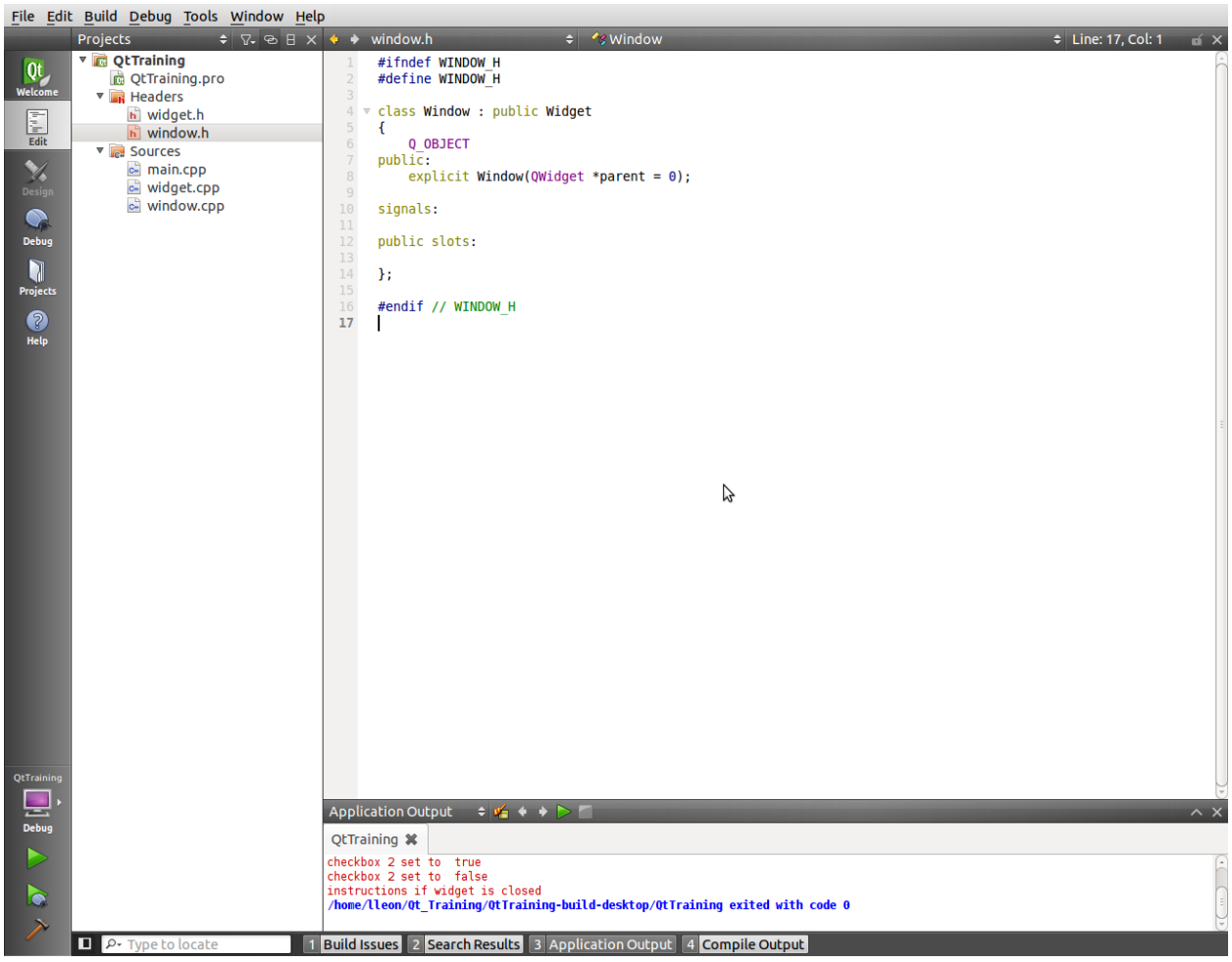
Click finish to complete the process.



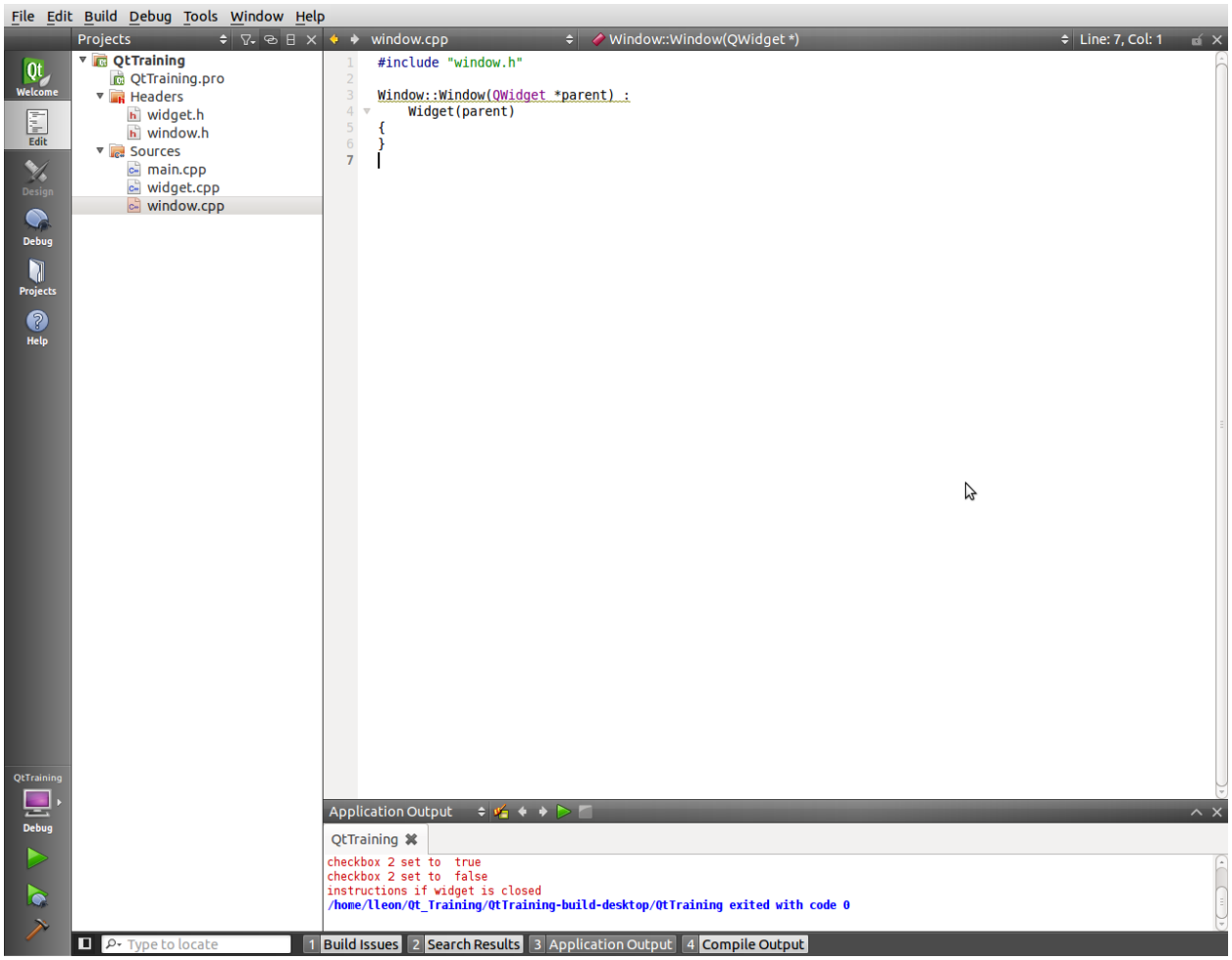
A new class will be added to the *.pro file as defined by the “window.h” and “window.cpp” files.



This is the auto-generated header file. Verify that the base class for Window is Widget, and not QWidget. Otherwise, the compiler will complain about this.



And this is the auto-generated *.cpp file

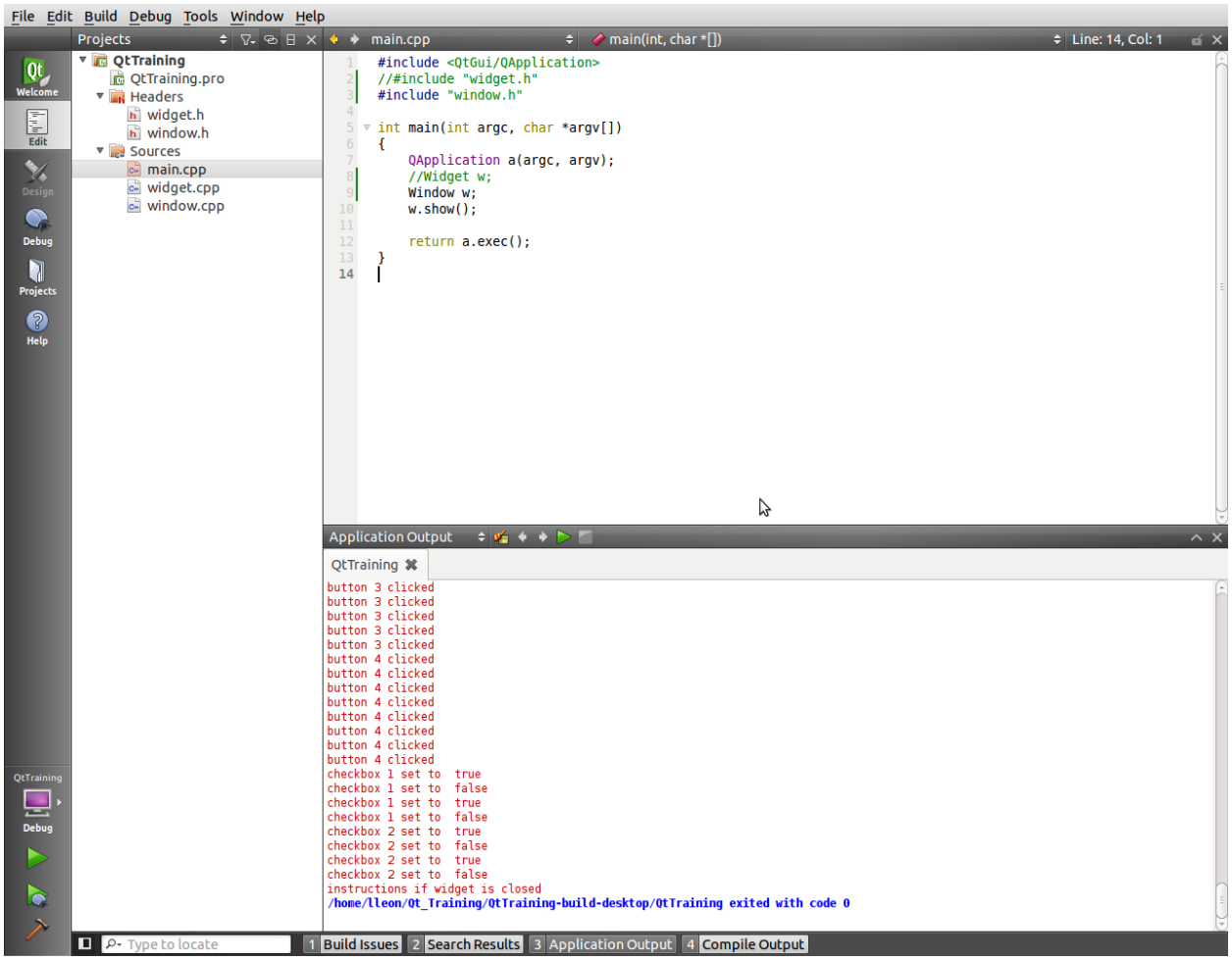


Both the *.cpp and the *.h file is empty. We will eventually add code to this.

But before this, we need to edit the main.cpp file since “Window” is the main code we want to execute. Since I defined “Window” to inherit from “widget”, and to automatically create an instance of “Widget” during construction, “Widget” will also be created when I create “Window”.

All you need to do is comment out “`Widget w;`” and put in “`Window w;`”

You will also need to include “window.h” in the include statement at the beginning. In the code below, I commented out `#include “widget.h”`, although you don’t have to.



Next, we want to connect the code from “Window” to the signals from “Widget”. This means defining “Window” slots for the “Widget” signals to be connected to. In other words, we will connect the signals from Widget to code from Window.

First, change “window.h” to the following

```
#ifndef WINDOW_H
#define WINDOW_H
#include "widget.h"
/* Window inherits from Widget, not QWidget. Therefore, make Widget the base class */
class Window : public Widget
{
    Q_OBJECT
public:
    explicit Window(QWidget *parent = 0);
    // add destructor
    ~Window();
    // connect signals from the widget to this class
    void connect_signals();
signals:
public slots:
    void slot_button1();
    void slot_button2();
    void slot_button3();
    void slot_button4();
    void slot_checkbox1(bool flg);
    void slot_checkbox2(bool flg);
};
#endif // WINDOW_H
```

The main changes from the auto-generated header file is: (1) included “widget.h”, (2) added a default destructor, (3) prototyped slot functions to connect the signals from Widget, (4) prototype function to do the connecting

For convenience, the header file is found in “27_Qt_new_class_connected.h”

Now, change “window.cpp” to the following. For convenience, the cpp file can be found in “28_Qt_New_Class_connected.cpp”

```
#include "window.h"
Window::Window(QWidget *parent) :
    Widget(parent)
{
    this->connect_signals();
}
Window::~Window()
{
}
void Window::connect_signals()
{
    connect(this, SIGNAL(signal_button1()), this, SLOT(slot_button1()));
    connect(this, SIGNAL(signal_button2()), this, SLOT(slot_button2()));
    connect(this, SIGNAL(signal_button3()), this, SLOT(slot_button3()));
    connect(this, SIGNAL(signal_button4()), this, SLOT(slot_button4()));
    connect(this, SIGNAL(signal_checkbox1(bool)), this, SLOT(slot_checkbox1(bool)));
    connect(this, SIGNAL(signal_checkbox2(bool)), this, SLOT(slot_checkbox2(bool)));
}
void Window::slot_button1()
{
    qDebug() << "slot_button1()";
}
void Window::slot_button2()
{
    qDebug() << "slot_button2()";
}
void Window::slot_button3()
{
    qDebug() << "slot_button3()";
}
void Window::slot_button4()
{
```

```
    qDebug() << "slot_button4()";
}
void Window::slot_checkbox1(bool flg)
{
    qDebug() << "slot_checkbox1() with " << flg;
}
void Window::slot_checkbox2(bool flg)
{
    qDebug() << "slot_checkbox2() with " << flg;
}
```

Notice that the only thing this class does (i.e., the Window class) is connect the signals from the Widget to the slots of the Window. While this is clearly a trivial case, perhaps button1 from the Widget will be used to perform some computations. You can change the slot to perform these computations. Now imagine you have a different purpose for button1, perhaps in a different application (say computer vision tasks). Without changing the widget at all (that is, keeping the same UI), you can connect the signal from button1, to a different function from a different class.