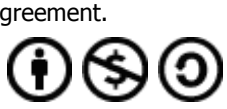
Exercises Lecture 1 – An introduction to Qt

Aim: This exercise will help you setup a working Qt development environment and introduce you to some of the basic features of Qt as well as the basics of the Qt Creator environment.

Duration: 1h



Install the Qt SDK

Visit http://qt.nokia.com/downloads , and download the LGPL version of Qt SDK suitable for your system. Then proceed to install the SDK. This gives you a Qt Creator icon on your desktop and in your program menu. Please refer to the platform specific tips and tricks below if you are experiencing any issues with this.

Windows

If altering the installation path of the Qt SDK, make sure to use a path without spaces. Spaces can in some situations confuse the build tool-chain.

Mac OS X

Installation should be straight forward.

Before you can develop Qt applications, you must make sure you have installed the developer tools from the DVD that came with your Mac.

Unix/Linux X11

Before you can develop Qt software, you must ensure that you have a C++ toolchain installer. This usually means GCC with support for C++. This comes as a part of your distribution and you must install it to be able to use the SDK.

If your Linux distribution has a prepackaged version of Qt Creator and the Qt development libraries, it is often better to use those libraries. This course has been developed with Qt 4.6+ and Qt Creator 1.2+ in mind.

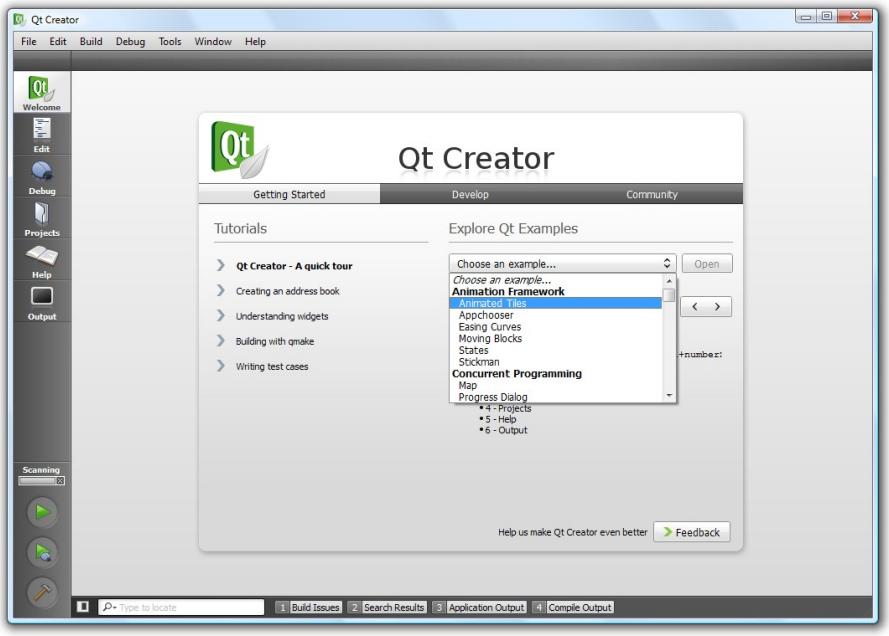
If you choose to use the installer downloaded from the web, you must make the installer executable before you can run it. You can either do this by setting the executable bit (sometimes just X-bit) of the file through your file manager, or by running `chmod` from a command prompt.

```
chmod u+x qt-sdk-linux-*.bin
```

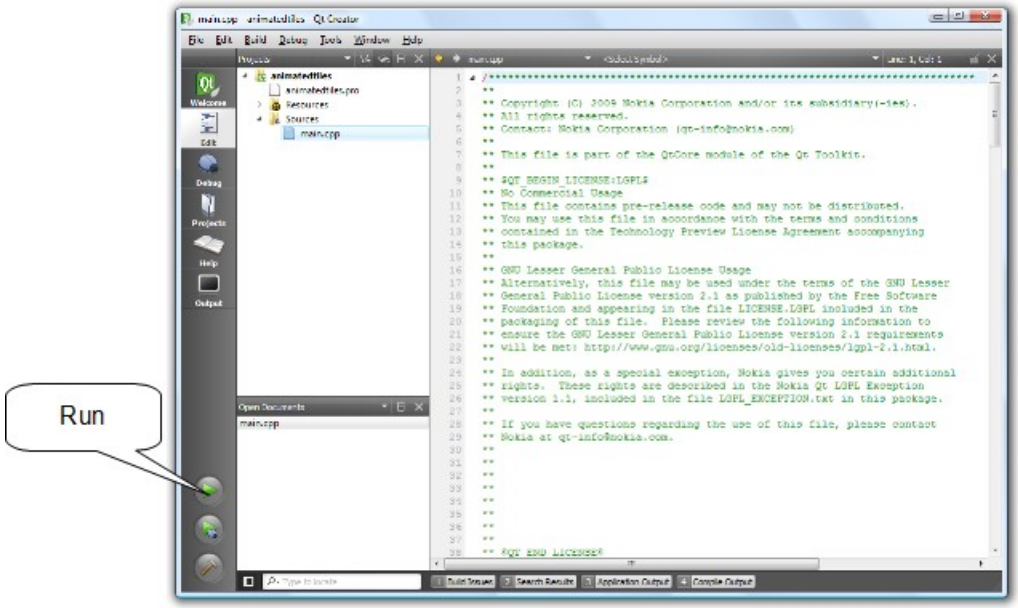
When running the installer, make sure that you have write access to the directory where you chose to install the SDK.

Testing the tool-chain

To ensure that the SDK has been properly installed, please try to build and debug some of the examples shipped with Qt. On the welcome screen of Qt Creator, make sure that you are on the “Getting Started” page shown below. From the list of Qt examples, pick the “Animated Tiles” example from the “Animation Framework” group.

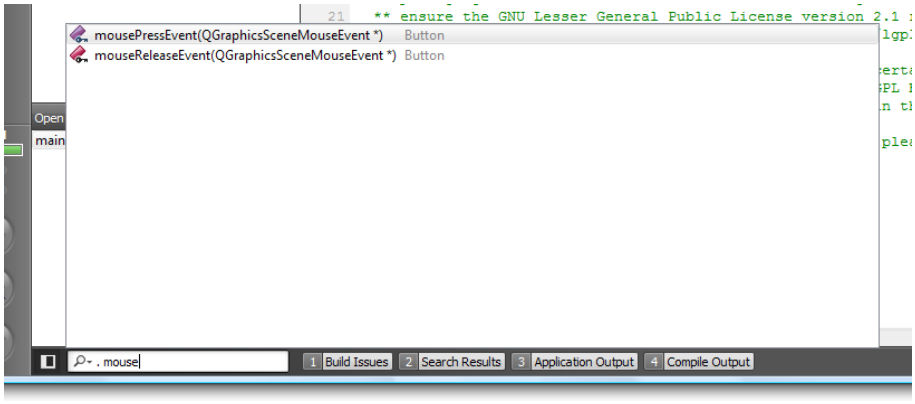


Having opened the project, ensure that you are in *edit mode* (Ctrl+2), then look for `main.cpp` in the Sources folder. Understanding the source code is not the purpose of this exercise, so leave that for later. Instead, try pressing the *run button* (Ctrl+R) to start the example. This will trigger Qt Creator to first build the example, then run it. When the example has been built, you should see a window with the demo running.

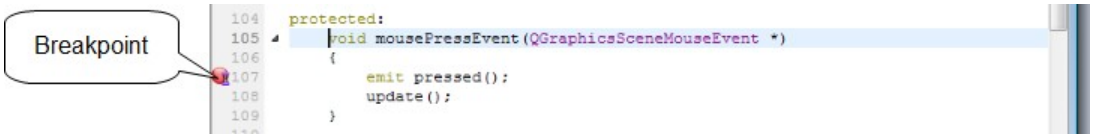


Having run the project, you have verified that the compiler and linker work. The next step is to test the debugging features.

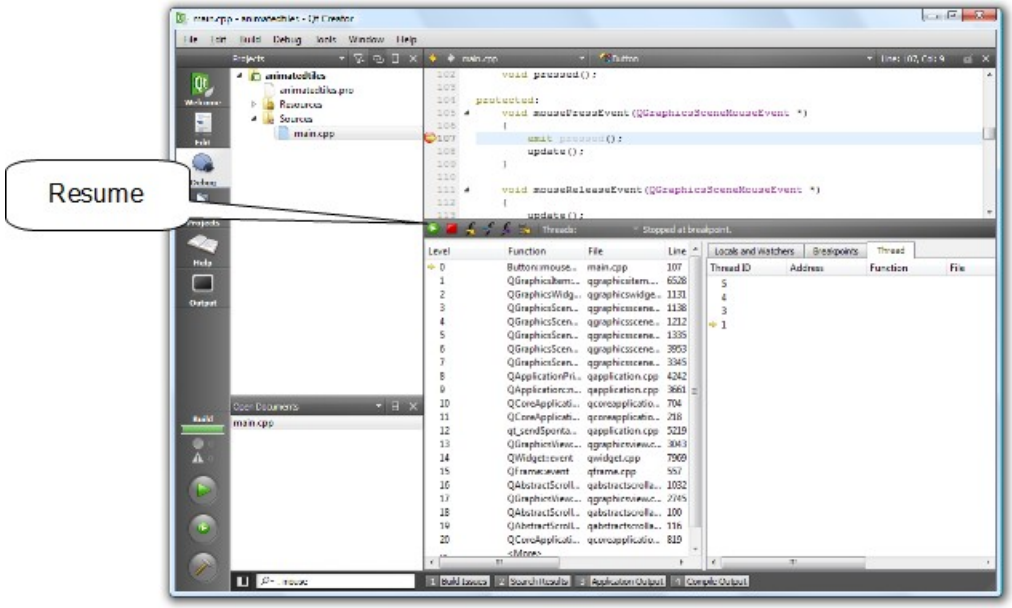
To find a point in the code to insert a break point at, we will use the *locator*. The locator lets you search within your project. It can be used to look for classes, methods and files within or outside your project. In this case, you are looking for the `mousePressEvent` in the `main.cpp` file. To locate a method, start your search with a dot followed by a space ". ". Then start typing mousePressEvent until you find the method in the list, then press *enter* to go there.



Having located the `Button::mousePressEvent` in the source code, you will see something like the image below. Try setting a break point as shown below (by clicking just left of the line number or right clicking on the line number and picking "Set Breakpoint" from the pop-up menu).



To start debugging click the debugging button (F5) just below the run button. This will start a debugging session. The example program will start as expected, but as soon as you click one of the buttons, the program execution will break and the editor will show the current callstack, breakpoint and so on.



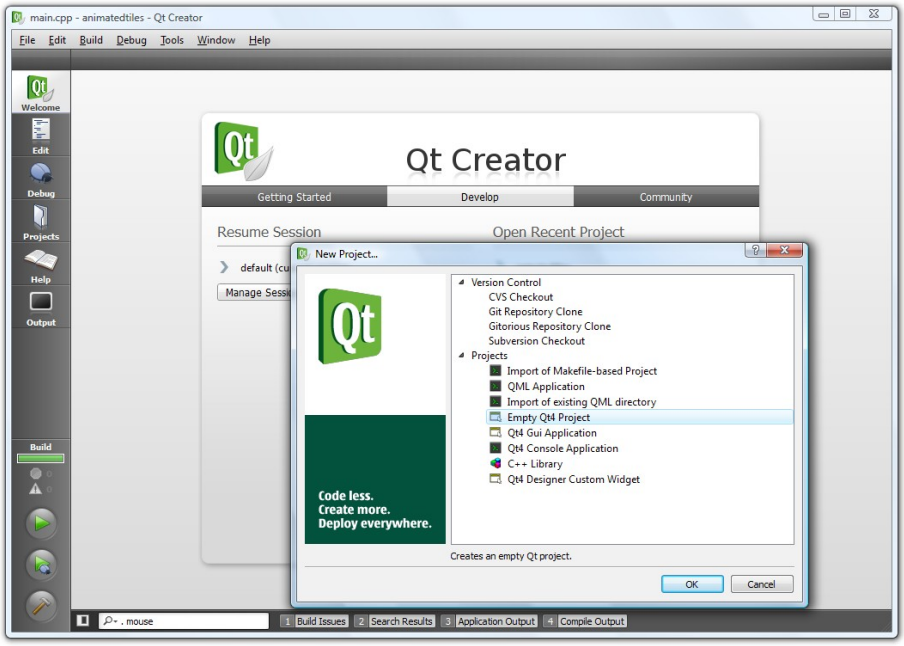
To resume execution, press the resume button on the debugging bar. Exiting the example program

will end the debug session.

When you have run and debugged this example, you have verified that your development setup is in order and you are ready to continue.

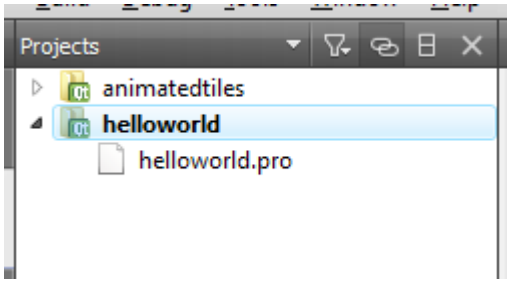
Create a Hello World project

To create a project of your own, go the the welcome screen using the top-left button (Ctrl+1), pick the page “Develop” and click “Create New Project...”.



From the dialog that pops up, create an “Empty Qt4 Project”, then specify a location for your project. Notice that Qt Creator will create a sub-directory with your project's name, so if you choose to name the project `helloworld` and create it in `/home/user/code`, your source files will end up in `/home/user/code/helloworld`.

If you have other projects open (from earlier exercises), make sure to either close them (right click on the project node and choose “Close Project ...” from the menu) or ensure that you are working with your project by right clicking on your project in the projects tree view and pick your project as the current *run configuration*. Your project should be marked as bold as shown below.

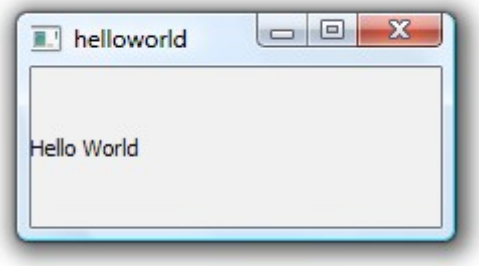


Now, right click on your project and choose “Add New...” from the menu. In the following dialogs, create a “C++ Source File” and name it `main.cpp`. In the file, enter the following code and save it.

```
#include <QApplication>
#include <QLabel>

int main(int argc, char **argv)
{
    QApplication a(argc, argv);
    QLabel l("Hello World");
    l.show();
    return a.exec();
}
```

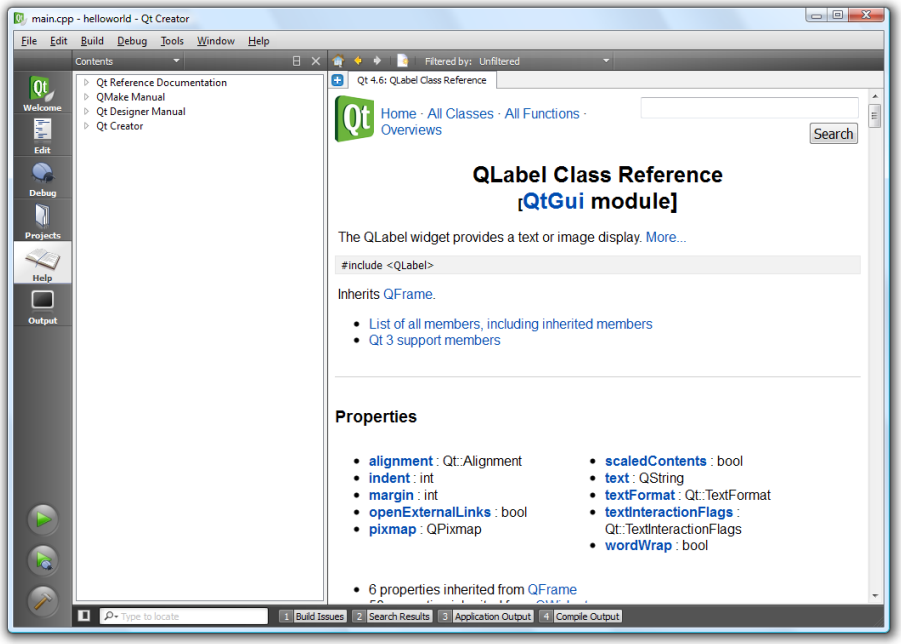
Now, run the application (Ctrl+R) and try stretching and moving the resulting window about.



Documentation and Text Alignment

The last task resulted in a window with some text. Notice that the text is left aligned, but vertically centered. Would it not be better if it was properly centered in all directions?

To center the text, you will have to locate the documentation for the appropriate method and then make changes to your source code. To view the documentation for the `QLabel` class, place your cursor over the word `QLabel` and press F1. This will take you to the help page of the `QLabel` class.



All Qt classes are documented according to a specific structure. The top of each page refers to the class in question and the module. It lists the required header file, super-classes and classes inheriting from the specific class. Then follows a list of properties, functions, slots and signals. After this list follows the “Detailed Description” section. This part of the documentation tells you how to use each class and refers to other classes that might be good to know about.

Reading the detailed description for the `QLabel` class you learn that “By default, labels display left-aligned, vertically-centered text... The positioning of the content within the QLabel widget area can be tuned with setAlignment()...”. Now, look up the `setAlignment` method and modify your code so that the label is centered in all directions.

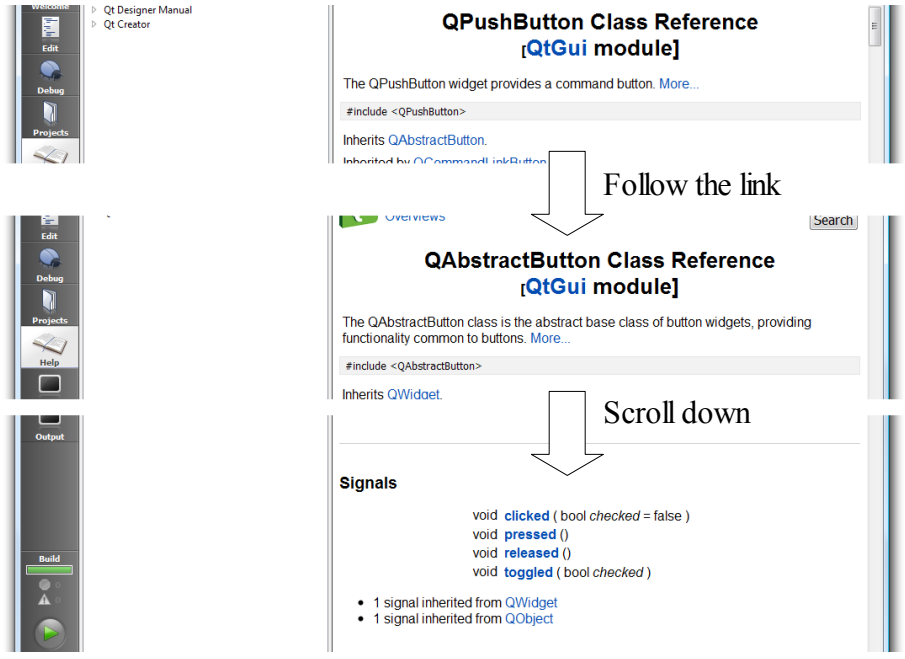
Adding a Button

As a final step of this exercise we will introduce a button in the example application. The button will reside in a window of its own. When clicked, it will quit the application.

First, add the button by inserting the following lines after the call to `show` the `QLabel`. This will show the button in a window of its own. Make sure to add the appropriate header file before using the `QPushButton` class, then try compiling and running the application.

```
QPushButton b("Close All");
b.show();
```

The next step is to make the button do something. If you visit the documentation for `QPushButton`, you will not find any signals for this. This is because this functionality is added in the super-class `QAbstractButton`. Follow the link there from the top of the `QPushButton` class documentation, then scroll down to the “Signals” section. There you will find the `clicked` signal.



We will connect this signal to the `quit` slot of the `QApplication` class (defined in the super-class `QCoreApplication`). This is done using the `QObject::connect(`*`src`*`, `*`signal`*`, `*`dest`*`, `*`slot`*`)` call shown below. Insert the following command just before the `return` statement and replace the ***`bold-italic`*** parts to make the connection described above.

```
QObject::connect(&b, SIGNAL(clicked()), source, SLOT(slot));
```

Now compile and test your application. Notice that connections can pass compilation without failing. Instead, a run-time message such as the following is displayed. These messages are shown in the “Application Output” tab (Alt+3) available at the bottom of the Qt Creator window.

```
Object::connect: No such slot QApplication::foo() in main.cpp:13
```

In the example above, the `SLOT(foo())` part of the connect command is causing a problem, as the specified slot is not available.

When your connection works as intended, the application will close both its windows and quit when the button is clicked.

Solution Tips

Step 4

Add the following line before showing the `QLabel`.

```
    l.setAlignment(Qt::AlignCenter);
```

Step 5

The connection line should read:

```
    QObject::connect(&b, SIGNAL(clicked()), &a, SLOT(quit()));
```