Foundations of Qt Development

Johan Thelin

Apress®

Foundations of Qt Development

Copyright © 2007 by Johan Thelin

ISBN-13 (pbk): 978-1-59059-831-3

ISBN-10 (pbk): 1-59059-831-8

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Jason Gilmore
Technical Reviewer: Witold Wysota
Editorial Board: Steve Anglin, Ewan Buckingham, Gary Cornell, Jonathan Gennick, Jason Gilmore, Jonathan Hassell, Chris Mills, Matthew Moodie, Jeffrey Pepper, Ben Renow-Clarke, Dominic Shakeshaft, Matt Wade, Tom Welsh
Senior Project Manager: Tracy Brown Collins
Copy Edit Manager: Nicole Flores
Copy Editor: Nancy Sixsmith
Assistant Production Director: Kari Brooks-Copony
Production Editor: Kelly Winquist
Compositor: Dina Quan
Proofreader: Paulette McGee
Indexer: Brenda Miller
Artist: April Milne
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail orders-ny@springer-sbm.com, or visit http://www.springeronline.com.

For information on translations, please contact Apress directly at 2855 Telegraph Avenue, Suite 600, Berkeley, CA 94705. Phone 510-549-5930, fax 510-549-5939, e-mail info@apress.com, or visit http://www.apress.com.

The source code for this book is available to readers at http://www.apress.com in the Source Code/Download section.

Till Åsa.

Contents at a Glance

Contents

Foreword

My very first computer, a ZX81, did not have a graphical user interface. Compared with today's offerings, I'd say it hardly had graphics at all. That computer never got me excited about programming, mostly because the manuals were in English and I didn't yet know how to read the language.

Then I met the ABC80, a Swedish computer from Luxor. It had the same Z80 processor, 16 kilobytes of RAM, and no real graphics to talk about. It did have an introduction to BASIC in Swedish, though, so it got me started with programming.

My next computer experience was an Atari ST. I must admit that in the beginning I used it mostly for gaming. But as time passed I was thrilled about the possibilities of the Atari for programming. I wrote games, utilities, and painting applications. I also ran into something that I learned to like: an API for handling windows and drawing graphics.

Moving on, I got a PC. I learned C and C++, as well as how to do 3D graphics in software (this was before 3D graphics cards). I was introduced to the Internet and learned lots of new things from newsgroups and FAQs. I also got my first paid job as a programmer, processing scientific data using FORTRAN.

At Chalmers University I met Tru64 UNIX and X Windows. The API for doing graphics felt awkward, so I went looking for something better. That was when I found Qt. Back then, it just solved my problem of the day: showing a couple of dialogs and drawing some graphics. But the architecture got me hooked.

Over time, I used Qt more and more. I soon tried to figure out what it was that made Qt so easy to use. The flexibility of the signals and slots concept that enabled me to connect widgets and objects to each other was one reason. As was the up-to-date reference documentation—nothing was left undocumented. And the naming made it easy to find the class and method I was looking for. The name said it all.

Qt brought me to KDE and Linux. I learned to love GCC, Makefiles, and shell scripting. The thing that thrilled me about Qt was that no matter what the task was, it fit right into its architecture. Today, with Qt 4.0, the API covers most of the tasks that you might want to perform. Graphics, files, databases, networking, printing—you name it. Qt helps me solve my problems quickly and easily.

I've recently become more and more involved in the Qt community. It all started with my original "Independent Qt Tutorial" that introduced Qt 3.0 (you can still find it at `www.thelins.se/qt`). I'm also a part of the administration team at QtCentre, which is where I met the technical reviewer of this book, Witold Wysota. QtCentre (`www.qtcentre.org`) is a community-driven forum, a wiki, and a news site—the natural meeting place for Qt developers. Just over a year ago, Apress posted this question in the jobs section: Is there anyone who wants to write a book about Qt? That was the starting point of the book that you are reading right now.

Johan Thelin
M.Sc.E.E.

About the Author

JOHAN THELIN has worked with software development since 1995 and has experience ranging from embedded systems to server-side enterprise software. He started using Qt in 2000 and has loved using it ever since. Since 2002 Johan has provided the Qt community with tutorials, articles, and help (most notably, he wrote the “Independent Qt Tutorial”). He currently works as a consultant focusing on embedded systems, FPGA design, and software development.

About the Technical Reviewer

WITOLD WYSOTA, Institute of Computer Science, Warsaw University of Technology, was born in Wroclaw, Poland. He has a Master of Science degree in Computer Science from the Warsaw University of Technology (WUT), where he is currently a PhD candidate. As such, he gives lectures about Qt and conducts exercises using Qt for programming interactive applications. Witold has been a Qt user since 2004 and was an active contributor to QtForum.org community forum before January 2006—when he established QtCentre.org with Axel Jäger, Daniel Kish, Jacek Piotrowski, and Johan Thelin. It has since become the biggest actively maintained, community-based Qt-related site and forum.

Witold has been practicing the traditional Seven Star Praying Mantis Kung-Fu style since 1989 and has achieved success in domestic tournaments. He is interested in IT, sports, martial arts, astrophysics, and history. He lives in Warsaw.

Acknowledgments

There are so many people I want to thank—everybody involved in the project has been helpful, positive, and supportive. It has been a great time working with all of you.

First, many thanks go to Witold Wysota, who has provided me with feedback, technical input, and kind words. Without his support I could not have completed this project. I would also like to thank Jason Gilmore from Apress for his excellent feedback and writing tips. Thanks to him, the text is far more enjoyable to read.

Jasmin Blanchette of Trolltech helped me by producing screenshots from the Mac. The excellent support team at Trolltech also clarified unclear issues and fixed bugs. Everyone at Trolltech has been very positive and supportive.

I want to thank all the people at Apress: Matt Wade, who gave me the chance to do this; Elizabeth Seymour, Grace Wong, and Tracy Brown Collins for managing the project. An extra thanks to Tracy who pushed me the last mile to get the project done on time.

Without the help of Nancy Sixsmith's language skills, the text would not have been as easy to read. Thanks to her attention to detail and excellent writing abilities, the text reads as well as it does today.

There are so many people involved in this project that I have not worked with so closely. I'm still very grateful to their efforts and appreciate their skills. Many thanks go to Kelly Winquist, Dina Quan, Brenda Miller, April Milne, and Paulette McGee.

PART 1

Getting to Know Qt

In the first few chapters of this book, you will get acquainted with the Qt way of doing things—including using available classes as well as creating your own classes that interact with the existing ones. You will also learn about the build system and some of the tools available to help make the lives of Qt developers easier.

CHAPTER 1

The Qt Way of C++

Qt is a cross-platform, graphical, application development toolkit that enables you to compile and run your applications on Windows, Mac OS X, Linux, and different brands of Unix. A large part of Qt is devoted to providing a platform-neutral interface to everything, ranging from representing characters in memory to creating a multithreaded graphical application.

Note Even though Qt was originally developed to help C++ programmers, bindings are available for a number of languages. Trolltech provides official bindings for C++, Java, and JavaScript. Third parties provide bindings for many languages, including Python, Ruby, PHP, and the .NET platform.

This chapter starts by taking an ordinary C++ class and integrating it with Qt to make it more reusable and easier to use. In the process, you have a look at the build system used to compile and link Qt applications as well as installing and setting up Qt on your platform.

The chapter then discusses how Qt can enable you to build components that can be interconnected in very flexible ways. This is what makes Qt such a powerful tool—it makes it easy to build components that can be reused, exchanged, and interconnected. Finally, you learn about the collection and helper classes offered by Qt.

Installing a Qt Development Environment

Before you can start developing Qt applications, you need to download and set up Qt. You will use the open source edition of Qt because it is freely available for all. If you have a commercial license for Qt, you have received installations instructions with it.

The installation procedure differs slightly depending on the platform that you are planning to use for development. Because Mac OS X and Linux are both based on Unix, the installation process is identical for the two (and all Unix platforms). Windows, on the other hand, is different and is covered separately. You can start all three platforms by downloading the edition suitable for your platform from `www.trolltech.com/products/qt/downloads`.

Installing on Unix Platforms

All platforms except Windows can be said to be Unix platforms. However, Mac OS X differs from the rest because it does not use the X Window System, more commonly known as X11, for handling graphics. So Mac OS X needs a different Qt edition; the necessary file (`qt-mac-opensource-src-`*`version`*`.tar.gz`) can be downloaded from Trolltech. The X11-based Unix platforms use the `qt-x11-opensource-src-`*`version`*`.tar.gz` file from Trolltech.

Note Qt depends on other components such as compilers, linkers, and development libraries. The requirements differ depending on how Qt is configured, so you should study the reference documentation if you run into problems.

When the file has been downloaded, the procedure goes like this: unpack, configure, and compile. Let's go through these steps one by one. The easiest way is to work from the command prompt.

To unpack the file, download it, place it in a directory, and go there in your command shell. Then type something like this (put **x11** or **mac** in place of *edition* and use the *version* that you have downloaded):

```
tar xvfz qt-edition-opensource-src-version.tar.gz
```

This code extracts the file archive to a folder named `qt-`*`edition`*`-opensource-src-`*`version`*. Use the `cd` command to enter that directory:

```
cd qt-edition-opensource-src-version
```

Before building Qt, you need to configure it using the `configure` script and its options. Run the script like this:

```
./configure options
```

There are lots of options to choose from. The best place to start is to use `-help`, which shows you a list of the available options. Most options can usually be left as the default, but the `-prefix` option is good to use. You can direct the installation to go to a specific location by specifying a path just after the option. For instance, to install Qt in a directory called `inst/qt4` in your home directory, use the following `configure` command:

```
./configure -prefix ~/inst/qt4
```

The Mac OS X platform has two other options that are important to note. First, adding the `-universal` option creates universal binaries using Qt. If you plan to use a PowerPC-based computer for your development, you have to add the `-sdk` option.

The `configure` script also makes you accept the open source license (unless you have a commercial license) before checking that all the dependencies are in place and starting to create configuration files in the source tree. When the script is done, you can build Qt using the following command:

```
make
```

This process will take a relatively long time to complete, but after it finishes you can install Qt by using the next line:

```
make install
```

Note The installation command might need root access if you try to install Qt outside your home directory.

When Qt has been installed, you need to add Qt to your `PATH` environment variable. If you are using a compiler that does not support `rpath`, you have to update the `LD_LIBRARY_PATH` environment variable as well.

If you used the `$HOME/inst/qt4` prefix when running `configure`, you need to add the path `$HOME/inst/qt4/bin` to `PATH`. If you are using a bash shell, change the variable using an assignment:

```
export PATH=$HOME/inst/qt4/bin:$PATH
```

If you want this command to run every time you start a command shell, you can add it to your `.profile` file just before a line that reads `export PATH`. This exports the new `PATH` environment variable to the command-line session.

Note The methods for setting up environment variables differ from shell to shell. If you are not using bash, please refer to the reference documentation on how to set the `PATH` variable for your system.

If you have several Qt versions installed at once, make sure that the version that you intend to use appears first in the `PATH` environment variable because the `qmake` binary used knows where Qt has been installed.

If you have to change the `LD_LIBRARY_PATH` environment variable, add the `$HOME/inst/qt4/lib` directory to the variable. On Mac OS X and Linux (which use the Gnu Compiler Collection [GCC]), this step is not needed.

Installing on Windows

If you plan to use the Windows platform for your Qt development, download a file called `qt-win-opensource-`*`version`*`-mingw.exe` from Trolltech. This file is an installer that will set up Qt and a mingw environment.

Note *mingw*, which is short for Minimalist GNU for Windows, is a distribution of common GNU tools for Windows. These tools, including GCC and make, are used by the open source edition of Qt for compiling and linking.

The installer works as a guide, asking you where to install Qt. Make sure to pick a directory path free from spaces because that can cause you problems later. After you install Qt, you see a Start menu folder called `Qt by Trolltech (OpenSource)`. This folder contains entries for the Qt tools and documentation as well as a Qt command prompt. It is important that you access Qt from this command prompt because it sets up the environment variables such as `PATH` correctly. Simply running the command prompt found in the `Accessories` folder on the Start menu will fail because the variables are not properly configured.

Making C++ "Qt-er"

Because this is a book on programming, you will start with some code right away (see Listing 1-1).

Listing 1-1. *A simple C++ class*

```
#include <string>
using std::string;
class MyClass
{
public:
  MyClass( const string& text );

  const string& text() const;
  void setText( const string& text );

  int getLengthOfText() const;

private:
  string m_text;
};
```

The class shown in Listing 1-1 is a simple string container with a method for getting the length of the current text. The implementation is trivial, `m_text` is simply set or returned, or the size of `m_text` is returned. Let's make this class more powerful by using Qt. But first, take a look at the parts that already are "Qt-ish":

- The class name starts with an uppercase letter and the words are divided using *Camel-Casing*. That is, each new word starts with an uppercase letter. This is the common way to name Qt classes.
- The names of the methods all start with a lowercase letter, and the words are again divided by using CamelCasing. This is the common way to name Qt methods.
- The getter and setter methods of the property text are named `text` (getter) and `setText` (setter). This is the common way to name getters and setters.

They are all traits of Qt. It might not seem like a big thing, but having things named in a structured manner is a great timesaver when you are actually writing code.

Inheriting Qt

The first Qt-specific adjustment you will make to the code is really simple: you will simply let your class inherit the `QObject` class, which will make it easier to manage instances of the class dynamically by giving instances parents that are responsible for their deletion.

■Note All Qt classes are prefixed by a capital Q. So if you find the classes `QDialog` and `Dialog`, you can tell right away that `QDialog` is the Qt class, whereas `Dialog` is a part of your application or third-party code. Some third-party libraries use the `QnnClassName` naming convention, which means that the class belongs to a library extending Qt. The `nn` from the prefix tells you which library the class belongs to. For example, the class `QwtDial` belongs to the Qt Widgets for Technical Applications library that provides classes for graphs, dials, and so on. (You can find out more about this and other third-party extensions to Qt in the appendixes.)

The changes to the code are minimal. First, the definition of the class is altered slightly, as shown in Listing 1-2. The `parent` argument is also added to the constructor as a convenience because `QObject` has a function, `setParent`, which can be used to assign an object instance to a parent after creation. However, it is common—and recommended—to pass the parent as an argument to the constructor as the first default argument to avoid having to type `setParent` for each instance created from the class.

Listing 1-2. *Inheriting* `QObject` *and accepting a parent*

```
#include <QObject>
#include <string>
using std::string;

class MyClass : public QObject
{
public:
  MyClass( const string& text, QObject *parent = 0 );
...
};
```

■Note To access the `QObject` class, the header file `<QObject>` has to be included. This works for most Qt classes; simply include a header file with the same name as the class, omitting the `.h`, and everything should work fine.

The parent argument is simply passed on to the `QObject` constructor like this:

```
MyClass::MyClass( const string& text, QObject *parent ) : QObject( parent )
```

Let's look at the effects of the change, starting with Listing 1-3. It shows a `main` function using the `MyClass` class dynamically without Qt.

Listing 1-3. *Dynamic memory without Qt*

```
#include <iostream>
int main( int argc, char **argv )
{
  MyClass *a, *b, *c;

  a = new MyClass( "foo" );
  b = new MyClass( "ba-a-ar" );
  c = new MyClass( "baz" );

  std::cout << a->text() << " (" << a->getLengthOfText() << ")" << std::endl;
  a->setText( b->text() );
  std::cout << a->text() << " (" << a->getLengthOfText() << ")" << std::endl;

  int result = a->getLengthOfText() - c->getLengthOfText();

  delete a;
  delete b;
  delete c;

  return result;
}
```

Each `new` call must be followed by a call to `delete` to avoid a memory leak. Although it is not a big issue when exiting from the `main` function (because most modern operating systems free the memory when the application exits), the destructors are not called as expected. In locations other than loop-less `main` functions, a leak eventually leads to a system crash when the system runs out of free memory. Compare it with Listing 1-4, which uses a parent that is automatically deleted when the `main` function exits. The parent is responsible for calling `delete` for all children and—ta-da!—the memory is freed.

■ **Note** In the code shown in Listing 1-4, the `parent` object is added to show the concept. In real life, it would be an object performing some sort of task—for example, a `QApplication` object, or (in the case of a dialog box or a window) the `this` pointer of the `window` class.

Listing 1-4. *Dynamic memory with Qt*

```
#include <QtDebug>
int main( int argc, char **argv )
{
  QObject parent;
  MyClass *a, *b, *c;

  a = new MyClass( "foo", &parent );
  b = new MyClass( "ba-a-ar", &parent );
  c = new MyClass( "baz", &parent );

  qDebug() << QString::fromStdString(a->text())
           << " (" << a->getLengthOfText() << ")";
  a->setText( b->text() );
  qDebug() << QString::fromStdString(a->text())
           << " (" << a->getLengthOfText() << ")";

  return a->getLengthOfText() - c->getLengthOfText();
}
```

You even saved the extra step of having to keep the calculated result in a variable because the dynamically created objects can be used directly from the `return` statement. It might look odd to have a parent object like this, but most Qt applications use a `QApplication` object to act as a parent.

■ **Note** Listing 1-4 switched from using `std::cout` for printing debugging messages to `qDebug()`. The nice thing about using `qDebug()` is that it sends the message to the right place on all platforms. It is also easy to turn off: simply define the `QT_NO_DEBUG_OUTPUT` symbol when compiling. If you have debugging messages after which you want to terminate the application, Qt provides the `qFatal()` function, which works just like `qDebug()`, but terminates the application after the message. The compromise between the two is to use `qWarning()`, which indicates something more serious than a debug message, but nothing fatal. The Qt functions for debugging messages automatically appends a line break after each call, so you do not have to include the `std::endl` any more.

When comparing the code complexity in Listing 1-3 and Listing 1-4, look at the different memory situations, as shown in Figure 1-1. The parent is gray because it is allocated on the stack and thus automatically deleted, whereas the instances of `MyClass` are white because they are on the heap and must be handled manually. Because you use the parent to keep track of the children, you trust the parent to delete them when it is being deleted. So you no longer have to keep track of the dynamically allocated memory as long as the root object is on the stack (or if you keep track of it).

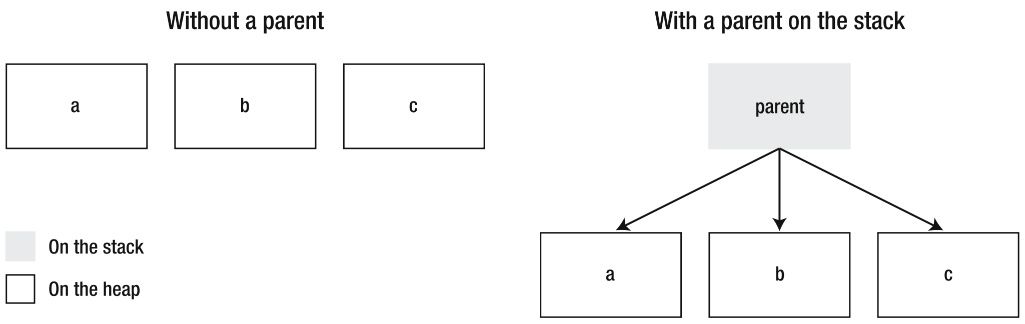


Figure 1-1. *Difference between dynamic memory with a parent and without a parent on the stack*

Using a Qt String

Another step toward using Qt is to replace any classes from the C++ standard template library (STL) with the corresponding Qt class. Although it is not required (Qt works great alongside the STL), it does make it possible to avoid having to rely on a second framework. The benefit of not using the STL is that you use the same containers, strings, and helpers as Qt does, so the resulting application will most likely be smaller. You also avoid having to track down compatibility issues and strange deviations from the STL standard when moving between platforms and compilers—you can even develop on platforms that do not have implementations of the STL.

Looking at the class as it currently stands, spot the `string` class as the only STL class used. The corresponding Qt class is called `QString`. You can mix `QString` objects and `string` objects seamlessly, but using only `QString` means performance gains and more features. For example, `QString` supports Unicode on all platforms, making it a lot easier for international users to use your application.

Listing 1-5 shows how your code looks after replacing all occurrences of `string` with `QString`. As you can see, the changes to the class are minimal.

Listing 1-5. `MyClass` *using* `QString` *instead of* `string`

```
#include <QString>
#include <QObject>

class MyClass : public QObject
{
public:
  MyClass( const QString& text, QObject *parent = 0 );

  const QString& text() const;
  void setText( const QString& text );

  int getLengthOfText() const;
private:
  QString m_text;
};
```

Tip When mixing `string` and `QString`, use the `QString` methods `toStdString` and `fromStdString` to convert to and from the Qt Unicode format to the ASCII representation used by the `string` class.

Building a Qt Program

Compiling and building this application should not be any different from building the original application. All that you have to do is make sure that the compiler can find the Qt headers and that the linker can find the Qt library files.

To handle all this smoothly and in a cross-platform manner, Qt comes with the QMake tool, which can create Makefiles for a range of different compilers. It even creates the project definition file for you if you want it to.

Try this by building a simple application. Start by creating a directory called `testing`. Then put the code from Listing 1-6 inside this directory. You can call the file anything as long as it has the `cpp` extension.

Listing 1-6. *A trivial example*

```
#include <QtDebug>

int main( )
{
    qDebug() << "Hello Qt World!";

    return 0;
}
```

Now open a command line and change your working directory to the one that you just created. Then type **`qmake -project`** and press Enter, which should generate a file named `testing.pro`. My version of that file is shown in Listing 1-7.

Tip If you are running the open-source version of Qt in Windows, you have an application called something like Qt 4.2.2 Command Prompt in the Start menu folder that was created when you installed Qt. Run this application and use the `cd` command to change the directory. For example, first locate your folder using Explorer; then copy the entire path (it should be similar to `c:\foo\bar\baz\testing`). Now type **`cd`**, followed by a space at the command prompt before you right-click, select Paste, and then press Enter. That should get you to the right working directory in a snap.

Listing 1-7. *A generated project file*

```
######################################################################
# Automatically generated by qmake (2.00a) to 10. aug 17:06:34 2006
######################################################################

TEMPLATE = app
TARGET +=
DEPENDPATH += .
INCLUDEPATH += .

# Input
SOURCES += anything.cpp
```

The file consists of a set of variables that are set by using `=` or extended by using `+=`. The interesting part is the `SOURCES` variable, which tells you that QMake has found the `anything.cpp` file. The next step is to generate a platform-specific Makefile using QMake. Because the working directory contains only one project file, simply type **`qmake`** and press Enter. This should give you a Makefile and platform-specific helper files.

Note On GNU/Linux, the result is a single file called `Makefile`. On Windows, if you use the open-source edition and mingw you get `Makefile`, `Makefile.Release`, `Makefile.Debug`, and two directories: `debug` and `release`.

The last step is to build the project from the generated Makefile. How to do this depends on which platform and compiler you are using. You should usually type **`make`** and press Enter, but `gmake` (common on Berkeley Software Distribution [BSD] systems) and `nmake` (on Microsoft compilers) are other common alternatives. Try looking in your compiler manual if you cannot get it to work at the first try.

Tip When running Windows, applications do not get a console output by default. This means that Windows applications cannot, by default, write output to the command-line users. To see any output from `qDebug()`, you must add a line reading `CONFIG += console` to the project file. If you built the executable and then saw this tip, try fixing the project file; then run `make clean` followed by `make`. This process ensures that the project is completely rebuilt and that the new configuration is taken into account.

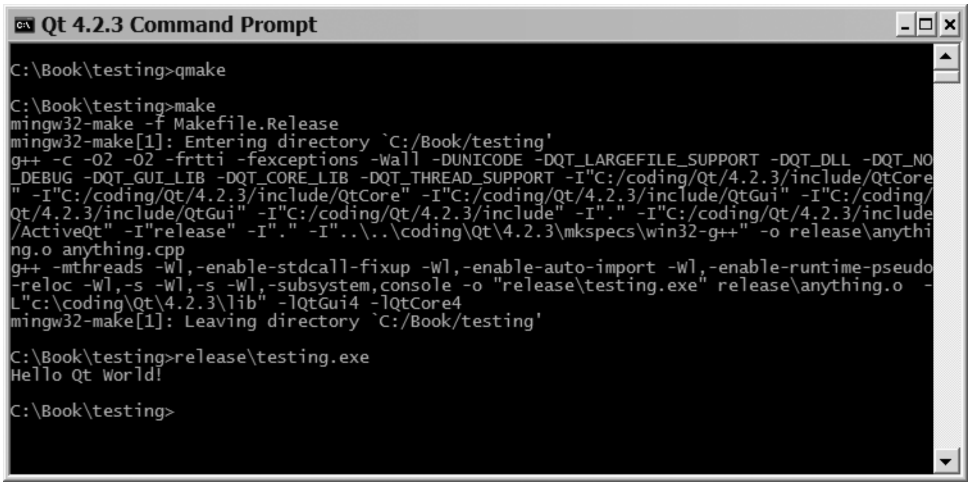
The only thing left to do now is to run the application and watch this message: `Hello Qt World!`. The executable will have the same name as the directory that you used. For Windows users, the executable ends up in the release directory with the `exe` file name extension, so you start it by running the following command:

```
release\testing.exe
```

On other platforms it is usually located directly in the working directory, so you start it by typing the following:

```
./testing
```

On all platforms the result is the same: the `Hello Qt World!` message is printed to the console. The resulting command prompt on the Windows platform is shown in Figure 1-2.



```
C:\Book\testing>qmake

C:\Book\testing>make
mingw32-make -f Makefile.Release
mingw32-make[1]: Entering directory `C:/Book/testing'
g++ -c -O2 -O2 -frtti -fexceptions -Wall -DUNICODE -DQT_LARGEFILE_SUPPORT -DQT_DLL -DQT_NO
_DEBUG -DQT_GUI_LIB -DQT_CORE_LIB -DQT_THREAD_SUPPORT -I"C:/coding/Qt/4.2.3/include/QtCore
" -I"C:/coding/Qt/4.2.3/include/QtCore" -I"C:/coding/Qt/4.2.3/include/QtGui" -I"C:/coding/
Qt/4.2.3/include/QtGui" -I"C:/coding/Qt/4.2.3/include" -I"." -I"C:/coding/Qt/4.2.3/include
/ActiveQt" -I"release" -I"." -I"..\..\coding\Qt\4.2.3\mkspecs\win32-g++" -o release\anythi
ng.o anything.cpp
g++ -mthreads -Wl,-enable-stdcall-fixup -Wl,-enable-auto-import -Wl,-enable-runtime-pseudo
-reloc -Wl,-s -Wl,-s -Wl,-subsystem,console -o "release\testing.exe" release\anything.o  -
L"c:\coding\Qt\4.2.3\lib" -lQtGui4 -lQtCore4
mingw32-make[1]: Leaving directory `C:/Book/testing'

C:\Book\testing>release\testing.exe
Hello Qt World!

C:\Book\testing>
```

Figure 1-2. *A Qt application running from the command prompt*

Signals, Slots, and Meta-Objects

Two of the biggest strengths that Qt brings to C++ are *signals* and *slots*, which are very flexible ways to interconnect objects and help to make code easy to design and reuse.

A *signal* is a method that is emitted rather than executed when called. So from your viewpoint as a programmer, you declare prototypes of signals that might be emitted. Do not implement signals; just declare them in the class declaration in the `signals` section of your class.

A *slot* is a member function that can be invoked as a result of signal emission. You have to tell the compiler which methods to treat as slots by putting them in one of these sections: `public slots`, `protected slots`, or `private slots`. The protection level protects the slot only when it is being used as a method. You can still connect a `private` slot or a `protected` slot to a signal that you receive from another class.

When it comes to connecting signals and slots, you can connect any number of signals to any number of slots. This means that a single slot can be connected to many signals, and a single signal can be connected to many slots. There are no limitations to how you interconnect your objects. When a signal is emitted, all slots connected to it are called. The order of the calls is undefined, but they do get called. Let's look at some code that shows a class declaring both a signal and a slot (see Listing 1-8).

Listing 1-8. *A class with a signal and a slot*

```
#include <QString>
#include <QObject>
class MyClass : public QObject
{
  Q_OBJECT

public:
  MyClass( const QString &text, QObject *parent = 0 );

  const QString& text() const;
  int getLengthOfText() const;

public slots:
  void setText( const QString &text );

signals:
  void textChanged( const QString& );

private:
  QString m_text;
};
```

The code is a new incarnation of the class `MyClass` you have been working with throughout the chapter. There are changes related to the signals and slots in the three emphasized areas of the listing. Start from the bottom with the new section labeled `signals:`. This tells you that the functions declared in this section will not be implemented by you; they are simply prototypes for the signals that this class can emit. This class has one signal: `textChanged`.

Moving upward, there is another new section: `public slots:`. Slots can be public, protected, or private like any other member—just add the appropriate protection level before the `slots` keyword. Slots can be considered a member function that can be connected to a signal. There is really no other difference; it is declared and implemented just like any other member function of the class.

■ **Tip** Setter methods are natural slots. By making all setters slots, you guarantee that you can connect signals to all interesting parts of your class. The only time when a setter should not also be a slot is when the setter accepts some very custom type that you are sure will never come from a signal.

At the very top of the class declaration you find the `Q_OBJECT` macro. It is important that this macro appears first in the body of the class declaration because it marks the class as a class that needs a meta-object. Let's look at what meta-objects are before continuing.

The word *meta* indicates that the word prefixed is about itself. So a *meta-object* is an object describing the object. In the case of Qt, meta-objects are instances of the class

QMetaObject and contain information about the class such as its name, its super classes, its signals, its slots, and many other interesting things. The important thing to know now is that the meta-object knows about the signals and slots.

This leads into the next implication of this feature. Until now, all the examples have fitted nicely into a single file of source code. It is possible to go on like this, but the process is much smoother if you separate each class into a header and a source file. A Qt tool called the *meta-object compiler*, moc, parses the class declaration and produces a C++ implementation file from it. This might sound complex, but as long as you use QMake to handle the project, there is no difference to you.

This new approach means that the code from Listing 1-8 goes into a file called myclass.h. The implementation goes into myclass.cpp, and the moc generates another C++ file from the header file called moc_myclass.cpp. The contents from the generated file can change between Qt versions and is nothing to worry about. Listing 1-9 contains the part of the implementation that has changed because of signals and slots.

Listing 1-9. *Implementing* MyClass *with signals and slots*

```
void MyClass::setText( const QString &text )
{
  if( m_text == text )
    return;

  m_text = text;
  emit textChanged( m_text );
}
```

The changes made to emit the signal textChanged can be divided into two parts. The first half is to check that the text actually has changed. If you do not check this before you connect the textChanged signal to the setText slot of the same object, you will end up with an infinite loop (or as the user would put it, the application will hang). The second half of the change is to actually emit the signal, which is done using the Qt keyword emit followed by the signal's name and arguments.

SIGNALS AND SLOTS UNDER THE HOOD

Signals and slots are implemented by Qt using function pointers. When calling emit with the signal as argument, you actually call the signal. The signal is a function implemented in the source file generated by the moc. This function calls any slots connected to the signal using the meta-objects of the objects holding the connected slots.

The meta-objects contain function pointers to the slots, along with their names and argument types. They also contain a list of the available signals and their names and argument types. When calling connect, you ask the meta-object to add the slot to the signal's calling list. If the arguments match, the connection is made.

When matching arguments, the match is checked only for the arguments accepted by the slot. This means that a slot that does not take any arguments matches all signals. The arguments not accepted by the slot are simply dropped by the signal-emitting code.

Making the Connection

To try out the signals and slots in `MyClass`, the `a`, `b`, and `c` instances are created:

```
QObject parent;
MyClass *a, *b, *c;

a = new MyClass( "foo", &parent );
b = new MyClass( "bar", &parent );
c = new MyClass( "baz", &parent );
```

Now connect them. To connect signals and slots, the `QObject::connect` method is used. The arguments are `source object`, `SIGNAL(`*`source signal`*`)`, `destination object`, `SLOT(`*`destination slot`*`)`. The macros `SIGNAL` and `SLOT` are required; otherwise, Qt refuses to establish the connection. The source and destination objects are pointers to `QObject`s or objects of classes inheriting `QObject`. The source signal and destination slot are the name and argument types of the signal and slot involved. The following shows how it looks in the code. Figure 1-3 shows how the object instances are connected.

```
QObject::connect(
  a, SIGNAL(textChanged(const QString&)),
  b, SLOT(setText(const QString&)) );
QObject::connect(
  b, SIGNAL(textChanged(const QString&)),
  c, SLOT(setText(const QString&)) );
QObject::connect(
  c, SIGNAL(textChanged(const QString&)),
  b, SLOT(setText(const QString&)) );
```

■ **Caution** Trying to specify signal or slot argument *values* when connecting will cause your code to fail at run-time. The `connect` function understands only the argument `types`.

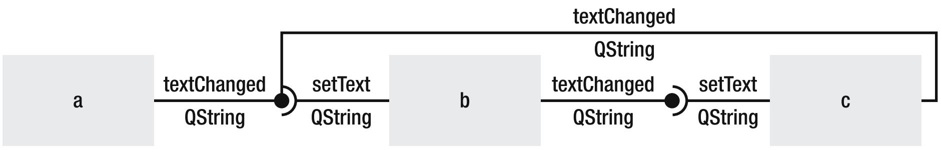


Figure 1-3. *The connections between* `a`, `b`, *and* `c`

The following line shows a call to one of the objects:

```
b->setText( "test" );
```

Try tracing the call from `b`, where there is a change from `"bar"` to `"test"`; through the connection to `c`, where there is a change from `"baz"` to `"test"`; and through the connection to `b`,