

# Compiling Your First Qt Application - HOWTO

Jan Schaumann

July 18, 2001

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	Requirements . . . . .	3
1.2	How to use this document . . . . .	3
1.3	Translations . . . . .	4
1.4	Comments and Suggestions . . . . .	4
1.5	Copyright and Disclaimer . . . . .	4
<b>2</b>	<b>Hello, World</b>	<b>4</b>
2.1	Compiling by hand . . . . .	5
2.1.1	In one step . . . . .	5
2.1.2	In two steps . . . . .	5
2.2	Using tmake . . . . .	6
<b>3</b>	<b>A (slightly) more complex application</b>	<b>7</b>
3.1	Compiling by hand . . . . .	8
3.2	Using tmake . . . . .	9
<b>4</b>	<b>Ressources</b>	<b>9</b>

## Abstract

This document serves as a brief introduction to compiling your first application using Trolltech's Qt Framework. Two examples are discussed – a very simple "Hello, World" example, and a slightly more complex example consisting of multiple files. It is assumed that you have Qt and gcc installed as per their respective documentation.

# 1 Introduction

This document briefly explains how to compile your first Qt-Application, as the otherwise very helpful documentation from Trolltech does not elaborate on the exact syntax.

This is Version 0.1 of the "Compiling Your First Qt Application - HOWTO", dated 2001-07-06.

## 1.1 Requirements

Naturally, you will need the Qt API. You can download it from:

<http://www.trolltech.com>.

Follow the instructions that accompany the package and read the documentation, available at:

<http://doc.trolltech.com/installation.html>

Needless to say that you will also require a working C++ compiler - you can get gcc from:

<http://gcc.gnu.org>.

Again, follow the instructions to find out how to install the package properly on your system.

## 1.2 How to use this document

This document tries to guide even unexperienced users to the success of compiling their first Qt-Application. Some basic unix-commands are assumed to be understood. Furthermore, this document often refers to various websites for more detailed information; the reader should be able to easily follow the steps described in this document, while at the same time possibly keeping a browser-window open for other information.

## 1.3 Translations

If you wish to translate this document to any other language, please email the author. He'll be thrilled.

## 1.4 Comments and Suggestions

The latest version of this document should always be available at <http://www.netmeister.org/>. If you have any questions, comments, suggestions etc, please do not hesitate to email the author at [jschauma@netmeister.org](mailto:jschauma@netmeister.org). While this is a very simple document, I would like to keep it as accurate and complete as possible; all input will therefore be highly appreciated.

## 1.5 Copyright and Disclaimer

The author does not take any responsibility for anything that might happen as a result of following the instructions in this document. Feel free to redistribute this document as you see fit, modifications, however, need to be discussed with the author (read: "Copyright 2001 by Jan Schaumann").

# 2 Hello, World

Let's assume we wish to follow the tutorial as described on <http://doc.trolltech.com/tutorial.html>. Using your favorite editor, create a file called "hello.cpp" and enter the following code:

```
#include <qapplication.h>
#include <qpushbutton.h>

int main( int argc, char **argv )
{
    QApplication a( argc, argv );
    QPushButton hello( "Hello world!", 0 );
    hello.resize( 100, 30 );
    a.setMainWidget( &hello );
    hello.show();
    return a.exec();
}
```

Save the file and exit your editor. Again, read through <http://doc.trolltech.com/t1.html> for a detailed explanation of the code. The sentence "You should now try to

compile and run this program." is where this document attempts to fill the gap.

## 2.1 Compiling by hand

### 2.1.1 In one step

Assuming you have successfully and correctly installed the Qt-libraries and headers and set the appropriate environment variables accordingly, you should now be able to compile the application as follows.

```
$> g++ -Wall -I$QTDIR/include -L$QTDIR/lib hello.cpp -o hello -lqt
```

Ok, let's explain what we are doing here in detail, what does all this mean? First off, `g++`, is of course the compiler. We tell it to warn us about everything that looks even remotely suspicious by passing the `-Wall` flag. Next, we tell the compiler where to look for the header files. As mentioned above, the environment variable `$QTDIR` must be set as suggested by the Qt Installation Documentation.

Suppose you have installed Qt in `/usr/local/qt`, then you could also use `-I/usr/local/qt/include` instead of `-I$QTDIR/include`. Note that there is no space between the `-I` and the `$QTDIR`!

Since this is a simple application, we can compile and link the application in one step, so the next flag tells the compiler where to search for a library (in addition to the standard directories): `-L$QTDIR/lib`. Again, note that there is no space between the flag and the directory!

Next, we pass the input file to the compiler (`hello.cpp`) and specify that the generated executable should be named "hello" (`-o hello`), as otherwise the compiler would produce a file called "a.out".

And finally, The next flag tells the compiler that we wish to link the application to a library which is called "libqt.so". When linking, the "lib" and the file extension are left out, so that we end up with `-lqt`. Note that the library parameters are position sensitive, and that it therefore is a good idea to place the libraries at the end of the instructions.

For a more detailed documentation of the various compiler flags and an overview of the other flags you might want to pass to `g++`, refer to `gcc(1)`.

### 2.1.2 In two steps

While the above example will work for a simple application that consists of only one file, you will of course encounter larger projects that require several

source-files that need to be compiled and whose object-files then need to be linked together with the shared libraries. It is therefore a good idea to get used to separating the two steps (compiling and linking):

```
$> g++ -Wall -I$QTDIR/include -c hello.cpp
$> g++ -Wall -L$QTDIR/lib hello.o -lqt -o hello
```

The only difference here is that in the first step, we compile without linking, but instead we pass the `-c` flag to the compiler. This flag instructs the compiler to NOT attempt to link. Per default, the compiler will create an object-file with the “.o” ending.

In the second step, we take all object files that were compiled in the first step (one, in this case) and link them together with the appropriate libraries (`-lqt`, in this case) to form the executable “hello”.

Again, refer to *gcc(1)* for details.

## 2.2 Using tmake

As you develop more and more complex programs, you will find the need to split the project over several files. Now you could, of course, always type in the same commands over and over again for all your source-files, but naturally, this becomes tedious very soon. This is where *make* comes into play. (If you don’t know what *make* is, please read through *make(1)* and the documentation given in the “References” section of this document.) Creating proper Makefiles is not for the faint of heart, which is why Trolltech developed *tmake*, a little tool that will take over this task for you.

Once you installed *tmake*, please read through the documentation - you will find that it is incredibly easy to manage projects with this tool. For our little “Hello, World” example, we would need to create a *hello.pro* file with the following content:

```
HEADERS      =
SOURCES      = hello.cpp
TARGET       = hello
```

Then let *tmake* create the Makefile and finally let *make* build the program for you:

```
$> tmake hello.pro -o Makefile
$> make
```

Et voilà - “hello” has been built, much in the same way as we did by hand!

For additional information on the flags used by and on tmake in general, please refer to the tmake documentation.

### 3 A (slightly) more complex application

As mentioned above, you will eventually encounter much more complex projects consisting of multiple files, including your own customary header-files. Since Qt utilizes some macros that need to be pre-processed before compilation, the procedure to build such a project is slightly different from the above.

The header files, or any other file that contains some of the Qt-specific macros, need to be processed by *moc*, the Meta-Object Compiler, before they can be compiled by g++.

So let’s assume we modify the code from above as follows. First, we create a file called “main.cpp”:

```
#include <qapplication.h>
#include "myButton.h"

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

    myButton button( "Hello World!" );

    a.setMainWidget( &button );
    button.show();
    return a.exec();
}
```

Here, we are making use of a class called “myButton”, which is defined in the header “myButton.h”:

```
#ifndef MYBUTTON_H
#define MYBUTTON_H

#include <qpushbutton.h>
```

```

class myButton : public QPushButton
{
    Q_OBJECT
    public:
        myButton(const QString& text, QWidget *parent=0);
};

#endif

```

The actual code for all of “myButton”’s functionality goes into “myButton.cpp”:

```

#include "myButton.h"

myButton::myButton(const QString& text, QWidget *parent)
    : QPushButton(text, parent)
{
    resize( 100, 30 );
    connect(this, SIGNAL(clicked()), this, SLOT(close()));
}

```

Granted, this does not do very much and could easily be achieved in our first example in a much more obvious way. However, let’s take this example to show how to compile a program that consists of several files. Now we have:

1. main.cpp - the main sources
2. myButton.h - defining the “myButton” class
3. myButton.cpp - implementation code of “myButton”

### 3.1 Compiling by hand

Again, just to teach you what exactly is happening here, I will walk you through the steps necessary to compile and link this application. Once you understand what is going on here, you should however take advantage of “tmake” as described below.

The first step is to compile the *.cpp* files, which is identical to the commands used above:

```

$> g++ -Wall -I$QTDIR/include -c myButton.cpp
$> g++ -Wall -I$QTDIR/include -c main.cpp

```

Now, before we can continue and link the object files together, we need to use the meta-object compiler to process the header files:

```
$> moc myButton.h -o myButton.moc.cpp
```

This creates a new `.cpp` file, which in turn we need to compile into an object file:

```
$> g++ -Wall -I$QTDIR/include -c myButton.moc.cpp
```

Finally, we have all our object files and continue to link the application similar to our first example:

```
$> g++ -Wall -L$QTDIR/lib myButton.o main.o myButton.moc.o -lqt -o h
```

## 3.2 Using tmake

Now that we understand the commands necessary, we can take advantage of the power of *make* and use “tmake” to create the initial Makefile. The input to tmake, “hello.pro” will now look as follows:

```
HEADERS      = myButton.h
SOURCES      = myButton.cpp main.cpp
TARGET      = hello
```

Run tmake and make just as in the example above:

```
$> tmake hello.pro -o Makefile
$> make
```

And once again: voilà!

## 4 Ressources

- Qt Reference Documentation - <http://doc.trolltech.com/index.html>
- Installing Qt / X11 - <http://doc.trolltech.com/install-x11.html>
- Qt Tutorial "Hello, World" - <http://doc.trolltech.com/t1.html>
- Qt Interest Mailinglist - [qt-interest@trolltech.com](mailto:qt-interest@trolltech.com)
- Qt Interest Mailinglist Archive - <http://qt-interest.trolltech.com>
- GNU Make - [http://www.gnu.org/manual/make-3.79.1/html\\_mono/make.html](http://www.gnu.org/manual/make-3.79.1/html_mono/make.html)
- tmake - <http://www.trolltech.com/products/download/freebies/tmake.html>
- The Meta-Object Compiler (moc) - <http://doc.trolltech.com/moc.html>