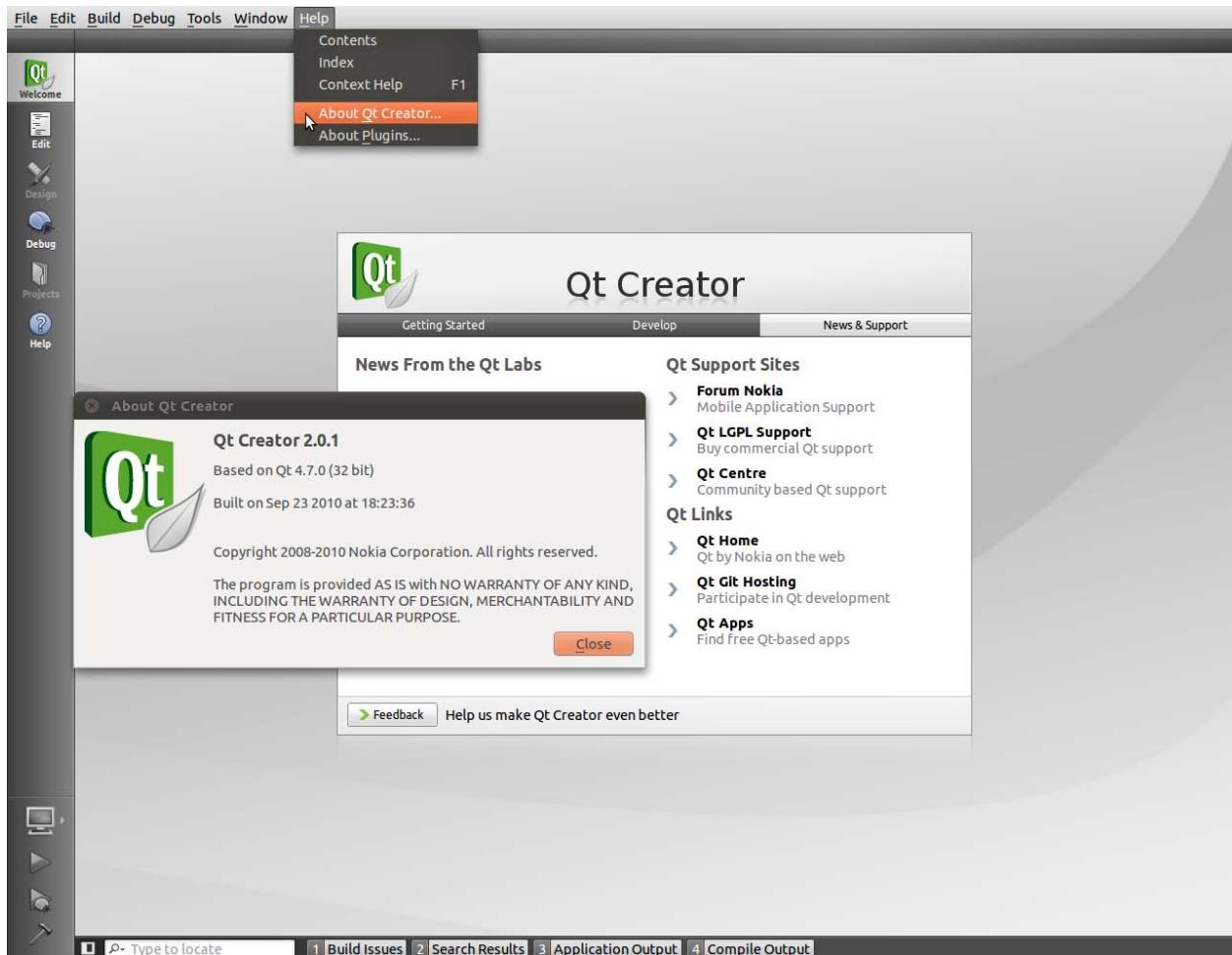
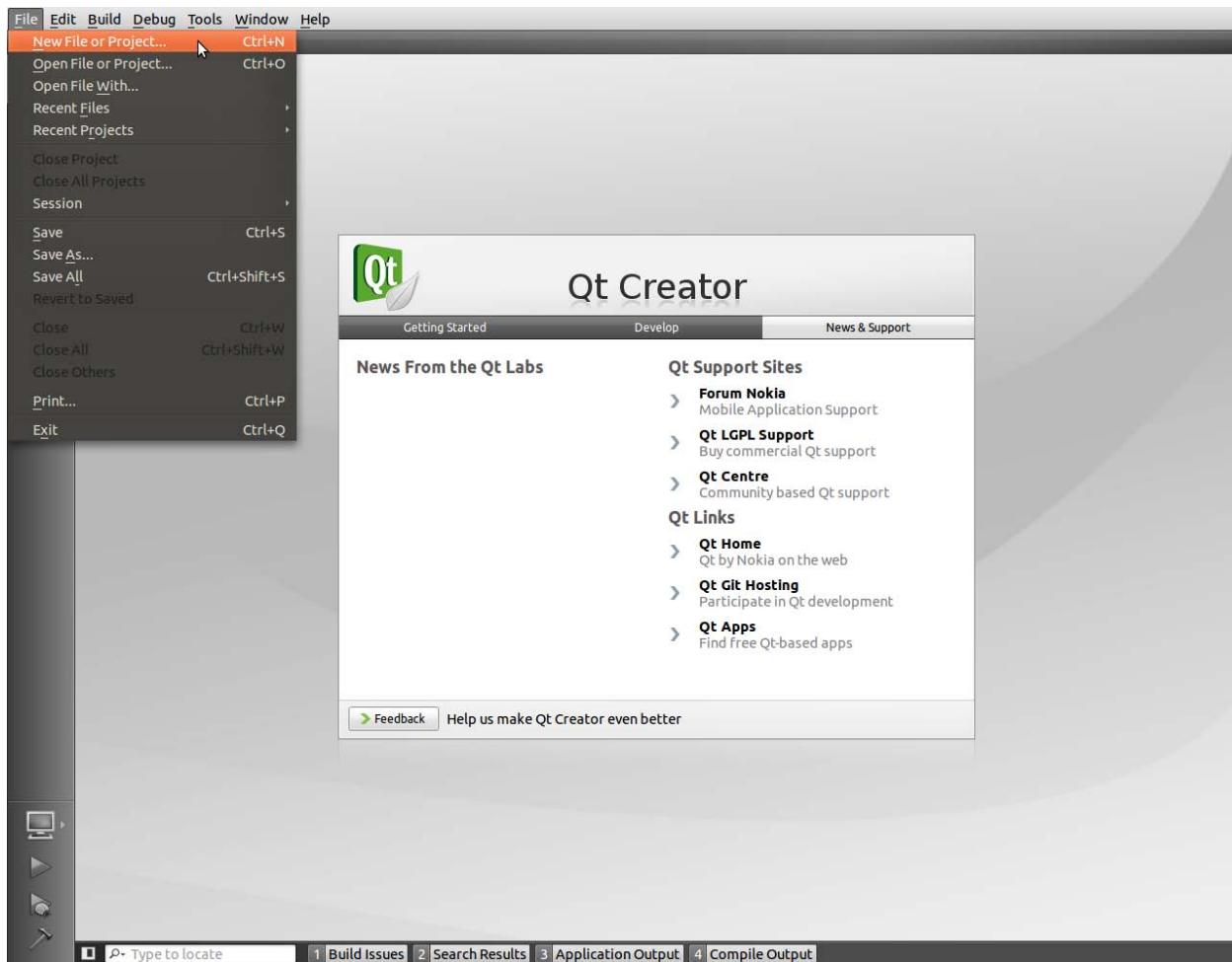


## Qt Tutorial

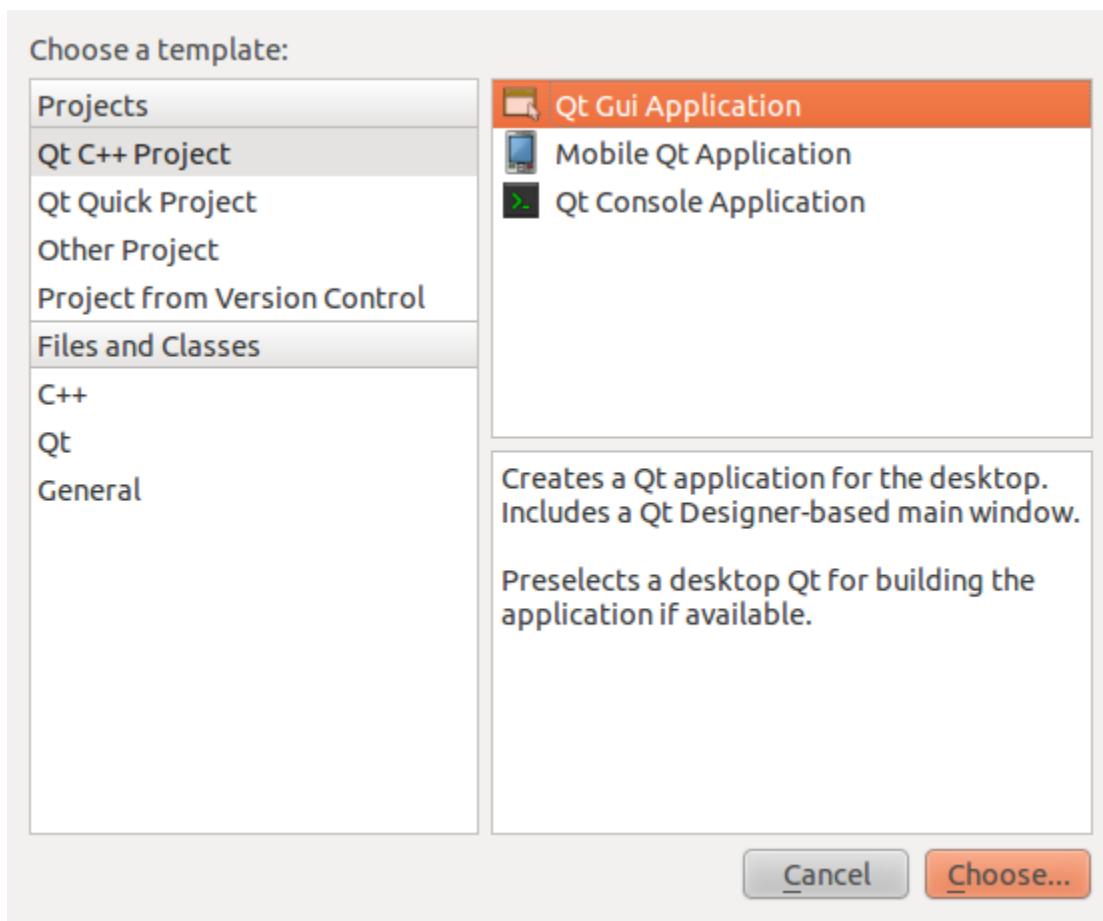
This tutorial is based on Qt Creator version 2.01 running on Ubuntu 10.10. Current versions of Qt Creator have many more features, but the basic operation should be about the same. You can install Qt Creator on your Ubuntu install by searching for “QtCreator” through the *Ubuntu Software Center* or the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



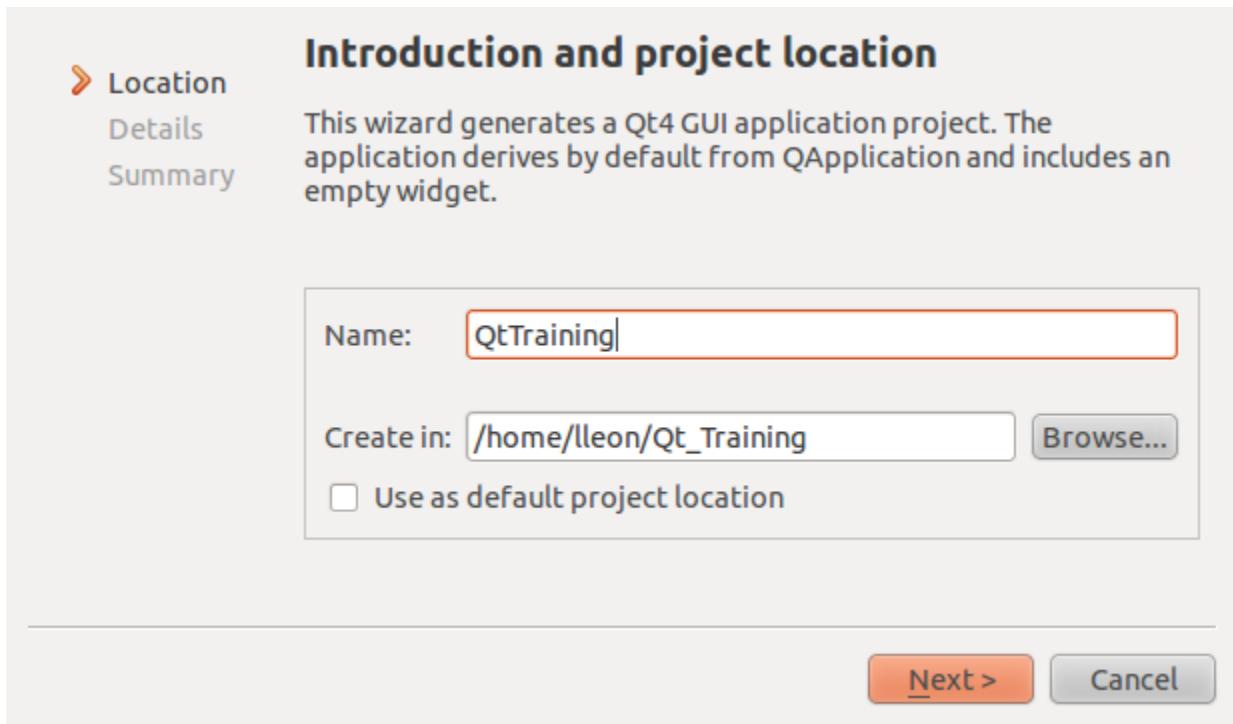
After you load Qt Creator, create a new project. It will load up a project manager wizard to step you through the options for creating a Qt project.



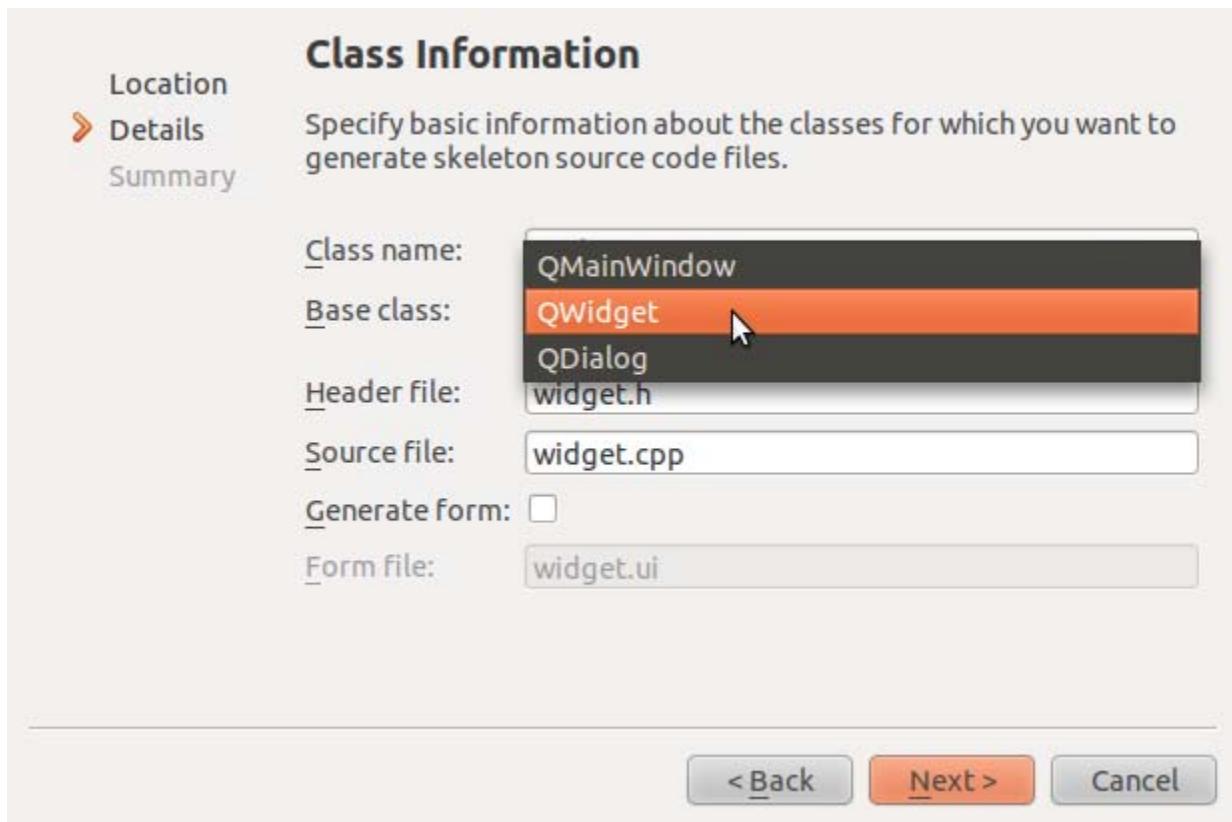
The following dialog appears. Select the project template for “Qt Gui Application”.



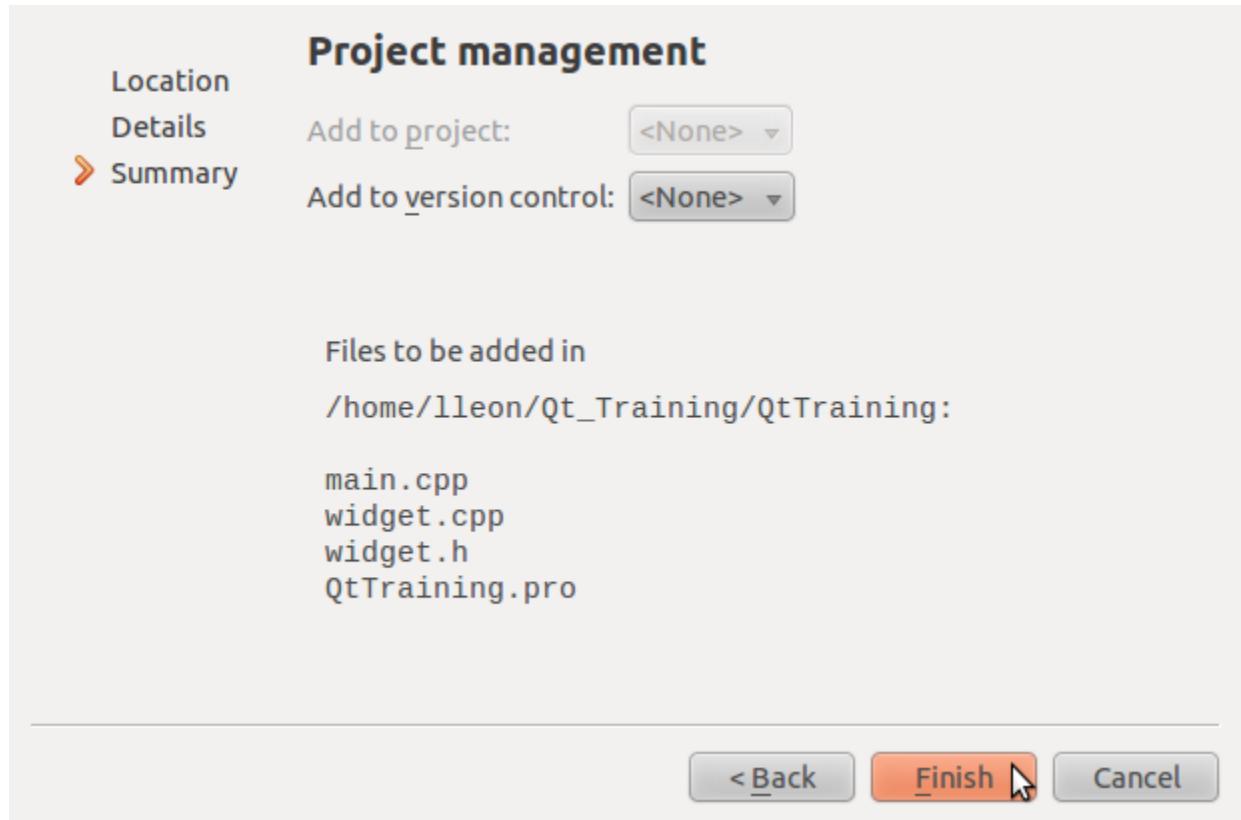
Enter information about your project. In this tutorial, the settings below will create the folder "QtTraining" in the "/home/lleon/Qt\_Training" folder. Once you have entered your project information, select "next".



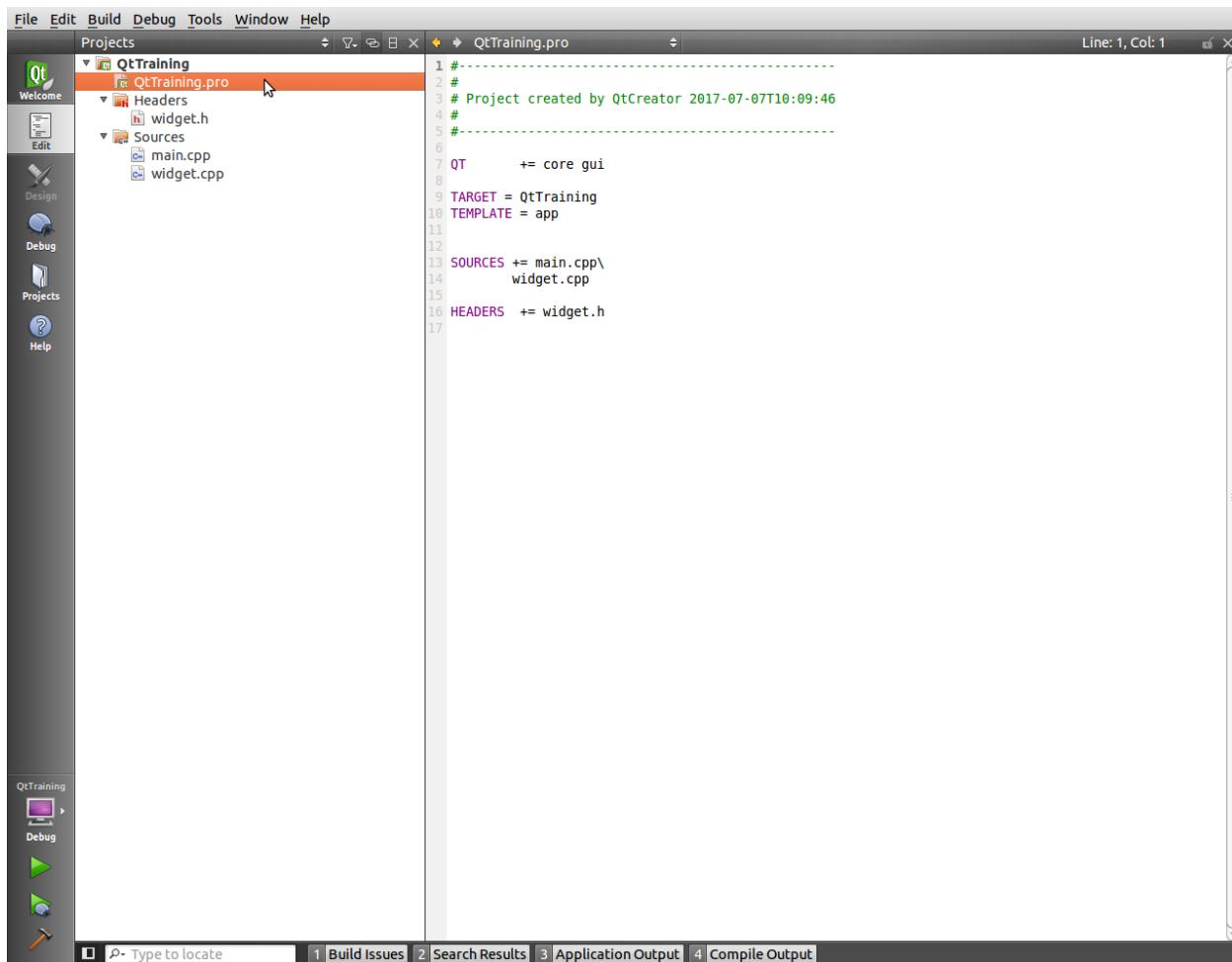
Select QWidget for the base class. Also, you will need to enter the name of the class. In this example, “Widget” is the name of the class (which you don’t see), and will be defined by “widget.h” and “widget.cpp”. Lastly, this tutorial assumes that GUI development will not be done through a Qt form, which 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 consider 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.

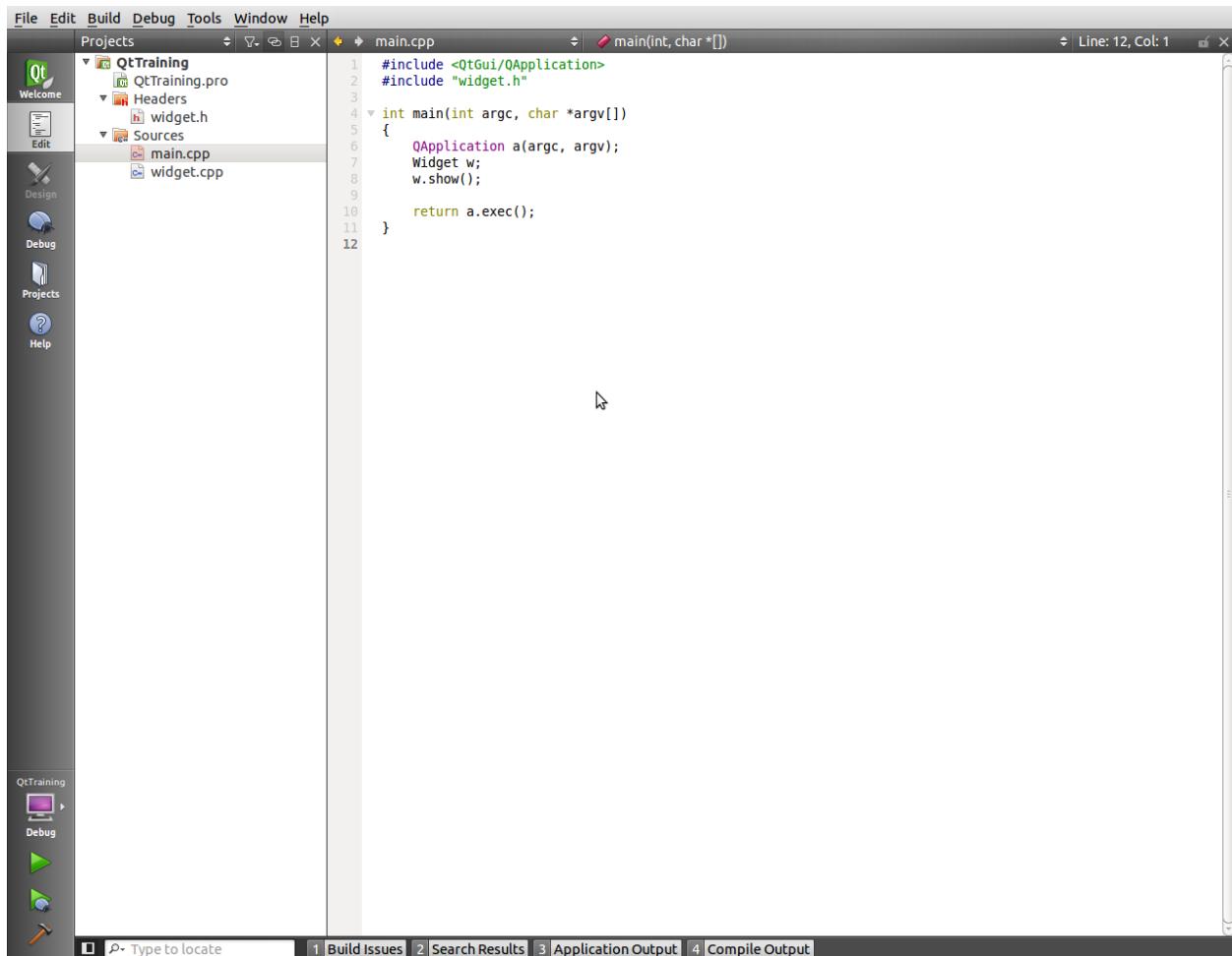


The screenshot shows the Qt Creator interface with the following details:

- File Menu:** File, Edit, Build, Debug, Tools, Window, Help
- Projects View:** Shows the "QtTraining" project with its structure:
  - QtTraining.pro (selected)
  - Headers: widget.h
  - Sources: main.cpp, widget.cpp
- Code Editor:** Displays the contents of the QtTraining.pro file:

```
1 #-----  
2 #  
3 # Project created by QtCreator 2017-07-07T10:09:46  
4 #  
5 #-----  
6 QT      += core gui  
7 TARGET  = QtTraining  
8 TEMPLATE = app  
9  
10 SOURCES += main.cpp  
11           widget.cpp  
12  
13 HEADERS  += widget.h
```
- Bottom Bar:** Type to locate, Build Issues, Search Results, Application Output, Compile Output

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 screenshot shows the Qt Creator IDE interface. The top menu bar includes File, Edit, Build, Debug, Tools, Window, and Help. The Projects tab is selected, displaying a project named "QtTraining". Under "Sources", the "main.cpp" file is selected, showing its code. The code is as follows:

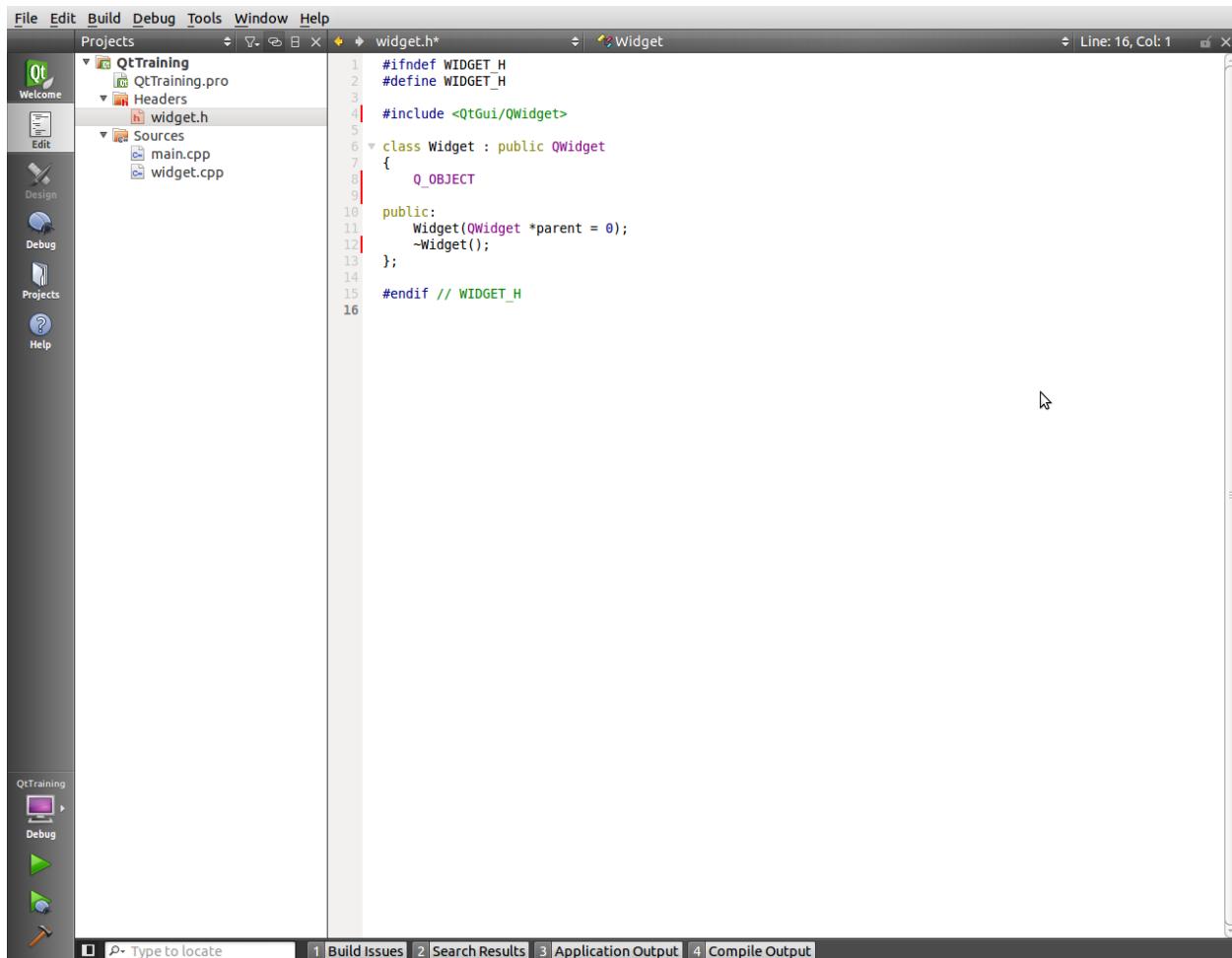
```
#include <QtGui/QApplication>
#include "widget.h"

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    Widget w;
    w.show();

    return a.exec();
}
```

The bottom status bar shows "Line: 12, Col: 1". The footer bar includes icons for Welcome, Edit, Design, Debug, Projects, and Help, along with tabs for QtTraining, Debug, Run, and Stop. The bottom navigation bar has tabs for Type to locate, Build Issues, Search Results, Application Output, and Compile Output.

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 screenshot shows the Qt Creator IDE interface. The top menu bar includes File, Edit, Build, Debug, Tools, Window, and Help. The title bar displays "Projects" and "widget.h\*". The left sidebar has icons for Welcome, Edit, Design, Debug, Projects, and Help. The main area shows a project tree for "QTTraining" with "QTTraining.pro", "Headers" (containing "widget.h"), and "Sources" (containing "main.cpp" and "widget.cpp"). The code editor on the right contains the following C++ code:

```
#ifndef WIDGET_H
#define WIDGET_H

#include <QtGui/QWidget>

class Widget : public QWidget
{
    Q_OBJECT
public:
    Widget(QWidget *parent = 0);
    ~Widget();
};

#endif // WIDGET_H
```

The status bar at the bottom shows "Type to locate" and tabs for Build Issues, Search Results, Application Output, and Compile Output. A cursor is visible in the code editor area.

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 screenshot shows the Qt Creator IDE interface. The top menu bar includes File, Edit, Build, Debug, Tools, Window, and Help. The left sidebar has sections for Qt Welcome, Projects, Design, Debug, and Help. The main area displays the project structure under 'QtTraining' with files like QtTraining.pro, Headers/widget.h, Sources/main.cpp, and Sources/widget.cpp. The code editor window is open with the file 'widget.h'. The code defines a class 'Widget' that inherits from 'QWidget'. It includes includes for QWidget, QCheckBox, QPushButton, QHBoxLayout, QVBoxLayout, QCloseEvent, and QDebug. It also includes a private section with pointers to QCheckBox and QPushButton objects, and public section with constructors and methods for creating buttons, checkboxes, and layouts.

```
#ifndef WIDGET_H
#define WIDGET_H

#include <QtGui/QWidget>
/* add other widget libraries that you might need */
#include <QCheckBox>           // to add a checkbox
#include <QPushButton>          // to add a button
#include <QHBoxLayout>          // to arrange widgets horizontally
#include <QVBoxLayout>          // to arrange widgets vertically
#include <QCloseEvent>          // to detect when widget is closed and initiate clean up routine
#include <QDebug>               // to add messages to the console (similar to std::cout)

class Widget : public QWidget
{
    Q_OBJECT

private:
    QCheckBox * checkBox1;
    QCheckBox * checkBox2;

    QPushButton * button1;
    QPushButton * button2;
    QPushButton * button3;
    QPushButton * button4;

    QHBoxLayout * hlayout1;
    QHBoxLayout * hlayout2;
    QHBoxLayout * hlayout3;

    QVBoxLayout * mainLayout;

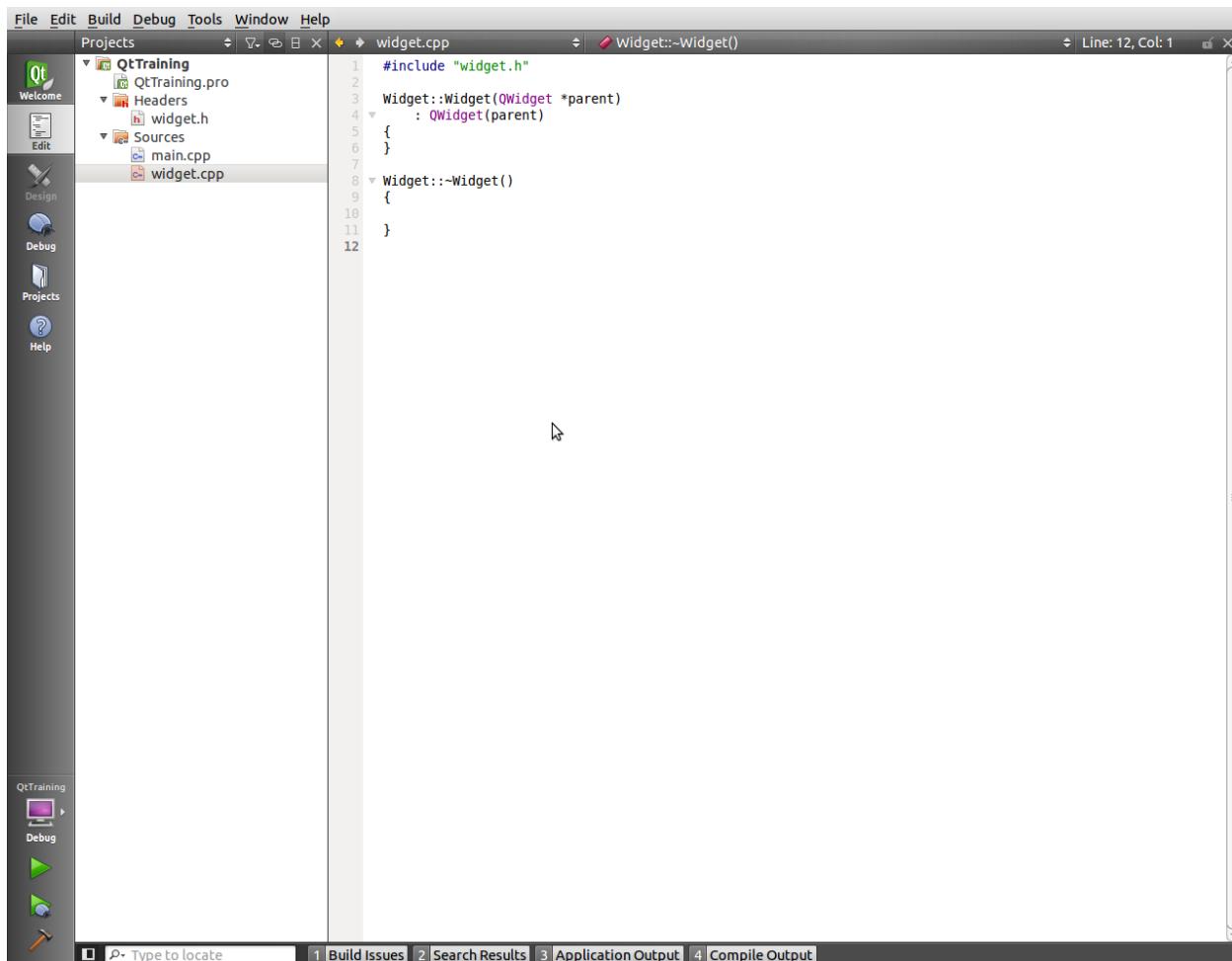
public:
    Widget(QWidget *parent = 0);
    ~Widget();

    void create_buttons();
    void create_checkboxes();
    void create_layouts();
};

#endif // WIDGET_H
```

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.



The screenshot shows the Qt Creator IDE interface. The top menu bar includes File, Edit, Build, Debug, Tools, Window, and Help. The title bar displays "File Edit Build Debug Tools Window Help" and "Projects" with a dropdown arrow, followed by "widget.cpp" and "Widget::Widget()". A status bar at the bottom right indicates "Line: 12, Col: 1".

The left sidebar contains icons for Welcome, Edit, Design, Debug, Projects, and Help. The Projects section shows a project named "QtTraining" with files "QtTraining.pro", "Headers", "Sources", "main.cpp", and "widget.cpp".

The main central area is a code editor showing the following C++ code:

```
#include "widget.h"
Widget::Widget(QWidget *parent)
    : QWidget(parent)
{
}
Widget::~Widget()
{
}
```

At the bottom of the interface, there is a toolbar with icons for Run, Stop, and Build, along with tabs for "Type to locate", "Build Issues", "Search Results", "Application Output", and "Compile Output".

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.

```

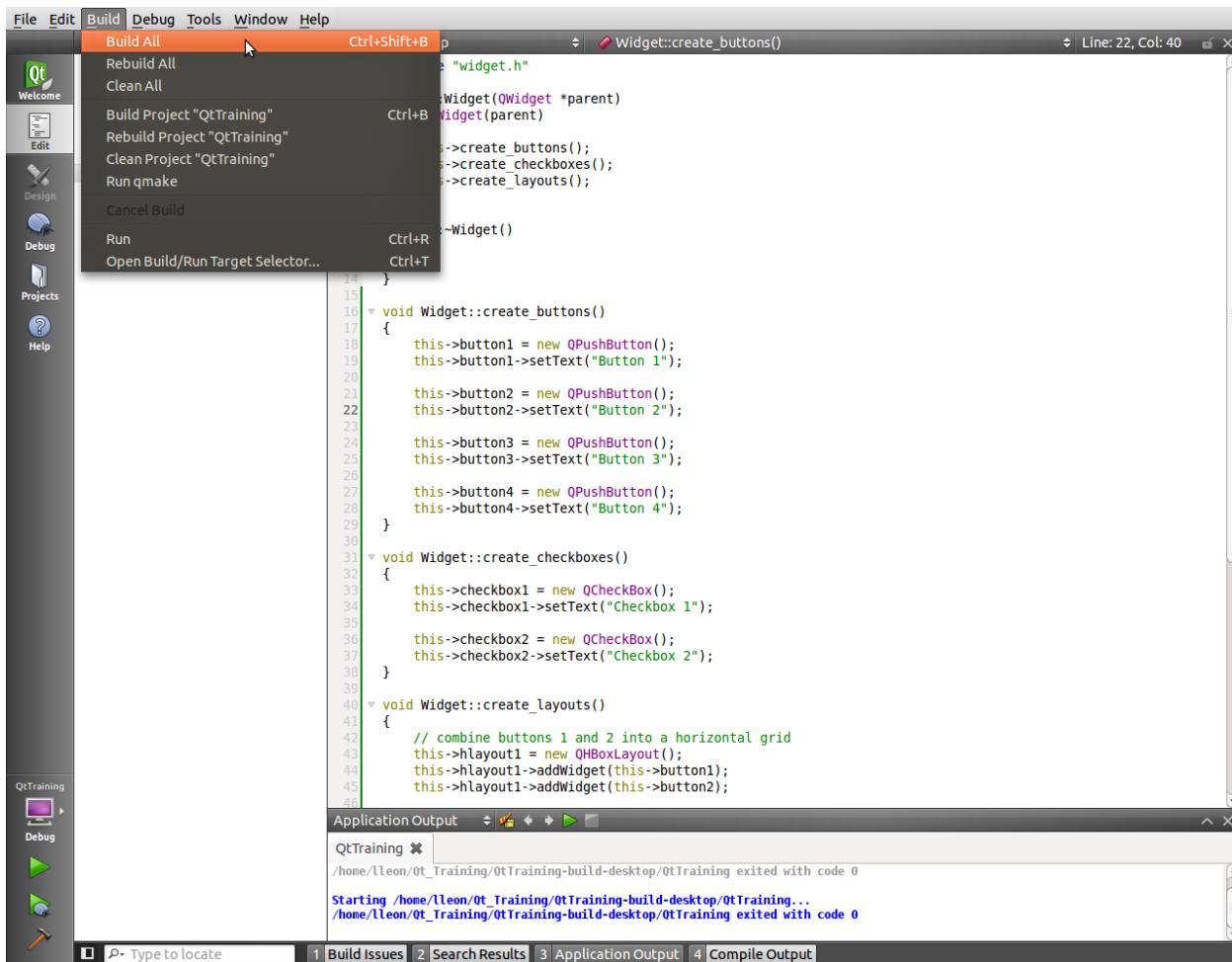
Widget::Widget(QWidget *parent)
    : QWidget(parent)
{
    this->create_buttons();
    this->create_checkboxes();
    this->create_layouts();
}

```

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()”.

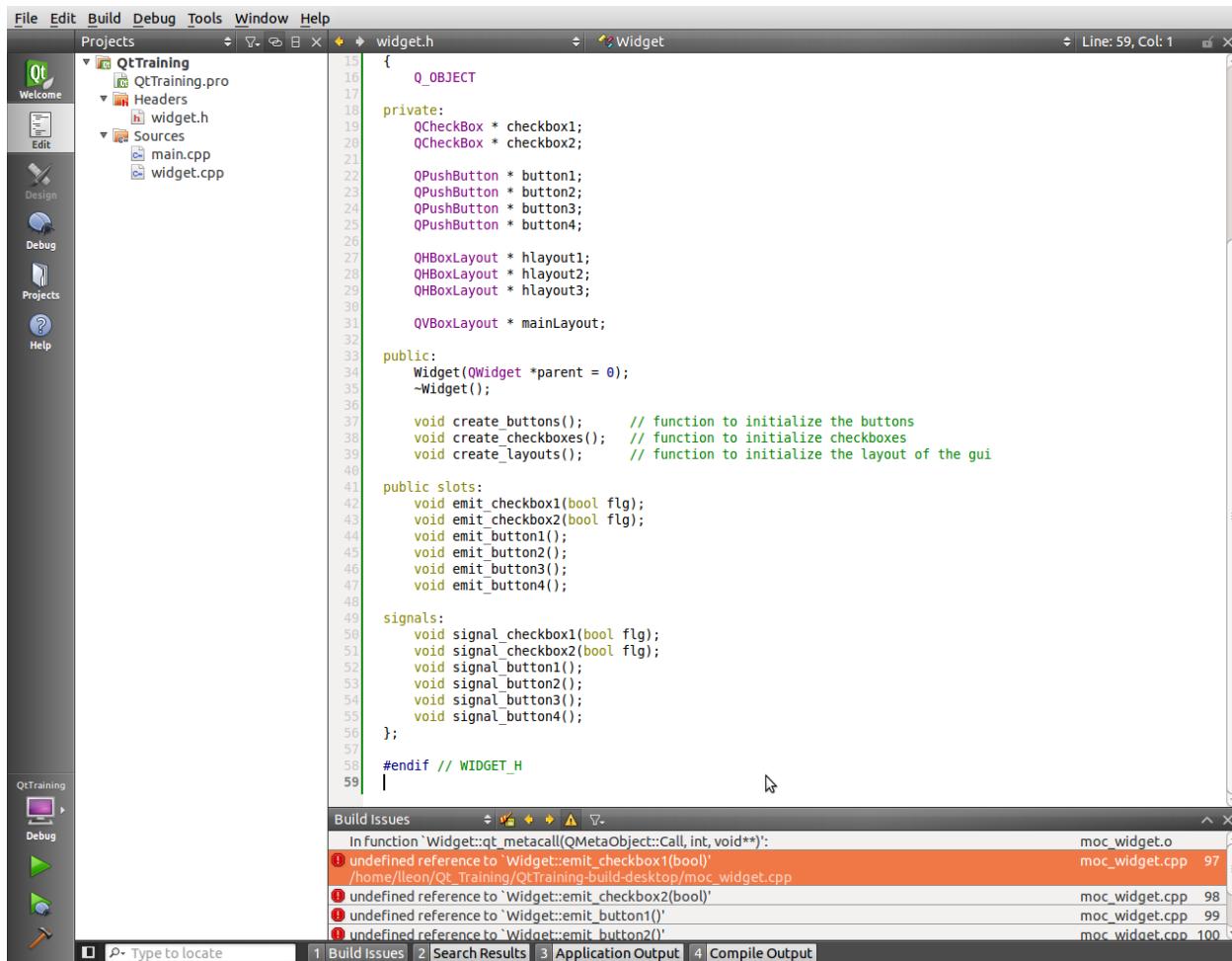
The code is provided in the file “12\_Qt\_cpp.cpp” if you want to copy from there.

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.



```

File Edit Build Debug Tools Window Help
File Edit Build Debug Tools Window Help
Projects Projects QTTraining QTTraining.pro Headers Headers Sources Sources main.cpp widget.cpp
Line: 59, Col: 1
QTTraining
QTTraining.pro
Headers
widget.h
Sources
main.cpp
widget.cpp
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59

Widget.h
Widget
{
    Q_OBJECT

private:
    QCheckBox * checkbox1;
    QCheckBox * checkbox2;

    QPushButton * button1;
    QPushButton * button2;
    QPushButton * button3;
    QPushButton * button4;

    QHBoxLayout * layout1;
    QHBoxLayout * layout2;
    QHBoxLayout * layout3;

    QVBoxLayout * mainLayout;

public:
    Widget(QWidget *parent = 0);
    ~Widget();

    void create_buttons(); // function to initialize the buttons
    void create_checkboxes(); // function to initialize checkboxes
    void create_layouts(); // function to initialize the layout of the gui

public slots:
    void emit_checkbox1(bool flg);
    void emit_checkbox2(bool flg);
    void emit_button1();
    void emit_button2();
    void emit_button3();
    void emit_button4();

signals:
    void signal_checkbox1(bool flg);
    void signal_checkbox2(bool flg);
    void signal_button1();
    void signal_button2();
    void signal_button3();
    void signal_button4();
};

#endif // WIDGET_H

```

Build Issues

In Function `Widget::qt_metacall(QMetaObject::Call, int, void*)`:	
undefined reference to 'Widget::emit_checkbox1(bool)' /home/leon/Qt_Training/QtTraining-build-desktop/moc_widget.cpp	moc_widget.o moc_widget.cpp 97
undefined reference to 'Widget::emit_checkbox2(bool)' /home/leon/Qt_Training/QtTraining-build-desktop/moc_widget.cpp	moc_widget.o moc_widget.cpp 98
undefined reference to 'Widget::emit_button1()' /home/leon/Qt_Training/QtTraining-build-desktop/moc_widget.cpp	moc_widget.o moc_widget.cpp 99
undefined reference to 'Widget::emit_button2()' /home/leon/Qt_Training/QtTraining-build-desktop/moc_widget.cpp	moc_widget.o moc_widget.cpp 100

Type to locate 1 Build Issues 2 Search Results 3 Application Output 4 Compile Output

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 to 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.

The screenshot shows the Qt Creator IDE interface. The top menu bar includes File, Edit, Build, Debug, Tools, Window, and Help. The left sidebar has sections for Welcome, Projects, Edit, Design, Debug, Projects, and Help. The central area shows a code editor with the file `widget.cpp` open. The code contains a `closeEvent` slot:

```

1 #include "widget.h"
2
3 Widget::Widget(QWidget *parent)
4 : QWidget(parent) {...}
5
6 Widget::~Widget() {...}
7
8 void Widget::closeEvent(QCloseEvent *event)
9 {
10     qDebug() << "instructions if widget is closed";
11 }
12
13 void Widget::create_buttons() {...}
14
15 void Widget::create_checkboxes() {...}
16
17 void Widget::create_layouts() {...}
18
19 /* code for slots */
20 void Widget::emit_checkbox1(bool flg) {...}
21
22 void Widget::emit_checkbox2(bool flg) {...}
23
24 void Widget::emit_button1() {...}
25
26 void Widget::emit_button2() {...}
27
28 void Widget::emit_button3() {...}
29
30 void Widget::emit_button4() {...}
31
32 /* connect the slots to the signals */
33 void Widget::connect_slots() {...}
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123

```

The bottom right corner of the code editor shows "Line: 19, Col: 52". Below the code editor is the Application Output tab, which displays the following log message:

```

QtTraining
Starting /home/lleon/Ot_Training/OtTraining-build-desktop/OtTraining...
instructions if widget is closed
/home/lleon/Ot_Training/OtTraining-build-desktop/OtTraining exited with code 0

```

The bottom navigation bar includes tabs for Type to locate, Build Issues, Search Results, Application Output, and Compile Output.

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

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

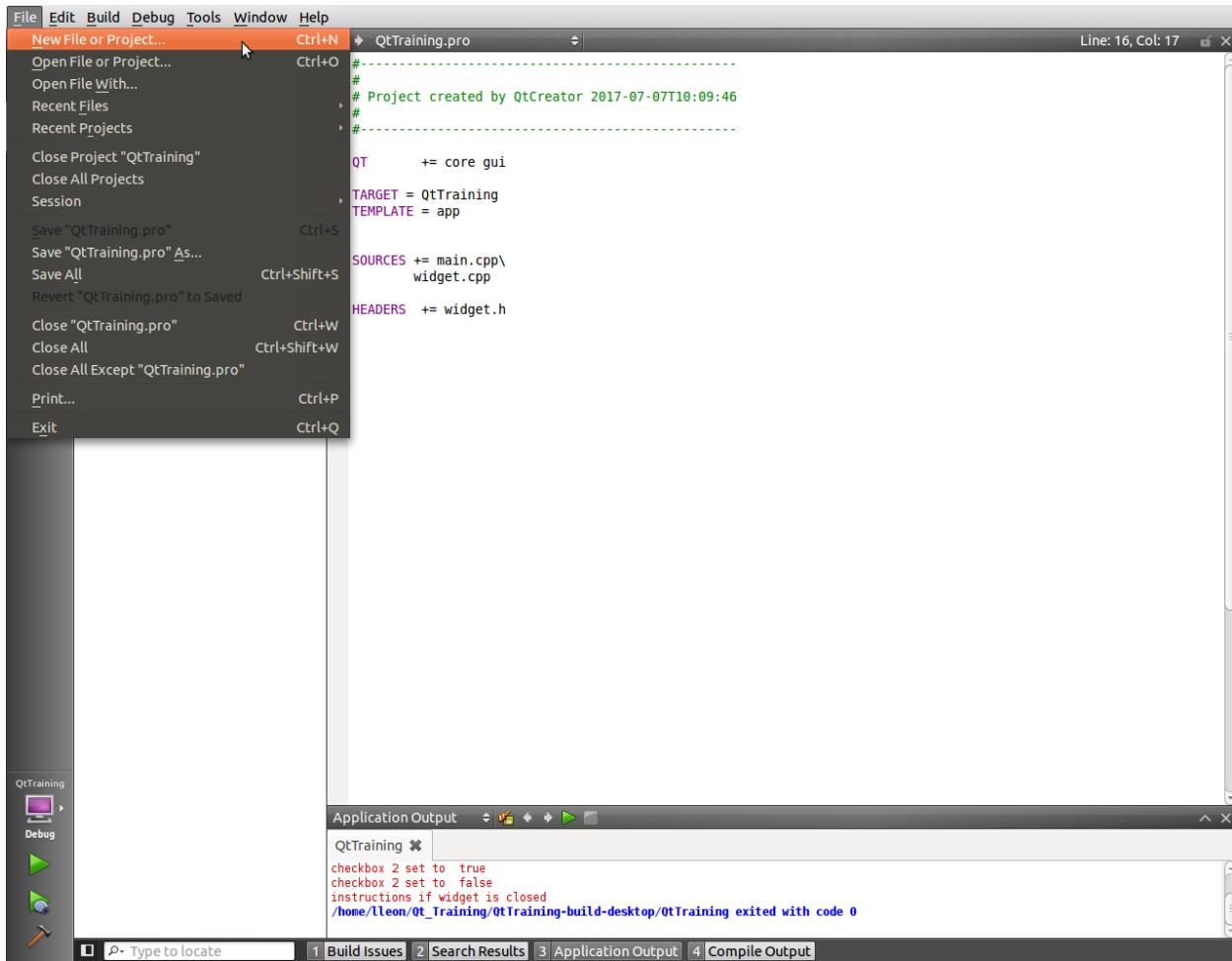
and place it after the default destructor `~Widget();`

The full header file is found in "17\_Qt\_close\_event.h" for your convenience.

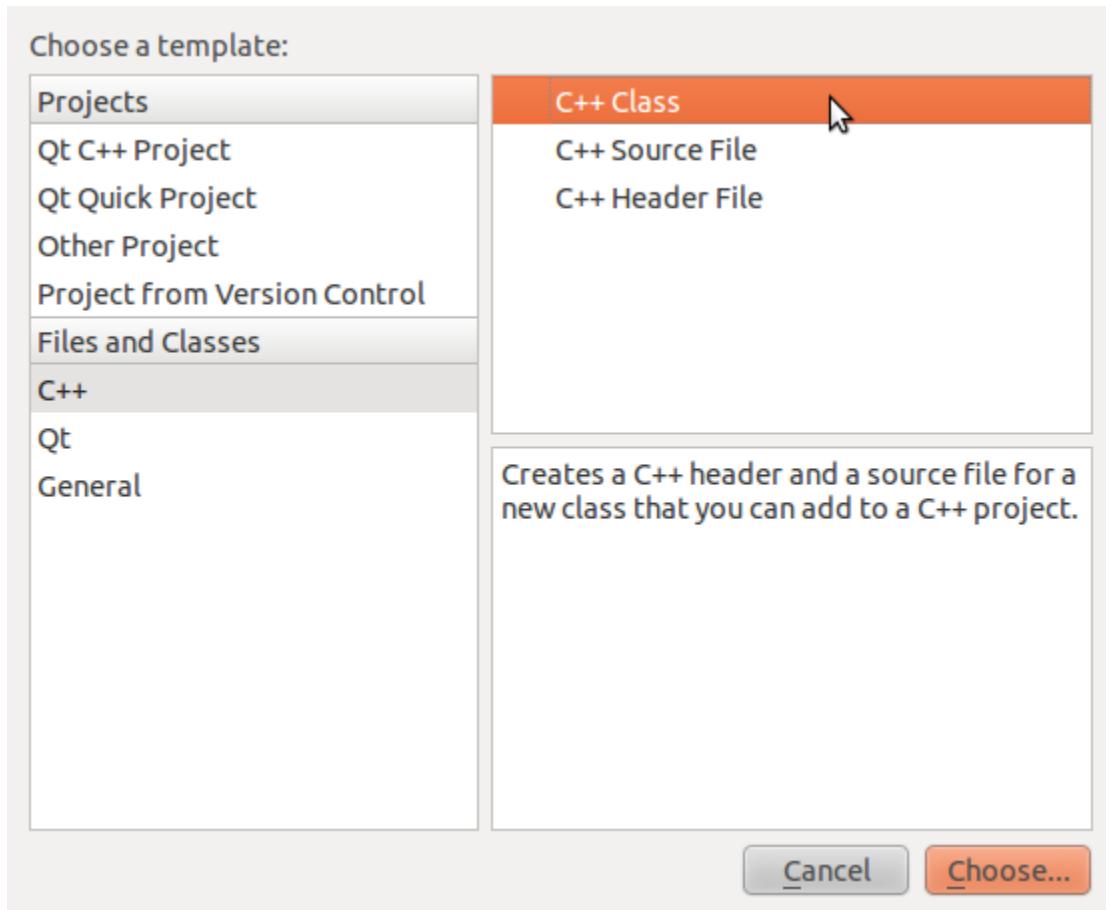
So far we've just created a widget that does stuff when you activate the UI features. In this final section, we will connect the signals generated by the widget to code that is defined in a different class altogether. The power of this feature is that the signal generated by the widget (and keep in mind 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.

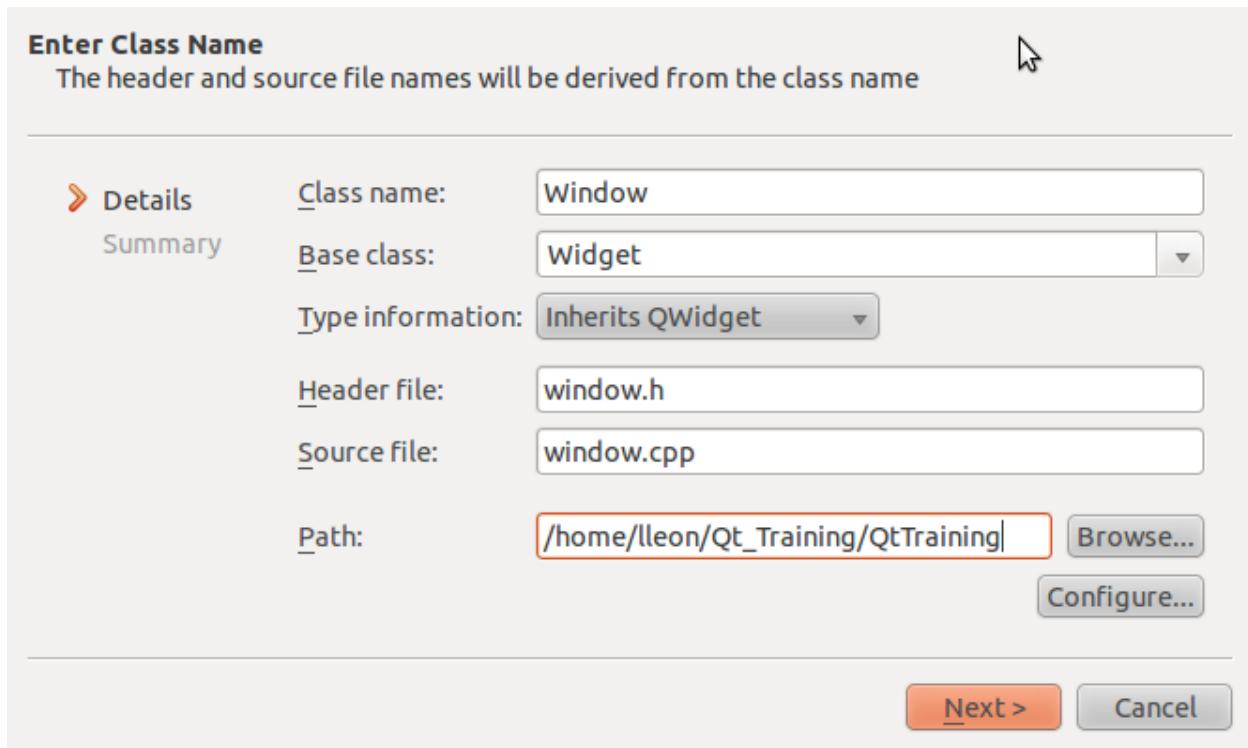
First, let's define a new class that inherits from the Widget class just created.



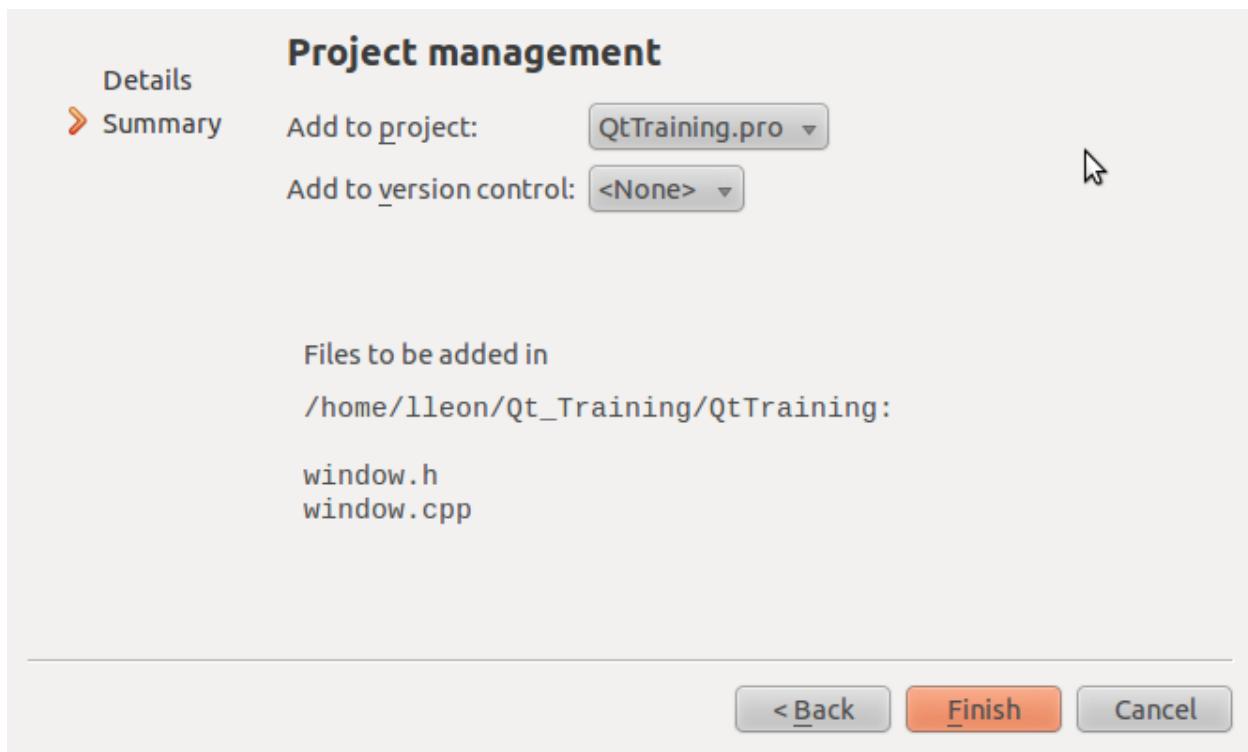
Select C++ Class. We are not creating a new project. We are simply creating a new class in the same project.



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.

The screenshot shows the Qt Creator IDE interface. The top menu bar includes File, Edit, Build, Debug, Tools, Window, Help, and a status bar indicating Line: 19, Col: 1. The left sidebar has icons for Welcome, Edit, Design, Debug, Projects, and Help. The Projects panel shows a project named "QtTraining" with a sub-project "QtTraining.pro". Inside "QtTraining.pro", there are sections for Headers (containing "widget.h" and "window.h") and Sources (containing "main.cpp", "widget.cpp", and "window.cpp"). The main code editor area displays the "QtTraining.pro" file content:

```
1 #-
2 #
3 # Project created by QtCreator 2017-07-07T10:09:46
4 #
5 #-
6
7 QT      += core gui
8
9 TARGET = QtTraining
10 TEMPLATE = app
11
12
13 SOURCES += main.cpp \
14             widget.cpp \
15             window.cpp
16
17 HEADERS += widget.h \
18             window.h
19 |
```

The bottom right corner of the code editor has a small mouse cursor icon. Below the code editor is the "Application Output" window, which displays the following log output:

```
QtTraining x
checkbox 2 set to true
checkbox 2 set to false
instructions if widget is closed
/home/Leon/Qt_Training/QtTraining-build-desktop/QtTraining exited with code 0
```

The bottom navigation bar of the application window includes tabs for Type to locate, Build Issues, Search Results, Application Output, and Compile Output.

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.

The screenshot shows the Qt Creator IDE interface. The top menu bar includes File, Edit, Build, Debug, Tools, Window, Help, and a status bar indicating Line: 17, Col: 1. The left sidebar has icons for Welcome, Edit, Design, Debug, Projects, and Help. The Projects panel shows a single project named "QtTraining" with a sub-project "QTTraining.pro". The main editor area displays the contents of "window.h":

```
#ifndef WINDOW_H
#define WINDOW_H

class Window : public Widget
{
    Q_OBJECT
public:
    explicit Window(QWidget *parent = 0);

signals:

public slots:

};

#endif // WINDOW_H
```

The bottom Application Output panel shows the following log:

```
QtTraining x
checkbox 2 set to true
checkbox 2 set to false
instructions if widget is closed
/home/Leon/Qt_Training/QtTraining-build-desktop/QtTraining exited with code 0
```

At the bottom of the interface, there is a toolbar with icons for Type to locate, Build Issues, Search Results, Application Output, and Compile Output.

And this is the auto-generated \*.cpp file

The screenshot shows the Qt Creator IDE interface. The top menu bar includes File, Edit, Build, Debug, Tools, Window, Help, Projects, and a search bar. The left sidebar has icons for Welcome, Edit, Design, Debug, Projects, and Help. The central Projects panel shows a project named "QtTraining" with files QTTraining.pro, Headers (widget.h, window.h), and Sources (main.cpp, widget.cpp, window.cpp). The main code editor window displays the contents of window.cpp:

```
#include "window.h"
Window::Window(QWidget *parent) :
    QWidget(parent)
{}
```

The status bar at the bottom shows "Line: 7, Col: 1". Below the code editor is the Application Output panel, which displays the following log message:

```
checkbox 2 set to true
checkbox 2 set to false
instructions if widget is closed
/home/lleon/Qt_Training/QtTraining-build-desktop/QtTraining exited with code 0
```

The bottom navigation bar includes tabs for Type to locate, Build Issues, Search Results, Application Output (which is active), and Compile Output.

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.

The screenshot shows the Qt Creator IDE interface. The top menu bar includes File, Edit, Build, Debug, Tools, Window, Help, Projects, and a Welcome section. On the left, there's a vertical toolbar with icons for Qt Welcome, Edit, Design, Debug, Projects, and Help. The Projects panel shows a project named "QtTraining" with files "QtTraining.pro", "Headers" containing "widget.h" and "window.h", and "Sources" containing "main.cpp", "widget.cpp", and "window.cpp". The main editor window displays the "main.cpp" source code:

```
#include <QtGui/QApplication>
#include "widget.h"
#include "window.h"

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    //Widget w;
    Window w;
    w.show();

    return a.exec();
}
```

The bottom part of the interface shows the "Application Output" panel with the title "QtTraining". It displays the following log output:

```
button 3 clicked
button 4 clicked
checkbox 1 set to true
checkbox 1 set to false
checkbox 1 set to true
checkbox 1 set to false
checkbox 2 set to true
checkbox 2 set to false
checkbox 2 set to true
checkbox 2 set to false
instructions if widget is closed
/home/leon/Qt_Training/QtTraining-build-desktop/QtTraining exited with code 0
```

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.