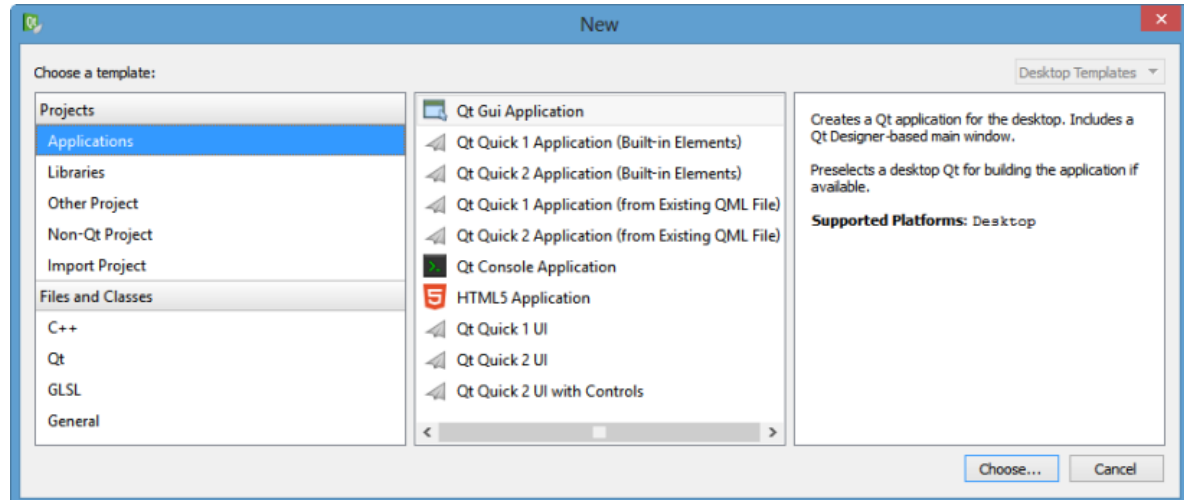


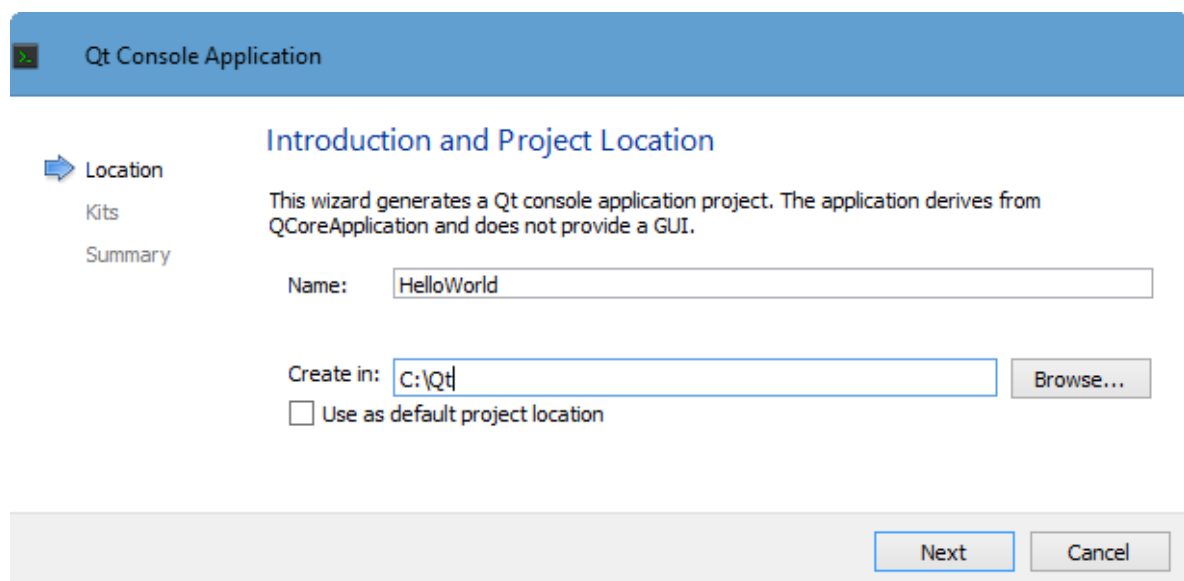
Hello World

In this tutorial, we will print out simple text "Hello World".

File->New File or Project...



Applications->Qt Console Application->Choose...



Let's see the **main.cpp** file that Creator made for us:

```
#include <QCoreApplication>

int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);

    return a.exec();
}
```

The **QCoreApplication** class provides an event loop for console Qt applications. This class is used by non-GUI applications to provide their event loop. For non-GUI application that uses Qt, there should be exactly one **QCoreApplication** object. For GUI applications, we will use **QApplication**.

When calling **a.exec()** the event loop is launched.

Let's compile this application. By clicking on the green arrow on the bottom left, Qt Creator will compile and execute it. And what happened ? The application seems to be launched and not responding. It is actually normal. The event loop is running and waiting for events such as mouse click, but we did not provide any event to be processed, so it will run indefinitely.

When we compile the project, behind the scene, our Qt app is compiled in the following steps:

1. **qmake** parses the **.pro** file, and generates **makefile**.
2. The program is build using **make** (**jom** on windows).

Let's print out "Hello World"i by modifying the source:

```
#include <QCoreApplication>
#include <QDebug>
int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);

    qDebug() << "Hello World";

    return a.exec();
}
```

If we run the code again, it prints out "Hello World".

Setting Up Signals and Slots

File->New File or Project...

Applications->Qt Gui Application->Choose...

We keep the class as **MainWindow** as given by default.

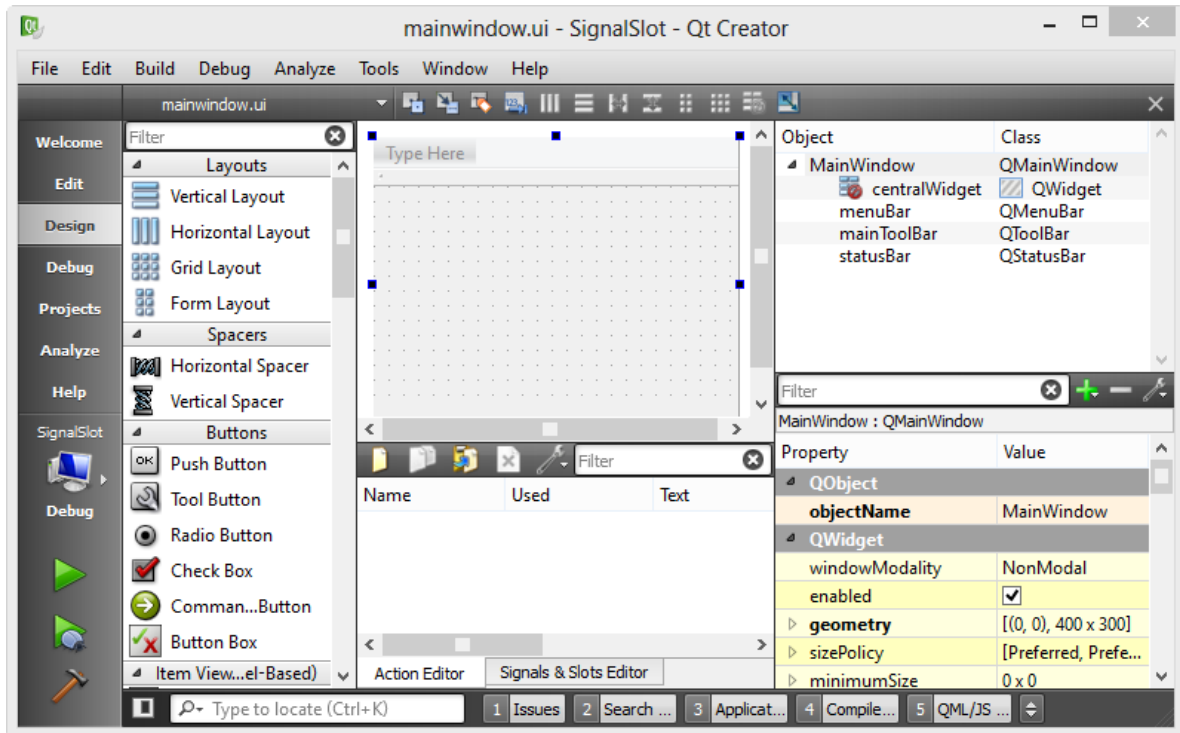
The screenshot shows the 'Qt Gui Application' wizard in Qt Creator. The 'Class Information' tab is selected, showing fields for Class name, Base class, Header file, Source file, Generate form, and Form file. The 'Details' tab is also visible on the left sidebar.

Class Information	
Specify basic information about the classes for which you want to generate skeleton source code files.	
Class name:	MainWindow
Base class:	QMainWindow
Header file:	mainwindow.h
Source file:	mainwindow.cpp
Generate form:	<input checked="" type="checkbox"/>
Form file:	mainwindow.ui

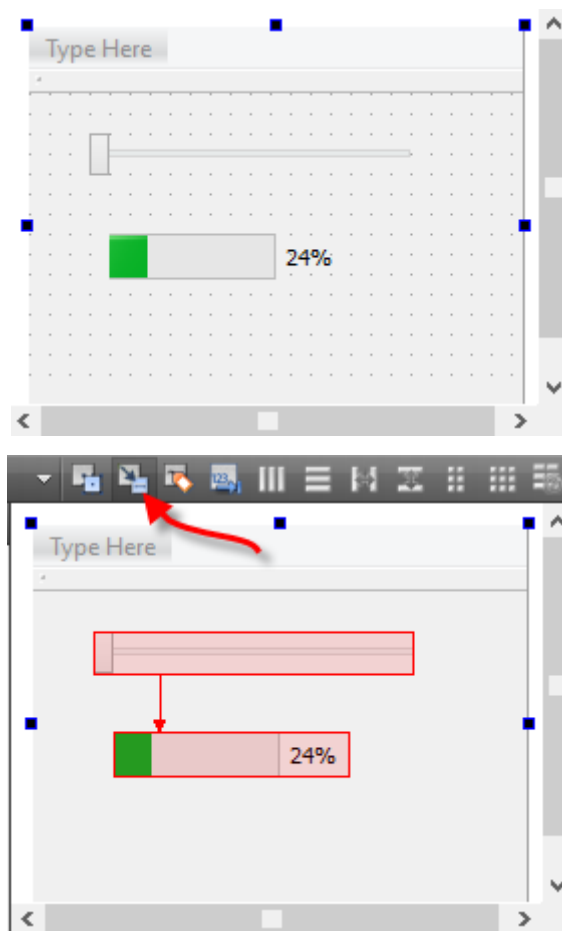
Next Cancel

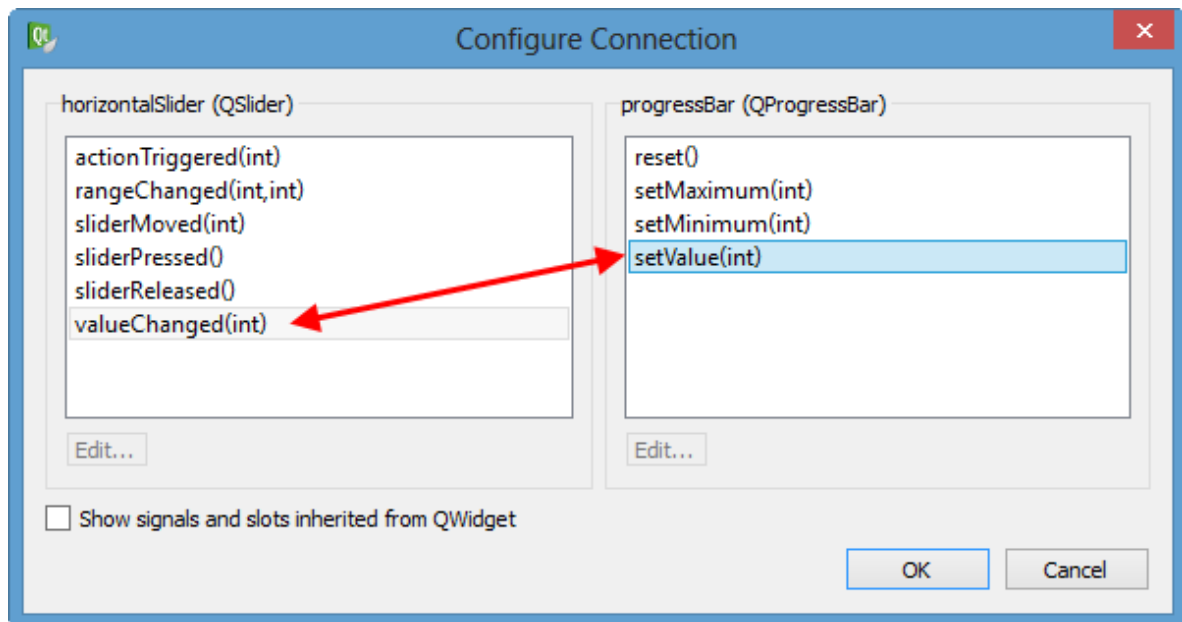
Next->Finish

Let's open up **Forms** by double-clicking the **mainwindow.ui** to put gui components:

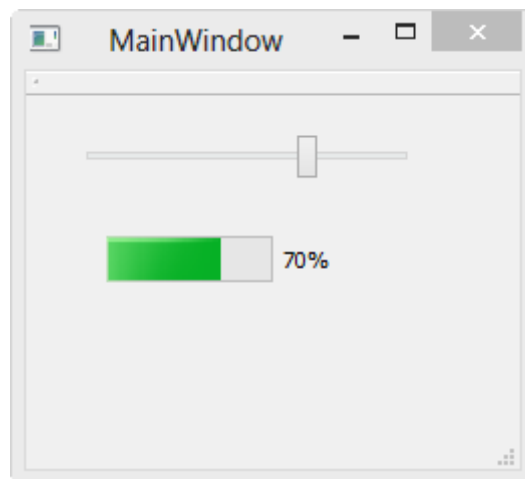


From the Widgets, drag Horizontal Slider and Progress Bar, and place them into the main window. Then,





Run the code. Now, if we move the slider, the progress will reflect the changes in the slider:



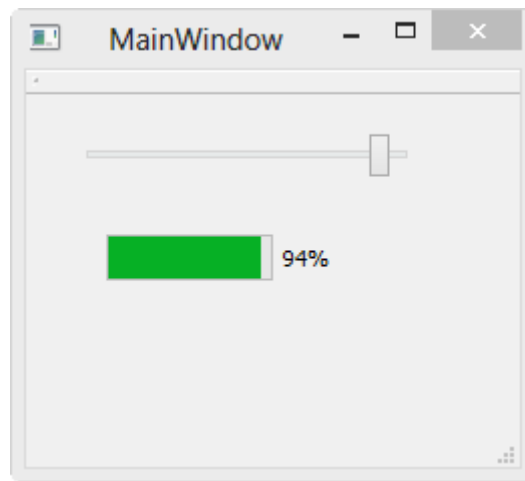
We did it via gui, but we can do it via direct programming.

Let's delete the signal and slot, and write the code for the signal and slot mechanism in the constructor of the **MainWindow** class as shown below:

```
#include "mainwindow.h"
#include "ui_mainwindow.h"

MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::MainWindow)
{
    ui->setupUi(this);
    connect(ui->horizontalSlider,
        SIGNAL(valueChanged(int)),
        ui->progressBar,
        SLOT(setValue(int)));
}

MainWindow::~MainWindow()
{
    delete ui;
}
```



Signal and Slot

Signals and slots are used for communication between objects. The signals and slots mechanism is a central feature of Qt and probably the part that differs most from the features provided by other frameworks.

In GUI programming, when we change one widget, we often want another widget to be notified. More generally, we want objects of any kind to be able to communicate with one another. For example, if a user clicks a Close button, we probably want the window's close() function to be called. Older toolkits achieve this kind of communication using callbacks. A callback is a pointer to a function, so if you want a processing function to notify you about some event you pass a pointer to another function (the callback) to the processing function. The processing function then calls the callback when appropriate. Callbacks have two fundamental flaws: Firstly, they are not type-safe. We can never be certain that the processing function will call the callback with the correct arguments. Secondly, the callback is strongly coupled to the processing function since the processing function must know which callback to call.

In Qt, we have an alternative to the callback technique: We use signals and slots. A signal is emitted when a particular event occurs. Qt's widgets have many predefined signals, but we can always subclass widgets to add our own signals to them. A slot is a function that is called in response to a particular signal. Qt's widgets have many predefined slots, but it is common practice to subclass widgets and add your own slots so that you can handle the signals that you are interested in.

The signals and slots mechanism is type safe: The signature of a signal must match the signature of the receiving slot. (In fact a slot may have a shorter signature than the signal it receives because it can ignore extra arguments.) Since the signatures are compatible, the compiler can help us detect type mismatches. Signals and slots are loosely coupled: A class which emits a signal neither knows nor cares which slots receive the signal. Qt's signals and slots mechanism ensures that if you connect a signal to a slot, the slot will be called with the signal's parameters at the right time. Signals and slots can take any number of arguments of any type. They are completely type safe.

All classes that inherit from **QObject** or one of its subclasses (e.g., **QWidget**) can contain signals and slots. Signals are emitted by objects when they change their state in a way that may be interesting to other objects. This is all the object does to communicate. It does not know or care whether anything is receiving the signals it emits. This is true information encapsulation, and ensures that the object can be used as a software component.

Slots can be used for receiving signals, but they are also normal member functions. Just as an object does not know if anything receives its signals, a slot does not know if it has any signals connected to it. This ensures that truly independent components can be created with Qt. You can connect as many signals as you want to a single slot, and a signal can be connected to as many slots as you need. It is even possible to connect a signal directly to another signal. (This will emit the second signal immediately whenever the first is emitted.)

Q_OBJECT Macro

The **Q_OBJECT Macro** is probably one of the weirdest things to whoever beginning to use Qt.

Qt [QObject Class](#) says:

The Q_OBJECT macro must appear in the private section of a class definition that declares its own signals and slots or that uses other services provided by Qt's meta-object system.

```
class MyClass : public QObject
{
    Q_OBJECT

public:
    MyClass(QObject *parent = 0);
    ~MyClass();

signals:
    void mySignal();

public slots:
    void mySlot();
};
```

So, it sounds like we need it to use signal and slot, and probably for other purposes (meta-object related) as well.

Another doc related to [moc](#) explains:

The Meta-Object Compiler, moc, is the program that handles Qt's C++ extensions.

The moc tool reads a C++ header file. If it finds one or more class declarations that contain the Q_OBJECT macro, it produces a C++ source file containing the meta-object code for those classes. Among other things, meta-object code is required for the signals and slots mechanism, the run-time type information, and the dynamic property system.

Another doc on [Signal and Slot](#):

All classes that contain signals or slots must mention Q_OBJECT at the top of their declaration. They must also derive (directly or indirectly) from QObject.

Note that when we want to make connection between signal and slot, we actually do it with **Object** scope:

```
QObject::connect(sender, signal, receiver, slot):
```

Is Q_OBJECT Macro always needed?

1. Quick answer: No.

2. Better answer: Well, just put it there always.

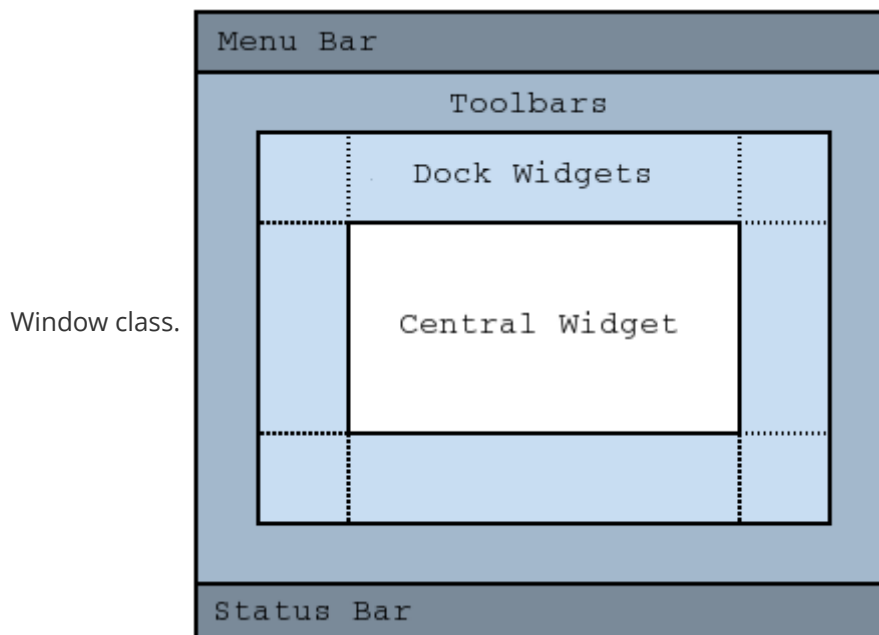
Actually, the `Q_OBJECT` macro is only required if meta-object code has to be produced by the moc tool in order to use the signals and slots mechanism, the run-time type information, the dynamic property system and translating features for internationalisation.

While it is possible to use `QObject` as a base class without the `Q_OBJECT` macro and without meta object code, neither signals and slots nor the other features described here will be available if the `Q_OBJECT` macro is not used.

In general, it seems that all Qt developers have been strongly recommend to use `Q_OBJECT` for every subclass of `QObject` whether or not they actually use the features listed above.

Main Window and Action



In this tutorial, we will learn how to setup the action from the menu and toolbar of the Main



File->New File or Project...

Applications->Qt Gui Application->Choose...

We keep the class as **MainWindow** as given by default.

 Qt Gui Application

Location
Kits
Details
Summary

Class Information

Specify basic information about the classes for which you want to generate skeleton source code files.

Class name:

Base class:

Header file:

Source file:



Generate form: ☒

Form file:

Next

Cancel

Hit Next.

 Qt Gui Application

Location
Kits
Details
Summary

Project Management

Add as a subproject to project:

Add to version control:

Files to be added in

C:\Qt\MainWindowAction:

```
main.cpp
mainwindow.cpp
mainwindow.h
mainwindow.ui
MainWindowAction.pro
```

Finish

Cancel

Hit Finish.

Here is the **main.cpp**:


```

#include "mainwindow.h"
#include <QApplication>

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

    return a.exec();
}

```

From the highlight line, we have an incident of the **MainWindow** class.

Here is the header file: **mainwindow.h**:

```

#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>

namespace Ui {
class MainWindow;
}

class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
    explicit MainWindow(QWidget *parent = 0);
    ~MainWindow();

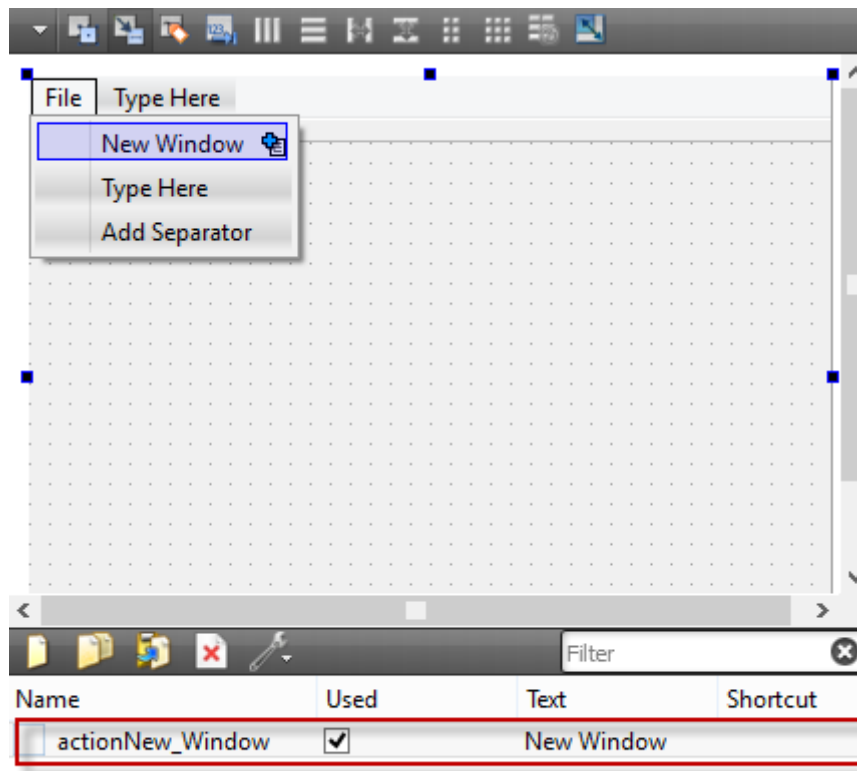
private:
    Ui::MainWindow *ui;
};

#endif // MAINWINDOW_H

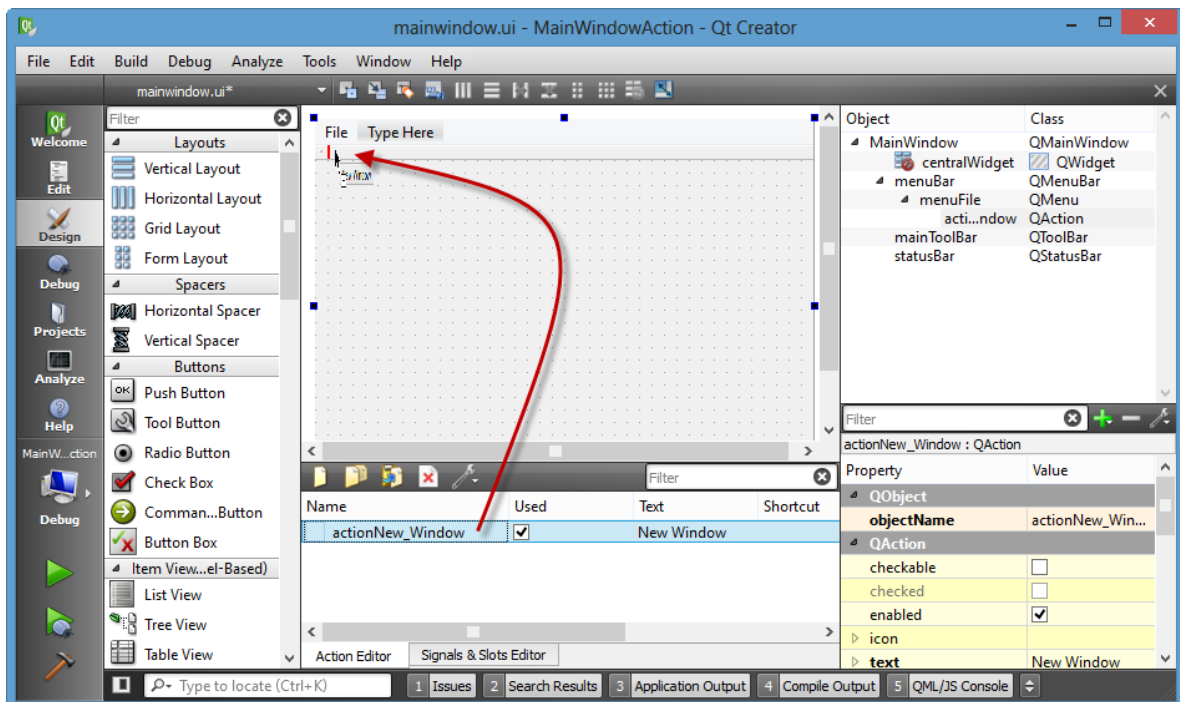
```

Note that we have a namespace **UI**, and our **MainWindow** is under the **UI** scope. Also, in the header, we declared a pointer to the **UI::MainWindow** as a private member, **ui**.

Let's open up **Designer** by double-clicking the **mainwindow.ui** to put menu bar and action. Type in "File" at the top menu, and "New Window..." under the "File" menu. We need to press Enter to make a real change in the menu. Notice that at the bottom, in the Action Editor, a new action called **actionNew_Window** has been created.

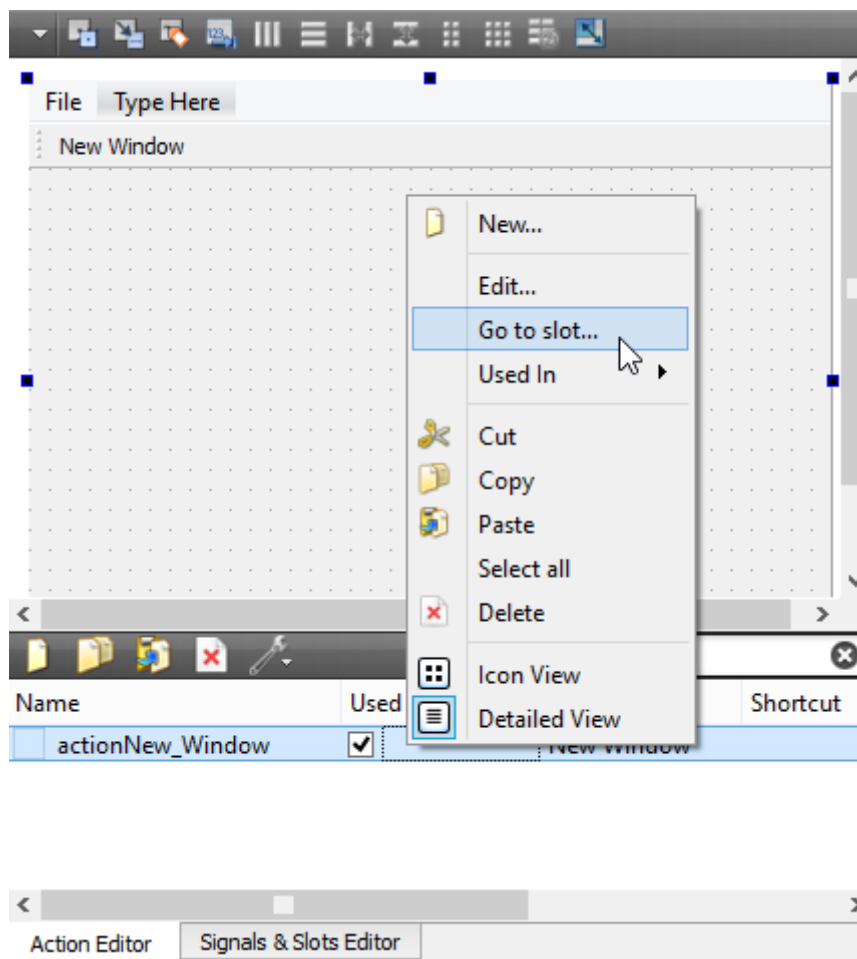


We can also drag the **actionNew_Window** action menu and drop in the toolbar:

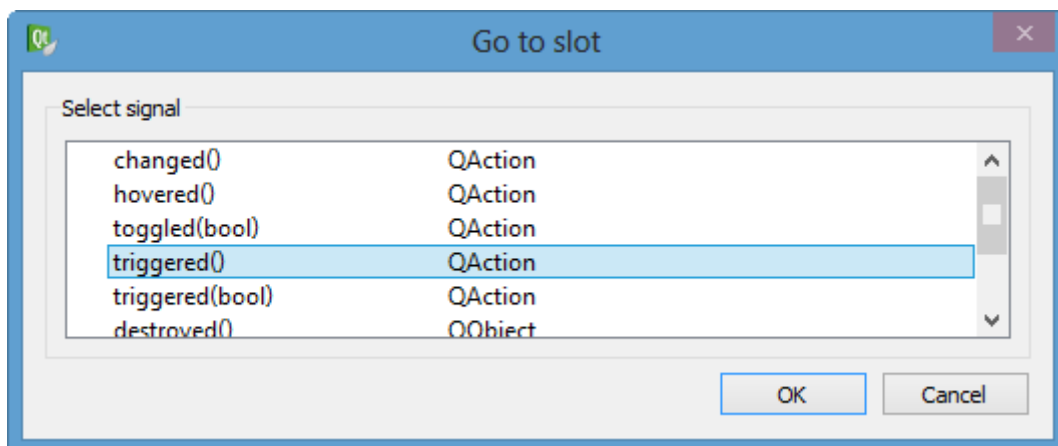


Now, we have two places to trigger **actionNew_Window** action.

Click the action on the Action Editor, and select "Go to Slot...".



It allows us to trigger an event. Select **triggered()** and hit OK.



Then, Creator will show the changes in the code (**mainwindow.cpp**) automatically:

```
#include "mainwindow.h"
#include "ui_mainwindow.h"

MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::MainWindow)
{
    ui->setupUi(this);
}

MainWindow::~MainWindow()
{
    delete ui;
}
```

```

}

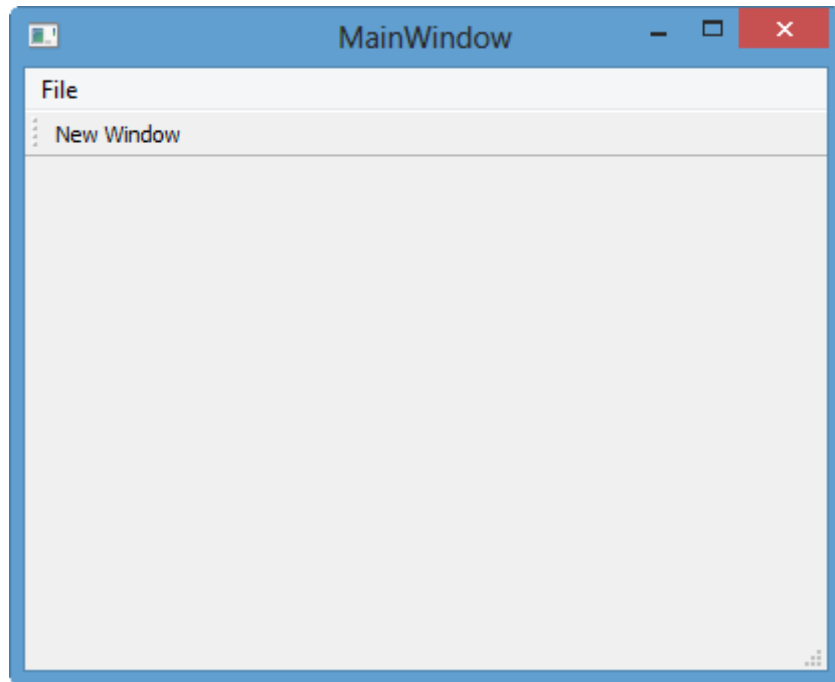
void MainWindow::on_actionNew_Window_triggered()
{

}

```

So, whenever we click either toolbar item or menu item, this event will be triggered.

If we run the code, we get:



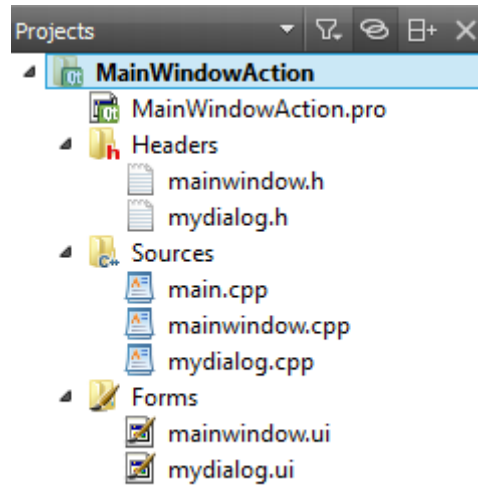
When we click the **New Window**, nothing happens. This is because we haven't put anything into the **MainWindow::on_actionNew_Window_triggered()** method.

Open a Dialog triggered by the action

Right click on MainWindowAction project->Add New->Qt->Qt Designer Form Class->Dialog without Buttons

 A screenshot of the "Qt Designer Form Class" dialog box. The dialog has a blue header bar with a back arrow, the Qt logo, and the text "Qt Designer Form Class". On the left, there are three tabs: "Form Template", "Class Details" (which is selected and highlighted with a blue arrow), and "Summary". The main area is titled "Choose a Class Name". It contains several input fields: "Class name:" with the text "MyDialog", "Header file:" with "mydialog.h", "Source file:" with "mydialog.cpp", "Form file:" with "mydialog.ui", and "Path:" with "C:\Qt". There is a "Browse..." button next to the Path field. At the bottom right, there are "Next" and "Cancel" buttons.

New files related to our new dialog have been created:



Let's put the proper code for the triggered event.

We'll open up the new dialog in two types: **modal** and **modaless**.

Modal Dialog

```
#include "mainwindow.h"
#include "ui_mainwindow.h"
#include "mydialog.h"
MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::MainWindow)
{
    ui->setupUi(this);
}

MainWindow::~MainWindow()
{
    delete ui;
}

void MainWindow::on_actionNew_Window_triggered()
{
    MyDialog mDialog;
    mDialog.setModal(true);
    mDialog.exec();
}
```

If we run it, we get the modal type dialog:



Modaless Dialog

To get modeless, we edited the **mainwindow.cpp** as below:

```
void MainWindow::on_actionNew_Window_triggered()
{
    MyDialog mDialog;
    /*
    mDialog.setModal(false);
    mDialog.exec();
    */
    mDialog.show();
}
```

But it disappears as soon as it is created. That's because unlike modal type dialog, modeless gives control to the main app, the stack variable goes away when it gets out of **on_actionNew_Window_triggered()**. So, we need to set a pointer as a member of the **MainWindow** class:

mainwindow.cpp:

```
#include "mainwindow.h"
#include "ui_mainwindow.h"
#include "mydialog.h"

MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::MainWindow)
{
    ui->setupUi(this);
}

MainWindow::~MainWindow()
{
    delete ui;
}
```

```
void MainWindow::on_actionNew_Window_triggered()
{
    mDialog = new MyDialog(this);
    mDialog->show();
}
```

mainwindow.h:

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>
#include "mydialog.h"

namespace Ui {
class MainWindow;
}

class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
    explicit MainWindow(QWidget *parent = 0);
    ~MainWindow();

private slots:
    void on_actionNew_Window_triggered();

private:
    Ui::MainWindow *ui;
    MyDialog *mDialog;
};

#endif // MAINWINDOW_H
```

Now, modeless dialog does not disappear.

Main Window for ImageViewer

In this tutorial, we will learn how to setup the action from the menu and toolbar of the Main Window class as in the previous tutorial. But this time, it is a little bit more complicated and closer to the real application. This is largely based on the Qt Tutorial but intended to be more kind with more detail: building ui using designer rather than directly typing the code.

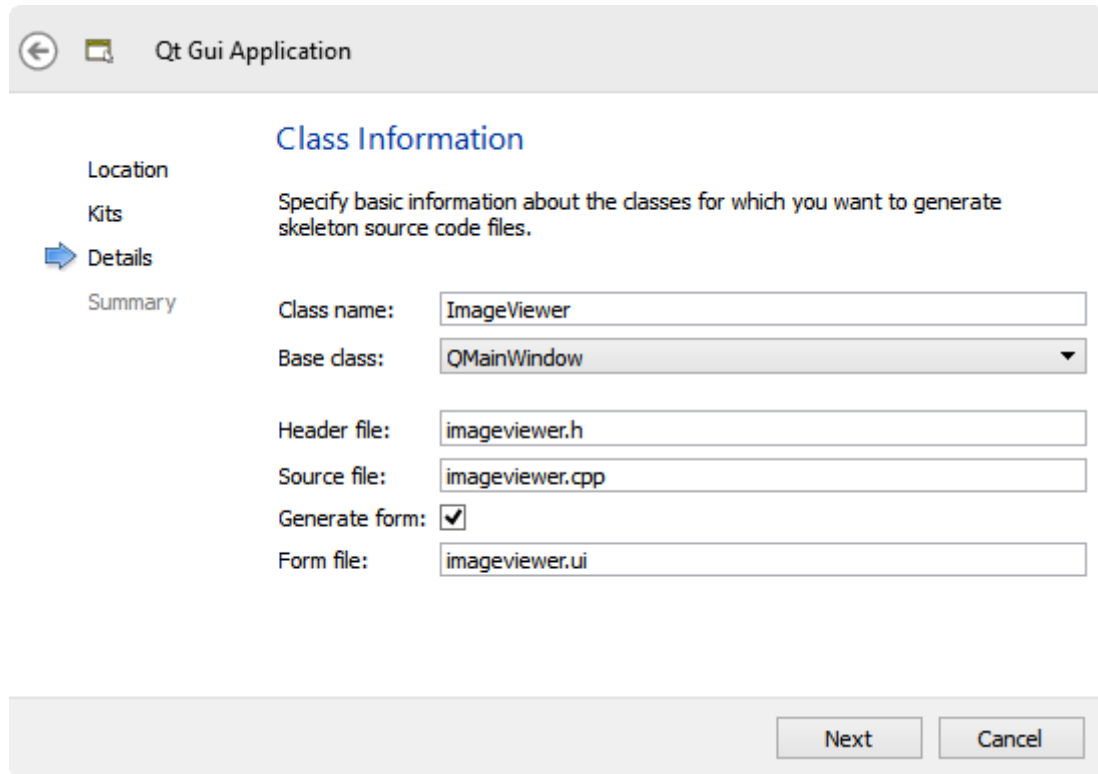
The example demonstrates how QLabel's ability to scale its contents (QLabel::scaledContents), and QScrollArea's ability to automatically resize its contents (QScrollArea::widgetResizable), can be used to implement zooming and scaling features. In addition the example shows how to use QPainter to print an image.

Start

File->New File or Project...

Applications->Qt Gui Application->Choose...

We rename the class as **ImageViewer**.



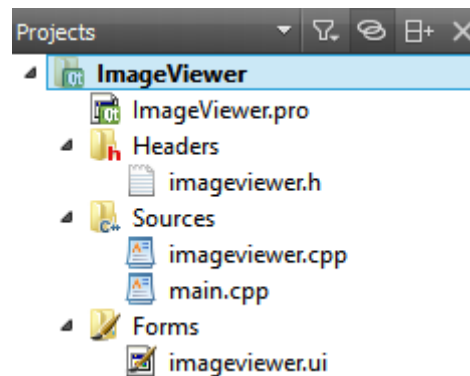
The image shows the 'Class Information' dialog box in Qt. The title bar says 'Qt Gui Application'. On the left, there are tabs: 'Location', 'Kits', 'Details' (selected), and 'Summary'. The main area is titled 'Class Information' and contains the following fields:

- Class name:
- Base class:
- Header file:
- Source file:
- Generate form: ☒
- Form file:

At the bottom right, there are 'Next' and 'Cancel' buttons.

Hit Next ->Finish.

These are the files we get:



Let's open up **imageviewer.pro**, add a line to support print:

```
QT      += core gui
QT      += printsupport

greaterThan(QT_MAJOR_VERSION, 4): QT += widgets

TARGET = ImageViewer
TEMPLATE = app

SOURCES += main.cpp\
           imageviewer.cpp

HEADERS  += imageviewer.h

FORMS    += imageviewer.ui
```


Here is the **main.cpp**:

```
#include "imageviewer.h"
#include <QApplication>

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

    return a.exec();
}
```

From the highlight line, we have an incident of the **ImageViewer** class.

Here is the header file: **imageviewer.h**:

```
#ifndef IMAGEVIEWER_H
#define IMAGEVIEWER_H

#include <QMainWindow>

namespace Ui {
class ImageViewer;
}

class ImageViewer : public QMainWindow
{
    Q_OBJECT

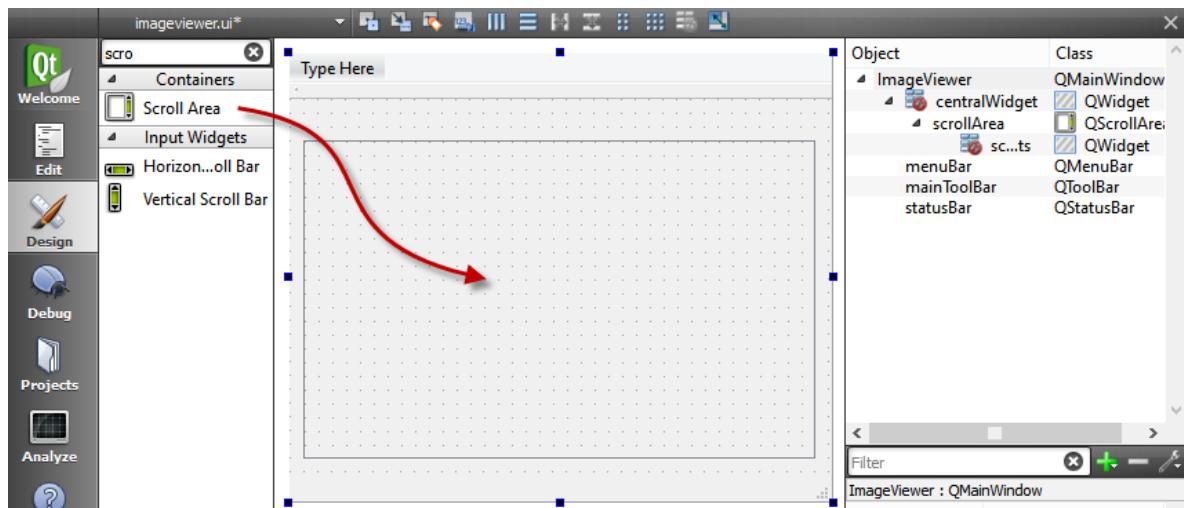
public:
    explicit ImageViewer(QWidget *parent = 0);
    ~ImageViewer();

private:
    Ui::ImageViewer *ui;
};

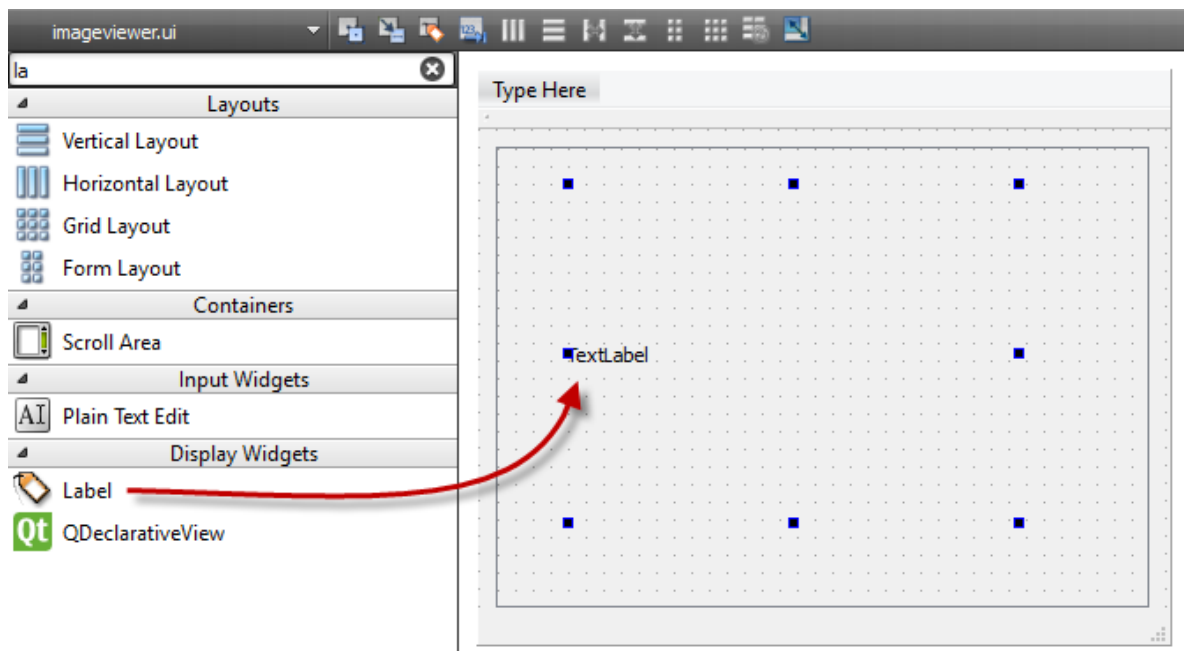
#endif // IMAGEVIEWER_H
```

Note that we have a namespace **UI**, and our **ImageViewer** is under the **UI** scope. Also, in the header, we declared a pointer to the **Ui::ImageViewer** as a private member, **ui**.

Let's open up **Designer** by double-clicking the **imageviewer.ui**. We want to put Scroll Area into the Main Window:



Then, a Label widget as well.



QLabel is typically used for displaying text, but it can also display an image.

QScrollArea provides a scrolling view around another widget. If the child widget exceeds the size of the frame, QScrollArea automatically provides scroll bars.

Initial Preview of ImageViewer

Let's do some check how it will work.

Here is the minimal version just for a display. The code below is the constructor of the **ImageViewer** which is the bare minimum.

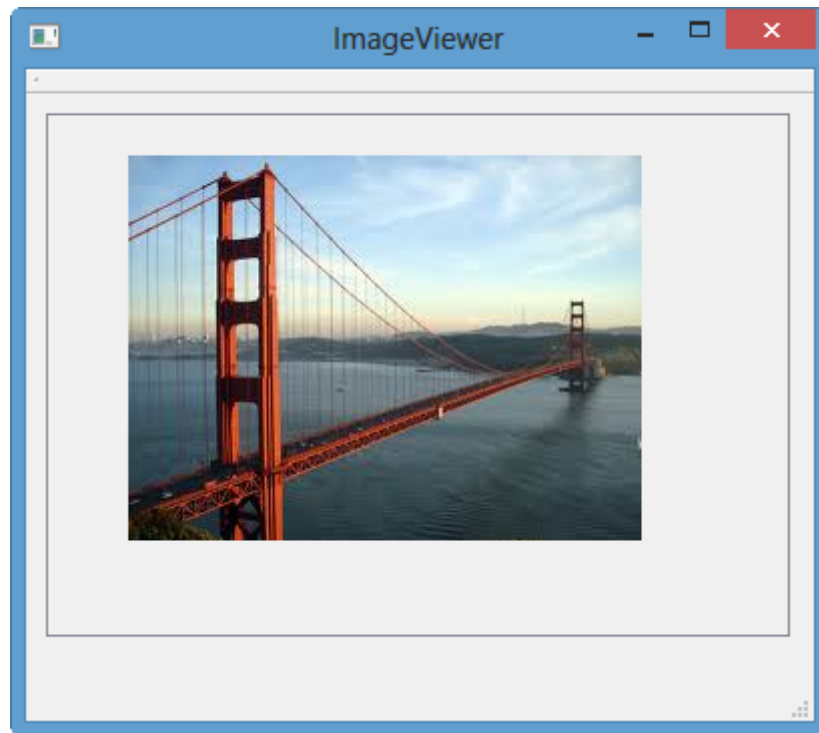
```
ImageViewer::ImageViewer(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::ImageViewer)
{
    ui->setupUi(this);

    QImage image("C:/TEST/GoldenGate.png");
    ui->imageLabel->setPixmap(QPixmap::fromImage(image));
}
```

Of course, we need the two pointers to QScroll and QLabel in **imageviewer.h**:

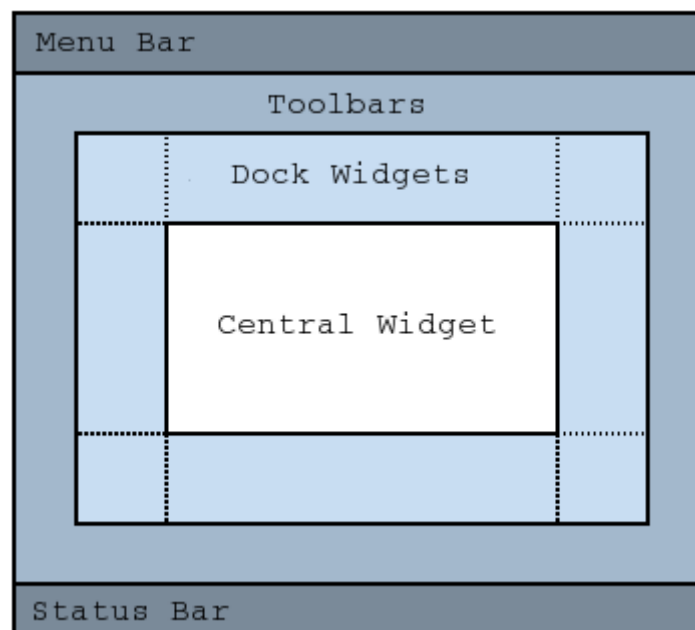
```
private:
    Ui::ImageViewer *ui;
    QLabel *imageLabel;
    QScrollArea *scrollArea;
```

When we run the code, we get:



Now, we know what to do.

1. Menus to select an image
2. To do that, we need a kind of file open dialog
3. Also, we may want to play with the image such as zoom-in/out
4. If possible, we want print the image



More work on the Scroll area and the Label

This time, we set up the QScrollArea and the QLabel using code but not Designer.

```
// imageview.cpp

#include "imageviewer.h"
#include "ui_imageviewer.h"

ImageViewer::ImageViewer(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::ImageViewer)
{
    ui->setupUi(this);

    imageLabel = new QLabel;
    imageLabel->setBackgroundRole(QPalette::Base);
    imageLabel->setSizePolicy(QSizePolicy::Ignored, QSizePolicy::Ignored);
    imageLabel->setScaledContents(true);

    scrollArea = new QScrollArea;
    scrollArea->setBackgroundRole(QPalette::Dark);
    scrollArea->setWidget(imageLabel);
    setCentralWidget(scrollArea);

    setWindowTitle(tr("Image Viewer"));
    resize(500, 400);
}

ImageViewer::~ImageViewer()
{
    delete ui;
}
```

The **setBackgroundRole()** method let us use a colour role for the background, which means one of the predefined colour of the style applied to the widget.

The **setSizePolicy()** sets the size policy to policy. The size policy describes how the item should grow horizontally and vertically when arranged in a layout. We set **imageLabel**'s size policy to ignored, making the users able to scale the image to whatever size they want when the **Fit to Window** option is turned on. Otherwise, the default size policy (preferred) will make scroll bars appear when the scroll area becomes smaller than the label's minimum size hint.

The **setScaledContents(bool)** decides whether the label will scale its contents to fill all available space. When enabled and the label shows a pixmap, it will scale the pixmap to fill the available space. This property's default is false. Here, we ensure that the label will scale its contents to fill all available space, to enable the image to scale properly when zooming. If we omitted to set the **imageLabel**'s **scaledContents** property, zooming in would enlarge the QLabel, but leave the pixmap at its original size, exposing the QLabel's background.

The **setCentralWidget(ui->scrollArea)** make the scrollArea as a central widget of MainWindow.

Building Menus

With the Image Viewer application, the users can view an image of their choice. The File menu gives the user the possibility to:

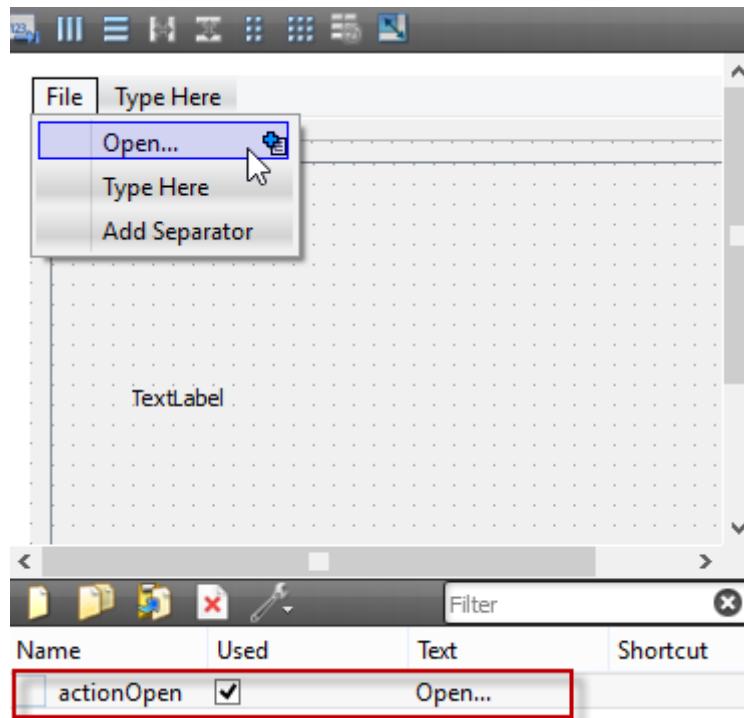
1. Open... - Open an image file
2. Print... - Print an image
3. Exit - Exit the application

Once an image is loaded, the View menu allows the users to:

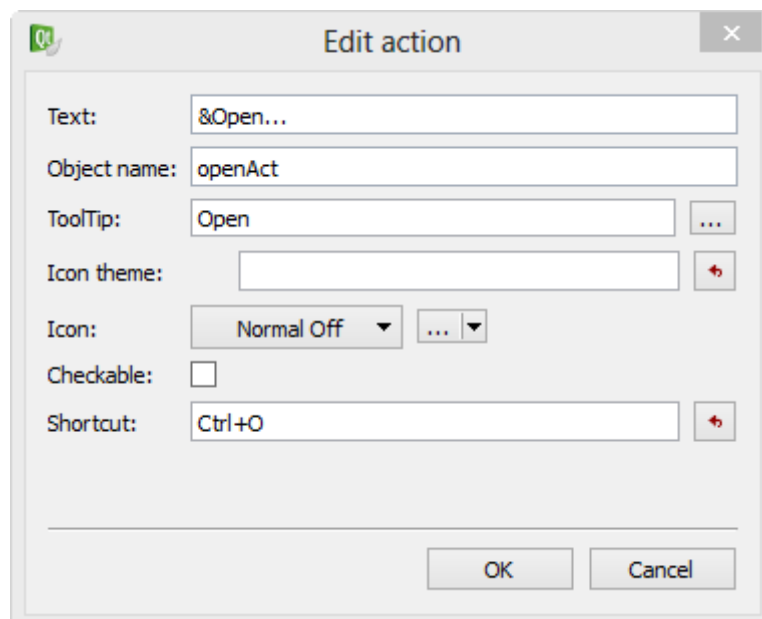
1. Zoom In - Scale the image up by 25%
2. Zoom Out - Scale the image down by 25%
3. Normal Size - Show the image at its original size
4. Fit to Window - Stretch the image to occupy the entire window

In addition we may have the Help menu, and About to provide the users with information about the Image Viewer example

Let's open up **Designer** by double-clicking the **imageviewer.ui** to put menu bar and actions. Type in "File" at the top menu, and "Open..." under the "File" menu. We need to press Enter to make a real change in the menu. Notice that at the bottom, in the Action Editor, a new action called **actionOpen** has been created.



Let's change the object name to **openAct**, and make a shortcut (we should type in real key combination, not typing in each character) as shown in the picture below:



Note that we can get the "Edit action" window either by double click the item in the Action Editor window or right mouse click on the item, and then select "Edit..."

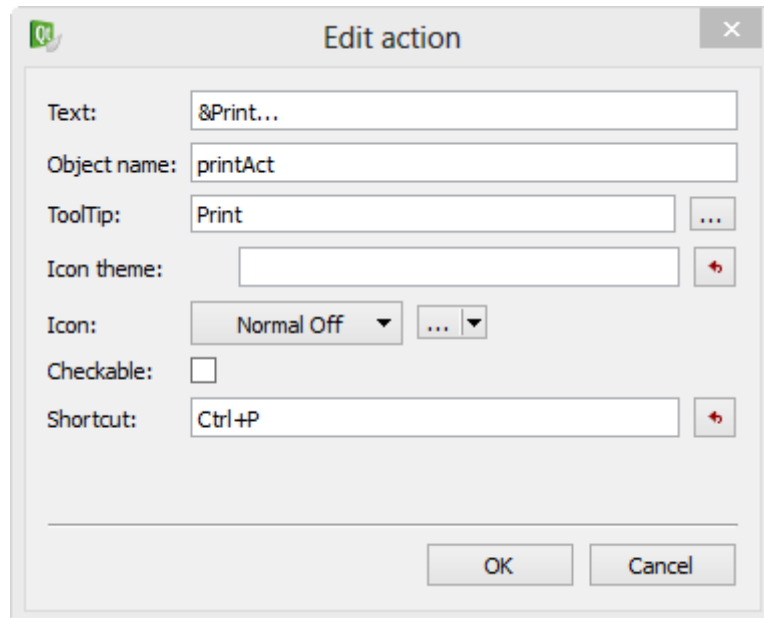
The equivalent code in C++ would be like this:

```
openAct = new QAction(tr("&Open...;"), this);
openAct->setShortcut(tr("Ctrl+O"));
```

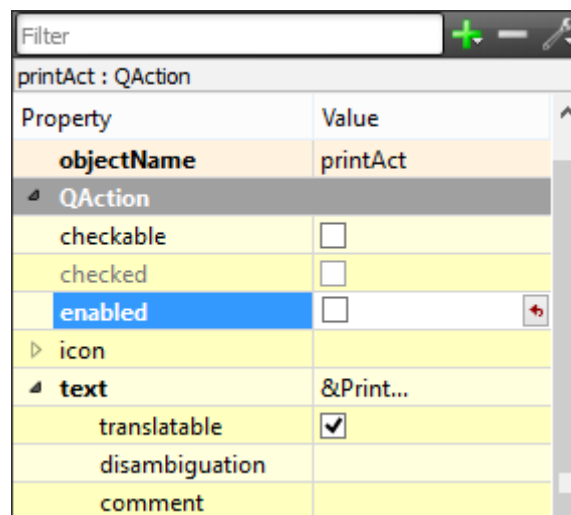
The translation, **tr()**, is set enabled by default.

More work with Creator than just coding!

The next is **Print...**:



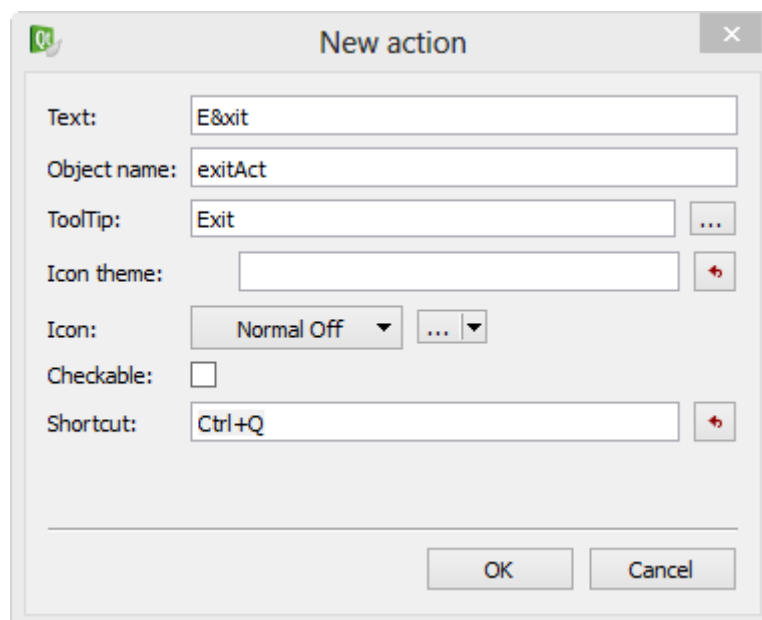
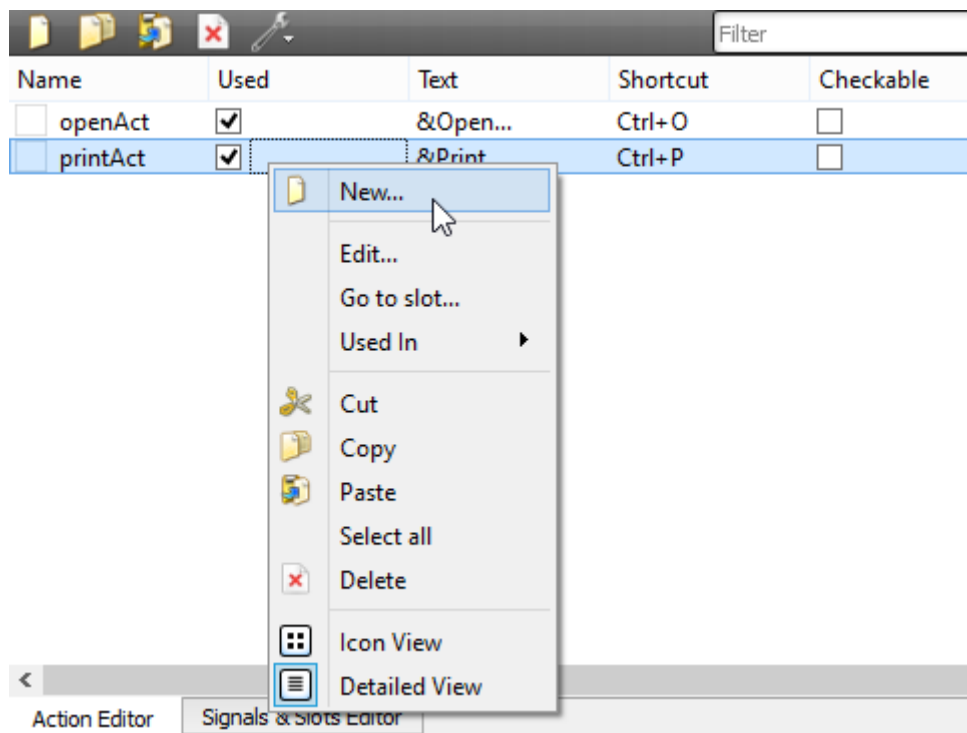
For the Print menu, we need to set it disabled (unchecked) initially. Later, after the image is loaded it will be reset as enabled:



Again, the equivalent code should look like this:

```
printAct = new QAction(tr("&Print...;"), this);
printAct->setShortcut(tr("Ctrl+P"));
printAct->setEnabled(false);
```

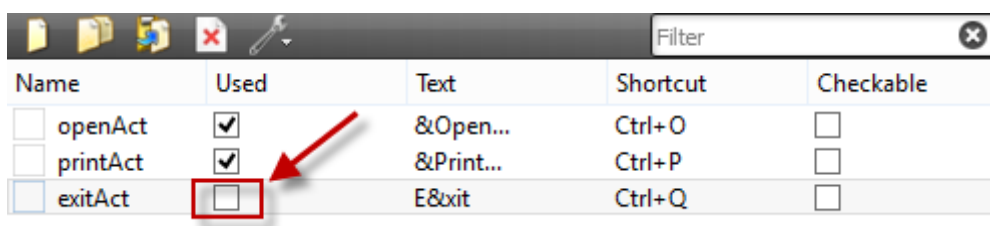
For the **Exit** menu, we open up the "Edit action" pop-up either by clicking the icon(New) or by right-click on the item -> select New... instead of directly double-clicking on the top menu area:



Equivalent code:

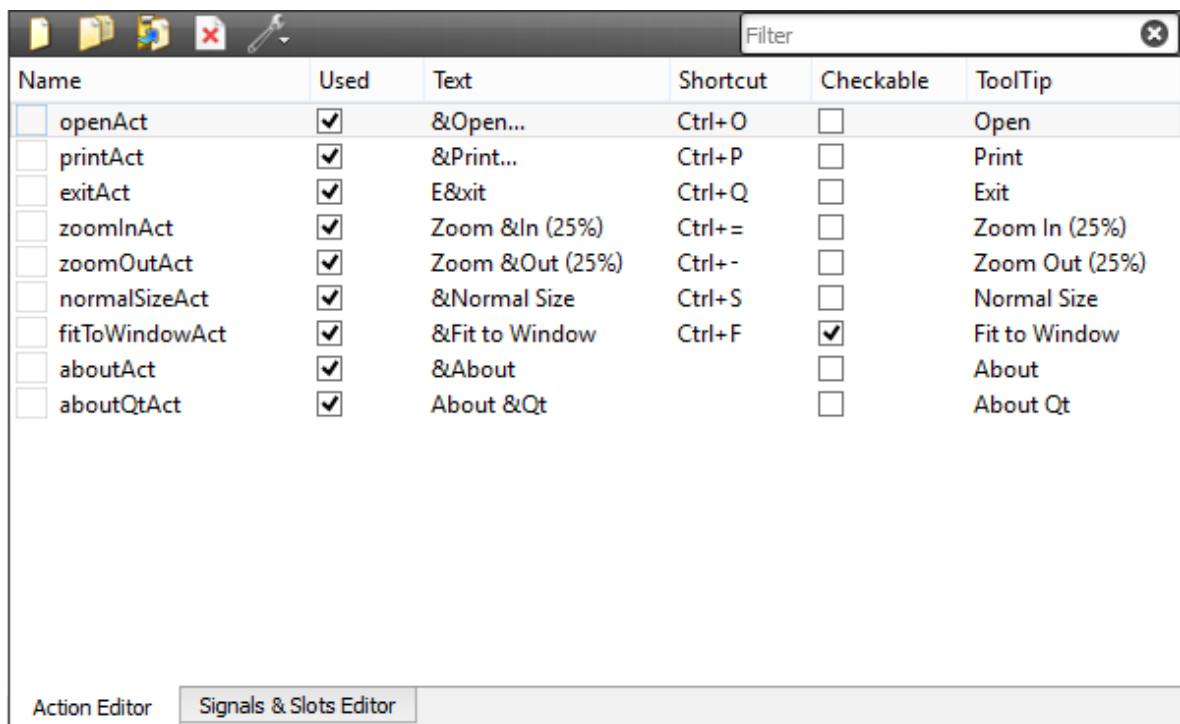
```
exitAct = new QAction(tr("E&xit;"), this);
exitAct->setShortcut(tr("Ctrl+Q"));
```

However, when we run the code and look at the menu, the "Exit" is not in the top menu. We can check carefully the Action Editor again:



It's marked as NOT used for the menu. So, we need to drag the **exitAct**, and then drop it under the top menu's **Print...** item. Then it will be in the top menu items and will be marked as used.

Here is the final version of Action Editor:



Equivalent code should look like this:

```
openAct = new QAction(tr("&Open...;"), this);
openAct->setShortcut(tr("Ctrl+O"));

printAct = new QAction(tr("&Print...;"), this);
printAct->setShortcut(tr("Ctrl+P"));
printAct->setEnabled(false);

exitAct = new QAction(tr("E&xit;"), this);
exitAct->setShortcut(tr("Ctrl+Q"));

zoomInAct = new QAction(tr("Zoom &In; (25%)"), this);
zoomInAct->setShortcut(tr("Ctrl+=")); // (Ctrl)(+)
zoomInAct->setEnabled(false);

zoomOutAct = new QAction(tr("Zoom &Out; (25%)"), this);
zoomOutAct->setShortcut(tr("Ctrl+-")); // (Ctrl)(-)
zoomOutAct->setEnabled(false);

normalSizeAct = new QAction(tr("&Normal; Size"), this);
normalSizeAct->setShortcut(tr("Ctrl+S"));
normalSizeAct->setEnabled(false);

fitToWindowAct = new QAction(tr("&Fit; to Window"), this);
fitToWindowAct->setEnabled(false);
fitToWindowAct->setCheckable(true);
fitToWindowAct->setShortcut(tr("Ctrl+F"));

aboutAct = new QAction(tr("&About;"), this);

aboutQtAct = new QAction(tr("About &Qt;"), this);
```

Actually the code above is not the same as we've done. We implicitly added Menu Bar and Each item of the Top menu has its submenus. So, more coding (shown below) is needed to achieve the same as we've done using Designer:


```

fileMenu = new QMenu(tr("&File;"), this);
fileMenu->addAction(openAct);
fileMenu->addAction(printAct);
fileMenu->addSeparator();
fileMenu->addAction(exitAct);

viewMenu = new QMenu(tr("&View;"), this);
viewMenu->addAction(zoomInAct);
viewMenu->addAction(zoomOutAct);
viewMenu->addAction(normalSizeAct);
viewMenu->addSeparator();
viewMenu->addAction(fitToWindowAct);

helpMenu = new QMenu(tr("&Help;"), this);
helpMenu->addAction(aboutAct);
helpMenu->addAction(aboutQtAct);

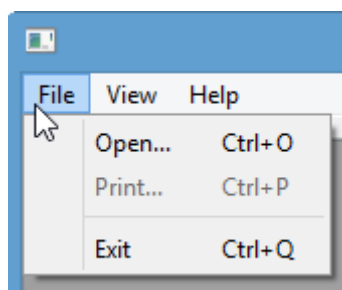
menuBar()->addMenu(fileMenu);
menuBar()->addMenu(viewMenu);
menuBar()->addMenu(helpMenu);

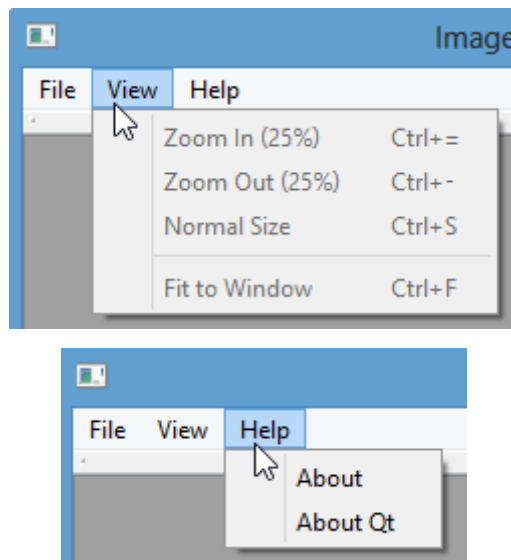
```

Actually, the menu structure looks like this:

Object	Class
▾ menuBar	QMenuBar
▾ fileMenu	QMenu
openAct	QAction
printAct	QAction
separator	QAction
exitAct	QAction
▾ viewMenu	QMenu
zoomInAct	QAction
zoomOutAct	QAction
normalSizeAct	QAction
separator	QAction
fitToWindowAct	QAction
▾ helpMenu	QMenu
aboutAct	QAction
aboutQtAct	QAction
mainToolBar	QToolBar
statusBar	QStatusBar

If we run the code, the menus we've made so far are looks like this:





Signal and Slot Implementation

We only enable the **openAct** and **exitAct** at the time of creation, the others are updated once an image has been loaded into the application. In addition we make the **fitToWindowAct** checkable.

Main Window for ImageViewer B

We'll finish the Actions from the menu we built in the previous tutorial. In other words, we need to connect each QAction to the appropriate slot.

Connecting Actions to Slots

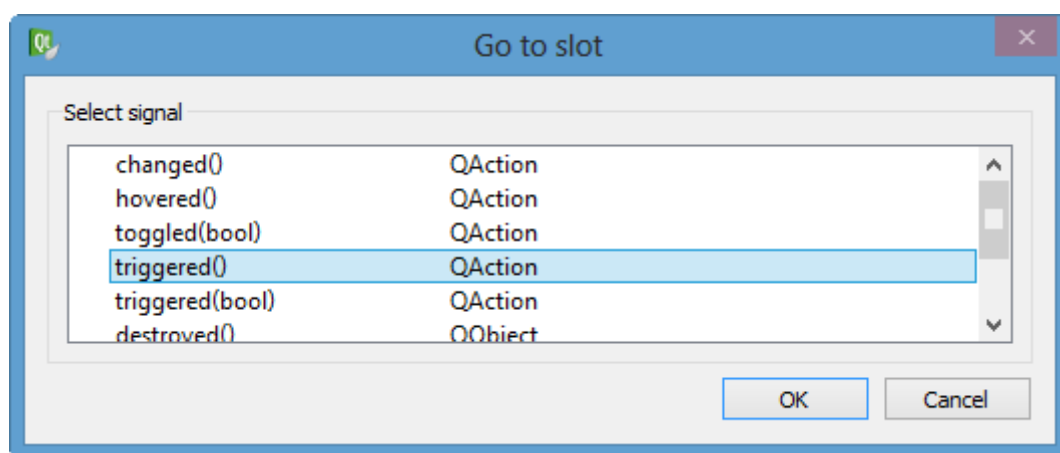
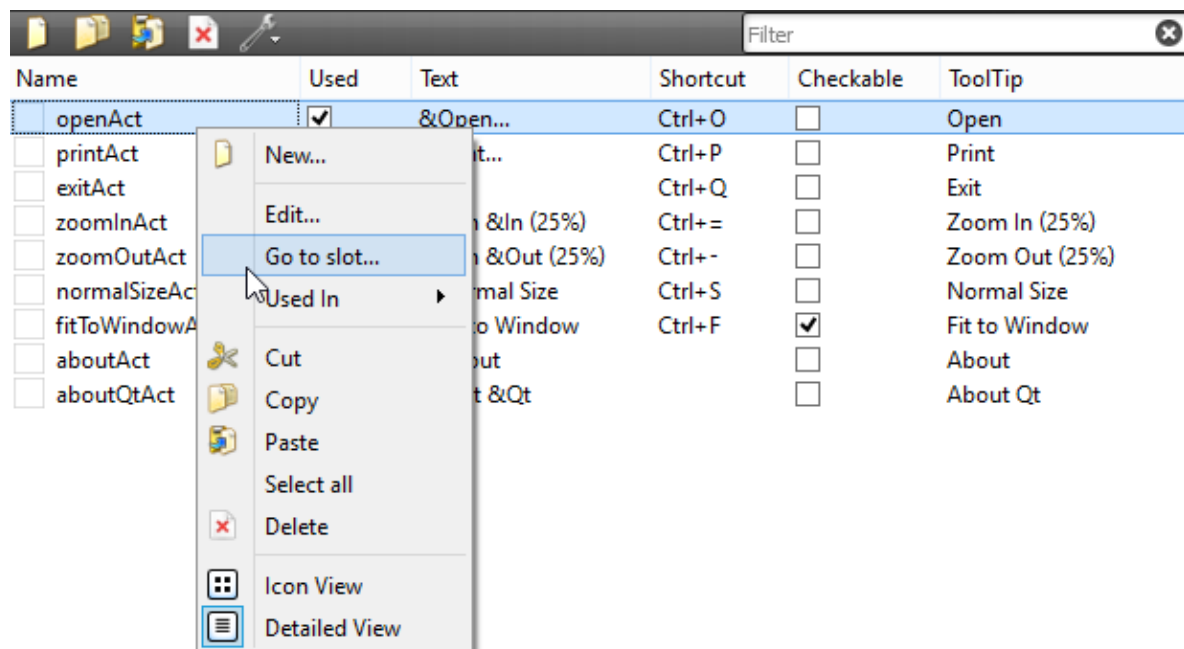
Make Connections

We're going to connect each action to a appropriate slot using Qt Designer. The equivalent code is shown below:

```
connect(openAct, SIGNAL(triggered()), this, SLOT(open()));
connect(printAct, SIGNAL(triggered()), this, SLOT(print()));
connect(exitAct, SIGNAL(triggered()), this, SLOT(close()));
connect(zoomInAct, SIGNAL(triggered()), this, SLOT(zoomIn()));
connect(zoomOutAct, SIGNAL(triggered()), this, SLOT(zoomOut()));
connect(normalSizeAct, SIGNAL(triggered()), this, SLOT(normalSize()));
connect(fitToWindowAct, SIGNAL(triggered()), this, SLOT(fitToWindow()));
connect(aboutAct, SIGNAL(triggered()), this, SLOT(about()));
connect(aboutQtAct, SIGNAL(triggered()), qApp, SLOT(aboutQt()));
```

Let's open up **Designer** by double-clicking the **imageviewer.ui** to connect actions to slots.

Right click on the openAct in the Action Editor, and select "Go to slot..."



Hit OK.

Then, it will generate a code for us into **imageviewer.cpp**:

```
void ImageViewer::on_openAct_triggered()
{
}

```

Designer also writes the slot declaration to the header file, **imageviewer.h**, as well.

```
void on_openAct_triggered();

```

Let's switch the name of the slot to **open()** as shown earlier in the equivalent code. It's not necessary but we just want to have the same code in the Qt Example.

Do the same to all the actions **triggered**.

Also, we need to put the pointers to those actions into **imageviewer.h**:

```

QAction *openAct;
QAction *printAct;
QAction *exitAct;
QAction *zoomInAct;
QAction *zoomOutAct;
QAction *normalSizeAct;
QAction *fitToWindowAct;
QAction *aboutAct;
QAction *aboutQtAct;

```

Pointer Assign

Even though we defined pointers to the widgets or actions of UI in header file as shown above, we haven't assigned them any objects. So, we need to do that now in **imageviewer.cpp**:

```

...
openAct = ui->openAct;
printAct = ui->printAct;
exitAct = ui->exitAct;
zoomInAct = ui->zoomInAct;
zoomOutAct = ui->zoomOutAct;
normalSizeAct = ui->normalSizeAct;
fitToWindowAct = ui->fitToWindowAct;
aboutAct = ui->aboutAct;
aboutQtAct = ui->aboutQtAct;
...

```

In that way, we can use the pointer **openAct->*** instead of **ui->openAct->***.

open()

From this point, the remaining process is almost the same as the original [Qt tutorial](#).

We want to implement the functionality of the **on_actionOpen_triggered()** slot. We changed the name to **open()**.

```

void ImageViewer::open()
{
    qDebug() << "open()";
    QString fileName = QFileDialog::getOpenFileName(this,
                                                    tr("Open File"), QDir::currentPath());

    if (!fileName.isEmpty()) {
        QImage image(fileName);
        if (image.isNull()) {
            QMessageBox::information(this, tr("Image Viewer"),
                                    tr("Cannot load %1.").arg(fileName));

            return;
        }
        imageLabel->setPixmap(QPixmap::fromImage(image));
        scaleFactor = 1.0;

        printAct->setEnabled(true);
        fitToWindowAct->setEnabled(true);
        updateActions();

        if (!fitToWindowAct->isChecked())

```

```
        imageLabel->adjustSize();
    }
}
```

The first step is asking the user for the name of the file to open. Qt comes with **QFileDialog**, which is a dialog from which the user can select a file.

The static **getOpenFileName()** function displays a modal file dialog. It returns the file path of the file selected, or an empty string if the user canceled the dialog. If the file could not be opened, we use **QMessageBox** to display a dialog with an error message. If the user presses Cancel, QFileDialog returns an empty string.

Unless the file name is an empty string, we check if the file's format is an image format by constructing a QImage which tries to load the image from the file. If the constructor returns a null image, we use a QMessageBox to alert the user.

The **QMessageBox** class provides a modal dialog with a short message, an icon, and some buttons. As with QFileDialog the easiest way to create a QMessageBox is to use its static convenience functions. QMessageBox provides a range of different messages arranged along two axes: severity (question, information, warning and critical) and complexity (the number of necessary response buttons). In this particular example an information message with an OK button (the default) is sufficient, since the message is part of a normal operation.

If the format is supported, we display the image in **imageLabel** by setting the label's pixmap. Then we enable the **Print** and **Fit to Window menu** entries and update the rest of the view menu entries. The **Open** and **Exit** entries are enabled by default.

If the Fit to Window option is turned off, the **QScrollArea::widgetResizable** property is false and it is our responsibility (not QScrollArea's) to give the QLabel a reasonable size based on its contents. We call **{QWidget::adjustSize()}adjustSize()** to achieve this, which is essentially the same as

```
imageLabel->resize(imageLabel->pixmap()->size());
```

Zoom

```
void ImageViewer::zoomIn()
{
    scaleImage(1.25);
}

void ImageViewer::zoomOut()
{
    scaleImage(0.8);
}

void ImageViewer::normalSize()
{
    imageLabel->adjustSize();
    scaleFactor = 1.0;
}
```

We implement the zooming slots using the private **scaleImage()** function. We set the scaling factors to 1.25 and 0.8, respectively. These factor values ensure that a Zoom In action and a Zoom Out action will cancel each other (since $1.25 * 0.8 == 1$), and in that way the normal image size can be restored using the zooming features.

When zooming, we use the QLabel's ability to scale its contents. Such scaling doesn't change the actual size hint of the contents. And since the **adjustSize()** function uses those size hints, the only thing we need to do to restore the normal size of the currently displayed image is to call **adjustSize()** and reset the scale factor to 1.0.

```
void ImageViewer::fitToWindow()
{
    bool fitToWindow = fitToWindowAct->isChecked();
    scrollArea->setWidgetResizable(fitToWindow);
    if (!fitToWindow) {
        normalSize();
    }
    updateActions();
}
```

The **fitToWindow()** slot is called each time the user toggled the Fit to Window option. If the slot is called to turn on the option, we tell the scroll area to resize its child widget with the **QScrollArea::setWidgetResizable()** function. Then we disable the Zoom In, Zoom Out and Normal Size menu entries using the private **updateActions()** function.

If the **QScrollArea::widgetResizable** property is set to false (the default), the scroll area honors the size of its child widget. If this property is set to true, the scroll area will automatically resize the widget in order to avoid scroll bars where they can be avoided, or to take advantage of extra space. But the scroll area will honor the minimum size hint of its child widget independent of the widget resizable property. So in this example we set **imageLabel**'s size policy to ignore in the constructor, to avoid that scroll bars appear when the scroll area becomes smaller than the label's minimum size hint.

If the slot is called to turn off the option, the **{QScrollArea::setWidgetResizable}** property is set to false. We also restore the image pixmap to its normal size by adjusting the label's size to its content. And in the end we update the view menu entries.

Updating Actions

```
void ImageViewer::updateActions()
{
    zoomInAct->setEnabled(!fitToWindowAct->isChecked());
    zoomOutAct->setEnabled(!fitToWindowAct->isChecked());
    normalSizeAct->setEnabled(!fitToWindowAct->isChecked());
}
```

The private **updateActions()** function enables or disables the Zoom In, Zoom Out and Normal Size menu entries depending on whether the Fit to Window option is turned on or off.

```

void ImageViewer::scaleImage(double factor)
{
    Q_ASSERT(imageLabel->pixmap());
    scaleFactor *= factor;
    imageLabel->resize(scaleFactor * imageLabel->pixmap()->size());

    adjustScrollBar(scrollArea->horizontalScrollBar(), factor);
    adjustScrollBar(scrollArea->verticalScrollBar(), factor);

    zoomInAct->setEnabled(scaleFactor < 3.0);
    zoomOutAct->setEnabled(scaleFactor > 0.333);
}

```

In **scaleImage()**, we use the factor parameter to calculate the new scaling factor for the displayed image, and resize imageLabel. Since we set the **scaledContents** property to true in the constructor, the call to **QWidget::resize()** will scale the image displayed in the label. We also adjust the scroll bars to preserve the focal point of the image.

At the end, if the scale factor is less than 33.3% or greater than 300%, we disable the respective menu entry to prevent the image pixmap from becoming too large, consuming too much resources in the window system.

```

void ImageViewer::adjustScrollBar(QScrollBar *scrollBar, double factor)
{
    scrollBar->setValue(int(factor * scrollBar->value()
                          + ((factor - 1) * scrollBar->pageStep()/2)));
}

```

Whenever we zoom in or out, we need to adjust the scroll bars in consequence. It would have been tempting to simply call

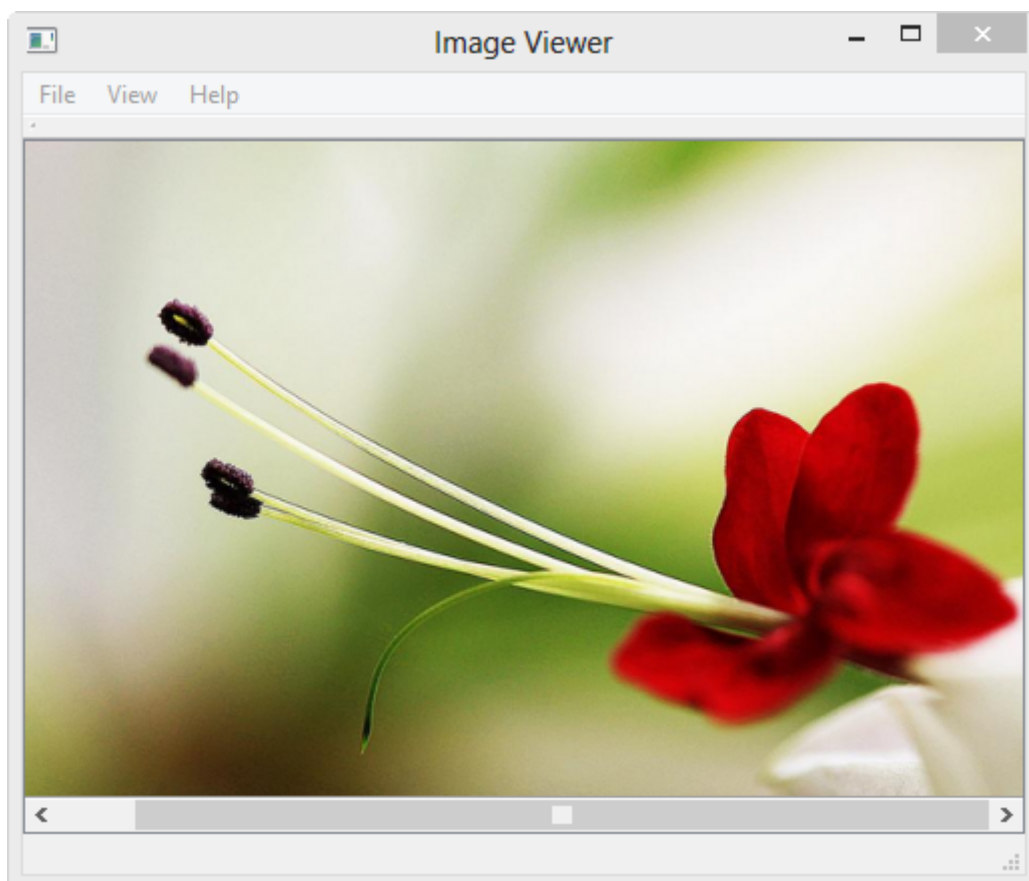
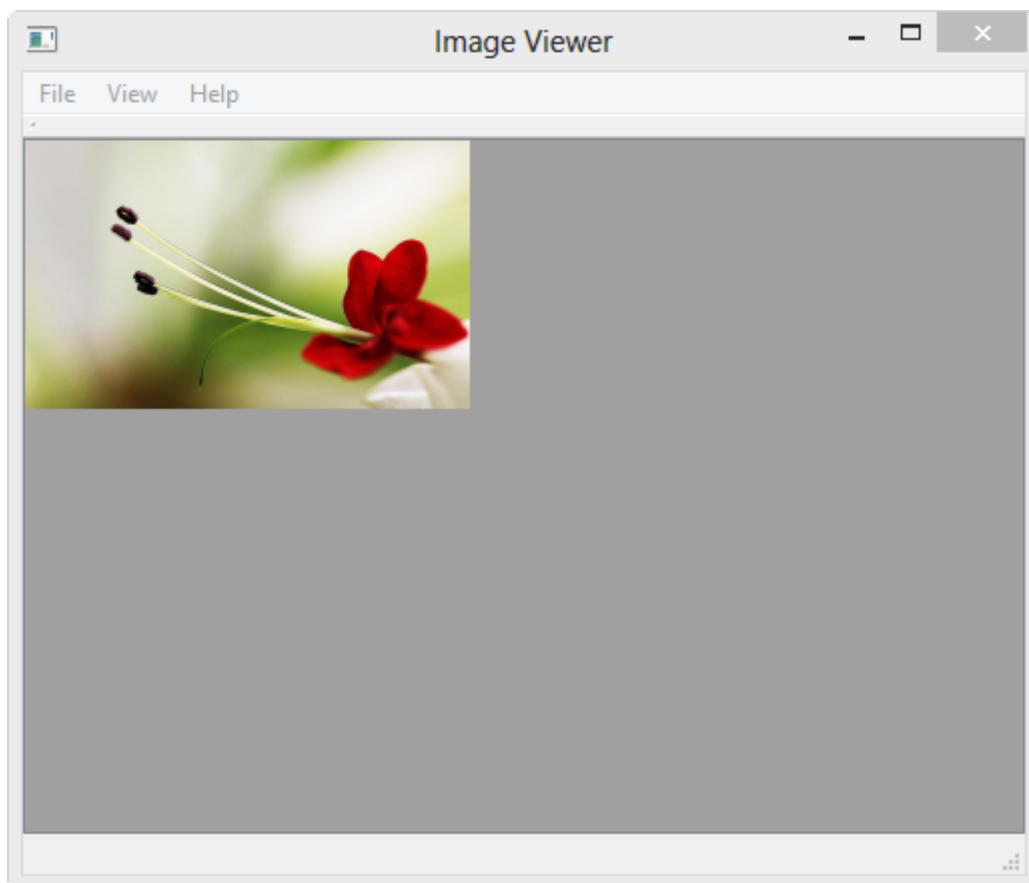
```

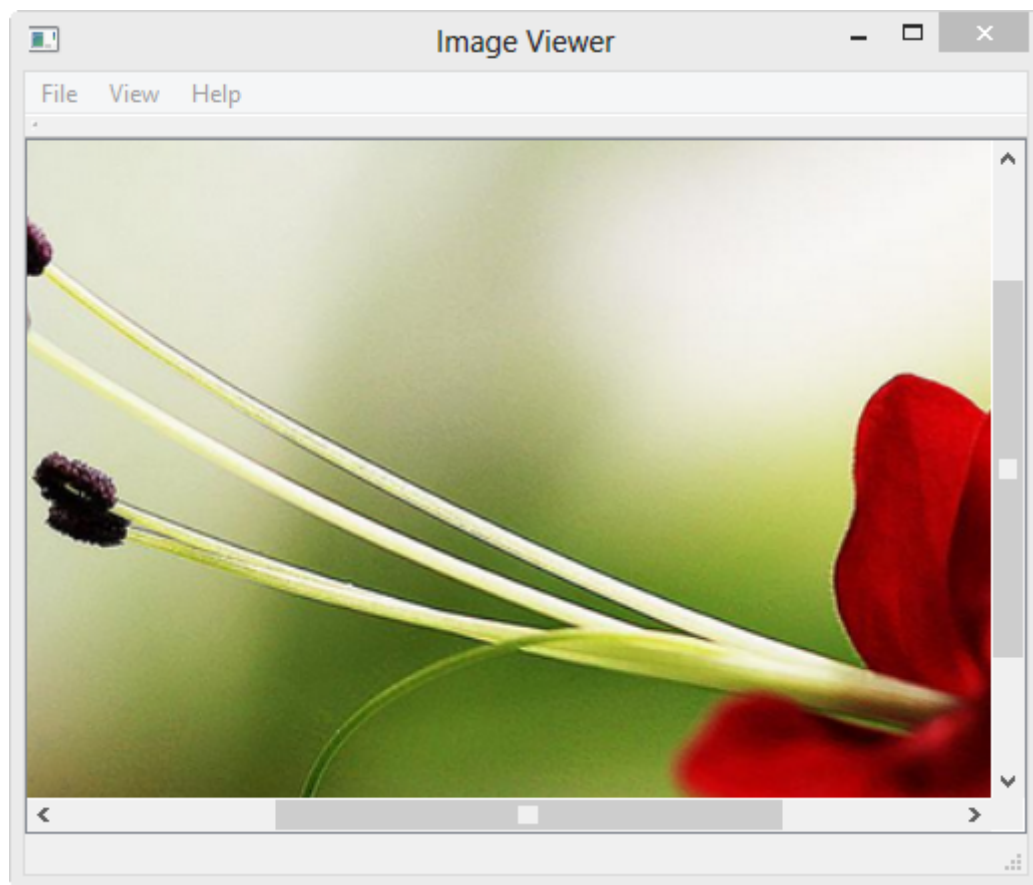
scrollBar->setValue(int(factor * scrollBar->value()));

```

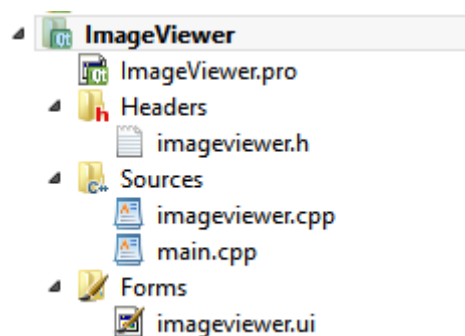
but this would make the top-left corner the focal point, not the center. Therefore we need to take into account the scroll bar handle's size (the page step).

Running the code





Source code



1. [ImageViewer.pro](#)
2. [main.cpp](#)
3. [imageviewer.ui](#)
4. [imageviewer.h](#)
5. [imageviewer.cpp](#)

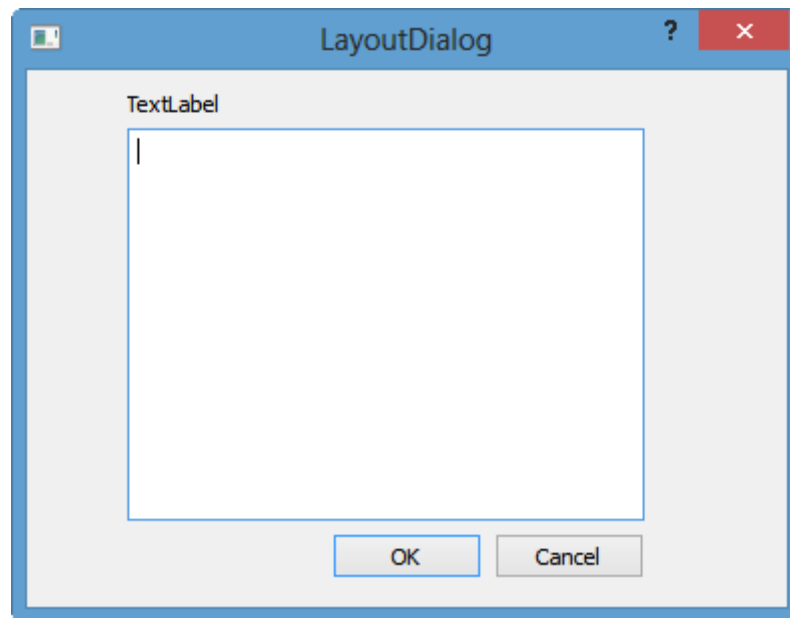
Layouts

In this tutorial, we will learn Layouts of Qt. We will use Designer rather than doing it programmatically.

File->New File or Project...

Applications->Qt Gui Application->Choose...

We will work with QDialog.

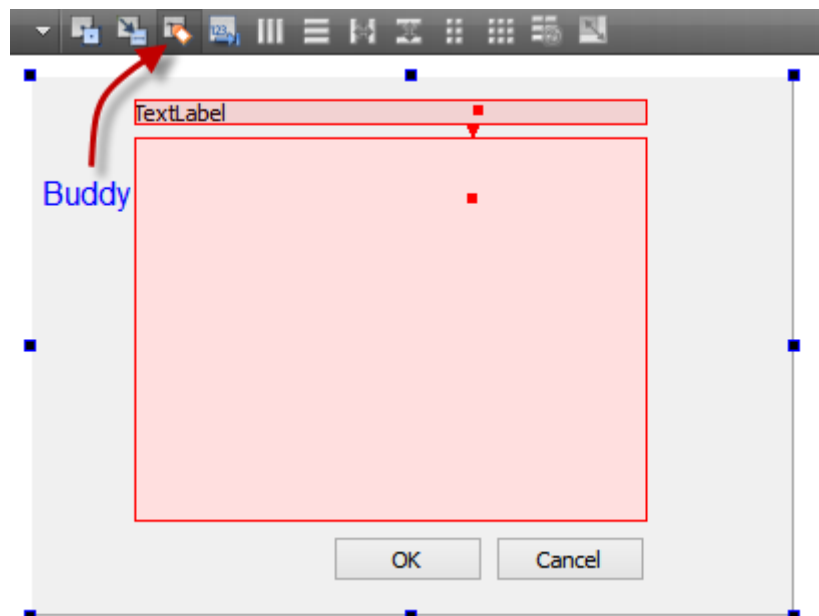


Note that we can do set layouts by typing the code directly.

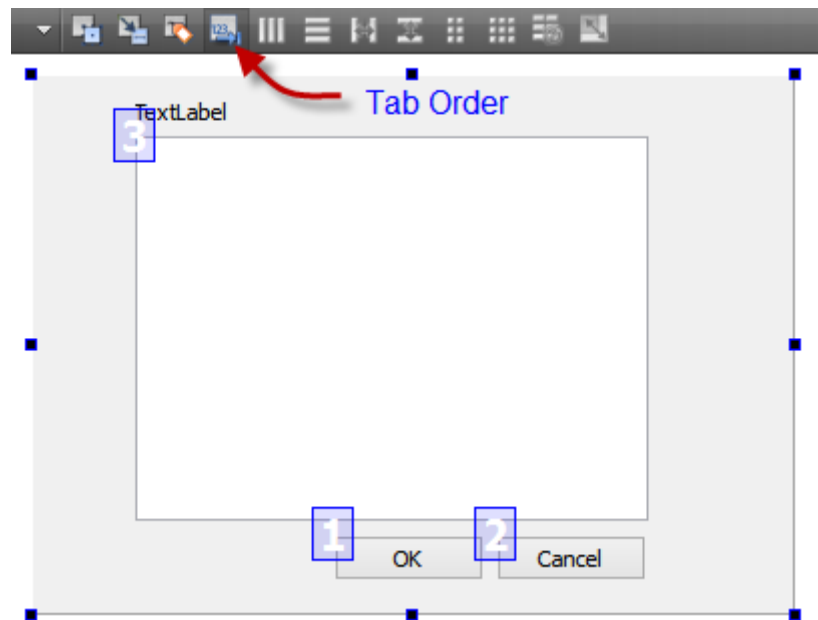
Buddies and Tab Order

What's buddies?

It is a way of linking widgets in terms of input controls. For example, in our case, if our label **TextLabel** can transfer focus to **TextEdit** if the Buddies has been setup between them.



We can also set or change Tab Orders.

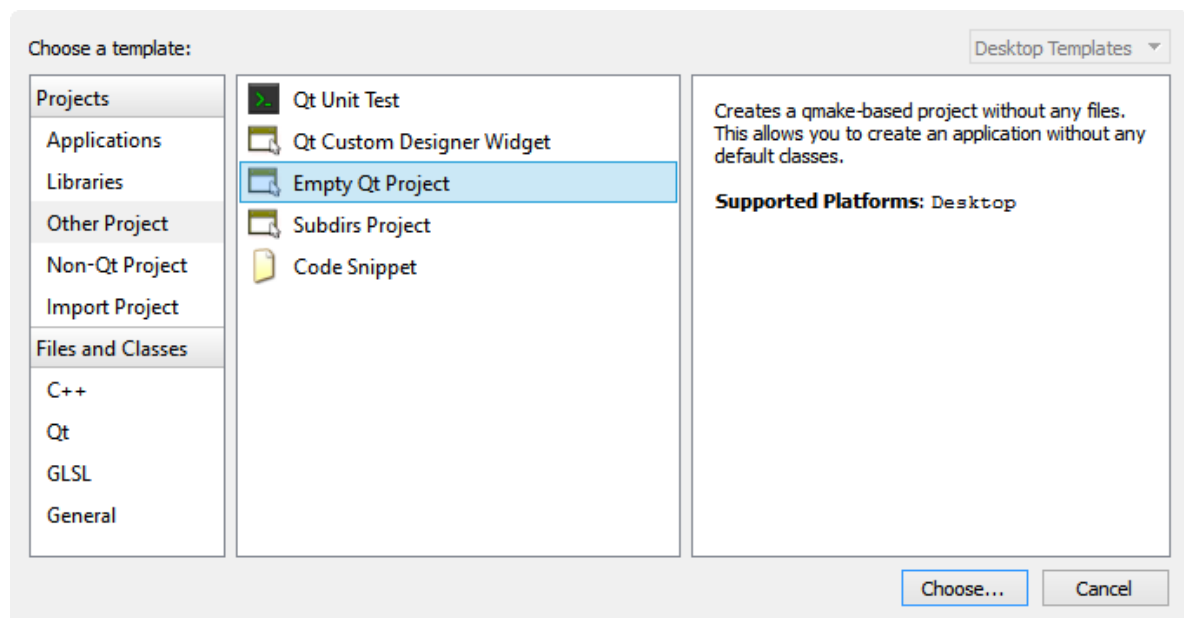


Layouts without Designer

In this tutorial, we will learn Layouts of Qt. We will add layouts to a form and add widgets to the layout programmatically instead of using Designer as was done in the [Layouts](#).

We will create an empty project and will create 6 PushButtons. Then, we will arrange 3 buttons with horizontal layout, and the other three with vertical layout. The two layouts will be added to the outer most layout, and then we pass the outer layout to our widget using **setLayout()**.

File->Other Project->Empty Qt Project



There is none in the Project explorer, and even the **.pro** has nothing in it. So, we need to create a file **main.cpp** by Add New...

```
#include <QApplication>
#include <QPushButton>
#include <QHBoxLayout>
#include <QVBoxLayout>

int main(int argc, char* argv[])
{
```

```

QApplication app(argc, argv);

// Horizontal layout with 3 buttons
QHBoxLayout *hLayout = new QHBoxLayout;
QPushButton *b1 = new QPushButton("A");
QPushButton *b2 = new QPushButton("B");
QPushButton *b3 = new QPushButton("C");
hLayout->addWidget(b1);
hLayout->addWidget(b2);
hLayout->addWidget(b3);

// Vertical layout with 3 buttons
QVBoxLayout *vLayout = new QVBoxLayout;
QPushButton *b4 = new QPushButton("D");
QPushButton *b5 = new QPushButton("E");
QPushButton *b6 = new QPushButton("F");
vLayout->addWidget(b4);
vLayout->addWidget(b5);
vLayout->addWidget(b6);

// Outer Layer
QVBoxLayout *mainLayout = new QVBoxLayout;

// Add the previous two inner layouts
mainLayout->addLayout(hLayout);
mainLayout->addLayout(vLayout);

// Create a widget
QWidget *w = new QWidget();

// Set the outer layout as a main layout
// of the widget
w->setLayout(mainLayout);

// Window title
w->setWindowTitle("layouts");

// Display
w->show();

// Event loop
return app.exec();
}

```

We also need to edit the **.pro** file:

```

QT += core gui
QT += widgets
SOURCES += \
    main.cpp

```

Note that we could have taken out the line:

```

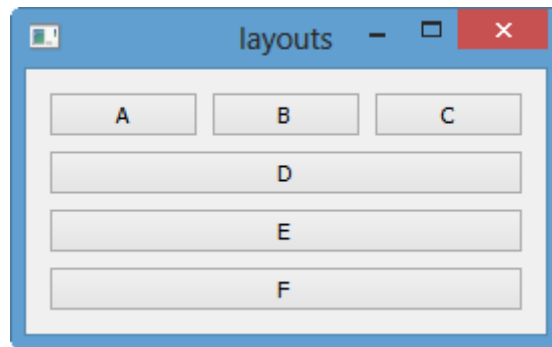
QT += core gui

```

and instead, we could have added include statements in the **main.cpp**:

```
#include <QtCore>
#include <QtGui>
```

Run the code:



Grid Layout

In this tutorial, we will learn Grid Layout of Qt. We will modify the code we used in the previous tutorial, [Layouts without Designer](#).

Grid Layout provides a way of dynamically arranging items in a grid.

If the GridLayout is resized, all items in the layout will be rearranged. It is similar to the widget-based QGridLayout. All children of the GridLayout element will belong to the layout. If you want a layout with just one row or one column, you can use the RowLayout or ColumnLayout. These offers a bit more convenient API, and improves readability.

By default items will be arranged according to the flow property. The default value of the flow property is GridLayout.LeftToRight.

If the columns property is specified, it will be treated as a maximum limit of how many columns the layout can have, before the auto-positioning wraps back to the beginning of the next row. The columns property is only used when flow is GridLayout.LeftToRight.

Here is our new file **main.cpp**:

```
#include <QApplication>
#include <QtCore>
#include <QtGui>
#include <QPushButton>
#include <QGridLayout>

int main(int argc, char* argv[])
{
    QApplication app(argc, argv);

    // Grid layout with 3 buttons
    QGridLayout *gridLayout = new QGridLayout;
    QPushButton *b1 = new QPushButton("A");
    QPushButton *b2 = new QPushButton("B");
    QPushButton *b3 = new QPushButton("C");
    QPushButton *b4 = new QPushButton("D");
    QPushButton *b5 = new QPushButton("E");
    QPushButton *b6 = new QPushButton("F");

    // addWidget(*Widget, row, column, rowspan, colspan)
```

```

// 0th row
gridLayout->addWidget(b1,0,0,1,1);
gridLayout->addWidget(b2,0,1,1,1);
gridLayout->addWidget(b3,0,2,1,1);

// 1st row
gridLayout->addWidget(b4,1,0,1,1);

// 2nd row with 2-column span
gridLayout->addWidget(b5,2,0,1,2);

// 3rd row with 3-column span
gridLayout->addWidget(b6,3,0,1,3);

// Create a widget
QWidget *w = new QWidget();

// Set the grid layout as a main layout
w->setLayout(gridLayout);

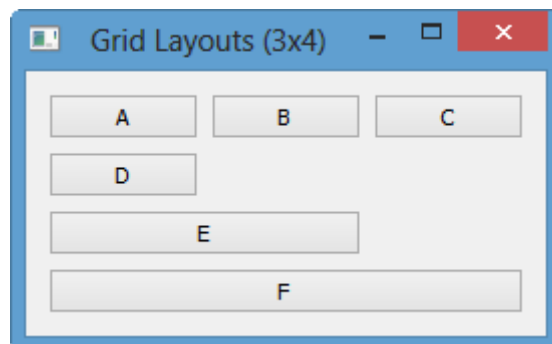
// Window title
w->setWindowTitle("Grid Layouts (3x4)");

// Display
w->show();

// Event loop
return app.exec();
}

```

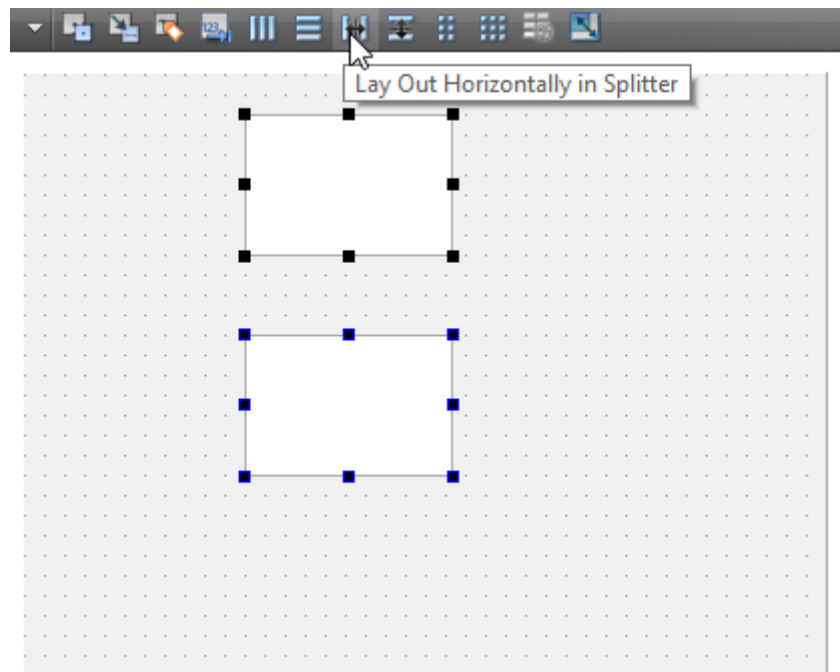
Run the code:



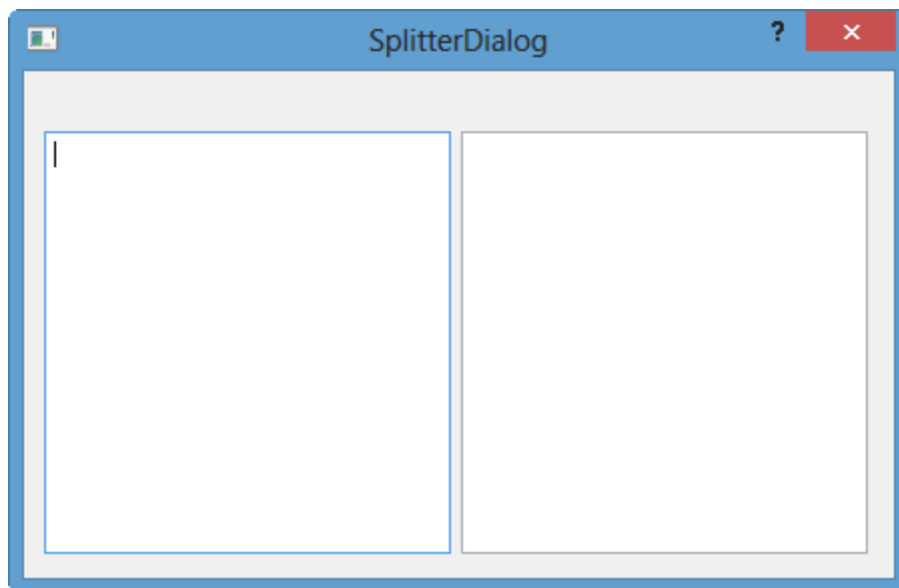
Splitter

In this tutorial, we will learn Splitter.

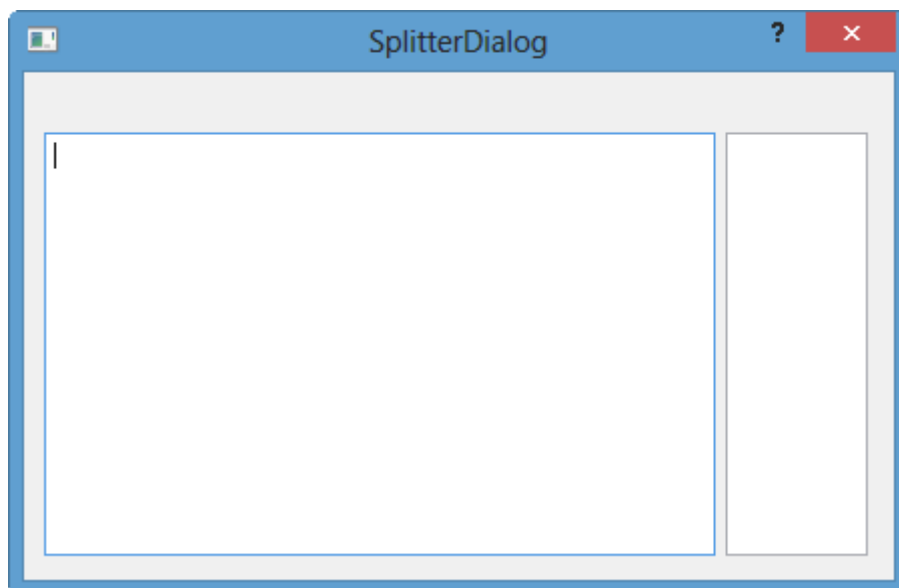
The dialog has two PlainTextEdits. We can split the two widgets in two ways: horizontally or vertically. Below is the Horizontally split example:



Run the code, we get:



We can move the splitter left and right by dragging it:



QDir

In this tutorial, we will learn QDir.

The QDir class provides access to directory structures and their contents.

A QDir is used to manipulate path names, access information regarding paths and files, and manipulate the underlying file system. It can also be used to access Qt's resource system.

Qt uses "/" as a universal directory separator in the same way that "/" is used as a path separator in URLs. If you always use "/" as a directory separator, Qt will translate your paths to conform to the underlying operating system.

QDir::exists()

The following code checks if the two directories exist. In this example, the first one does, but the second one does not:

```
#include <QCoreApplication>
#include <QDir>
#include <QDebug>

int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);

    // forward slash for directory separator
    QDir dir1("C:/Qt");
    QDir dir2("C:/Qt/test");

    // output: true false
    qDebug() << dir1.exists() << dir2.exists();

    return a.exec();
}
```

QFileInfoList QDir::drives()

QFileInfoList QDir::drives() returns a list of the root directories on this system.

On Windows this returns a list of QFileInfo objects containing "C:/", "D:/", etc. On other operating systems, it returns a list containing just one root directory (i.e. "/").

```
#include <QCoreApplication>
#include <QDir>
#include <QDebug>

int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);

    // forward slash for directory separator
    QDir dir1("C:/Qt");
    QDir dir2("C:/Qt/test");
    qDebug() << dir1.exists() << dir2.exists();
}
```

```

QDir dir3;
foreach(QFileInfo item, dir3.drives() )
{
    qDebug() << item.absoluteFilePath();
}

return a.exec();
}

```

Output:

```

true false
"C:/"
"D:/"
"E:/"

```

bool QDir::mkpath()

bool QDir::mkpath(const QString & dirPath) const creates the directory path dirPath.

The function will create all parent directories necessary to create the directory.

mkpath() returns true if successful; otherwise returns false.

If the path already exists when this function is called, it will return true.

The code below checks if a directory exists. In this example, a new directory will be created because it does not exist. If we run it again, the code will give a message telling the directory already exists.

```

#include <QCoreApplication>
#include <QDir>
#include <QDebug>
#include <QString>

int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);

    QString path = "C:/Qt/test";

    QDir dir(path);
    if(!dir.exists())
    {
        qDebug() << "Creating " << path << "directory";
        dir.mkpath(path);
    }
    else
    {
        qDebug() << path << " already exists";
    }

    return a.exec();
}

```

If we run the code, we get the output:

```
Creating "C:/Qt/test" directory
```

If we run it again, we get the following message:

```
"C:/Qt/test" already exists
```

QFileInfoList QDir::entryInfoList()

QFileInfoList QDir::entryInfoList(const QStringList & nameFilters, Filters filters = NoFilter, SortFlags sort = NoSort) const returns a list of QFileInfo objects for all the files and directories in the directory, ordered according to the name and attribute filters previously set with setNameFilters() and setFilter(), and sorted according to the flags set with setSorting().

The name filter, file attribute filter, and sorting specification can be overridden using the nameFilters, filters, and sort arguments.

entryInfoList() returns an empty list if the directory is unreadable, does not exist, or if nothing matches the specification.

Here is the code using QDir::entryInfoList():

```
#include <QCoreApplication>
#include <QDir>
#include <QDebug>
#include <QString>

int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);

    QString path = "C:/Qt";
    QDir dir(path);

    foreach(QFileInfo item, dir.entryInfoList() )
    {
        if(item.isDir())
            qDebug() << "Dir: " << item.absoluteFilePath();
        if(item.isFile())
            qDebug() << "File: " << item.absoluteFilePath();
    }

    return a.exec();
}
```

Output:

```
Dir: "C:/Qt"
Dir: "C:/"
...
Dir: "C:/Qt/build-QDir-Desktop_Qt_5_1_0_MSVC2012_OpenGL_64bit-Debug"
...
File: "C:/Qt/FileA.txt"
File: "C:/Qt/FileB.txt"
'''
Dir: "C:/Qt/test"
```

QFile

In this tutorial, we will learn **QFile**.

The QFile class provides an interface for reading from and writing to files.

QFile is an I/O device for reading and writing text and binary files and resources. A QFile may be used by itself or, more conveniently, with a QTextStream or QDataStream.

The file name is usually passed in the constructor, but it can be set at any time using setFileName(). QFile expects the file separator to be '/' regardless of operating system. The use of other separators (e.g., '\') is not supported.

We can check for a file's existence using exists(), and remove a file using remove().

QTextStream takes care of converting the 8-bit data stored on disk into a 16-bit Unicode QString. By default, it assumes that the user system's local 8-bit encoding is used (e.g., UTF-8 on most unix based operating systems). This can be changed using setCodec().

To write text, we can use operator<<(), which is overloaded to take a QTextStream on the left and various data types (including QString) on the right.

In the following code, we open a new file with the given name and write a text into the file. Then, we read in the file back and print the content to our console.

```
#include <QCoreApplication>
#include <QFile>
#include <QString>
#include <QDebug>
#include <QTextStream>

void write(QString filename)
{
    QFile file(filename);
    // Trying to open in WriteOnly and Text mode
    if(!file.open(QFile::WriteOnly |
                  QFile::Text))
    {
        qDebug() << " Could not open file for writing";
        return;
    }

    // To write text, we use operator<<(),
    // which is overloaded to take
    // a QTextStream on the left
    // and data types (including QString) on the right

    QTextStream out(&file);
    out << "QFile Tutorial";
    file.flush();
    file.close();
}

void read(QString filename)
{
    QFile file(filename);
    if(!file.open(QFile::ReadOnly |
```

```

        QFile::Text))
    {
        qDebug() << " Could not open the file for reading";
        return;
    }

    QTextStream in(&file);
    QString myText = in.readAll();
    qDebug() << myText;

    file.close();
}

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);

    QString filename = "C:/Qt/MyFile.txt";
    write(filename);
    read(filename);

    return a.exec();
}

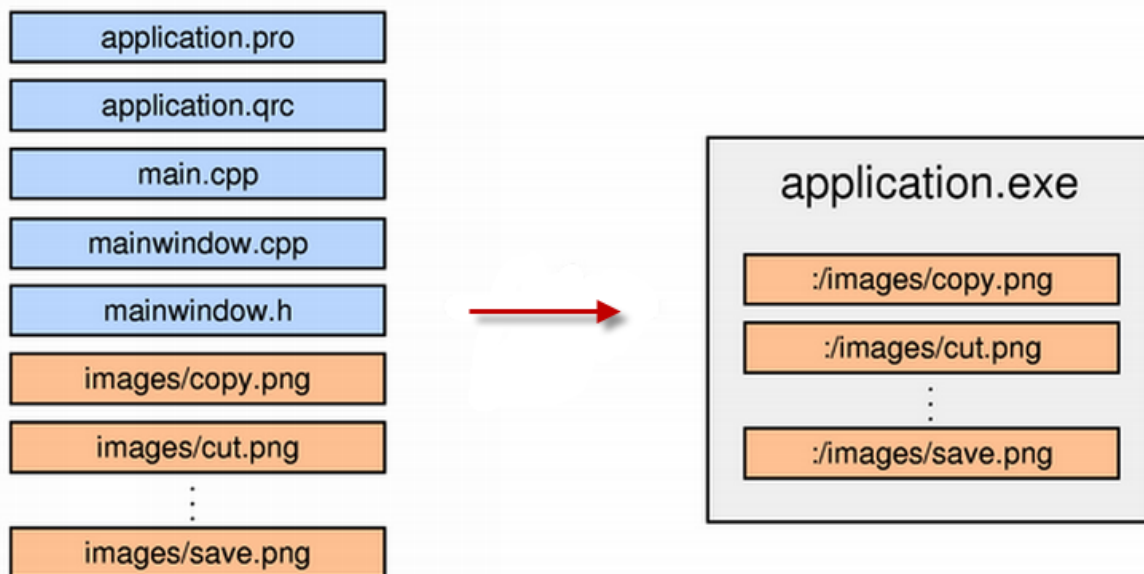
```

Output:

```
"QFile Tutorial"
```

Qt Resource Files

In this tutorial, we will learn **Qt Resource Files**. We'll set our .pro file as a resource, and then in the run time, we'll read it back and display it into the console.



Picture source: [The Qt Resource System](#).

As we see in the picture, we (application.exe) can access the resources during the run time.

The Qt resource system is a platform-independent mechanism for storing binary files in the application's executable. This is useful if our application always needs a certain set of files (icons, translation files, etc.) and we don't want to run the risk of losing the files.

The resource system is based on tight cooperation between qmake, rcc (Qt's resource compiler), and [QFile](#).

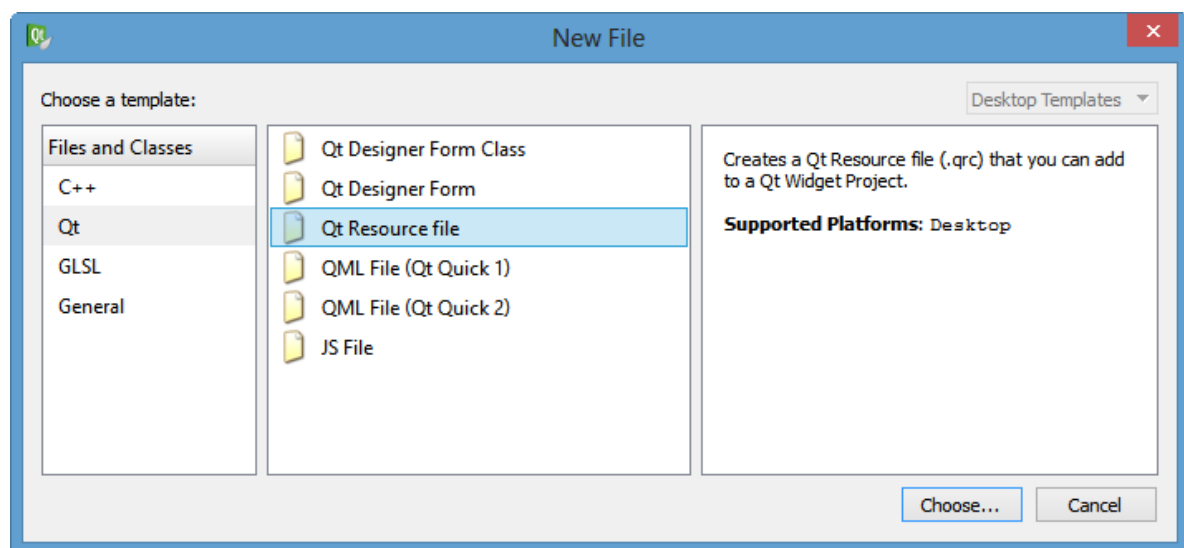
The resources associated with an application are specified in a .qrc file, an XML-based file format that lists files on the disk and optionally assigns them a resource name that the application must use to access the resource.

Here's an example .qrc file:

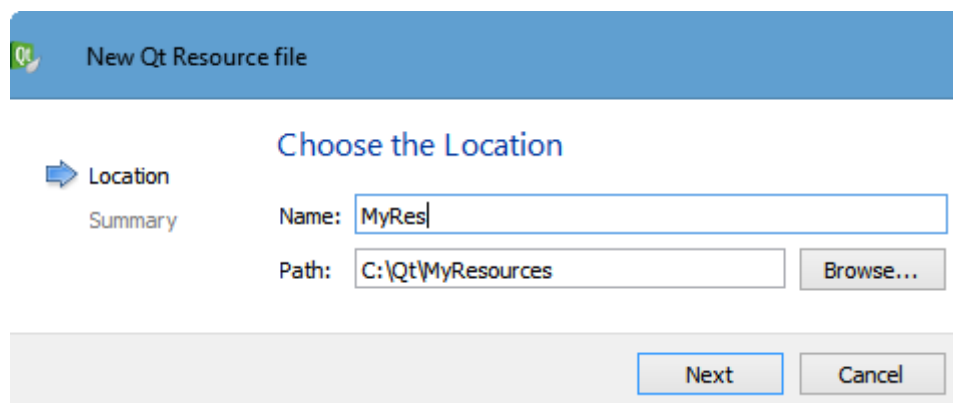
```
<!DOCTYPE RCC><RCC version="1.0">
<qresource>
  <file>images/copy.png</file>
  <file>images/cut.png</file>
  <file>images/new.png</file>
  <file>images/open.png</file>
  <file>images/paste.png</file>
  <file>images/save.png</file>
</qresource>
</RCC>
```

Let's make our resource file.

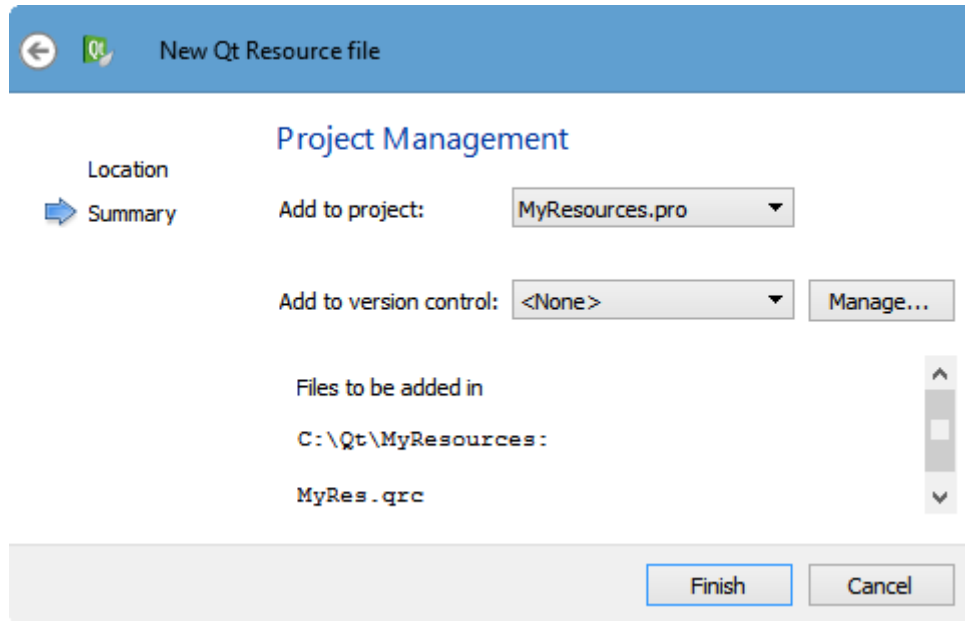
Right click on the Project->Add New...



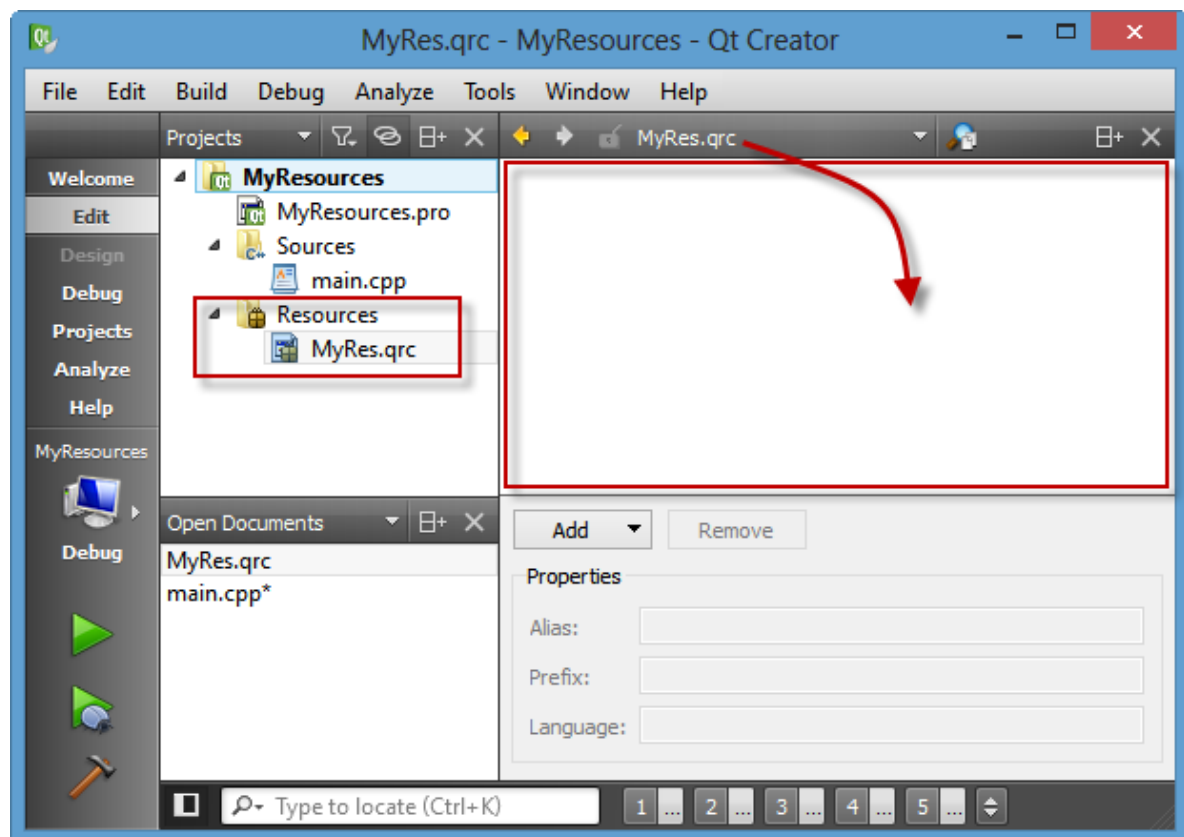
Choose...



Hit Next

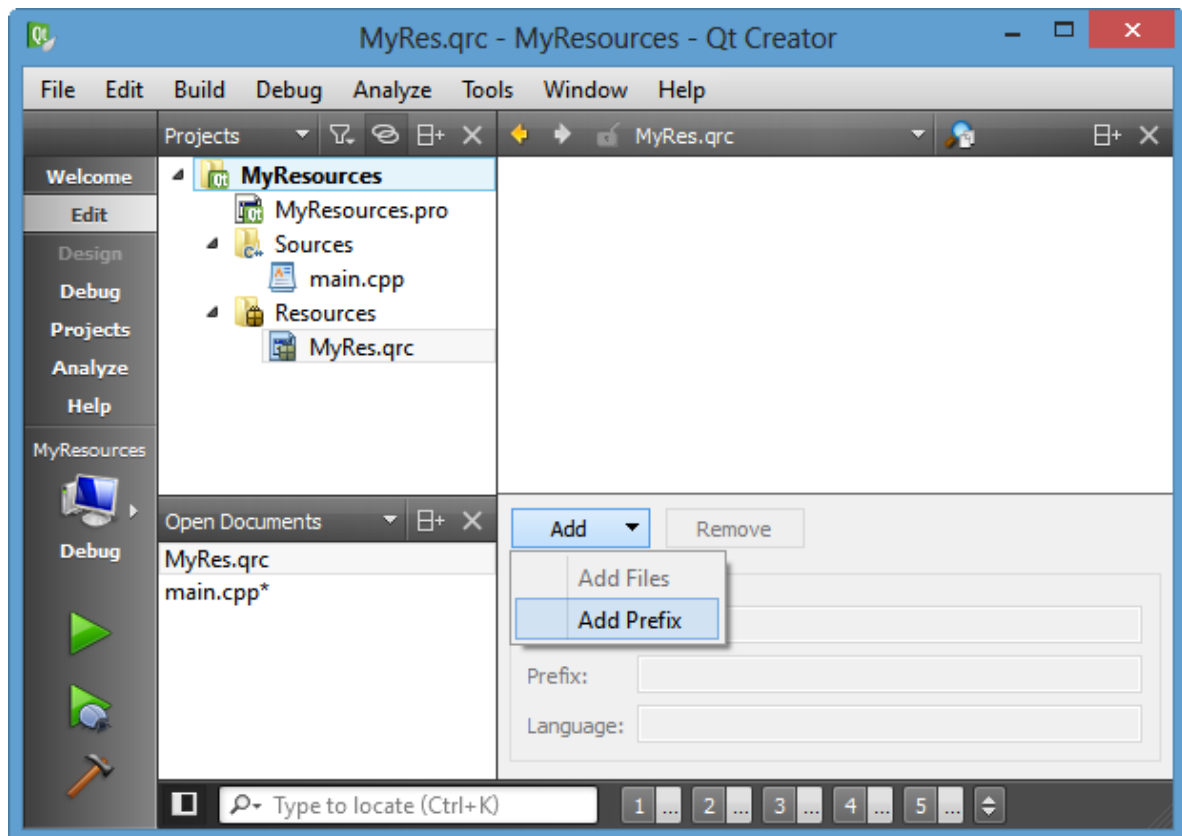


Finish. Then, the Designer will provide us with resource editor.

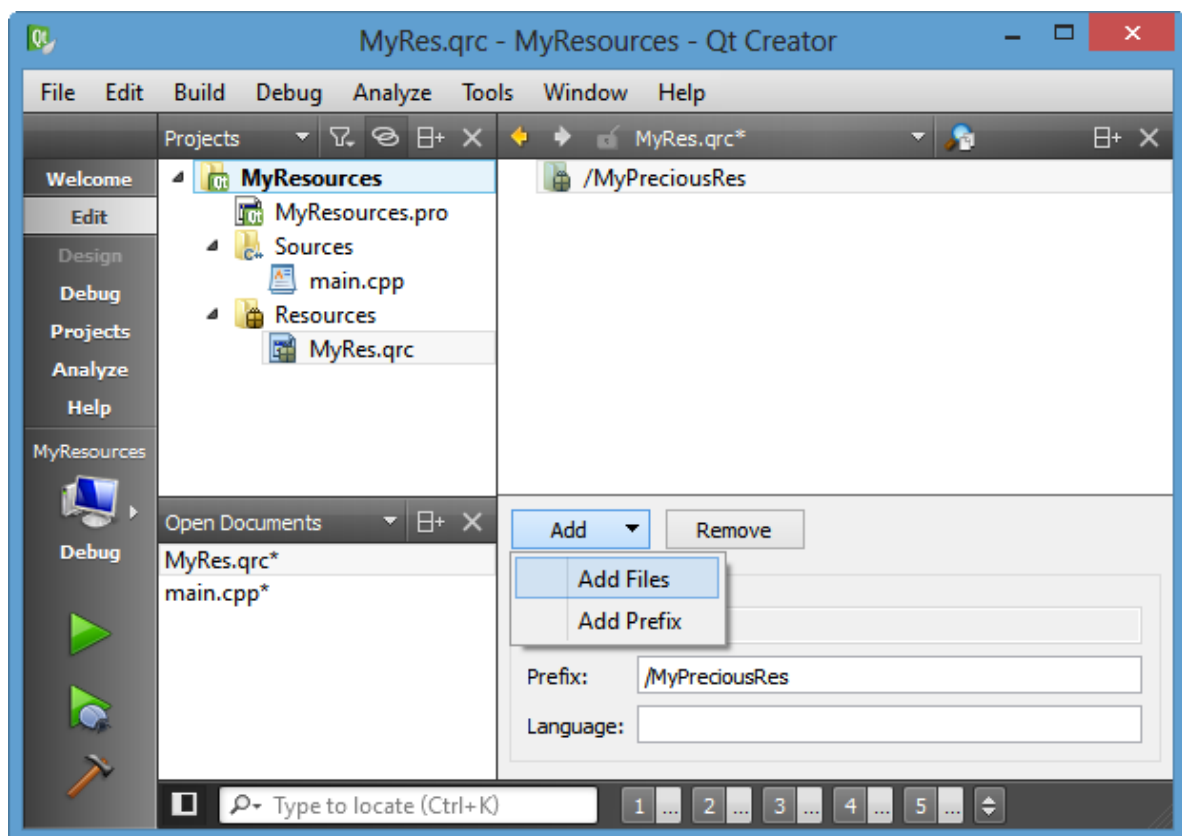


Let's make our resource file.

Add->Add Prefix

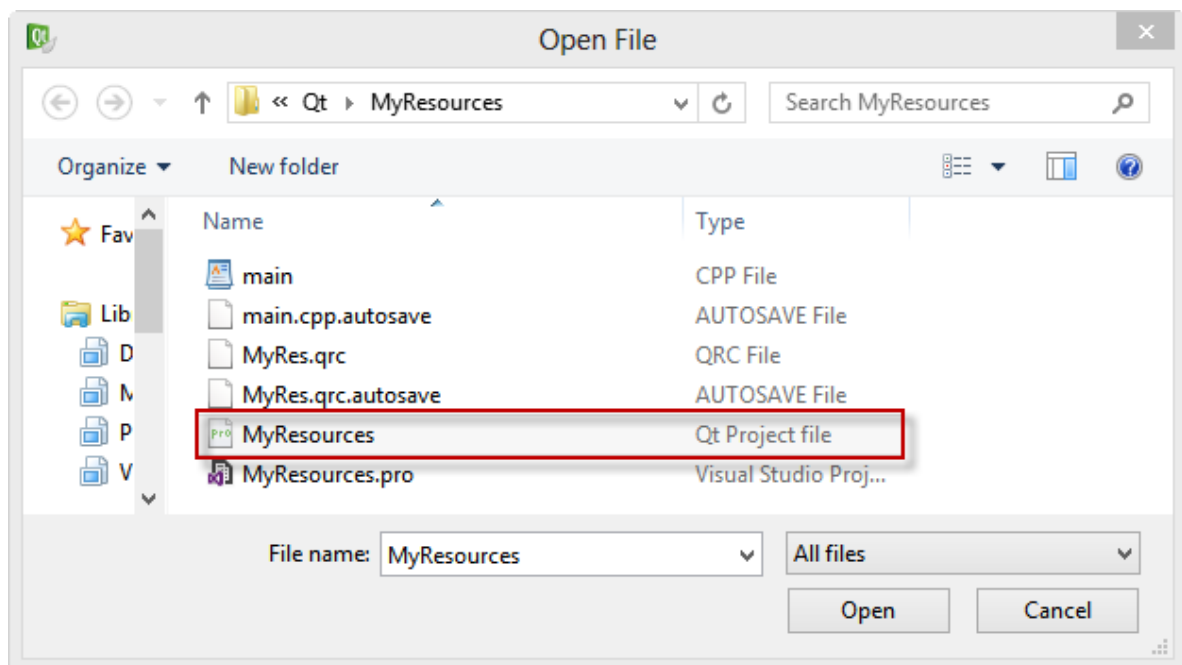


We can change the name but keep forward slash('/'). Then, Add->Add Files.

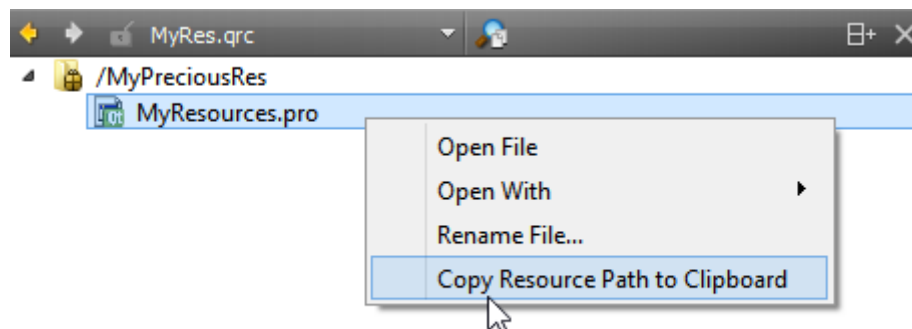


We'll use our **.pro** file as a resource.

We can change the name but keep forward slash('/'). Then, Add->Add Files.



We can put the path to the resource by right click on the file into a clipboard. Then, we can retrieve it from the clipboard.



In the code below, we set our .pro file as a resource. In the run time, we'll read it back and display it into the console. Here is the code:

```
#include <QCoreApplication>
#include <QFile>
#include <QString>
#include <QDebug>
#include <QTextStream>

void read(QString filename)
{
    QFile file(filename);
    if(!file.open(QFile::ReadOnly |
                  QFile::Text))
    {
        qDebug() << " Could not open the file for reading";
        return;
    }

    QTextStream in(&file);
    QString myText = in.readAll();

    // put QString into qDebug stream
    qDebug() << myText;
```

```

        file.close();
    }

    int main(int argc, char *argv[])
    {
        QCoreApplication a(argc, argv);

        read(":/MyPreciousRes/MyResources.pro");

        return a.exec();
    }

```

Run and we get the output which is our .pro file:

```

"#-----
#
# Project created by QtCreator 2013-09-13T11:02:43
#
#-----

QT      += core

QT      -= gui

TARGET = MyResources
CONFIG  += console
CONFIG  -= app_bundle

TEMPLATE = app

SOURCES += main.cpp

RESOURCES += \
    MyRes.qrc
"

```

Note the lines highlighted. Our **.pro** added lines for the resource!

QComboBox

In this tutorial, we will learn **QComboBox**.

Qt->Qt Gui Application:

Qt Gui Application

Class Information

Specify basic information about the classes for which you want to generate skeleton source code files.

Location
Kits
Details
Summary

Class name:

Base class:

Header file:

Source file:

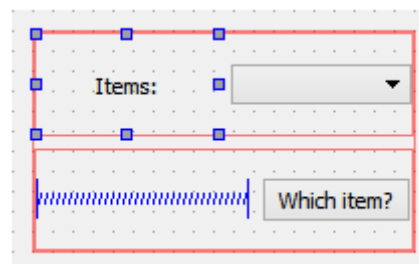
Generate form: ☒

Form file:

Next Cancel

Let's make the UI as shown in the picture below.

Forms->comboboxdialog.ui:



Here are the codes.

ComboDialog.cpp:

```
#include "comboboxdialog.h"
#include "ui_comboboxdialog.h"

ComboBoxDialog::ComboBoxDialog(QWidget *parent) :
    QDialog(parent),
    ui(new Ui::ComboBoxDialog)
{
    ui->setupUi(this);

    // Fill the items of the ComboBox
    for(int i = 0; i < 10; i++)
    {
        ui->comboBox->addItem("item " + QString::number(i));
    }
}

ComboBoxDialog::~ComboBoxDialog()
{
    delete ui;
}
```

```

}

// Message popup when the button is clicked
void ComboBoxDialog::on_pushButton_clicked()
{
    QMessageBox::information(this, "Item Selection",
                             ui->comboBox->currentText());
}

```

ComboDialog.h:

```

#ifndef COMBOBOXDIALOG_H
#define COMBOBOXDIALOG_H

#include <QDialog>
#include <QMessageBox>

namespace Ui {
class ComboBoxDialog;
}

class ComboBoxDialog : public QDialog
{
    Q_OBJECT

public:
    explicit ComboBoxDialog(QWidget *parent = 0);
    ~ComboBoxDialog();

private slots:
    void on_pushButton_clicked();

private:
    Ui::ComboBoxDialog *ui;
};

#endif // COMBOBOXDIALOG_H

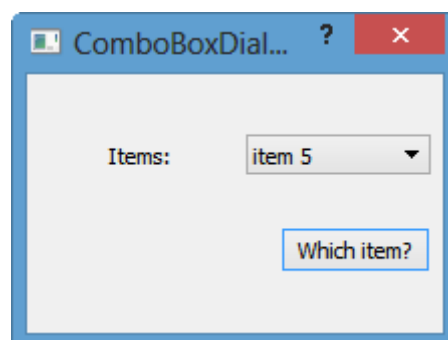
```

First, we populated the items of the QComboBox.

```

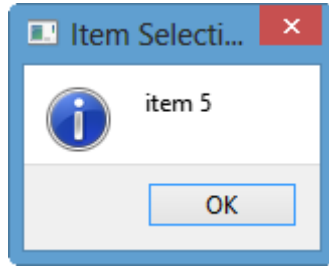
ui->comboBox->addItem("item " + QString::number(i));

```



Then, when we click the button, a message box will popup to give information about the item that's currently selected. This message is not needed because the combo is already displaying the current item. But it's there for getting used to the QMessageBox as well as signals/slots.

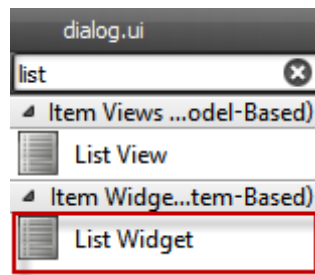
```
QMessageBox::information(this, "Item Selection",  
                        ui->comboBox->currentText());
```



QListWidget

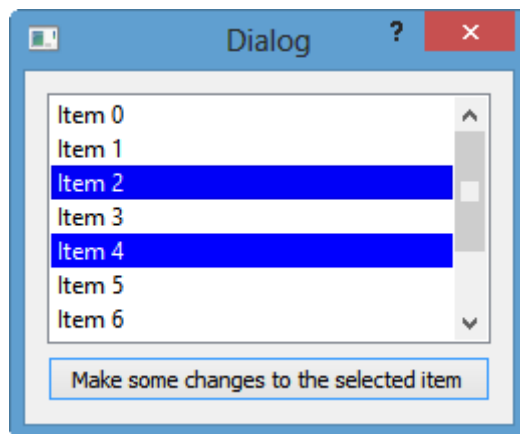
In this tutorial, we will learn **QListWidget**.

There is a big difference between **QListWidget** and **QListView**:



The **QListView** uses MVC, and will be covered in later section.

In this tutorial, we'll populate the items of the widget list and then when a button is clicked, it will make some changes on the currently selected item (text color and item background color).



Here is the code: **dialog.cpp**:

```
#include "dialog.h"  
#include "ui_dialog.h"  
  
Dialog::Dialog(QWidget *parent) :  
    QDialog(parent),  
    ui(new Ui::Dialog)  
{  
    ui->setupUi(this);  
  
    // populate the items of the list  
    for(int i = 0; i < 10; i++)  
    {
```

```

        ui->listWidget->addItem("Item " + QString::number(i));
    }
}

Dialog::~Dialog()
{
    delete ui;
}

void Dialog::on_pushButton_clicked()
{
    // Get the pointer to the currently selected item.
    QListWidgetItem *item = ui->listWidget->currentItem();

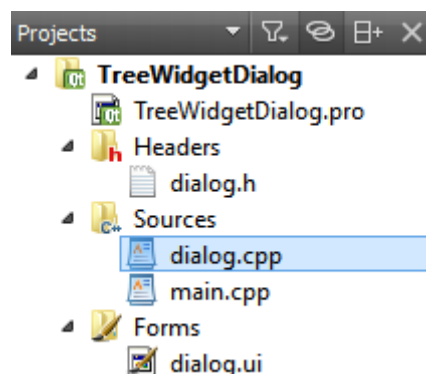
    // Set the text color and its background color using the pointer to the
    item.
    item->setTextColor(Qt::white);
    item->setBackgroundColor(Qt::blue);
}

```

Note that we set the color of the text and background color after we get the `ListWidgetItem`: by calling `QListWidgetItem * QListWidget::currentItem() const`.

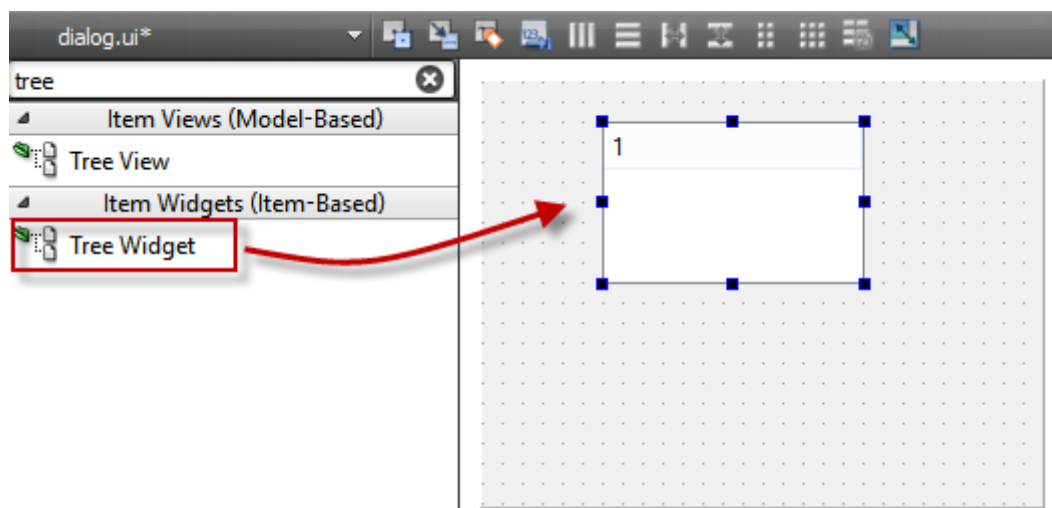
QTreeWidget

In this tutorial, we will learn **QTreeWidget**.

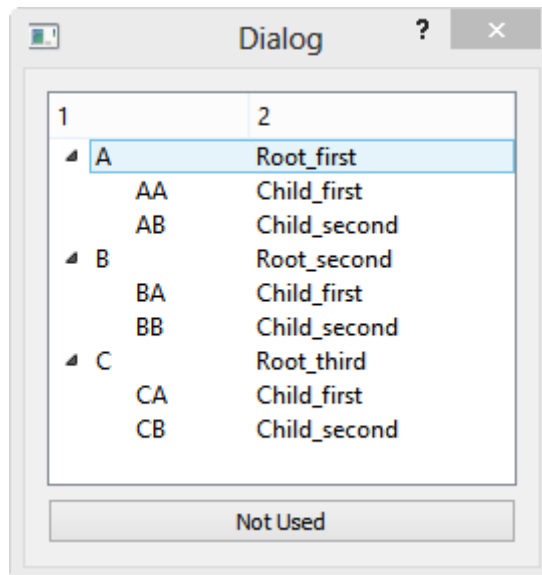


Note that the Project Explorer itself is a tree.

We're going to use Item-Based Tree widget:



In the example below, we'll construct 3 top-level tree nodes, and each of them has two child tree nodes as shown in the picture:



Here are the codes:

dialog.h:

```
#ifndef DIALOG_H
#define DIALOG_H

#include <QDialog>
#include <QString>
#include <QTreeWidget>

namespace Ui {
class Dialog;
}

class Dialog : public QDialog
{
    Q_OBJECT

public:
    explicit Dialog(QWidget *parent = 0);
    ~Dialog();

private:
    Ui::Dialog *ui;

    void addTreeRoot(QString name, QString description);
    void addTreeChild(QTreeWidgetItem *parent,
                     QString name, QString description);
};

#endif // DIALOG_H
```

dialog.cpp:

```
#include "dialog.h"
#include "ui_dialog.h"
```

```

Dialog::Dialog(QWidget *parent) :
    QDialog(parent),
    ui(new Ui::Dialog)
{
    ui->setupUi(this);

    // Set the number of columns in the tree
    ui->treeWidget->setColumnCount(2);

    // Add root nodes
    addTreeRoot("A", "Root_first");
    addTreeRoot("B", "Root_second");
    addTreeRoot("C", "Root_third");
}

Dialog::~Dialog()
{
    delete ui;
}

void Dialog::addTreeRoot(QString name, QString description)
{
    // QTreeWidgetItem(QTreeWidgetItem * parent, int type = Type)
    QTreeWidgetItem *treeItem = new QTreeWidgetItem(ui->treeWidget);

    // QTreeWidgetItem::setText(int column, const QString & text)
    treeItem->setText(0, name);
    treeItem->setText(1, description);
    addTreeChild(treeItem, name + "A", "Child_first");
    addTreeChild(treeItem, name + "B", "Child_second");
}

void Dialog::addTreeChild(QTreeWidgetItem *parent,
                          QString name, QString description)
{
    // QTreeWidgetItem(QTreeWidgetItem * parent, int type = Type)
    QTreeWidgetItem *treeItem = new QTreeWidgetItem();

    // QTreeWidgetItem::setText(int column, const QString & text)
    treeItem->setText(0, name);
    treeItem->setText(1, description);

    // QTreeWidgetItem::addChild(QTreeWidgetItem * child)
    parent->addChild(treeItem);
}

```

In the constructor of Dialog, we call **Dialog::addTreeRoot(QString name, QString description)** three times, and in each call we create a new root node which is **treeItem**. Note that **ui->treeWidget** is the parent of the three root nodes.

Also, we call **Dialog::addTreeChild(QTreeWidgetItem *parent, QString name, QString description)** two times in **addTreeRoot()**. Note that when we make a child node, we do not explicitly specify the parent node:

```
QTreeWidgetItem *treeItem = new QTreeWidgetItem();
```


We set the relationship at this line:

```
parent->addChild(treeItem);
```

where the **parent** is the passed in root node.

QTreeWidgetItem - Header and Customizing

In this section, we'll learn how to set the headers of the QTreeWidgetItem. Also we'll customize the columns of the tree.

The new codes should look like this:

dialog.h:

```
#ifndef DIALOG_H
#define DIALOG_H

#include <QDialog>
#include <QString>
#include <QTreeWidgetItem>
#include <QBrush>

namespace Ui {
class Dialog;
}

class Dialog : public QDialog
{
    Q_OBJECT

public:
    explicit Dialog(QWidget *parent = 0);
    ~Dialog();

private slots:
    void on_pushButton_clicked();

private:
    Ui::Dialog *ui;

    void addTreeRoot(QString name, QString description);
    void addTreeChild(QTreeWidgetItem *parent,
                     QString name, QString description);
};

#endif // DIALOG_H
```

dialog.cpp:

```
#include "dialog.h"
#include "ui_dialog.h"

Dialog::Dialog(QWidget *parent) :
    QDialog(parent),
    ui(new Ui::Dialog)
{
```

```

        ui->setupUi(this);

        // Set the number of columns in the tree
        ui->treeWidget->setColumnCount(2);

        // Set the headers
        ui->treeWidget->setHeaderLabels(QStringList() << "ABC" << "123");

        // Add root node
        addTreeRoot("A", "Root_first");
        addTreeRoot("B", "Root_second");
        addTreeRoot("C", "Root_third");
    }

Dialog::~Dialog()
{
    delete ui;
}

void Dialog::addTreeRoot(QString name, QString description)
{
    // QTreeWidgetItem(QTreeWidgetItem * parent, int type = Type)
    QTreeWidgetItem *treeItem = new QTreeWidgetItem(ui->treeWidget);

    // QTreeWidgetItem::setText(int column, const QString & text)
    treeItem->setText(0, name);
    treeItem->setText(1, description);
    addTreeChild(treeItem, name + "A", "Child_first");
    addTreeChild(treeItem, name + "B", "Child_second");
}

void Dialog::addTreeChild(QTreeWidgetItem *parent,
                          QString name, QString description)
{
    // QTreeWidgetItem(QTreeWidgetItem * parent, int type = Type)
    QTreeWidgetItem *treeItem = new QTreeWidgetItem();

    // QTreeWidgetItem::setText(int column, const QString & text)
    treeItem->setText(0, name);
    treeItem->setText(1, description);

    // QTreeWidgetItem::addChild(QTreeWidgetItem * child)
    parent->addChild(treeItem);
}

void Dialog::on_pushButton_clicked()
{
    QBrush brush_red(Qt::red);
    ui->treeWidget->currentItem()->setBackground(0, brush_red);
    QBrush brush_green(Qt::green);
    ui->treeWidget->currentItem()->setBackground(1, brush_green);
}

```

If we run the code:

