GAME PROGRAMMING USING QT 6

BEGINNER'S GUIDE

INSTALLATION & BACKGROUND INFO

Create Amazing Games With QT 6, C++ AND Qt Quick

Preface

As a leading cross-platform toolkit for all significant desktop, mobile, and embedded platforms, Qt is becoming more popular by the day. This book will help you learn the nitty-gritty of Qt and will equip you with the necessary toolsets to build apps and games. This book is designed as a beginner's guide to take programmers new to Qt from the basics, such as objects, core classes, widgets, and new features in version 5.9, to a level where they can create a custom application with the best practices of programming with Qt.

From a brief introduction of how to create an application and prepare a working environment for both desktop and mobile platforms, we will dive deeper into the basics of creating graphical interfaces and Qt's core concepts of data processing and display. As you progress through the chapters, you'll learn to enrich your games by implementing network connectivity and employing scripting. Delve into Qt Quick, OpenGL, and various other tools to add game logic, design animation, add game physics, handle gamepad input, and build astonishing UIs for games. Toward the end of this book, you'll learn to exploit mobile device features, such as sensors and geolocation services, to build engaging user experiences.

Who this book is for

This book will be interesting and helpful to programmers and application and UI developers who have basic knowledge of C++. Additionally, some parts of Qt allow you to use JavaScript, so basic knowledge of this language will also be helpful. No previous experience with Qt is required. Developers with up to a year of Qt experience will also benefit from the topics covered in this book.

I dedicate this book to all people who are passionate about programming; live long and prosper

– Pavel Strakhov

Conventions used

There are a number of text conventions used throughout this book.

`CodeInText`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "This API is centered on `QNetworkAccessManager`,which handles the complete communication between your game and the Internet."

A block of code is set as follows:

```
QNetworkRequest request;
request.setUrl(QUrl("http://localhost/version.txt"));
request.setHeader(QNetworkRequest::UserAgentHeader, "MyGame");
m_manager->get(request);
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
void FileDownload::downloadFinished(QNetworkReply *reply) {
    const QByteArray content = reply->readAll();
    _edit->setPlainText(content);
    reply->deleteLater();
}
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "On the Select Destination Location screen, click on Next to accept the default destination."

Warnings or important notes appear like this.

Tips and tricks appear like this.

Introduction to Qt

In this chapter, you will learn what Qt is and how it evolved. We will describe the structure of the Qt framework and the differences between its versions. Finally, you will learn how to decide which Qt licensing scheme is right for your projects.

The main topics covered in this chapter are:

- Qt history
- Supported platforms
- Structure of the Qt framework
- Qt versions
- Qt licenses

A journey through time

The development of Qt started in 1991 by two Norwegians—Eirik Chambe-Eng and Haavard Nord—who were looking to create a cross-platform GUI programming toolkit. The first commercial client of Trolltech (the company that created the Qt toolkit) was the European Space Agency. The commercial use of Qt helped Trolltech sustain further development. At that time, Qt was available for two platforms—Unix/X11 and Windows—however, developing with Qt for Windows required buying a proprietary license, which was a significant drawback in porting the existing Unix/Qt applications.

A major step forward was the release of Qt Version 3.0 in 2001, which saw the initial support for Mac as well as an option to use Qt for Unix and Mac under a liberal GPL license. Still, Qt for Windows was only available under a paid license. Nevertheless, at that time, Qt had support for all the important players in the market—Windows, Mac, and Unix desktops, with Trolltech's mainstream product and Qt for embedded Linux.

In 2005, Qt 4.0 was released, which was a real breakthrough for a number of reasons. First, the Qt API was completely redesigned, which made it cleaner and more coherent. Unfortunately, at the same time, it made the existing Qt-based code incompatible with 4.0, and many applications needed to be rewritten from scratch or required much effort to be adapted to the new API. It was a difficult decision, but from the time perspective, we can see it was worth it. Difficulties caused by changes in the API were well countered by the fact that Qt for Windows was finally released under GPL. Many optimizations were introduced that made Qt significantly faster. Lastly, Qt, which was a single library until now, was divided into a number of modules. This allowed programmers to only link to the functionality that they used in their applications, reducing the memory footprint and the dependencies of their software.

In 2008, Trolltech was sold to Nokia, which at that time was looking for a software framework to help it expand and replace its Symbian platform in the future. The Qt community became divided; some people were thrilled, others were worried after seeing Qt's development get transferred to Nokia. Either way, new funds were pumped into Qt, speeding up its progress and opening it for mobile platforms—Symbian and then Maemo and MeeGo.

For Nokia, Qt was not considered a product of its own, but rather a tool. Therefore, Nokia decided to introduce Qt to more developers by adding a very liberal **Lesser General Public License** (**LGPL**) that allowed the usage of the framework for both open and closed source development.

Bringing Qt to new platforms and less powerful hardware required a new approach to create user interfaces and to make them more lightweight, fluid, and attractive. Nokia engineers working on Qt came up with a new declarative language to develop such interfaces—the **Qt Modeling Language** (**QML**) and a Qt runtime for it called **Qt Quick**.

The latter became the primary focus of the further development of Qt, practically stalling all non-mobile-related work, channeling all efforts to make Qt Quick faster, easier, and more widespread. Qt 4 was already in the market for seven years, and it became obvious that another major version of Qt had to be released. It was decided to bring more engineers to Qt by allowing anyone to contribute to the project. The Qt Project founded by Nokia in 2011 provided an infrastructure for code review and introduced an open governance model, allowing outside developers to participate in decision making.

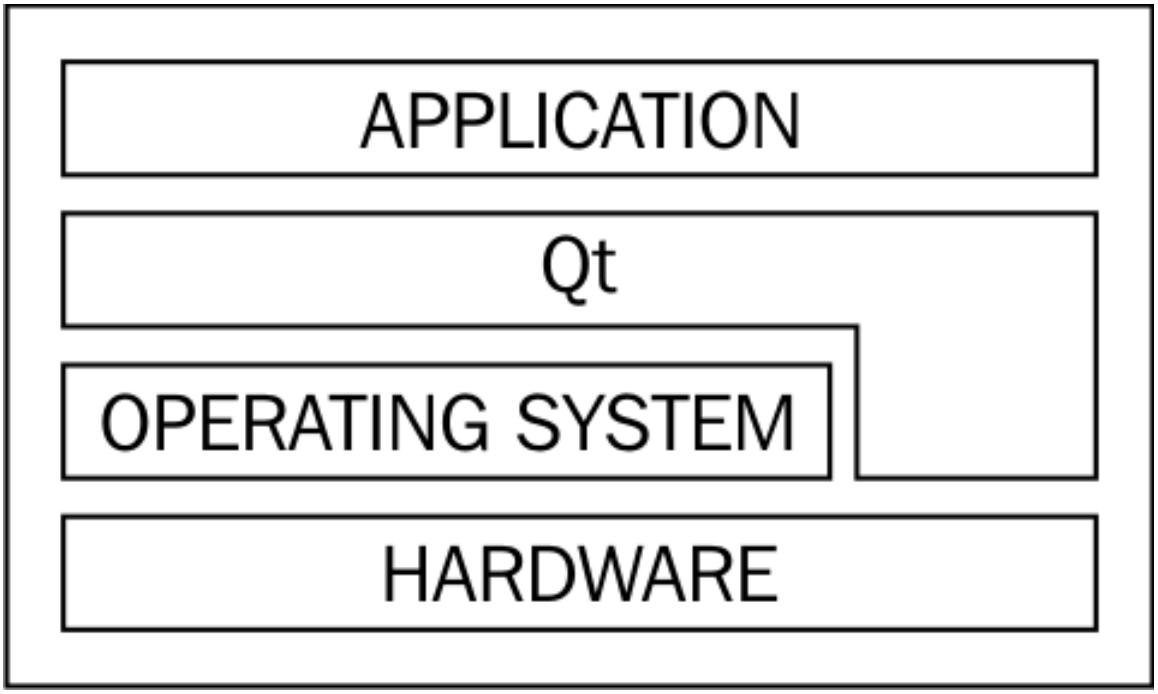
Nokia did not manage to finish working on Qt 5.0. As a result of an unexpected turnover of Nokia toward different technology in 2011, the Qt division was sold in mid 2012 to the Finnish company Digia that managed to complete the effort and release Qt 5.0, a completely restructured framework, in December of the same year. While Qt 5.0 introduced a lot of new features, it was mostly compatible with Qt 4 and allowed developers to seamlessly migrate to the new major version.

Qt 6 was officially released on December 8, 2020. The latest version of Qt is 6.5, released on April 3, 2023. Porting from Qt 5 to Qt 6 has been intentionally kept easy. There has been a conscious effort throughout the development of Qt 6 to maintain as much source compatibility with Qt 5 as possible.
Qt 6.2 was the next most important feature release in the Qt 6 generation, since it closed most gaps to Qt 5 in terms of available modules and functionality. Still, some effort is involved in porting. There are several resources available to help with porting from Qt 5 to Qt 6, including a dedicated page about this in the Qt Documentation - "Porting to Qt 6".

In 2014, Digia formed the Qt Company that is now responsible for Qt development, commercialization, and licensing. All Qt-related web resources scattered across Qt Project and Digia websites were eventually unified at https://www.qt.io/. Qt continues to receive bug fixes, new features, and new platform support. This book is based on Qt 6, which was released in 2023.

The cross-platform programming

Qt is an application-programming framework that is used to develop cross-platform applications. What this means is that software written for one platform can be ported and executed on another platform with little or no effort. This is obtained by limiting the application source code to a set of calls to routines and libraries available to all the supported platforms, and by delegating all tasks that may differ between platforms (such as drawing on the screen and accessing system data or hardware) to Qt. This effectively creates a layered environment (as shown in the following diagram), where Qt hides all platform-dependent aspects from the application code:



Of course, at times, we need to use some functionality that Qt doesn't provide. In such situations, it is important to use a conditional compilation for platform-specific code. Qt provides a wide set of macros specifying the current platform. We will return to this topic in Chapter 6, *Qt Core Essentials*.

Supported platforms

The framework is available for a number of platforms, ranging from classical desktop environments through embedded systems to mobile devices. Qt 5.9 supports the following platforms:

- Desktop platforms: Windows, Linux, and macOS
- Mobile platforms: UWP, Android, and iOS
- Embedded platforms: VxWorks, INTEGRITY, QNX, and Embedded Linux

It is likely that the list of supported platforms will change in future Qt versions. You should refer to the Supported Platforms documentation page for your Qt version for detailed information about supported versions of operating systems and compilers.

GUI scalability

For the most part of the history of desktop application development, specifying sizes of GUI elements in pixels was the common practice. While most operating systems had **dots per inch** (**DPI**) settings and APIs for taking it into account for a long time, the majority of existing displays had approximately the same DPI, so applications without high DPI support were common.

The situation changed when high-DPI displays became more common in the market—most notably in mobile phones and tablets, but also in laptops and desktops. Now, even if you only target desktop platforms, you should think about supporting different DPI settings. When you target mobile devices, this becomes mandatory.

If you are using Qt Widgets or Qt Quick, you often don't need to specify pixel sizes at all. Standard widgets and controls will use fonts, margins, and offsets defined by the style. If layouts are used, Qt will determine positions and sizes of all GUI items automatically. Avoid specifying constant sizes for GUI elements when possible. You may use sizes related to sizes of other GUI elements, the window, or the screen. Qt also provides an API for querying screen DPI, GUI style metrics, and font metrics, which should help to determine the optimal size for the current device.

On macOS and iOS, Qt Widgets and Qt Quick applications are scaled automatically using a virtual coordinate system. Pixel values in the application remain the same, but the GUI will scale according to the DPI of the current display. For example, if the pixel ratio is set to 2 (a common value for retina displays), creating a widget with 100 "pixels" width will produce a widget with 200 physical pixels. That means that the application doesn't have to be highly aware of DPI variations. However, this scaling does not apply to OpenGL, which always uses physical pixels.

Qt versions

Each Qt version number (for example, 5.9.2) consists of major, minor, and patch components. Qt pays special attention to forwards and backwards compatibility between different versions. Small changes which are both forwards and backwards compatible (typically bug fixes without changing any API) are indicated by changing only the patch version. New minor versions usually bring in new API and features, so they are not forwards compatible. However, all minor versions are backwards binary and source compatible. This means that if you're transitioning to a newer minor version (for example, from 5.8 to 5.9), you should always be able to rebuild your project without changes. You can even transition to a new minor version without rebuilding, by only updating shared Qt libraries (or letting the package manager of the OS do that). Major releases indicate big changes and may break backwards compatibility. However, the major release (5.0) was mostly source compatible with the previous version.

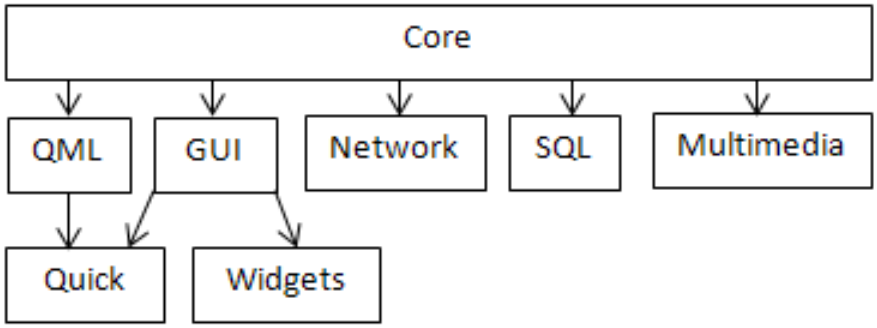
Qt declares **Long Term Support (LTS)** for certain versions. LTS versions receive patch-level releases with bug fixes and security fixes for three years. Commercial support is available for even longer periods. Current LTS releases at the time of writing are 6.4 and 6.5.

Structure of Qt framework

As Qt expanded over time, its structure evolved. At first, it was just a single library, then a set of libraries. When it became harder to maintain and update for the growing number of platforms that it supported, a decision was made to split the framework into much smaller modules contained in two module groups—Qt Essentials and Qt Add-ons. A major decision relating to the split was that each module could now have its own independent release schedule.

Qt Essentials

The Essentials group contains modules that are mandatory to implement for every supported platform. This implies that if you are implementing your system using modules from this group only, you can be sure that it can be easily ported to any other platform that Qt supports. The most important relations between Qt Essentials modules are shown in the following diagram:



Some of the modules are explained as follows:

- The **Qt Core** module contains the most basic Qt functionality that all other modules rely on. It provides support for event processing, meta-objects, data I/O, text processing, and threading. It also brings a number of frameworks, such as the Animation framework, the State Machine framework, and the Plugin framework.

- The **Qt GUI** module provides basic cross-platform support to build user interfaces. It contains the common functionality required by more high-level GUI modules (Qt Widgets and Qt Quick). Qt GUI contains classes that are used to manipulate windows that can be rendered using either the raster engine or OpenGL. Qt supports desktop OpenGL as well as OpenGL ES 1.1 and 2.0.
- **Qt Widgets** extends the GUI module with the ability to create a user interface using widgets, such as buttons, edit boxes, labels, data views, dialog boxes, menus, and toolbars, which are arranged using a special layout engine. Qt Widgets utilizes Qt's event system to handle input events in a cross-platform way. This module also contains the implementation of an object-oriented 2D graphics canvas called Graphics View.
- **Qt Quick** is an extension of Qt GUI, which provides a means to create lightweight fluid user interfaces using QML. It is described in more detail later in this chapter, as well as in Chapter 11, *Introduction to Qt Quick*.
- **Qt QML** is an implementation of the QML language used in Qt Quick. It also provides API to integrate custom C++ types into QML's JavaScript engine and to integrate QML code with C++.
- **Qt Network** brings support for IPv4 and IPv6 networking using TCP and UDP. It also contains HTTP, HTTPS, FTP clients, and it extends support for DNS lookups.
- **Qt Multimedia** allows programmers to access audio and video hardware (including cameras and FM radio) to record and play multimedia content. It also features 3D positional audio support.
- **Qt SQL** brings a framework that is used to manipulate SQL databases in an abstract way.

> *There are also other modules in this group, but we will not focus on them in this book. If you want to learn more about them, you can look them up in the Qt reference manual.*

Qt Add-ons

This group contains modules that are optional for any platform. This means that if a particular functionality is not available on some platform or there is nobody willing to spend time working on this functionality for a platform, it will not prevent Qt from supporting this platform. We'll mention some of the most important modules here:

- **Qt Concurrent**: This handles multi-threaded processing
- **Qt 3D**: This provides high-level OpenGL building blocks
- **Qt Gamepad**: This enables applications to support gamepad hardware
- **Qt D-Bus**: This allows your application to communicate with others via the D-Bus mechanism
- **Qt XML Patterns**: This helps us to access XML data

Many other modules are also available, but we will not cover them here.

qmake

Some Qt features require additional build steps during the compilation and linking of the project. For example, **Meta-Object Compiler** (**moc**), **User Interface Compiler** (**uic**), and **Resource Compiler** (**rcc**) may need to be executed to handle Qt's C++ extensions and features. For convenience, Qt provides the **qmake** executable that manages your Qt project and generates files required for building it on the current platform (such as Makefile for the make utility). qmake reads the project's configuration from a project file with the `.pro` extension. Qt Creator (the IDE that comes with Qt) automatically creates and updates that file, but it can be edited manually to alter the build process.

Alternatively, CMake can be used to organize and build the project. Qt provides CMake plugins for performing all the necessary build actions. Qt Creator also has fairly good support for CMake projects. CMake is more advanced and powerful than qmake, but it's probably not needed for projects with a simple build process.

Modern C++ standards

You can use modern C++ in your Qt projects. Qt's build tool (qmake) allows you to specify the C++ standard you want to target. Qt itself introduces an improved and extended API by using new C++ features when possible. For example, it uses ref-qualified member functions and introduces methods accepting initializer lists and rvalue references. It also introduces new macros that help you deal with compilers that may or may not support new standards.

If you use a recent C++ revision, you have to pay attention to the compiler versions you use across the target platforms because older compilers may not support the new standard. In this book, we will assume C++11 support, as it is widely available already. Thus, we'll use C++11 features in our code, such as range-based `for` loops, scoped enumerations, and lambda expressions.

Choosing the right license

Qt is available under two different licensing schemes—you can choose between a commercial license and an open source one. We will discuss both here to make it easier for you to choose. If you have any doubts regarding whether a particular licensing scheme applies to your use case, you better consult a professional lawyer.

An open source license

The advantage of open source licenses is that we don't have to pay anyone to use Qt; however, the downside is that there are some limitations imposed on how it can be used.

When choosing the open source edition, we have to choose between GPL 3.0 and LGPL 3. Since LGPL is more liberal, in this chapter we will focus on it. Choosing LGPL allows you to use Qt to implement systems that are either open source or closed source—you don't have to reveal the sources of your application to anyone if you don't want to.

However, there are a number of restrictions you need to be aware of:

- Any modifications that you make to Qt itself need to be made public, for example, by distributing source code patches alongside your application binary.
- LGPL requires that users of your application must be able to replace Qt libraries that you provide them with other libraries with the same functionality (for example, a different version of Qt). This usually means that you have to dynamically link your application against Qt so that the user can simply replace Qt libraries with his own. You should be aware that such substitutions can decrease the security of your system; thus, if you need it to be very secure, open source may not be the option for you.
- LGPL is incompatible with a number of licenses, especially proprietary ones, so it is possible that you won't be able to use Qt with some commercial components.

Some Qt modules may have different licensing restrictions. For example, Qt Charts, Qt Data Visualization, and Qt Virtual Keyboard modules are not available under LGPL and can only be used under GPL or the commercial license.

The open source edition of Qt can be downloaded directly from https://www.qt.io.

A commercial license

Most of the restrictions are lifted if you decide to buy a commercial license for Qt. This allows you to keep the entire source code a secret, including any changes you may want to incorporate into Qt. You can freely link your application statically against Qt, which means fewer dependencies, a smaller deployment bundle size, and a faster startup. It also increases the security of your application, as end users cannot inject their own code into the application by replacing a dynamically loaded library with their own.

Summary

In this chapter, you learned about the architecture of Qt. We saw how it evolved over time and we had a brief overview of what it looks like now. Qt is a complex framework and we will not manage to cover it all, as some parts of its functionality are more important for game programming than others that you can learn on your own in case you ever need them. Now that you know what Qt is, we can proceed with the next chapter, where you will learn how to install Qt on to your development machine.

Installation

In this chapter, you will learn how to install Qt on your development machine, including Qt Creator, an IDE tailored to use with Qt. You will see how to configure the IDE for your needs and learn the basic skills to use that environment. By the end of this chapter, you will be able to prepare your working environment for both desktop and embedded platforms using the tools included in the Qt release.

The main topics covered in this chapter are as follows:

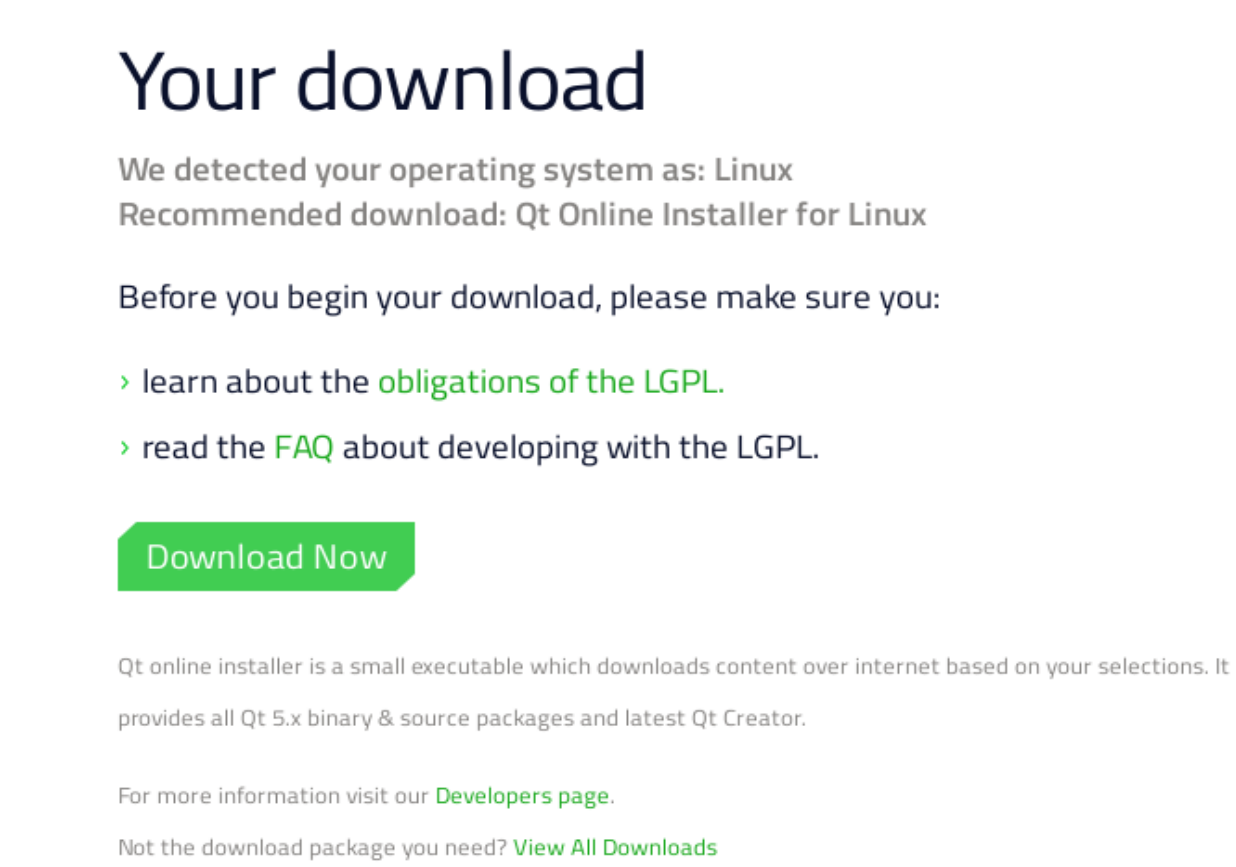
- Installing Qt and its developer tools
- Main controls of Qt Creator
- Qt documentation

Installing the Qt SDK

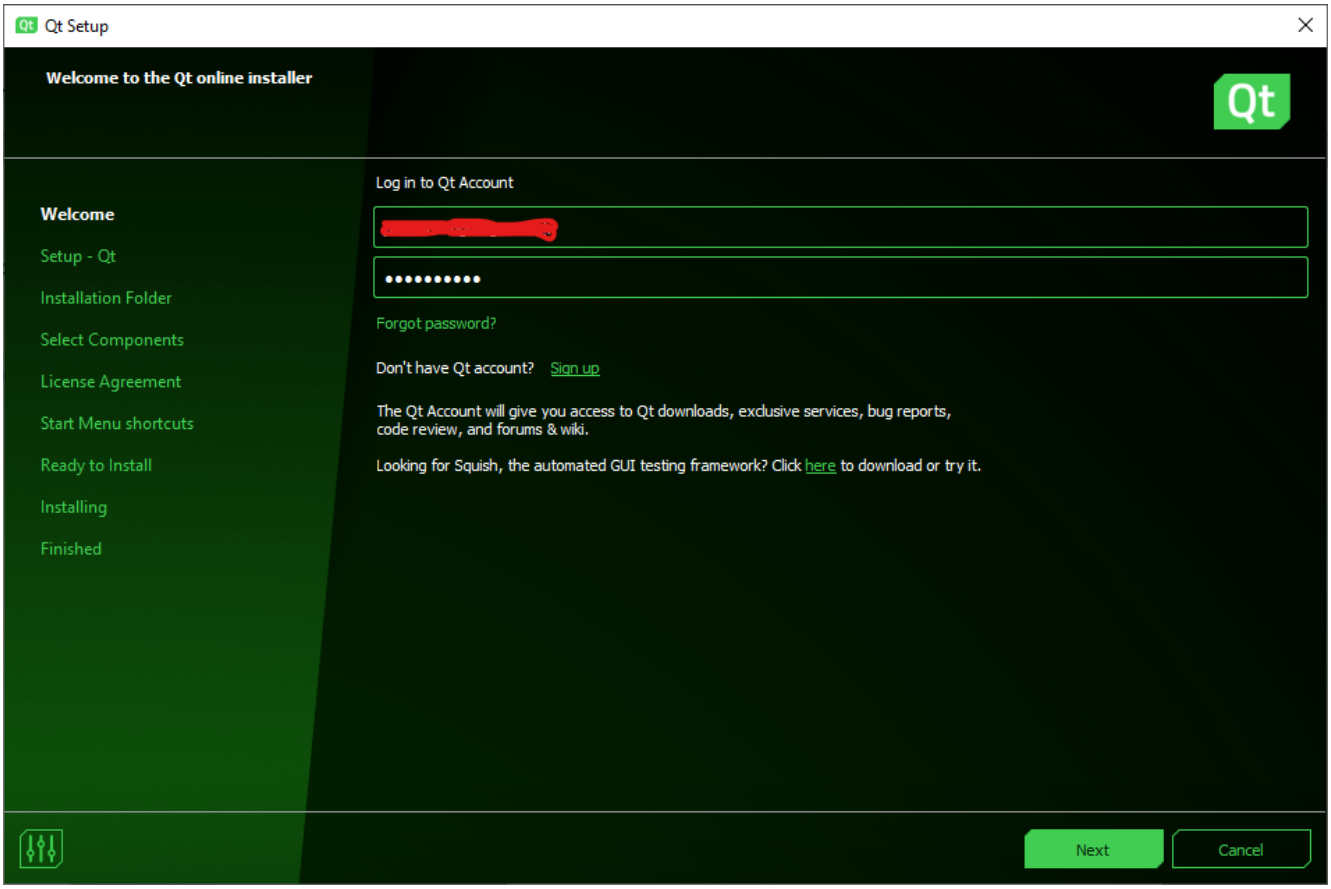
Before you can start using Qt on your machine, it needs to be downloaded and installed. Qt can be installed using dedicated installers that come in two flavors: the online installer, which downloads all the needed components on the fly, and a much larger offline installer, which already contains all the required components. Using an online installer is easier for regular desktop installs, so we prefer this approach.

Time for action – Installing Qt using an online installer

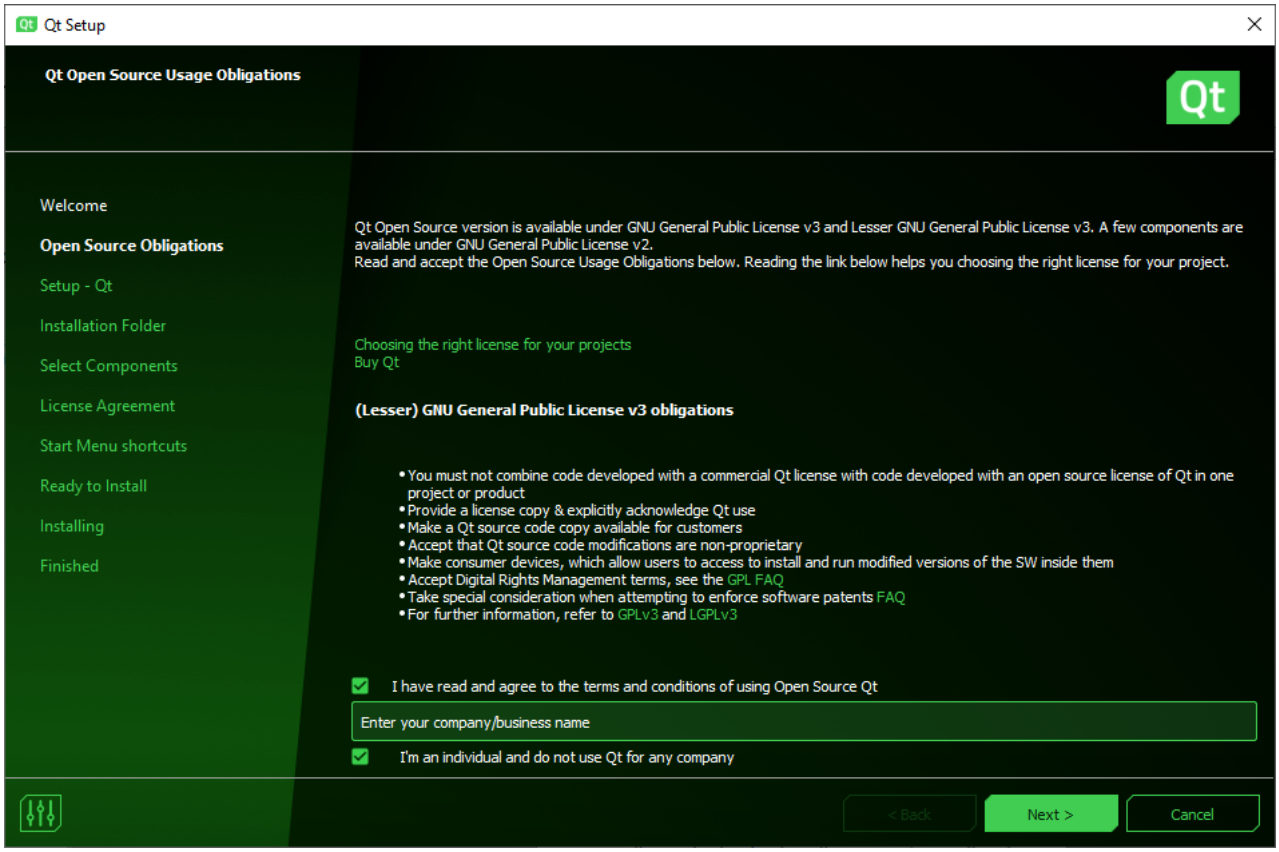
All Qt resources, including the installers, are available at https://qt.io. To obtain the open source version of Qt, go to https://www.qt.io/download-open-source/. The page suggests the online installer for your current operating system by default, as shown in the following screenshot. Click on the Download Now button to download the online installer, or click on View All Downloads to select a different download option:



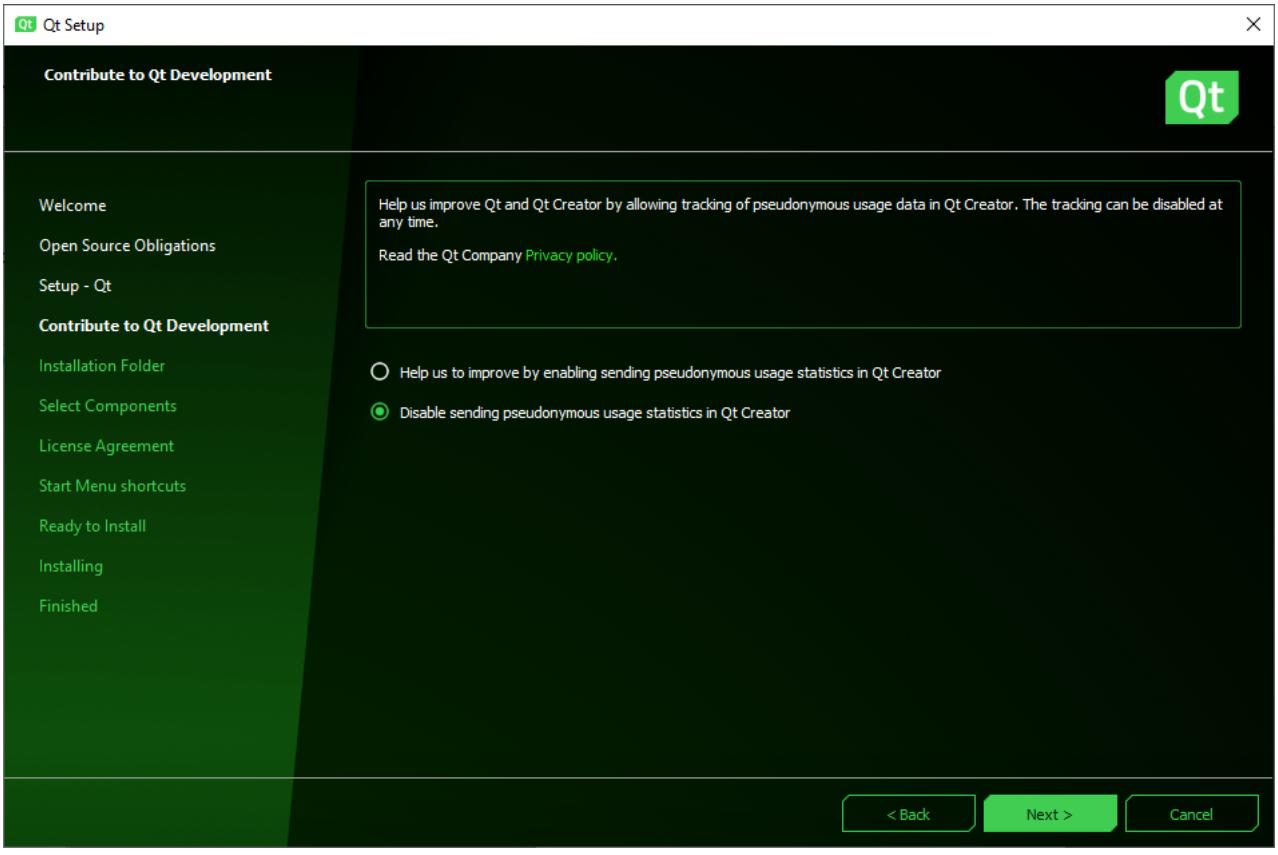
When the download is complete run the installer, as shown:



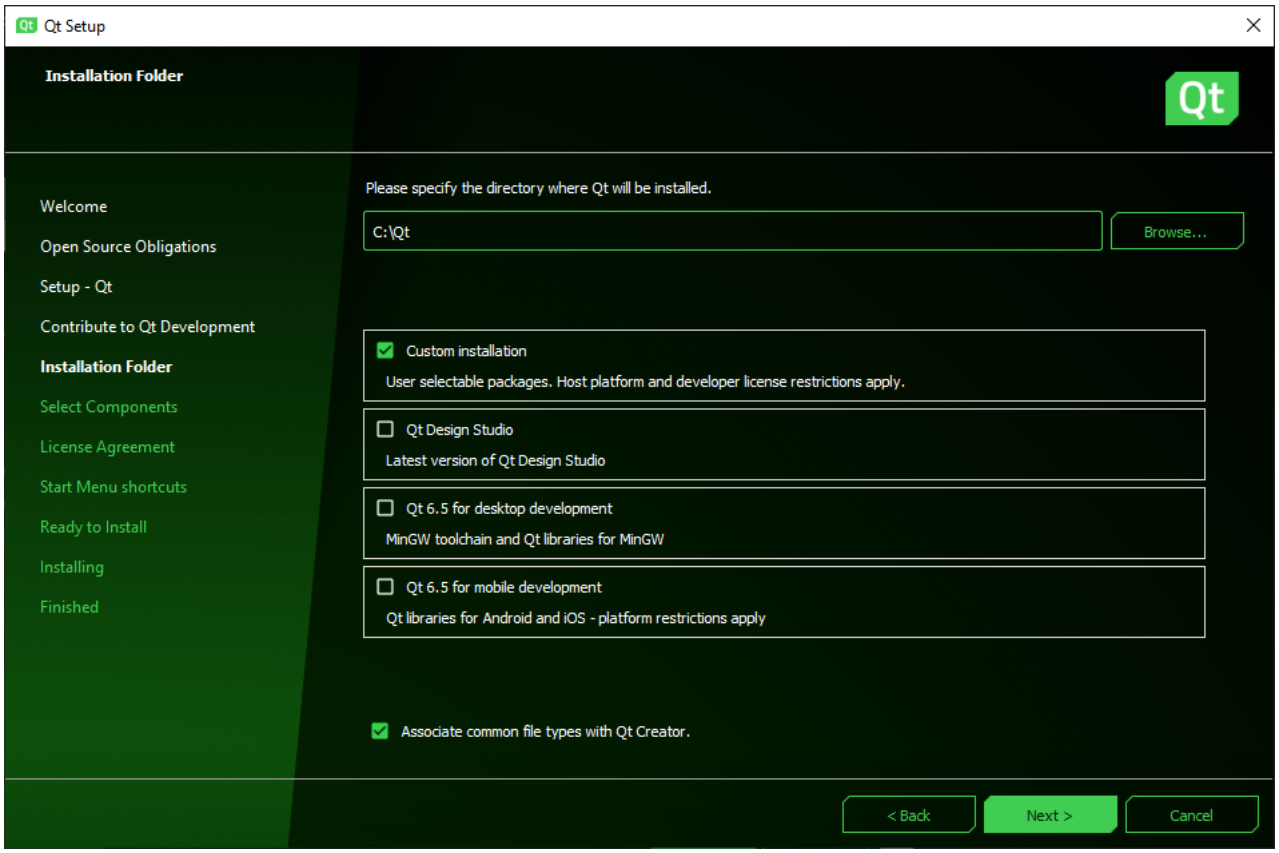
Then, either log into your Qt Account or or create a new account if you don't have one. Click on Next to begin the installation process. If you are using a proxy server, click on Settings and adjust your proxy configuration.



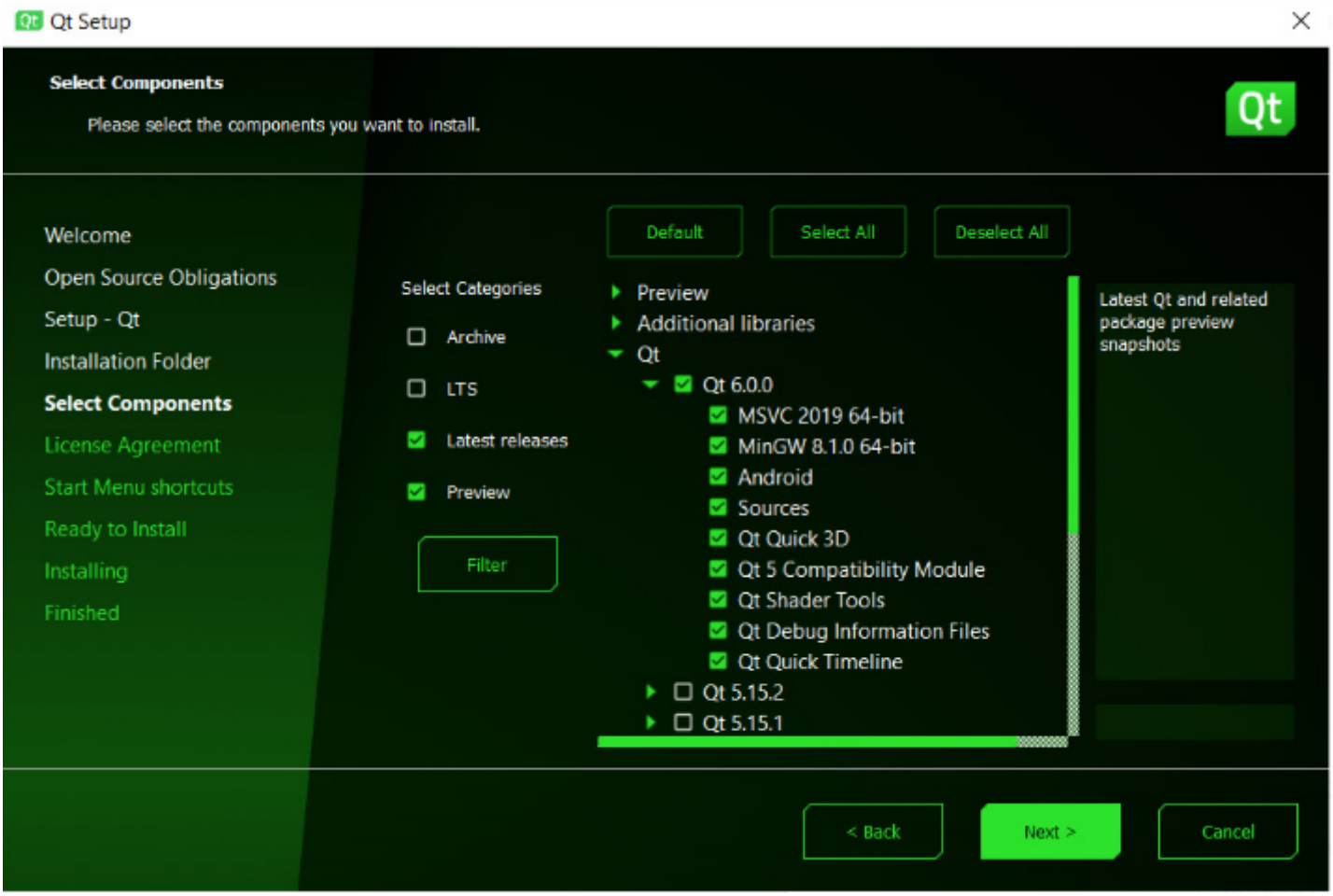
Accept the terms and conditions and enter a company name if you will be using Qt for a business or click on I'm an individual if its for personal use. Then click on next for next step. Click on Next again, and after a while of waiting as the downloader checks remote repositories, and on contribute section select the option you are comfortable with and click next



You'll be asked for the installation path. Ensure that you choose a path where you have write access and enough free space. It's best to put Qt into your personal directory, unless you ran the installer as the system administrator user. Clicking on Next again will present you with the choice of components that you wish to install, as shown in the following screenshot. You will be given different choices depending on your platform:



Select the components you want to use, you can either select custom installation and choose them manually and versions you want or select multiple choices for your use case between Design studio, desktop development or mobile development (choose one, two or all three) and click next and the installer will choose the appropriate components for you, if you want to select manually select custom installation then click next and follow the next instruction.



You need to choose which Qt version you want to install. We recommend that you use the most recent stable version, that is, the first item under the Qt section. Ignore the Preview section, as it contains pre-release packages that may be unstable. If you want to be fully consistent with the book, you can choose latest Qt 6, but it's not required. The installer also allows you to install multiple Qt versions at once.

Expand the section corresponding to the Qt version you want to install, and choose whichever platforms you need. Select at least one desktop platform to be able to build and run desktop applications. When in Windows, you have to make additional choices for the desktop builds. Select the 32-bit or 64-bit version and choose the compiler you want to be working with. If you have a Microsoft C++ compiler (provided with Visual Studio or Visual C++ Build Tools), you can select the build corresponding to the installed MSVC version. If you don't have a Microsoft compiler or you simply don't want to use it, choose the MinGW build and select the corresponding MinGW version in the Tools section of the package tree.

If you want to build Android applications, choose the option corresponding to the desired Android platform. In Windows, you can select a UWP build to create Universal Windows Platform applications.

The installer will always install Qt Creator—the IDE (integrated development environment) optimized for creating Qt applications. You may also select Qt add-ons that you want to use.

After choosing the required components and clicking on Next again, you will have to accept the licensing terms for Qt by marking an appropriate choice then click next.

After you click on Install, the installer will begin downloading and installing the required packages. Once this is done, your Qt installation will be ready. At the end of the process, you will be given an option to launch Qt Creator.

What just happened?

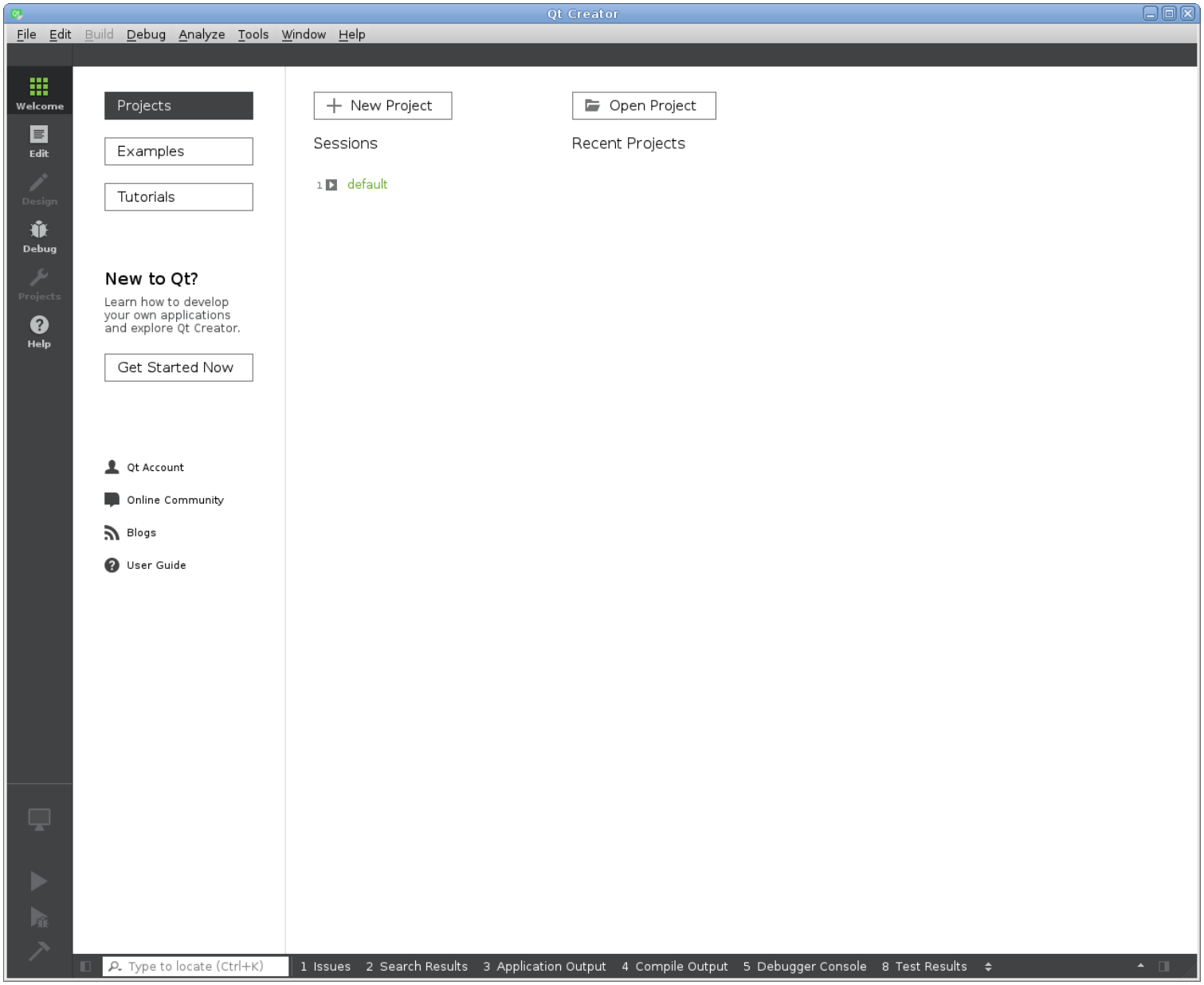
The process we went through results in the whole Qt infrastructure appearing on your disk. You can examine the directory you pointed to the installer to see that it created a number of subdirectories in this directory, one for each version of Qt chosen with the installer, and another one called `Tools` that contains Qt Creator. The Qt directory also contains a `MaintenanceTool` executable, which allows you to add, remove, and update the installed components. The directory structure ensures that if you ever decide to install another version of Qt, it will not conflict with your existing installation. Furthermore, for each version, you can have a number of platform subdirectories that contain the actual Qt installations for particular platforms.

Qt Creator

Now that Qt is installed, we will get familiar with Qt Creator and use it to verify the installation.

Qt Creator's modes

After Qt Creator starts, you should be presented with the following screen:



The panel on the left allows you to switch between different **modes** of the IDE:

- Welcome mode: Allows you to quickly open last sessions, projects, load examples, and tutorials.
- Edit mode: The main mode used to edit the source code of your applications.
- Design mode: Contains a visual form editor. Design mode is automatically activated when you create or open a Qt Widgets form file (`.ui`) or a QML form file (`.ui.qml`).
- Debug mode: Automatically activated when you launch the application under debugger. It contains additional views for displaying the call stack, the break point list, and values of local variables. More views (such as thread lists or values of registers) can be enabled when needed.
- Projects mode: Allows you to configure how Qt Creator will build and run your application. For example, you can choose which Qt version it will use or add command-line arguments here.
- Help mode: Provides access to the Qt documentation. We will focus on this topic later in the chapter.

Setting up compilers, Qt versions, and kits

Before Qt Creator can build and run projects, it needs to know which Qt builds, compilers, debuggers, and other tools are available. Fortunately, Qt installer will usually do it automatically, and Qt Creator is able to automatically detect tools that are available system-wide. However, let's verify that our environment is properly configured. From the Tools menu, choose Options. Once a dialog box pops up, choose Build & Run from the side list. This is the place where we can configure the way Qt Creator can build our projects. A complete build configuration is called a **kit**. It consists of a Qt installation and a compiler that will be executed to perform the build. You can see tabs for all the three entities in the Build & Run section of the Options dialog box.

Let's start with the Compilers tab. If your compiler was not autodetected properly and is not in the list, click on the Add button, choose your compiler type from the list, and fill the name and path to the compiler. If the settings were entered correctly, Creator will autofill all the other details. Then, you can click on Apply to save the changes.

Next, you can switch to the Qt Versions tab. Again, if your Qt installation was not detected automatically, you can click on Add. This will open a file dialog box where you will need to find your Qt installation's directory, where all the binary executables are stored (usually in the `bin` directory), and select a binary called `qmake`. Qt Creator will warn you if you choose a wrong file. Otherwise, your Qt installation and version should be detected properly. If you want, you can adjust the version name in the appropriate box.

The last tab to look at is the Kits tab. It allows you to pair a compiler with the Qt version to be used for compilation. In addition to this, for embedded and mobile platforms, you can specify a device to deploy to and a `sysroot` directory containing all the files needed to build the software for the specified embedded platform. Check that the name of each kit is descriptive enough so that you will be able to select the correct kit (or kits) for each of your applications. If needed, adjust the names of the kits.

Time for action – Loading an example project

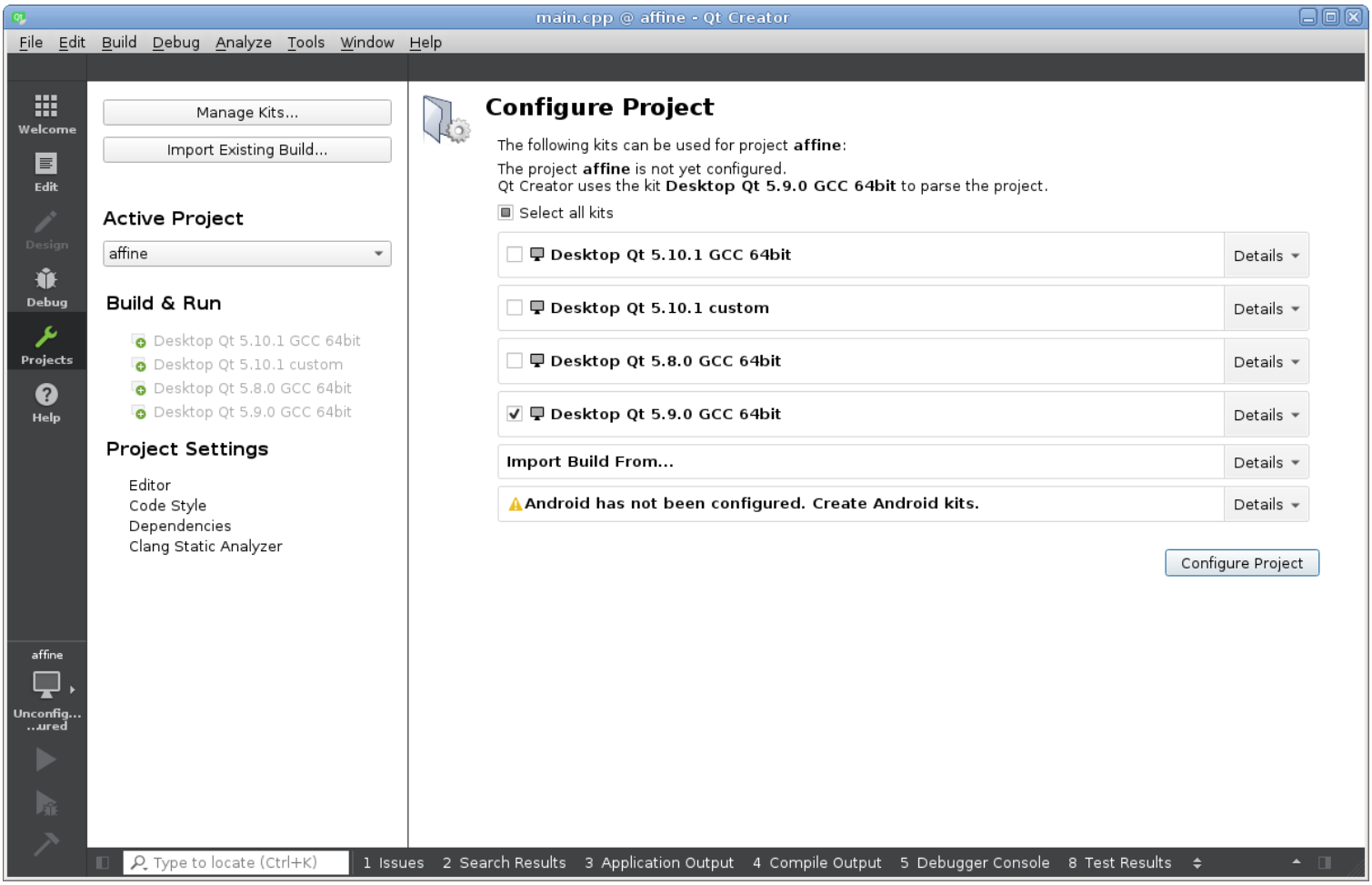
Examples are a great way to explore the capabilities of Qt and find the code required for some typical tasks. Each Qt version contains a large set of examples that are always up to date. Qt Creator provides an easy way to load and compile any example project.

Let's try loading one to get familiar with Qt Creator's project editing interface. Then, we will build the project to check whether the installation and configuration were done correctly.

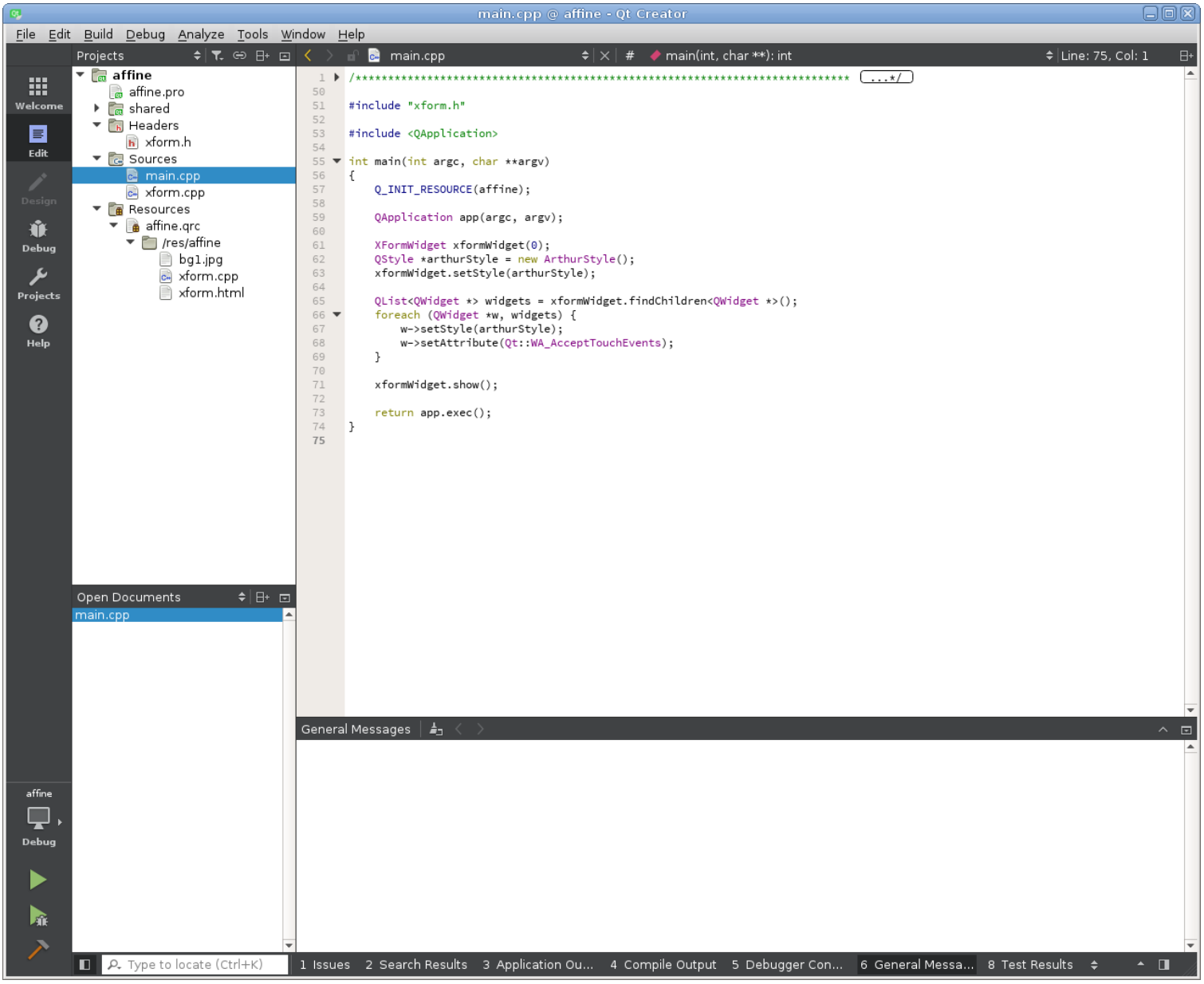
In Qt Creator, click on the Welcome button in the top-left corner of the window to switch to the Welcome mode. Click on the Examples button (refer to the previous screenshot) to open the list of examples with a search box. Ensure that the kit that you want to use is chosen in the drop-down list next to the search box. In the box, enter `aff` to filter the list of examples and click on Affine Transformations to open the project. If you are asked whether you want to copy the project to a new folder, agree.

After selecting an example, an additional window appears that contains the documentation page of the loaded example. You can close that window when you don't need it. Switch back to the main Qt Creator window.

Qt Creator will display the Configure Project dialog with the list of available kits:



Verify that the kits you want to use are marked with check boxes, and click on the Configure Project button. Qt Creator will then present you with the following window:



This is the Edit mode of Qt Creator. Let's go through the most important parts of this interface:

- **Project tree** is located at the top-left of the window. It displays all open projects and the hierarchy of files within them. You can double-click on a file to open it for editing. The context menu of projects, directories, and files in the project tree contains a lot of useful functions.
- At the bottom-left of the window, there's a list of **open documents**. The file selected in this list will appear in the code editor in the center of the window. If the selected file is a Qt Designer form, Qt Creator will automatically switch to the Design mode. Each file in the list has a close button.

- The Type to locate field is present at the left of the bottom panel. If you want to quickly navigate to another file in the project, type the beginning of its name in the field and select it in the pop-up list. Special prefixes can be used to enable other search modes. For example, the `c` prefix allows you to search for C++ classes. You can press *Ctrl* + *K* to activate this field.
- The buttons at the bottom of the left panel allow you to build and run your current project under debugger, or normally. The button above them displays names of the current project and the current build configuration (for example, Debug or Release) and allows you to change them.
- The output panes appear below the code editor when you select them in the bottom panel. The Issues pane contains compiler errors and other related messages. The Search Results pane allows you to run a text search in the entire project and view its results. The Application Output pane displays the text your application has printed to its standard output (`stderr` or `stdout`).

Qt Creator is highly configurable, so you can adjust the layout to your liking. For example, it's possible to change the locations of panes, add more panes, and change keyboard shortcuts for every action.

Qt documentation

Qt project has very thorough documentation. For each API item (class, method, and so on), there is a section in the documentation that describes that item and mentions things that you need to know. There are also a lot of overview pages describing modules and their parts. When you are wondering what some Qt class or module is made for or how to use it, the Qt documentation is always a good source of information.

Qt Creator has an integrated documentation viewer. The most commonly used documentation feature is context help. To try it out, open the `main.cpp` file, set the text cursor inside the `QApplication` text, and press *F1*. The help section should appear to the right of the code editor. It displays the documentation page for the `QApplication` class. The same should work for any other Qt class, method, macro, and so on. You can click on the Open in Help Mode button on top of the help page to switch to the Help mode, where you have more space to view the page.

Another important feature is the search in documentation index. To do that, go to the Help mode by clicking on the Help button on the left panel. In Help mode, in the top-left corner of the window, there is a drop-down list that allows you to select the mode of the left section: Bookmarks, Contents, Index, or Search. Select Index mode, input your request in the Look for: text field and see whether there are any search results in the list below the text field. For example, try typing `qt core` to search for the Qt Core module overview. If there are results, you can press *Enter* to quickly open the first result or double-click on any result in the list to open it. If multiple Qt versions are installed, a dialog may appear where you need to select the Qt version you are interested in.

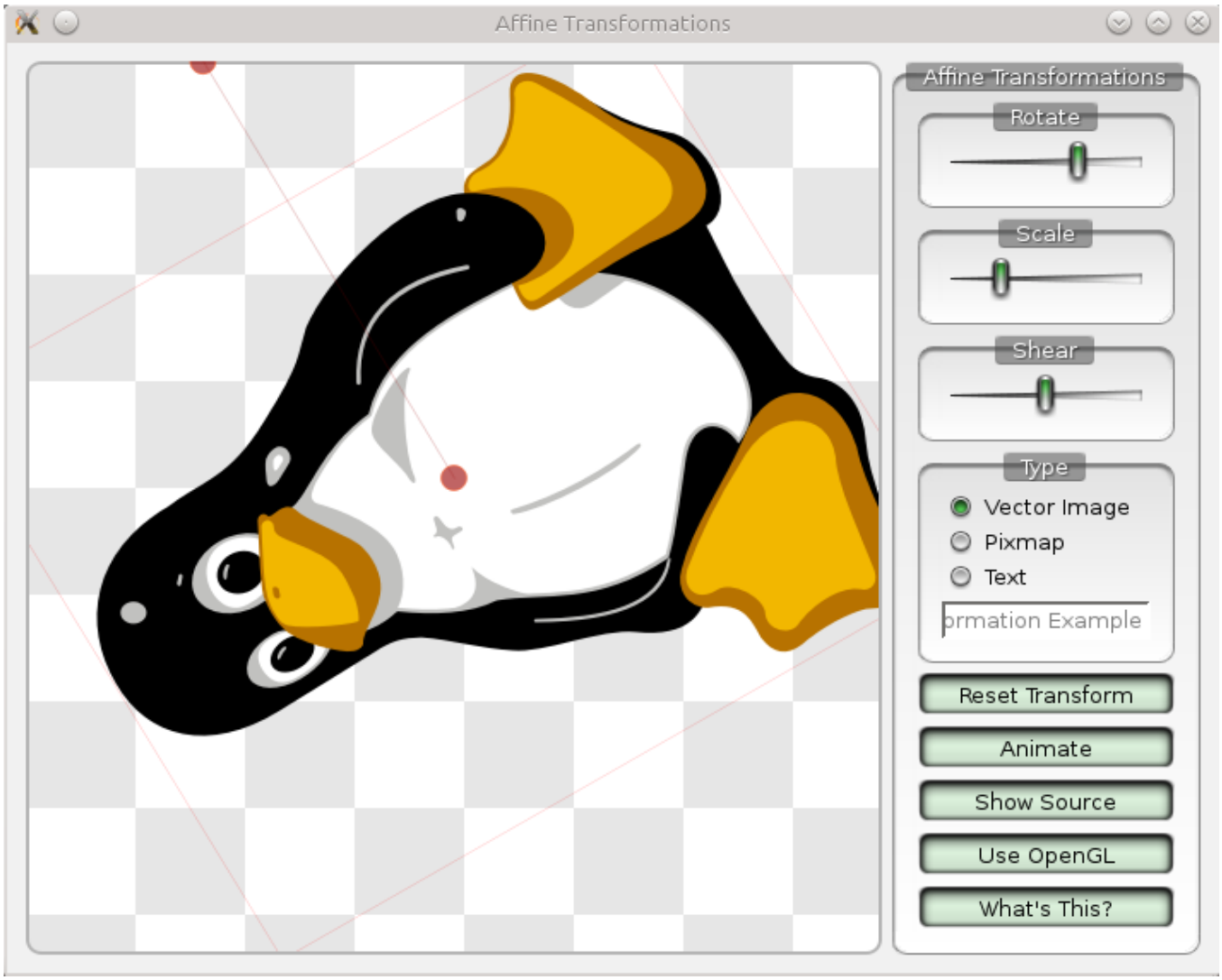
Later in this book, we will sometimes refer to Qt documentation pages by their names. You can use the method described previously to open these pages in Qt Creator.

Time for action – Running the Affine Transformations project

Let's try building and running the project to check whether the building environment is configured properly. To build the project, click on the hammer icon (Build) at the bottom of the left panel. At the right of the bottom panel, a grey progress bar will appear to indicate the build progress. When the build finishes, the progress bar turns green if the build was successful or red otherwise. After the application was built, click on the green triangle icon to run the project.

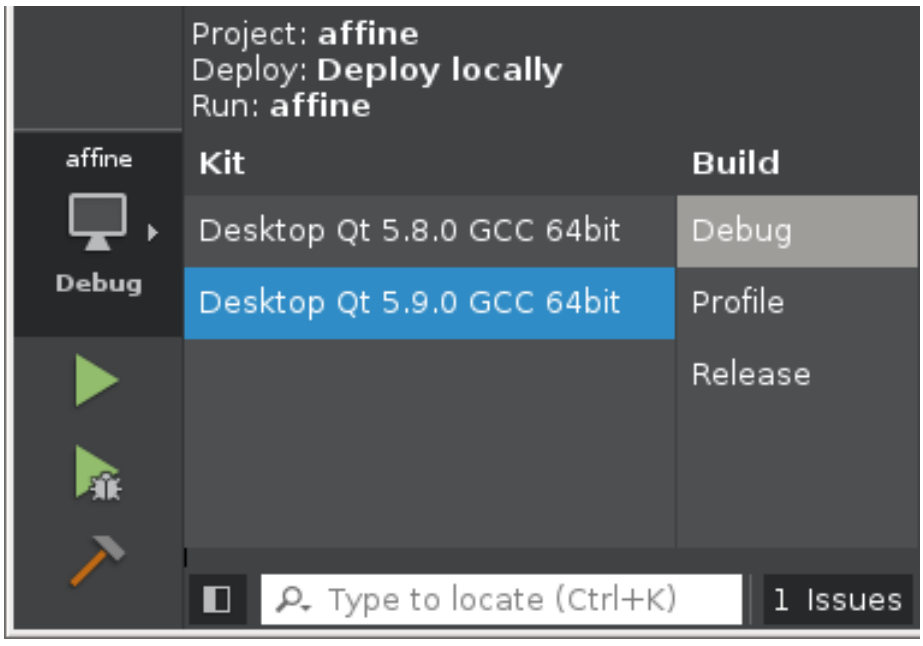
> *Qt Creator can automatically save all files and build the project before running it, so you can just hit the Run (Ctrl + R) or Start Debugging (F5) button after making changes to the project. To verify that this feature is enabled, click on Tools and Options in the main menu, go to the Build & Run section, go to the General tab, and check that the Save all files before build, Always build project before deploying it, and Always deploy project before running it options are checked.*

If everything works, after some time, the application should be launched, as shown in the next screenshot:



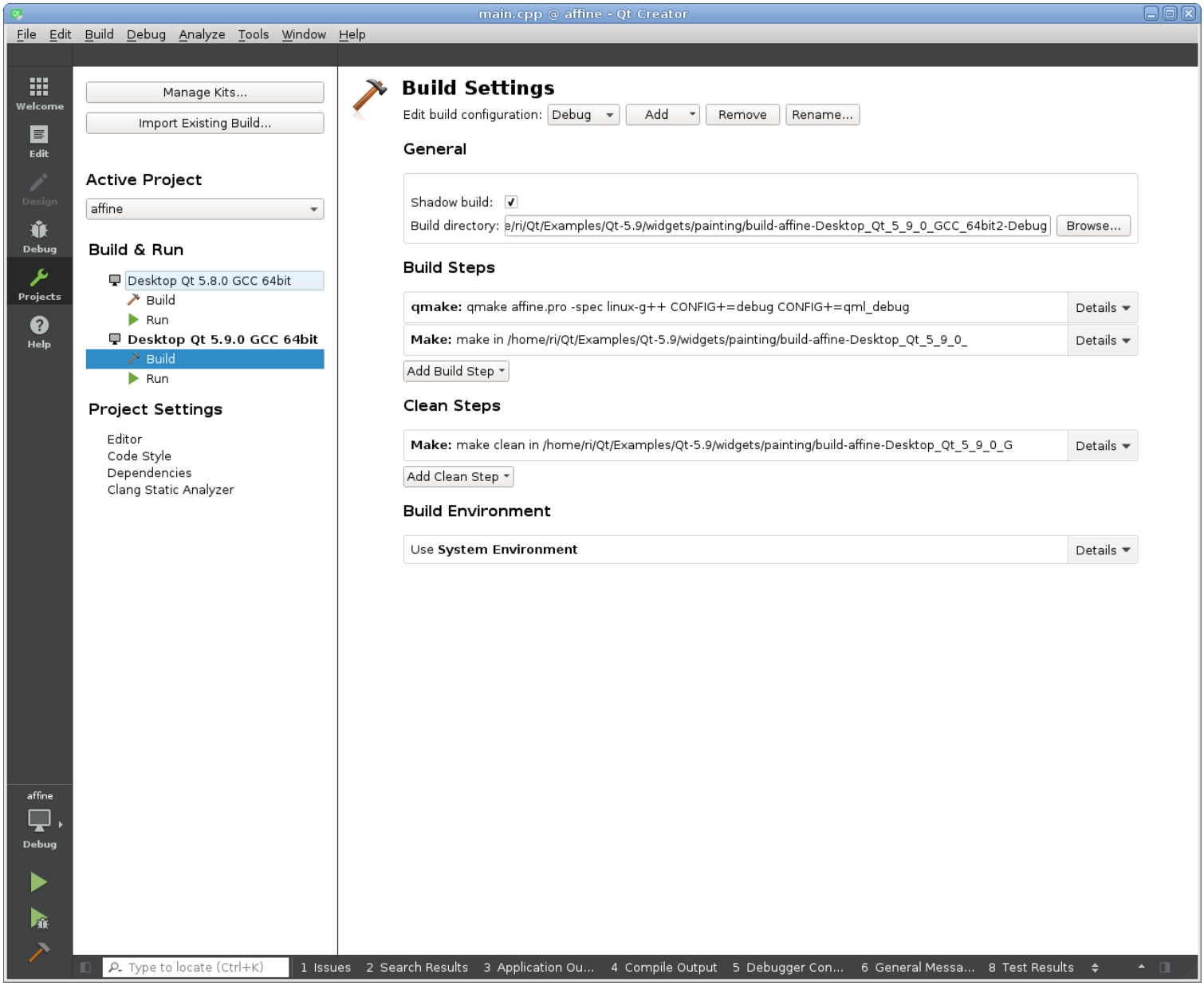
What just happened?

How exactly was the project built? To see which kit and which build configuration was used, click on the icon in the action bar directly over the green triangle icon to open the build configuration popup, as shown in the following screenshot:



The exact content that you get varies depending on your installation, but in general, on the left, you will see the list of kits configured for the project and on the right, you will see the list of build configurations defined for that kit. You can click on these lists to quickly switch to a different kit or a different build configuration. If your project is configured only for one kit, the list of kits will not appear here.

What if you want to use another kit or change how exactly the project is built? As mentioned earlier, this is done in the Projects mode. If you go to this mode by pressing the Projects button on the left panel, Qt Creator will display the current build configuration, as shown in the following screenshot:



The left part of this window contains a list of all kits. Kits that are not configured to be used with this project are displayed in gray color. You can click on them to enable the kit for the current project. To disable a kit, choose the Disable Kit option in its context menu.

Under each enabled kit, there are two sections of the configuration. The Build section contains settings related to building the project:

- Shadow build is a build mode that places all temporary build files in a separate build directory. This allows you to keep the source directory clean and makes your source files easier to track (especially if you use a version control system). This mode is enabled by default.
- Build directory is the location of temporary build files (only if shadow build is enabled). Each build configuration of the project needs a separate build directory.
- The Build steps section displays commands that will be run to perform the actual building of the project. You can edit command-line arguments of the existing steps and add custom build steps. By default, the build process consists of two steps: `qmake` (Qt's project management tool described in the previous chapter) reads the project's `.pro` file and produces a makefile, and then some variation of `make` tool (depending on the platform) reads the makefile and executes Qt's special compilers, the C++ compiler, and the linker. For more information about `qmake`, look up the `qmake Manual` in the documentation index.
- The Build environment section allows you to view and change environment variables that will be available to the build tools.

> *Most variations of the `make` tool (including `mingw32-make`) accept the `-j num_cores` command-line argument that allows `make` to spawn multiple compiler processes at the same time. It is highly recommended that you set this argument, as it can drastically reduce compilation time for big projects. To do this, click on Details at the right part of the Make build step and input `-j num_cores` to the Make arguments field (replace `num_cores` with the actual number of processor cores on your system). However, MSVC `nmake` does not support this feature. To fix this issue, Qt provides a replacement tool called `jom` that supports it.*

There can be multiple build configurations for each kit. By default, three configurations are generated: Debug (required for the debugger to work properly), Profile (used for profiling), and Release (the build with more optimizations and no debug information).

The Run section determines how the executable produced by your project will be started. Here, you can change your program's command-line arguments, working directory, and environment variables. You can add multiple run configurations and switch between them using the same button that allows you to choose the current kit and build configuration.

In most cases for desktop and mobile platforms, the binary release of Qt you download from the web page is sufficient for all your needs. However, for embedded systems, especially for ARM-based systems, there is no binary release available, or it is too heavy resource wise for such a lightweight system. Fortunately, Qt is an open source project, so you can always build it from sources. Qt allows you to choose the modules you want to use and has many more configuration options. For more information, look up Building Qt Sources in the documentation index.

Summary

By now, you should be able to install Qt on your development machine. You can now use Qt Creator to browse the existing examples and learn from them or to read the Qt reference manual to gain additional knowledge. You should have a basic understanding of Qt Creator's main controls. In the next chapter, we will finally start using the framework, and you will learn how to create graphical user interfaces by implementing our very first simple game.