

An Introduction to

Qt

*Gaurav Dixit,
PICT, Pune.*

In this tutorial, we will take a look at the Qt SDK and the Qt/C++ programming language. Even though Qt isn't difficult to pick up, it is recommended to have some experience with C++. We will cover the very basics of the Qt SDK to get you familiar with developing cross platform applications . You will notice that it takes very little effort and code to get up and running..

We will look at two different approaches for developing with Qt. Which approach to choose, depends entirely on the needs of the application and its complexity.

1. Using the *Qt designer*
2. Subclassing `QWidget`

Let's dive right in

1 The Drag and Drop Approach

1.1 What is Drag and Drop?

Qt provides rapid application development through this direct designing approach. We simply Drag and Drop Qt components , like Widgets, Dialogs, onto a form. The form thus acts like a canvas for the user interface.

1.2 Using *Qt Designer*

Qt Designer is a Qt tool for designing and building graphical user interfaces (GUIs) with Qt Widgets. You can compose and customize your widgets or dialogs in a what-you-see-is-what-you-get (WYSIWYG) manner, and test them using different styles and resolutions.

Widgets and forms created with *Qt Designer* integrate seamlessly with programmed code, using Qt's signals and slots mechanism, that lets you easily assign behavior to graphical elements. All properties set in *Qt Designer* can be changed dynamically within the code.

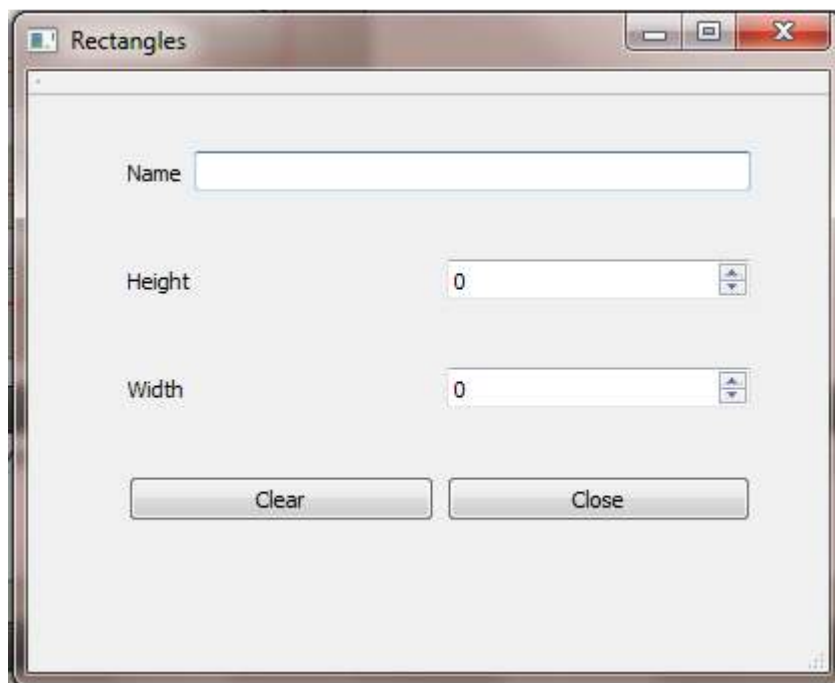
Most features of *Qt Designer* are accessible via the menu bar, the tool bar, or the widget box. Some features are also available through context menus that can be opened over the form windows. On most platforms, the right mouse is used to open context menus.

1.3 A Quick start to *Qt Designer*

Using *Qt Designer* involves four basic steps:

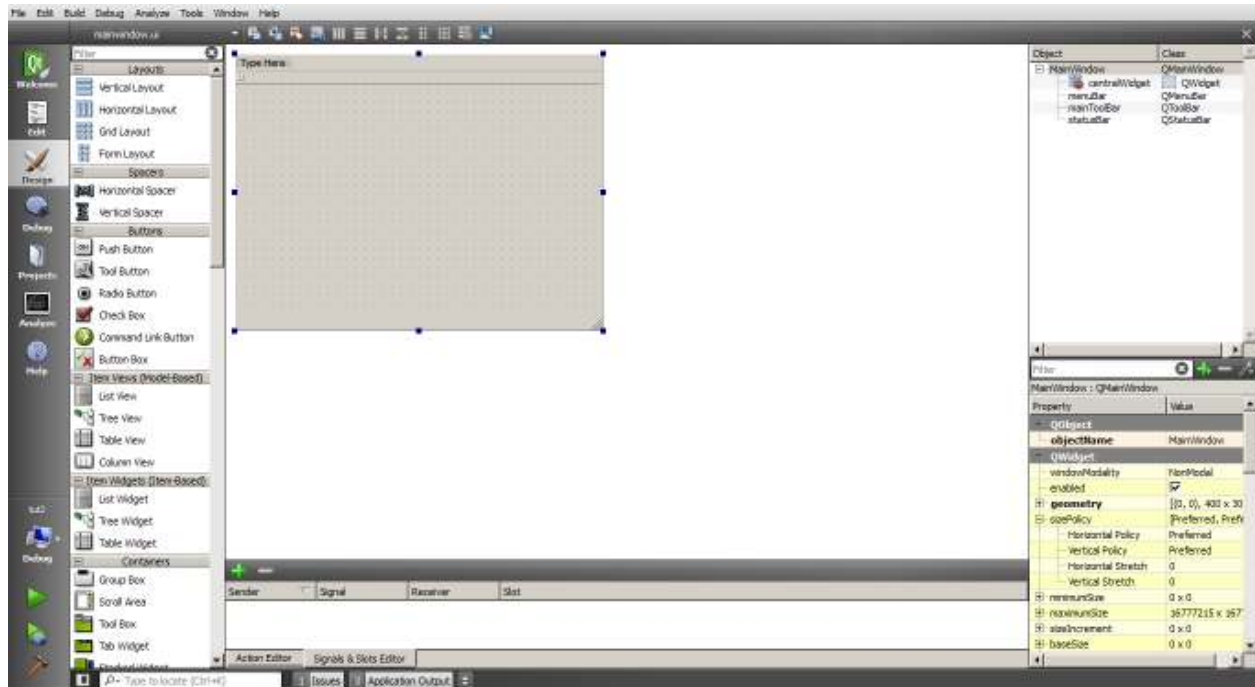
1. Choose the objects that you need
2. Lay the objects out on the form(canvas)
3. Make any required connections
4. Preview and Run

Suppose you want to design a widget that accepts a Name, Height and Width for a rectangle, and looks something like this



The image shows a Qt Designer window titled "Rectangles". The window contains a form with three input fields: "Name" (a text box), "Height" (a spin box with a value of 0), and "Width" (a spin box with a value of 0). Below these fields are two buttons: "Clear" and "Close".

To begin, create a new *Qt* Gui Project, and click on the form file (MainWindow.ui). This opens up the *Qt Designer* with a blank form.

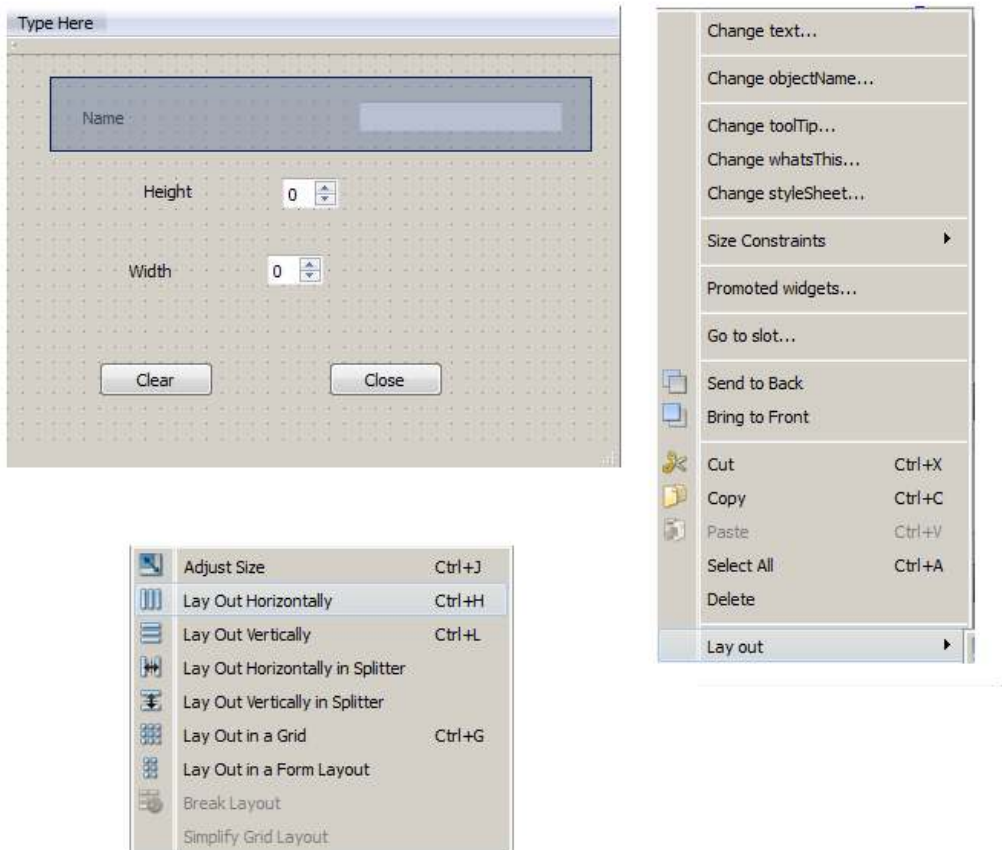


Placing Widgets on the form:

Drag three labels, a lineEdit, two spin boxes and two pushbuttons on to your form. To change a label's default text, simply double-click on it. You can arrange them according to how you would like them to be laid out.

To ensure that they are laid out exactly like this in your window, you need to add them to a layout. We will do this in **five** steps.

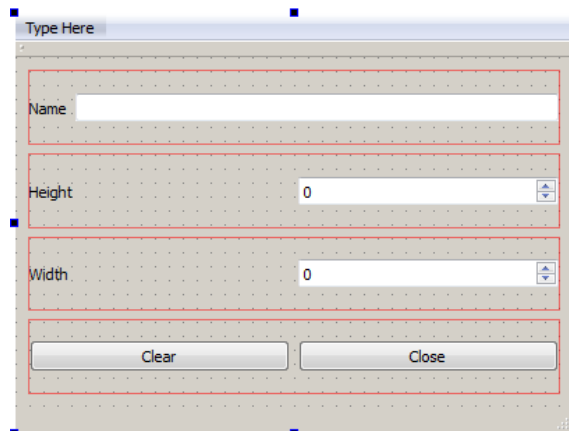
1. Select the first label and the lineEdit, right click to select a layout, and select Lay out Horizontally



2. Repeat the step for the groups of label, spinbox and the two buttons

Now the next step, is to combine all these layouts as the **main layout**.

It is important that your top level widget has a layout; otherwise, the widgets on your window will not resize when your window is resized. To set the layout, **Right click** anywhere on your form, outside of the four separate layouts, and select **Lay Out Vertically** -- you will see the arrangement (as shown below).



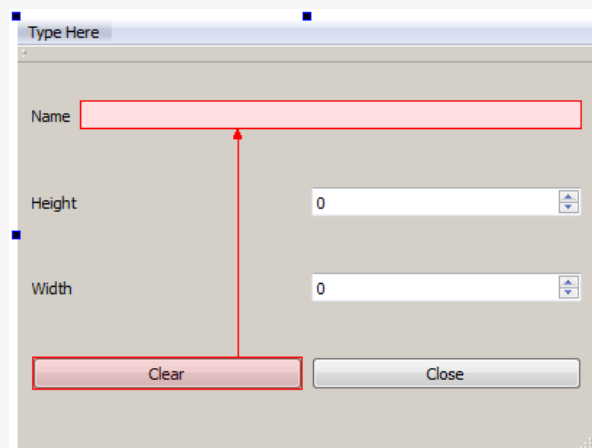
Now press **CTRL+R** , to run the application. You can see a window with the same arrangements as desired.

When you click on the *clear* button, you want the `lineEdit` to clear. To achieve this behavior, you need to connect the pushbutton's *Clicked()* signal with the `lineEdit`'s *clear()* slot.

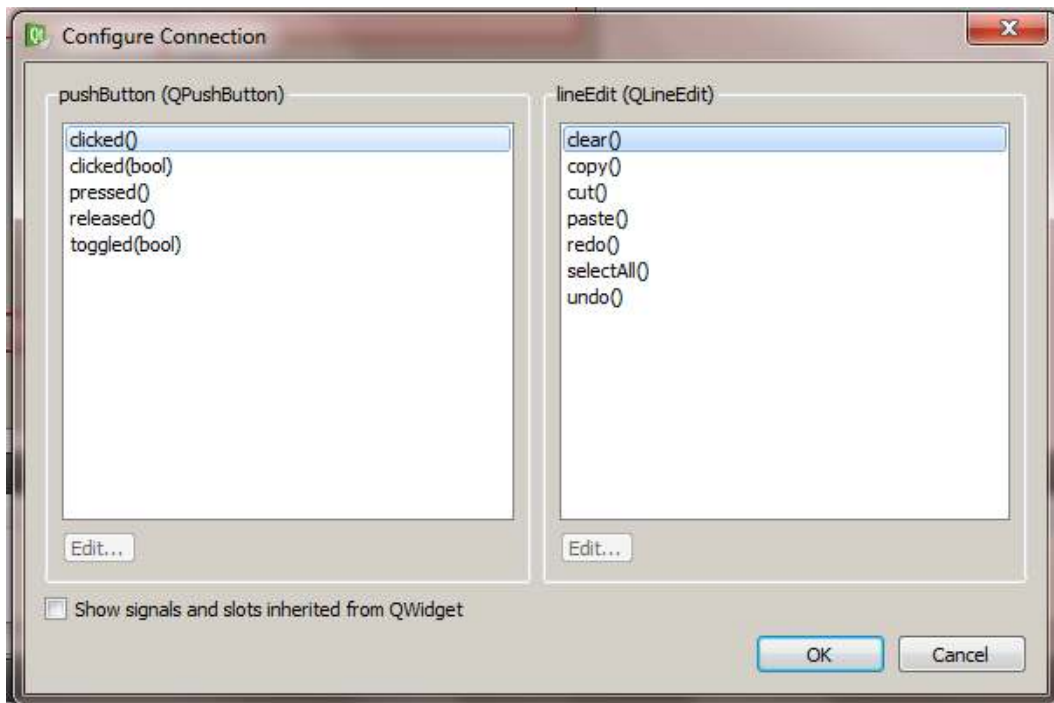
To do this, you have to switch to **Edit Signals/Slots** mode, by selecting **Edit Signals/Slots** from the **Edit** menu.

Connecting Signals to Slots:

Click on the button and drag the cursor towards the `lineEdit`.



The **Configure Connection** dialog, shown below, will pop up. Select the correct signal and slot and click **OK**.



Similarly we connect the second button's signal *clicked()* to the window's slot *close()*.

Run your form using **CTRL+R**, and resize the window, to see the auto-maintained layouts. Experiment with the signals, and sizes of child Widgets.

2 Programming with *Qt*

2.1 How is the programming approach different than Drag-and-Drop?

In this approach, we create our own *GUI* classes by subclassing `QWidget`s . We define every `QWidget` item onto the stack or heap, and place it in proper layouts. Finally we call `show()` method on an instance of our class.

Steps involved:

- 1. Designing the User Interface**
- 2. Defining one or more classes according to needs**
- 3. Implementing the classes**
- 4. Instantiating the parent class and running the application.**

2.2 Designing the User Interface

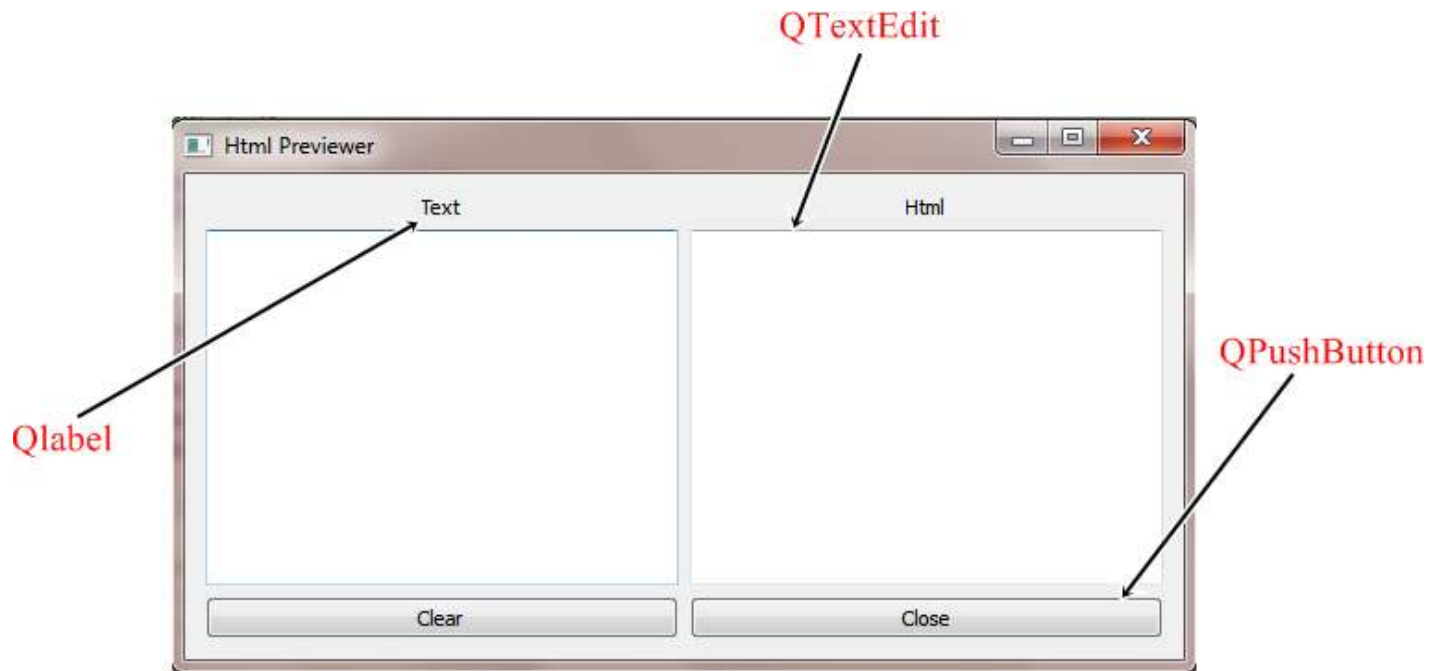
This first part covers the design of the basic graphical user interface (GUI) for HTML previewer application.

The first step in creating a GUI program is to design the user interface. Here the goal is to set up the text fields and labels to implement a basic previewer. The figure below is a screenshot of the expected output.



We start by noting down all the QWidgets that we will need to use for this output design.

We require two QTextEdit objects, editText and editHtml , as well as two QLabels, labelText and labelHtml, and two QPushButton objects, to enable the user to clear texts and exit the application. The widgets used and their positions are shown in the figure below.



There are three files used to implement this viewer:

- view.h - the definition file for the viewer class,
- view.cpp - the implementation file for the view class, and
- main.cpp - the file containing a main() function, with an instance of view.

2.3 Defining the *view* Class

We start by defining *view* as a `QWidget` subclass and declaring a constructor. We also use the `Q_OBJECT` macro to indicate that the class uses Qt's signals and slots features.

```
class view: public QWidget{  
    Q_OBJECT  
public:  
    view();
```

We need to declare the required `QTextEdits`, `Qlabels` and `QPushButtons`. One such declaration for each of these looks like

```
QTextEdit *editText;  
QLabel *labelText;  
QPushButton *buttonClear;
```

Now we declare two objects of `QVBoxLayout`, and an object of `QHBoxLayout`. These are needed later for handling layouts.

```
QVBoxLayout *vbText,*vbHtml;  
QHBoxLayout *hb;
```

We also declare two `QPushButtons` as

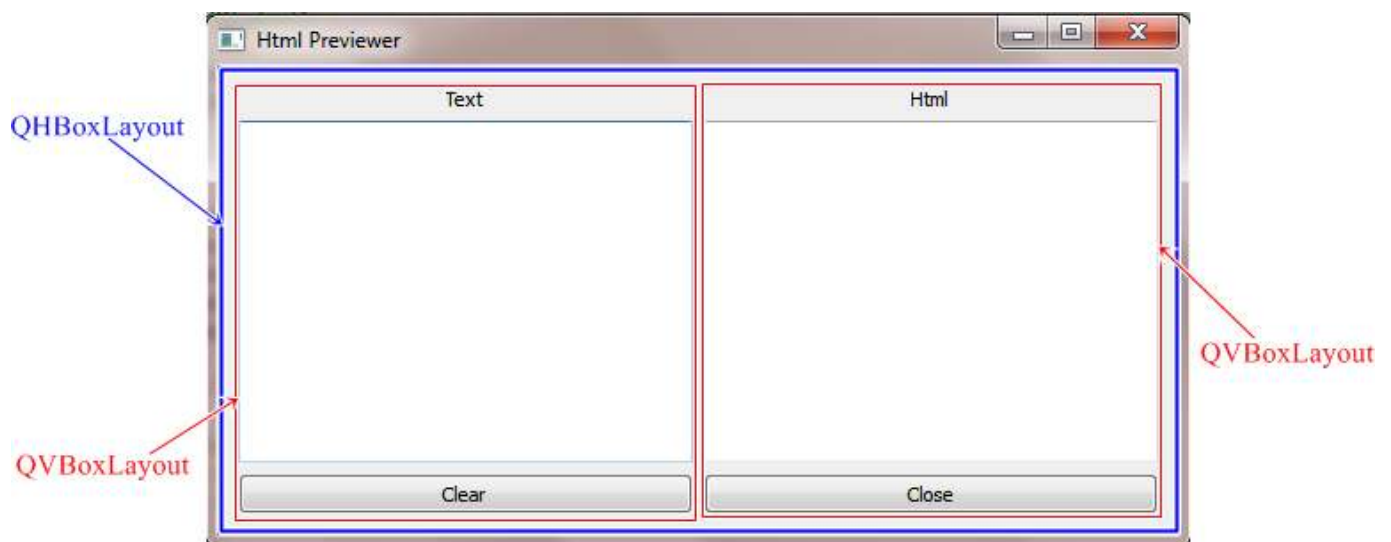
```
QPushButton *buttonClear,*buttonClose;
```

2.4 Implementing the *view* class

We now define the QWidgets in view's constructor

```
labelText=new QLabel("Text",this);  
labelHtml=new QLabel("Html",this);  
  
editText=new QTextEdit(this);  
editHtml=new QTextEdit(this);  
  
buttonClear=new QPushButton("Clear",this);  
buttonClose=new QPushButton("Close",this);
```

Once instantiation is completed, we need to add these objects to a layout, to obtain the required design.



The Layout objects are instantiated

```
vbText=new QVBoxLayout;  
vbHtml=new QVBoxLayout;  
hb=new QHBoxLayout;
```

Now, we add objects to the vbText Layout as

```
vbText->addWidget(labelText);  
vbText->addWidget(editText);  
vbText->addWidget(buttonClear);
```

Similarly, add objects to vbHtml.

The final Horizontal layout, consists of the earlier defined Vertical layouts.

```
hb->addLayout(vbText);  
hb->addLayout(vbHtml);  
  
setLayout(hb);
```

As the final Layout of our window is the Horizontal Layout, we call the `setLayout()` .

For interacting with the user, we need the QButtons to perform an operation when they are clicked. We call these functions as slots. They must be declared in `view.h` as

```
private slots:  
void onChange();  
void onClear();
```

And to connect these slots , we use the connect() method as shown in the constructor:

```
connect(editText,SIGNAL(textChanged()),this,SLOT(onChange()));  
connect(buttonClear,SIGNAL(clicked()),this,SLOT(onClear()));  
connect(buttonClose,SIGNAL(clicked()),this,SLOT(close()));
```

The first connect() connects editText's textChanged() signal to a slot, so that every time the input changes, we update the Html preview. The next connects are simply connected to slots when they are clicked.

The slots are defined as follows:

```
void view::onChange()  
{  
    QString text=editText->toPlainText();  
    editHtml->setHtml(text);  
}  
  
void view::onClear()  
{  
    editText->clear();  
}
```

The toPlainText() returns a QString from the editText. We set this QString as output to editHtml, by calling its setHtml(). The clear() method simply clears contents of the calling object.

This completes the implementation of the application. Now we need to instantiate this class in main.cpp.

2.5 Running the Application

A separate file, main.cpp, is used for the main() function.

Within this function, we instantiate a QApplication object, app. QApplication is responsible for running an event loop. Hence, there is always one QApplication object in every GUI application using Qt.

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

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

    return a.exec();
}
```

We construct a new view widget on the stack and invoke its show() function to display it. However, the widget will not be shown until the application's event loop is started. We start the event loop by calling the application's exec() function; the result returned by this function is used as the return value from the main() function.

Next, just build and run, and voila! , we have an HTML viewer ready!