Beginning C++ Programming

First published: April 2017

Production reference: 1180417

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.
ISBN 978-1-78712-494-3

www.packtpub.com

Credits

Author	**Copy Editor**
Richard Grimes	Safis Editing
Reviewer	**Project Coordinator**
Angel Hernandez	Vaidehi Sawant
Commissioning Editor	**Proofreader**
Aaron Lazar	Safis Editing
Acquisition Editor	**Indexer**
Nitin Dasan	Tejal Daruwale Soni
Content Development Editor	**Graphics**
Zeeyan Pinheiro	Abhinash Sahu
Technical Editor	**Production Coordinator**
Pavan Ramchandani	Shraddha Falebhai

About the Author

Richard Grimes has been programming in C++ for 25 years, working on projects as diverse as scientific control and analysis and finance analysis to remote objects for the automotive manufacturing industry. He has spoken at 70 international conferences on Microsoft technologies (including C++ and C#) and has written 8 books, 150 articles for programming journals, and 5 training courses for Microsoft. Richard was awarded Microsoft MVP for 10 years (1998-2007). He has a reputation for his deep understanding of the .NET framework and C++ and the frank way in which he assesses new technology.

For my wife Ellinor: it is only with your love and support that I am able to do anything at all

About the Reviewer

Angel Hernandez is a highly regarded senior solutions, architect and developer with over 15 years of experience, mainly in the consulting space. He is an 11-time Microsoft (2006-2016) MVP award recipient in Visual Studio and Development Technologies category (formerly, Visual C++), and he is currently a member of the Microsoft MVP Reconnect Program. Angel is also a TOGAF practitioner. He has deep knowledge of Microsoft and open source technologies (*nix Systems), and he's an expert in managed and native languages, C# and C++ being his favorites. He can be reached at `http://www.angelhernandezm.com`.

I'd like to thank, first and foremost, God and his son Jesus; Packt and the author for giving me the opportunity to review this book; and my family, Mery, Miranda, and Mikaela (the 3Ms) for being understanding and patient with me.

Customer Feedback

Table of Contents

Preface

C++ has been used for 30 years, and during that time, many new languages have come and gone, but C++ has endured. The big question behind this book is: Why? Why use C++? The answer lies in the ten chapters you see in front of you but, as a spoiler, it is the flexibility and power of the language and the rich, broad Standard Library.

C++ has always been a powerful language, giving you direct access to memory while providing high-level features such as the ability to create new types—classes—and to override operators to suit your needs. However, the more modern C++ standards added to this, generic programming through templates, and functional programming through function objects and lambda expressions. You can use as much or as little of these features as you want; you can write event-driven code with abstract interface pointers, or C-like procedural code.

In this book, we will take you through the features of the 2011 standard of C++ and the Standard Library provided with the language. The text explains how to use these features with short code snippets, and each chapter has a worked example illustrating the concepts. At the end of this book, you will be aware of all the features of the language and what can be possible with the C++ Standard Library. You will start this book as a beginner, and finish it informed and equipped to use C++.

What this book covers

`Chapter 1`, *Starting with C++*, explains the files used to write C++ applications, file dependencies, and the basics of C++ project management.

`Chapter 2`, *Understanding Language Features*, covers C++ statements and expressions, constants, variables, operators, and how to control execution flow in applications.

`Chapter 3`, *Exploring C++ Types*, describes C++ built-in types, aggregated types, type aliases, initializer lists, and conversion between types.

`Chapter 4`, *Working with Memory, Arrays, and Pointers*, covers how memory is allocated and used in C++ applications, how to use built-in arrays, the role of C++ references, and how to use C++ pointers to access memory.

`Chapter 5`, *Using Functions*, explains how to define functions, how to pass parameters-by-reference and by-value using a variable number of parameters, creating and using pointers to functions, and defining template functions and overloaded operators.

`Chapter 6`, *Classes*, describes how to define new types through classes and the various special functions used in a class, how to instantiate a class as an object and how to destroy them, and how to access objects through pointers and how to write template classes.

`Chapter 7`, *Introduction to Object-Orientated Programming*, explains inheritance and composition, and how this affects using pointers and references to objects and the access levels of class members and how they affect inherited members. This chapter also explains polymorphism through virtual methods, and inheritance programming through abstract classes.

`Chapter 8`, *Using Standard Library Containers*, covers all the C++ Standard Library container classes and how to use them with iterators and the standard algorithms so that you can manipulate the data in containers.

`Chapter 9`, *Using Strings*, describes the features of the standard C++ string class, converting between numeric data and strings, internationalizing strings, and using regular expressions to search and manipulate strings.

`Chapter 10`, *Diagnostics and Debugging*, explains how to prepare your code to provide diagnostics and to enable it to be debugged, how applications are terminated, abruptly or gracefully, and how to use C++ exceptions.

What you need for this book

This book covers the C++11 standard, and the associated C++ Standard Library. For the vast majority of this book, any C++11 compliant compiler is suitable. This includes compilers from Intel, IBM, Sun, Apple, and Microsoft, as well as the open source GCC compiler.

This book uses Visual C++ 2017 Community Edition because it is a fully featured compiler and environment, and it is provided as a free download. This is a personal choice of the author, but it should not restrict readers who prefer using other compilers. Some of the sections of the last chapter on *Diagnostics and Debugging* describe Microsoft-specific features, but these sections are clearly marked.

Who this book is for

This book is intended for experienced programmers who are new to C++. The reader is expected to understand what high-level languages are for and basic concepts such as modularizing code and controlling execution flow.

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "We can include other contexts through the use of the `include` directive."

A block of code is set as follows:

```
class point
{
public:
    int x, y;
};
```

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

```
class point
{
public:
    int x, y;
    point(int _x, int _y) : x(_x), y(_y) {}
};
```

Any command-line input or output is written as follows:

```
C:\> cl /EHsc test.cpp
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "Clicking the **Next** button moves you to the next screen."

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book-what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.
To send us general feedback, simply e-mail `feedback@packtpub.com`, and mention the book's title in the subject of your message.
If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for this book from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support`and register to have the files e-mailed directly to you.

You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the **SUPPORT** tab at the top.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box.
5. Select the book for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this book from.
7. Click on **Code Download**.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

The code bundle for the book is also hosted on GitHub at `https://github.com/PacktPublishing/Beginning-Cpp-Programming`. We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Downloading the color images of this book

We also provide you with a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output. You can download this file from `https://www.packtpub.com/sites/default/files/downloads/BeginningCppProgramming_ColorImages.pdf`.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books-maybe a mistake in the text or the code-we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to `https://www.packtpub.com/books/content/support`and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at `questions@packtpub.com`, and we will do our best to address the problem.

1 Starting with C++

Why C++? There will be as many reasons to use C++ as there will be readers of this book.

You may have chosen C++ because you have to support a C++ project. Over the 30 years of its lifetime there have been millions of lines of C++ written, and most popular applications and operating systems will be mostly written in C++, or will use components and libraries that are. It is nearly impossible to find a computer that does not contain some code that has been written in C++.

Or, you may have chosen C++ to write new code. This may be because your code will use a library written in C++, and there are thousands of libraries available: open source, shareware, and commercial.

Or it may be because you are attracted to the power and flexibility that C++ offers. Modern high-level languages have been designed to make it easy for programmers to perform actions; while C++ has such facilities, it also allows you to get as close to the machine as possible, gives you the (sometimes dangerous) power of direct memory access. Through language features such as classes and overloading, C++ is a flexible language that allows you to extend how the language works and write reusable code.

Whatever your reason for deciding on C++, you have made the right choice, and this book is the right place to start.

What will you find in this chapter?

Since this book is a hands-on book, it contains code that you can type, compile, and run. To compile the code, you will need a C++ compiler and linker, and in this book that means Visual Studio 2017 Community Edition, which provides Visual C++. This compiler was chosen because it is a free download, it is compliant with C++ standards and it has very wide range of tools to make writing code easier. Visual C++ provides C++11-compliant language features and almost all the language features of C++14 and C++17. Visual C++ is also provided with the C99 runtime library, C++11 standard library, and C++14 standard library. All of this mentions of **standard** means that the code that you learn to write in this book will compile with all other standard C++ compilers.

This chapter will start with details about how to obtain and install Visual Studio 2017 Community Edition. If you already have a C++ compiler, you can skip this section. Most of this book is vendor-neutral about the compiler and linker tools, but Chapter 10, *Diagnostics and Debugging*, which covers debugging and diagnostics, will cover some Microsoft-specific features. Visual Studio has a fully featured code editor, so even if you do not use it to manage your projects, you'll find it useful to edit your code.

After we've described the installation, you'll learn the basics of C++: how source files and projects are structured, and how you can manage projects with potentially thousands of files.

Finally, the chapter will finish with a step-by-step structured example. Here you will learn how to write simple functions that use the standard C++ library and one mechanism to manage files in the project.

What is C++?

The predecessor of C++ is C, which was designed by Dennis Richie at Bell Labs and first released in 1973. C is a widely used language and was used to write the early versions of Unix and Windows. Indeed, the libraries and software-development libraries of many operating systems are still written to have C interfaces. C is powerful because it can be used to write code that is compiled to a compact form, it uses a static type system (so the compiler does the work of type checking), and the types and structures of the language allow for direct memory access to computer architecture.

C, however, is procedural and based on functions, and although it has record types (`struct`) to encapsulate data, it does not have object-like behaviors to act on that encapsulated state. Clearly there was a need for the power of C but the flexibility and extensibility of object-oriented classes: a language that was C, with classes. In 1983, Bjarne Stroustrup released C++. The ++ comes from the C increment operator `++`.

Strictly, when postfixed to a variable, the `++` operator means *increment the variable, but return the variable's value before it was incremented*. So the C statements `int c = 1; int d = c++;` will result in variable `d` having a value of 1 and variable `c` having a value of 2. This does not quite express the idea that C++ is an increment on C.

Installing Visual C++

Microsoft's Visual Studio Community 2017 contains the Visual C++ compiler, the C++ standard libraries, and a collection of standard tools that you can use to write and maintain C++ projects. This book is not about how to write Windows code; it is about how to write standard C++ and how to use the C++ Standard Library. All the examples in this book will run on the command line. Visual Studio was chosen because it is a free download (although you do have to register an e-mail address with Microsoft), and it is standards-compliant. If you already have a C++ compiler installed, then you can skip this section.

Setting up

Before starting the installation, you should be aware that, as part of the agreement to install Visual Studio as part of Microsoft's community program, you should have a Microsoft account. You are given the option to create a Microsoft account the first time you run Visual Studio and if you skip this stage, you will be given a 30-day evaluation period. Visual Studio will be fully featured during this month, but if you want to use Visual Studio beyond this time you will have to provide a Microsoft account. The Microsoft account does not impose any obligation on you, and when you use Visual C++ after signing in, your code will still remain on your machine with no obligation to pass it to Microsoft.

Of course, if you read this book within one month, you will be able to use Visual Studio without having to sign in using your Microsoft account; you may view this as an incentive to be diligent about finishing the book!

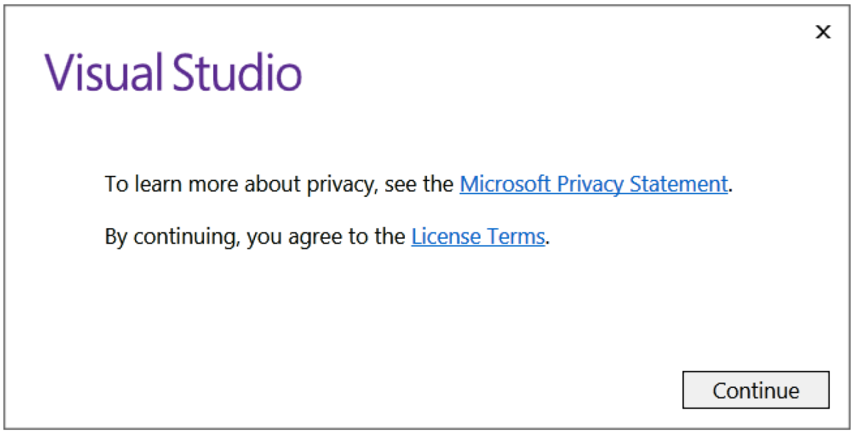
Downloading the installation files

To download the Visual Studio Community 2017 installer go to `https://www.visualstudio.com/vs/ community/`.

When you click on the Download Community 2017 button, your browser will download a 1 MB file called `vs_community__1698485341.1480883588.exe`. When you run this application, it will allow you to specify the languages and libraries that you want installed, and then it downloads and installs all the necessary components.

Installing Visual Studio

Visual Studio 2017 treats Visual C++ as an optional component, so you have to explicitly indicate that you want to install it through custom options. When you first execute the installer, you will see the following dialog box:

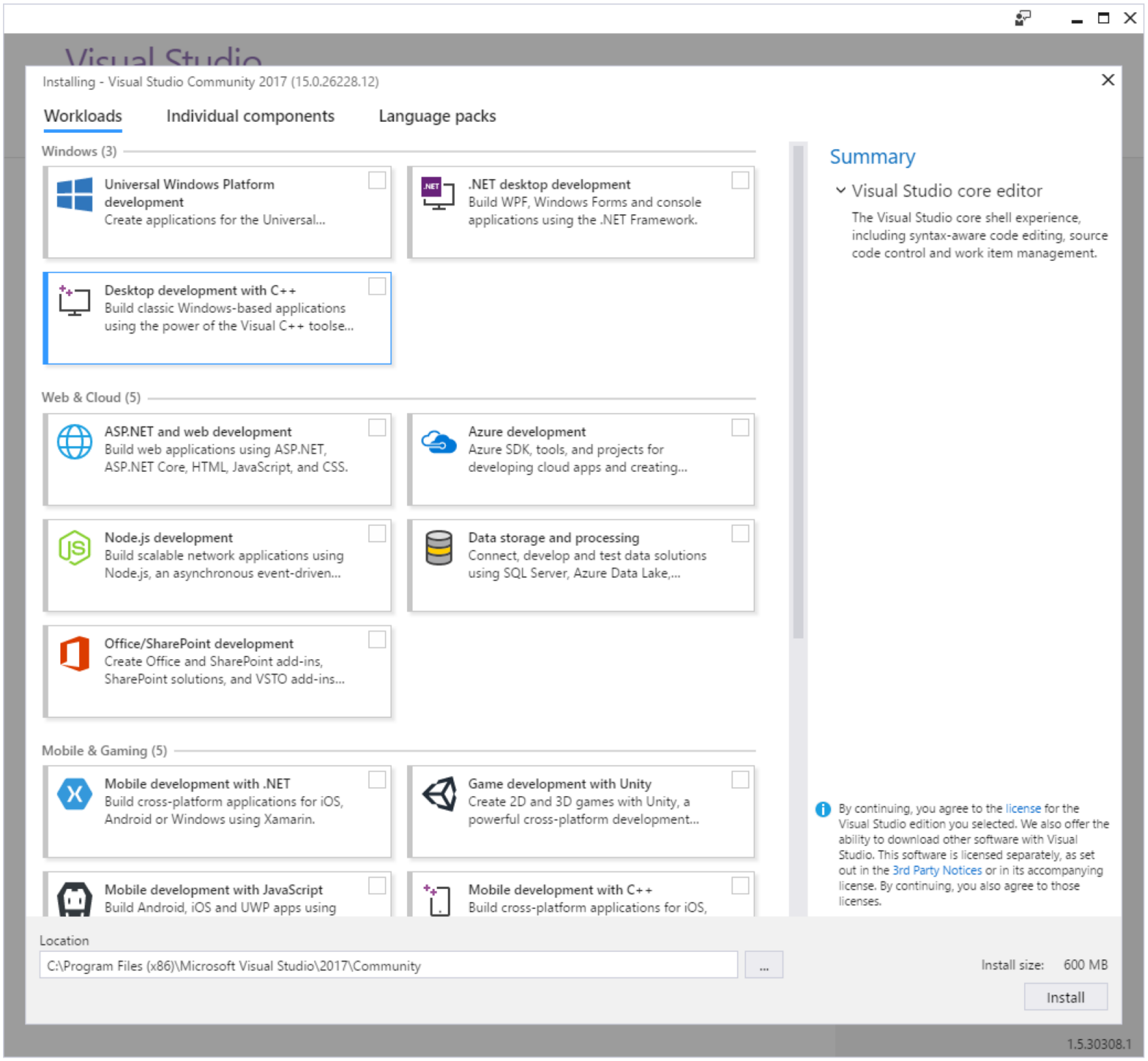


Visual Studio

To learn more about privacy, see the Microsoft Privacy Statement.

By continuing, you agree to the License Terms.

When you click on the **Continue** button the application will set up the installer, as shown here:


Visual Studio

Installing - Visual Studio Community 2017 (15.0.26228.12)

Workloads | Individual components | Language packs

Windows (3)

- Universal Windows Platform development — Create applications for the Universal...
- .NET desktop development — Build WPF, Windows Forms and console applications using the .NET Framework.
- Desktop development with C++ — Build classic Windows-based applications using the power of the Visual C++ toolse...

Web & Cloud (5)

- ASP.NET and web development — Build web applications using ASP.NET, ASP.NET Core, HTML, JavaScript, and CSS.
- Azure development — Azure SDK, tools, and projects for developing cloud apps and creating...
- Node.js development — Build scalable network applications using Node.js, an asynchronous event-driven...
- Data storage and processing — Connect, develop and test data solutions using SQL Server, Azure Data Lake,...
- Office/SharePoint development — Create Office and SharePoint add-ins, SharePoint solutions, and VSTO add-ins...

Mobile & Gaming (5)

- Mobile development with .NET — Build cross-platform applications for iOS, Android or Windows using Xamarin.
- Game development with Unity — Create 2D and 3D games with Unity, a powerful cross-platform development...
- Mobile development with JavaScript — Build Android, iOS and UWP apps using
- Mobile development with C++ — Build cross-platform applications for iOS,

Summary

Visual Studio core editor

The Visual Studio core shell experience, including syntax-aware code editing, source code control and work item management.

By continuing, you agree to the license for the Visual Studio edition you selected. We also offer the ability to download other software with Visual Studio. This software is licensed separately, as set out in the 3rd Party Notices or in its accompanying license. By continuing, you also agree to those licenses.

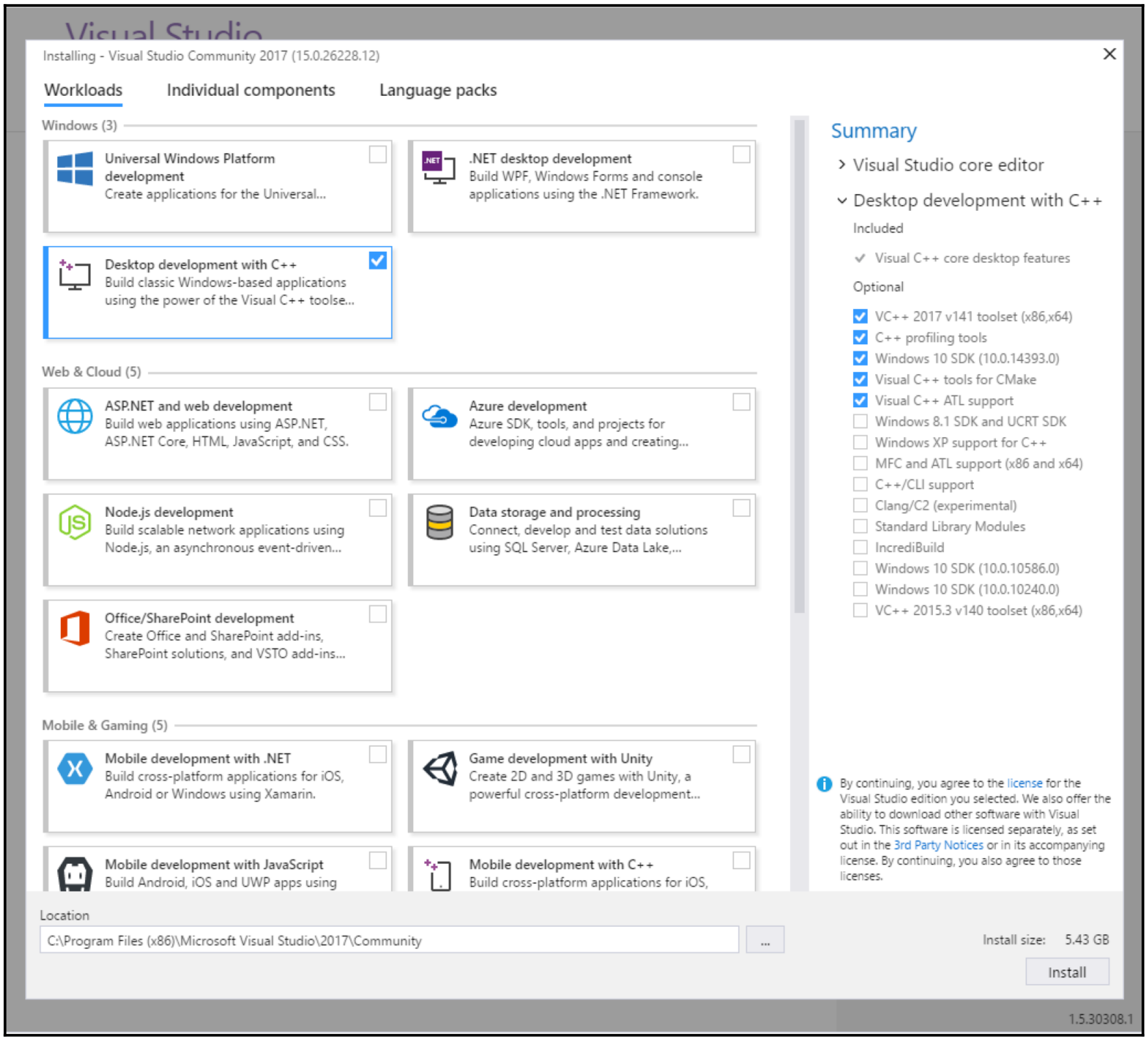
Location

C:\Program Files (x86)\Microsoft Visual Studio\2017\Community

Install size: 600 MB

Along the top are three tabs labeled Workloads, Individual Components and Language Packs. Make sure that you have selected the Workloads tab (as shown in the screenshot) and check the checkbox in the item called **Desktop development with C++**.

The installer will check that you have enough disk space for the selected option. The maximum amount of space Visual Studio will require is 8 GB, although for Visual C++ you will use a lot less. When you check **Desktop development with C++** item, you will see the right side of the dialog change to list the options selected and the disk size required, as follows:



Installing - Visual Studio Community 2017 (15.0.26228.12)

Workloads | Individual components | Language packs

Windows (3)

- [ ] Universal Windows Platform development — Create applications for the Universal...
- [ ] .NET desktop development — Build WPF, Windows Forms and console applications using the .NET Framework.
- [✓] Desktop development with C++ — Build classic Windows-based applications using the power of the Visual C++ toolse...

Web & Cloud (5)

- [ ] ASP.NET and web development — Build web applications using ASP.NET, ASP.NET Core, HTML, JavaScript, and CSS.
- [ ] Azure development — Azure SDK, tools, and projects for developing cloud apps and creating...
- [ ] Node.js development — Build scalable network applications using Node.js, an asynchronous event-driven...
- [ ] Data storage and processing — Connect, develop and test data solutions using SQL Server, Azure Data Lake,...
- [ ] Office/SharePoint development — Create Office and SharePoint add-ins, SharePoint solutions, and VSTO add-ins...

Mobile & Gaming (5)

- [ ] Mobile development with .NET — Build cross-platform applications for iOS, Android or Windows using Xamarin.
- [ ] Game development with Unity — Create 2D and 3D games with Unity, a powerful cross-platform development...
- [ ] Mobile development with JavaScript — Build Android, iOS and UWP apps using
- [ ] Mobile development with C++ — Build cross-platform applications for iOS,

Summary

› Visual Studio core editor

⌄ Desktop development with C++

Included

✓ Visual C++ core desktop features

Optional

- [✓] VC++ 2017 v141 toolset (x86,x64)
- [✓] C++ profiling tools
- [✓] Windows 10 SDK (10.0.14393.0)
- [✓] Visual C++ tools for CMake
- [✓] Visual C++ ATL support
- [ ] Windows 8.1 SDK and UCRT SDK
- [ ] Windows XP support for C++
- [ ] MFC and ATL support (x86 and x64)
- [ ] C++/CLI support
- [ ] Clang/C2 (experimental)
- [ ] Standard Library Modules
- [ ] IncrediBuild
- [ ] Windows 10 SDK (10.0.10586.0)
- [ ] Windows 10 SDK (10.0.10240.0)
- [ ] VC++ 2015.3 v140 toolset (x86,x64)

By continuing, you agree to the license for the Visual Studio edition you selected. We also offer the ability to download other software with Visual Studio. This software is licensed separately, as set out in the 3rd Party Notices or in its accompanying license. By continuing, you also agree to those licenses.

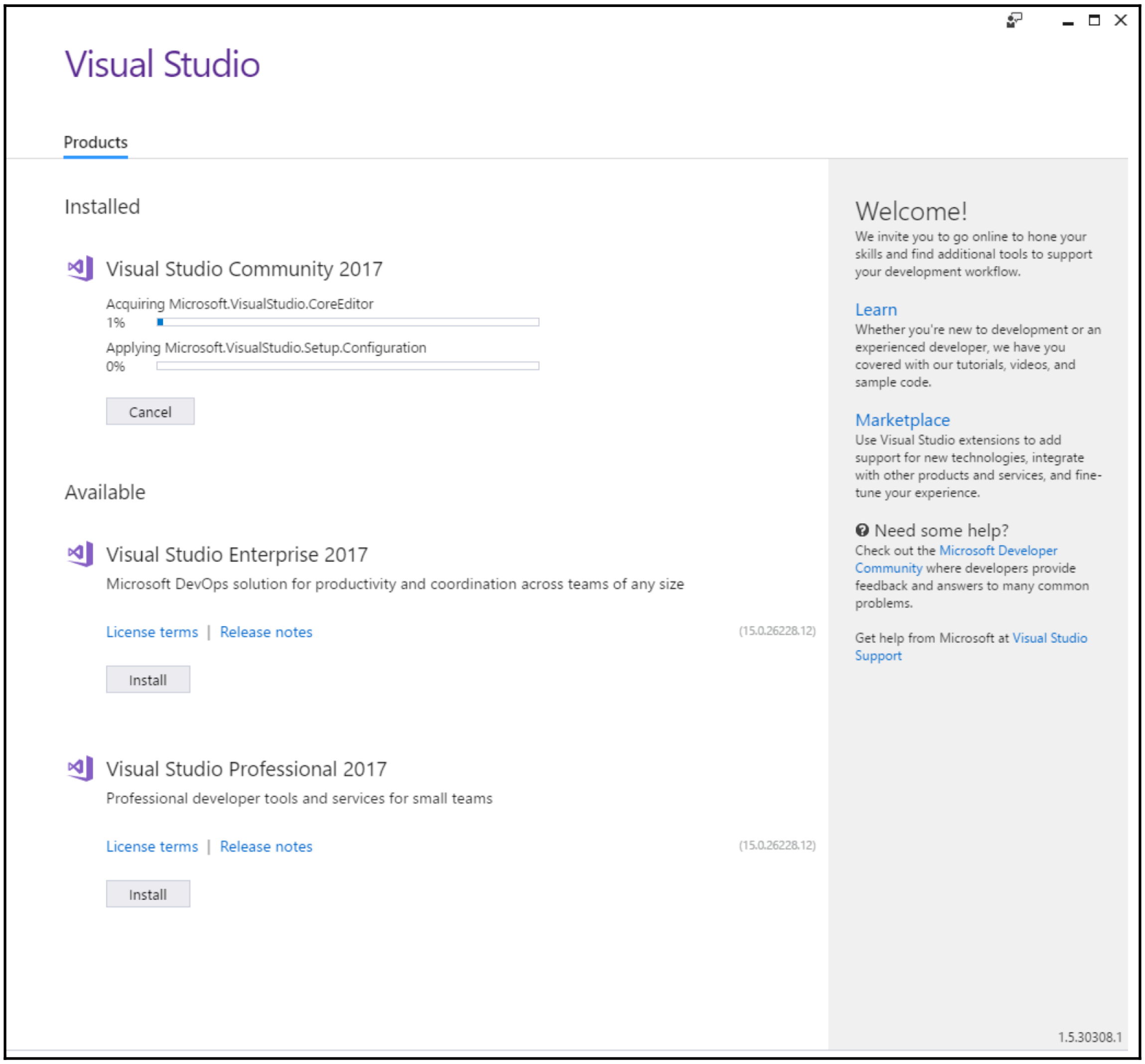
Location: C:\Program Files (x86)\Microsoft Visual Studio\2017\Community

Install size: 5.43 GB

Install

1.5.30308.1

For this book, leave the options selected by the installer and then click the Install button in the bottom right hand corner. The installer will download all the code it needs and it will keep you updated with the progress with the following dialog box:


Visual Studio

Products

Installed

Visual Studio Community 2017

Acquiring Microsoft.VisualStudio.CoreEditor
1%

Applying Microsoft.VisualStudio.Setup.Configuration
0%

Cancel

Available

Visual Studio Enterprise 2017

Microsoft DevOps solution for productivity and coordination across teams of any size

License terms | Release notes (15.0.26228.12)

Install

Visual Studio Professional 2017

Professional developer tools and services for small teams

License terms | Release notes (15.0.26228.12)

Install

Welcome!

We invite you to go online to hone your skills and find additional tools to support your development workflow.

Learn

Whether you're new to development or an experienced developer, we have you covered with our tutorials, videos, and sample code.

Marketplace

Use Visual Studio extensions to add support for new technologies, integrate with other products and services, and fine-tune your experience.

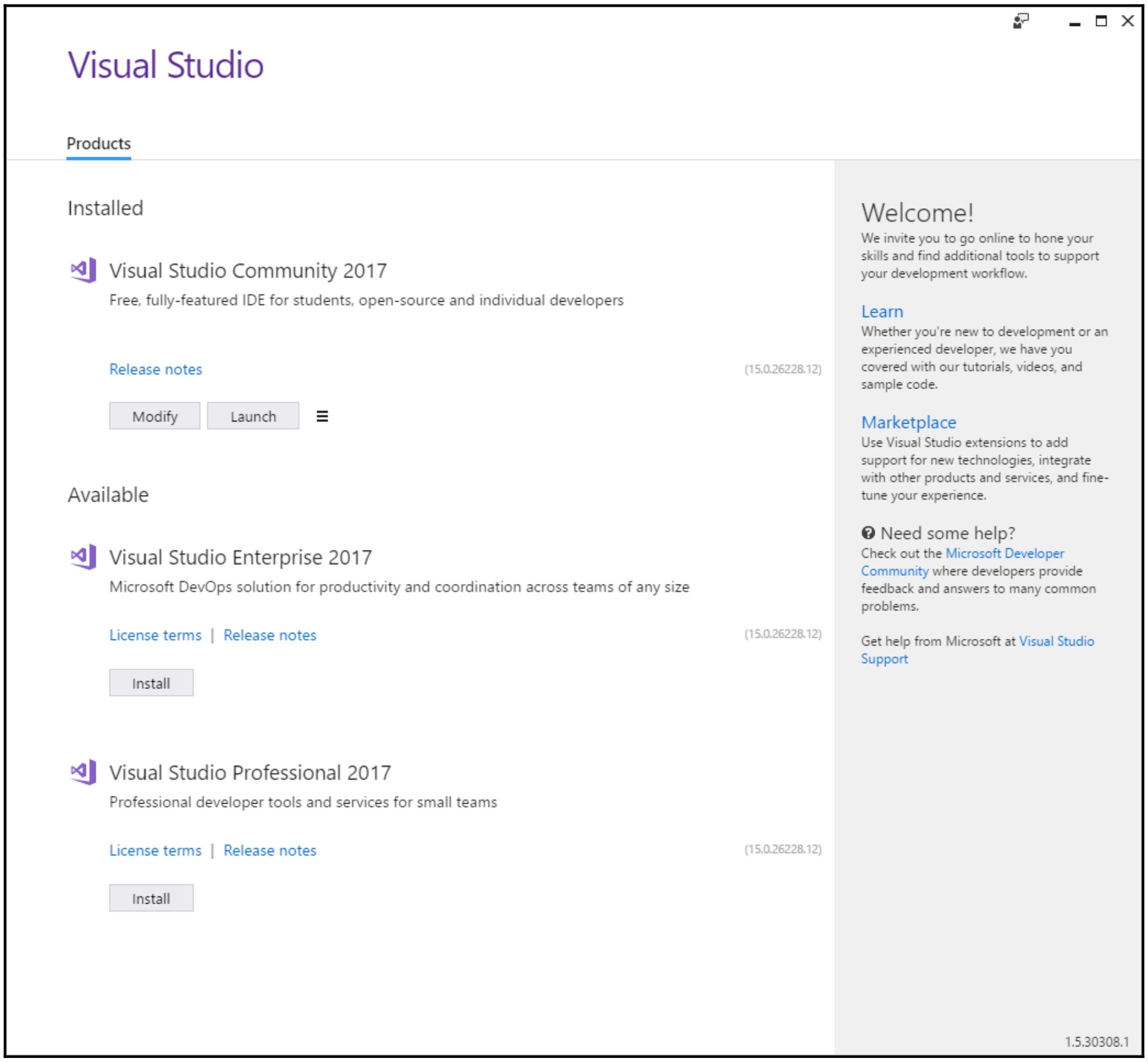
Need some help?

Check out the Microsoft Developer Community where developers provide feedback and answers to many common problems.

Get help from Microsoft at Visual Studio Support

1.5.30308.1

When the installation is complete the Visual Studio Community 2017 item will change to have two buttons, Modify and Launch, as showing here:



Visual Studio

Products

Installed

Visual Studio Community 2017

Free, fully-featured IDE for students, open-source and individual developers

Release notes (15.0.26228.12)

Available

Visual Studio Enterprise 2017

Microsoft DevOps solution for productivity and coordination across teams of any size

License terms | Release notes (15.0.26228.12)

Visual Studio Professional 2017

Professional developer tools and services for small teams

License terms | Release notes (15.0.26228.12)

Welcome!

We invite you to go online to hone your skills and find additional tools to support your development workflow.

Learn

Whether you're new to development or an experienced developer, we have you covered with our tutorials, videos, and sample code.

Marketplace

Use Visual Studio extensions to add support for new technologies, integrate with other products and services, and fine-tune your experience.

Need some help?

Check out the Microsoft Developer Community where developers provide feedback and answers to many common problems.

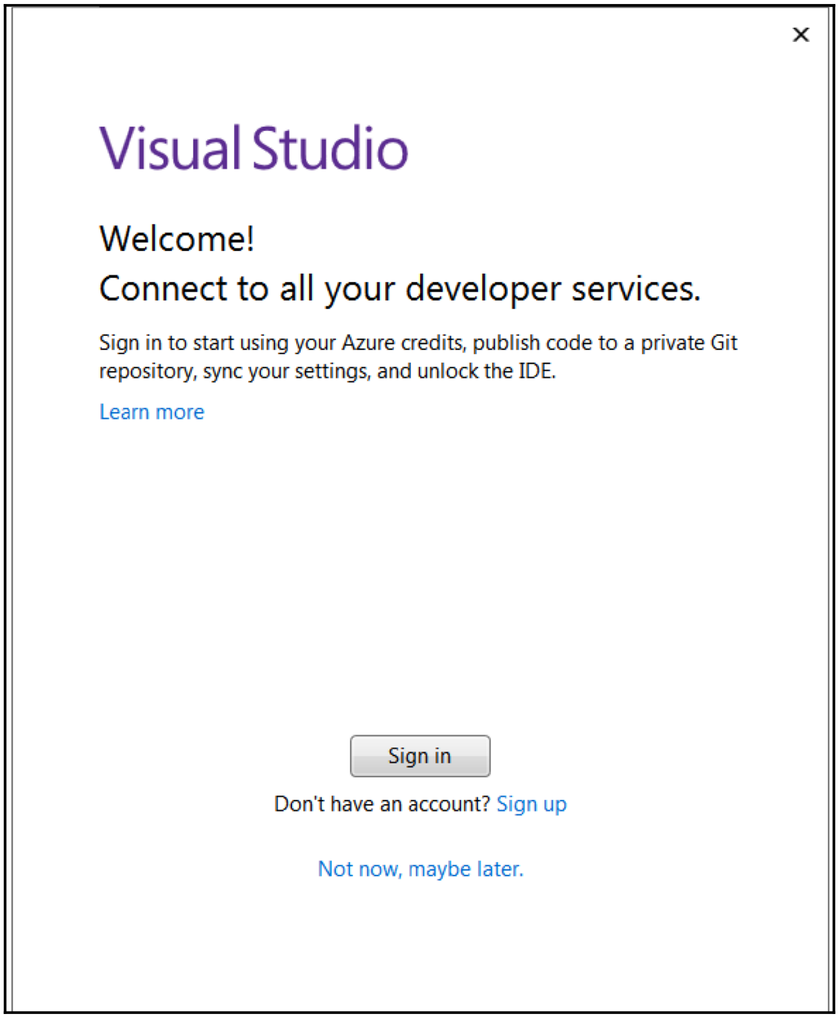
Get help from Microsoft at Visual Studio Support

1.5.30308.1

The **Modify** button allows you to add more components. Click on **Launch** to run Visual Studio for the first time.

Registering with Microsoft

The first time you run Visual Studio it will ask you to sign in to Microsoft services through the following dialog:


Visual Studio

Welcome!

Connect to all your developer services.

Sign in to start using your Azure credits, publish code to a private Git repository, sync your settings, and unlock the IDE.

Learn more

You do not have to register Visual Studio, but if you choose not to, Visual Studio will only work for 30 days. Registering with Microsoft places no obligations on you. If you are happy to register, then you may as well register now. Click on Sign in button provide your Microsoft credentials, or if you do not have an account then click on **Sign up** to create an account.

When you click on the **Launch** button a new window will open, but the installer window will remain open. You may find that the installer window hides the Welcome window, so check the Windows task bar to see if another window is open. Once Visual Studio has started you can close the installer window.

You will now be able to use Visual Studio to edit code, and will have the Visual C++ compiler and libraries installed on your machine, so you will be able to compile C++ code in Visual Studio or on the command line.

Examining C++ projects

C++ projects can contain thousands of files, and managing these files can be a task. When you build the project, should a file be compiled, and if so, by which tool? In what order should the files be compiled? What output will these compilers produce? How should the compiled files be combined to produce the executable?

Compiler tools will also have a large collection of options, as diverse as debug information, types of optimization, support for different language features, and processor features. Different combinations of compiler options will be used in different circumstances (for example, release builds and debug builds). If you compile from a command line, you have to make sure you choose the right options and apply them consistently across all the source code you compile.

Managing files and compiler options can get very complicated. This is why, for production code, you should use a make tool. Two are installed with Visual Studio: **MSBuild** and **nmake**. When you build a Visual C++ project in the Visual Studio environment, MSBuild will be used and the compilation rules will be stored in an XML file. You can also call MSBuild on the command line, passing it the XML project file. The nmake tool is Microsoft's version of the program maintenance utility common across many compilers. In this chapter, you will learn how to write a simple **makefile** to use with the nmake utility.

Before going through the basics of project management, first we have to examine the files that you will commonly find in a C++ project, and what a compiler will do to those files.

Compilers

C++ is a high-level language, designed to give you a wealth of language facilities and to be readable for you and other developers. The computer's processor executes low-level code, and it is the purpose of the compiler to translate C++ to the processor's machine code. A single compiler may be able to target several types of processor, and if the code is standard C++, it can be compiled with other compilers that support other processors.

However, the compiler does much more than this. As explained in `Chapter 4`, *Working With Memory, Arrays, and Pointers*, C++ allows you to split your code into functions, which take parameters and return a value, so the compiler sets up the memory used to pass this data. In addition, functions can declare variables that will only be used within that function (`Chapter 5`, *Using Functions*, will give more details), and will only exist while the function is executed. The compiler sets up this memory, called a **stack frame**. You have compiler options about how stack frames are created; for example, the Microsoft compiler options `/Gd`, `/Gr`, and `/Gz` determine the order in which function arguments are pushed onto the stack and whether the caller function or called function removes the arguments from the stack at the end of the call. These options are important when you write code that will be shared (but for the purpose of this book, the default stack construction should be used). This is just one area, but it should impress upon you that compiler settings give you access to a lot of power and flexibility.

The compiler compiles C++ code, and it will issue a compiler error if it comes across an error in your code. This is syntax checking of your code. It is important to point out that the code you write can be perfect C++ code from a syntax point of view, but it can still be nonsense. The syntax checking of the compiler is an important check of your code, but you should always use other checking. For example, the following code declares an integer variable and assigns it a value:

```
    int i = 1 / 0;
```

The compiler will issue an error `C2124 : divide or mod by zero`. However, the following code will perform the same action using an additional variable, which is logically the same, but the compiler will issue no error:

```
    int j = 0;
    int i = 1 / j;
```

When the compiler issues an error it will stop compiling. This means two things. Firstly, you get no compiled output, so the error will not find its way into an executable. Secondly, it means that, if there are other errors in the source code, you will only find out about it once you have fixed the current error and recompiled. If you want to perform a syntax check and leave compilation to a later time, use the `/Zs` switch.

The compiler will also generate warning messages. A warning means that the code will compile, but there is, potentially, a problem in the code that will affect how the executable will run. The Microsoft compiler defines four levels of warnings: level 1 is the most severe (and should be addressed) and level 4 is informational.

Warnings are often used to indicate that the language feature being compiled is available, but it needs a specific compiler option that the developer has not used. During development of code, you will often ignore warnings, since you may be testing language features. However, when you get closer to producing production code you should pay more attention to warnings. By default, the Microsoft compiler will display level 1 warnings, and you can use the `/W` option with a number to indicate the levels that you wish to see (for example, `/W2` means you wish to see level 2 warnings as well as level 1 warnings). In production code, you may use the `/Wx` option, which tells the compiler to treat warnings as errors so that you must fix the issues to be able to compile the code. You can also use the `pragmas` compiler (`pragmas` will be explained later) and compiler options to suppress specific warnings.

Linking the code

A compiler will produce an output. For C++ code, this will be object code, but you may have other compiler outputs, such as compiled resource files. On their own, these files cannot be executed; not least because the operating system will require certain structures to be set up. A C++ project will always be two-stage: compile the code into one or more object files and then link the object files into an executable. This means that your C++ compiler will provide another tool, called a linker.

The linker also has options to determine how it will work and specify its outputs and inputs, and it will also issue errors and warnings. Like the compiler, the Microsoft linker has an option, `/WX`, to treat warnings as errors in release builds.

Source files

At the very basic level, a C++ project will contain just one file: the C++ source file, typically with the extension `cpp` or `cxx`.

A simple example

The simplest C++ program is shown here:

```
#include <iostream>

// The entry point of the program
int main()
{
    std::cout << "Hello, world!n";
}
```

The first point to make is that the line starting with `//` is a comment. All the text until the end of the line is ignored by the compiler. If you want to have multiline comments, every line must start with `//`. You can also use C comments. A C comment starts with `/*` and ends with `*/` and everything between these two symbols is a comment, including line breaks.

C comments are a quick way to comment out a portion of your code.

The braces, `{}`, indicates a code block; in this case, the C++ code is for the function `main`. We know that this is a function because of the basic format: first, there is the type of the return value, then the name of the function with a pair of parentheses, which is used to declare the parameters passed to the function (and their types). In this example, the function is called `main` and the parentheses are empty, indicating that the function has no parameters. The identifier before the function name (`int`) says that the function will return an integer.

The convention with C++ is that a function called `main` is the **entry point** of the executable, that is, when you call the executable from the command line, this will be the first function in your code that will be called.

This simple example function immediately immerses you into an aspect of C++ that irritates programmers of other languages: the language may have rules, but the rules don't always appear to be followed. In this case, the `main` function is declared to return an integer, but the code returns no value. The rule in C++ is that, if the function declares that it returns a value, then it must return a value. However, there is a single exception to this rule: if the `main` function does not return a value, then a value of `0` will be assumed. C++ contains many quirks such as this, but you will soon learn what they are and get used to them.

The main function has just one line of code; this is a single statement starting with std and ending with the semicolon (;). C++ is flexible about the use of whitespace (spaces, tabs, and newlines) as will be explained in the next chapter. However, it is important to note that you have to be careful with literal strings (as used here), and every statement is delimited with a semicolon. Forgetting a required semicolon is a common source of compiler errors. An extra semicolon is simply an empty statement, so for a novice, having too many semicolons can be less fatal to your code than having too few.

The single statement prints the string Hello, world! (and a newline) to the console. You know that this is a string because it is enclosed in double quote marks (""). The string is *put to* the stream object std::cout using the operator <<. The std part of the name is a **namespace**, in effect, a collection of code with a similar purpose, or from a single vendor. In this case, std means that the cout stream object is part of the standard C++ library. The double colon :: is the **scope resolution** operator, and indicates that you want to access the cout object declared in the std namespace. You can define namespaces of your own, and in a large project you should define your own namespaces, since it will allow you to use names that may have been declared in other namespaces, and this syntax allows you to disambiguate the symbol.

The cout object is an instance of the ostream class and this has already been created for you before the main function is called. The << means that a function called operator << is called and is passed the string (which is an array of char characters). This function prints each character in the string to the console until it reaches a NUL characte.
This is an example of the flexibility of C++, a feature called **operator overloading**. The << operator is usually used with integers, and is used too shift the bits in the integer a specified number of places to the left; x << y will return a value which has every bit in x shifted left by y places, in effect returning a value that has been multiplied by 2^y. However, in the preceding code, in place of the integer x there is the stream object std::cout, and in place of the left shift index there is a string. Clearly, this does not make any sense in the C++ definition of the << operator. The C++ standard has effectively redefined what the << operator means when used with an ostream object on the left-hand side. Furthermore, the << operator in this code will print a string to the console, and so it takes a string on the right-hand side. The C++ Standard Library defines other << operators that allow other data types to be printed to the console. They are all called the same way; the compiler determines which function is compiled dependent upon the type of the parameter used.
Earlier we said that the std::cout object had already been created as an instance of the ostream class, but gave no indication of how this has occurred. This leads us to the last part of the simple source file not already explained: the first line starting with #include. The # here effectively indicates that a message of some kind will be given to the compiler.

There are various types of messages you can send (a few are `#define`, `#ifdef`, `#pragma`, which we will return to elsewhere in this book). In this case, `#include` tells the compiler to copy the contents of the specified file into the source file at this point, which essentially means the contents of that file will be compiled too. The specified file is called a **header file**, and is important in file management and the reuse of code through libraries.

The file `<iostream>` (note, no extension) is part of the Standard Library and can be found in the **include directory** provided with the C++ compiler. The angle brackets (<>) indicate that the compiler should look in the standard directories used to store header files, but you can provide the absolute location of a header file (or the location relative to the current file) using double quotes (""). The C++ Standard Library uses the convention of not using file extensions. You should use the extension `h` (or `hpp` and, rarely, `hxx`) when naming your own header files. The C Runtime Library (which is also available to your C++ code) also uses the extension `h` for its header files.

Creating source files

Start by finding the **Visual Studio 2017** folder on the Start Menu and click on the entry for **Developer Command Prompt for VS2017**. This will start a Windows command prompt and set up the environmental variables to use Visual C++ 2017. However, rather unhelpfully, it will also leave the command line in the Visual Studio folder under the Program Files folder. If you intend to do any development, you will want to move out of this folder to one where creating and deleting files will do no harm. Before you do that, move to the Visual C++ folder and list the files:

```
C:\Program Files\Microsoft Visual Studio\2017\Community>cd
%VCToolsInstallDir%
C:\Program Files\Microsoft Visual
Studio\2017\Community\VC\Tools\MSVC\14.0.10.2517>dir
```

Since the installer will place the C++ files in a folder that includes the current build of the compiler, it is safer to use the environment variable `VCToolsInstallDir` rather than specifying a specific version so that the latest version is used (in this case 14.0.10.2517). There are a few things to notice. First, the folders `bin`, `include`, and `lib`:

Folder	Description
`bin`	This contains, indirectly, the executables for Visual C++. The `bin` folder will contain separate folders for the CPU type you are using, so you will have to navigate below this to get to the actual folder containing the executables. The two main executables are `cl.exe`, which is the C++ compiler and `link.exe`, which is the linker.

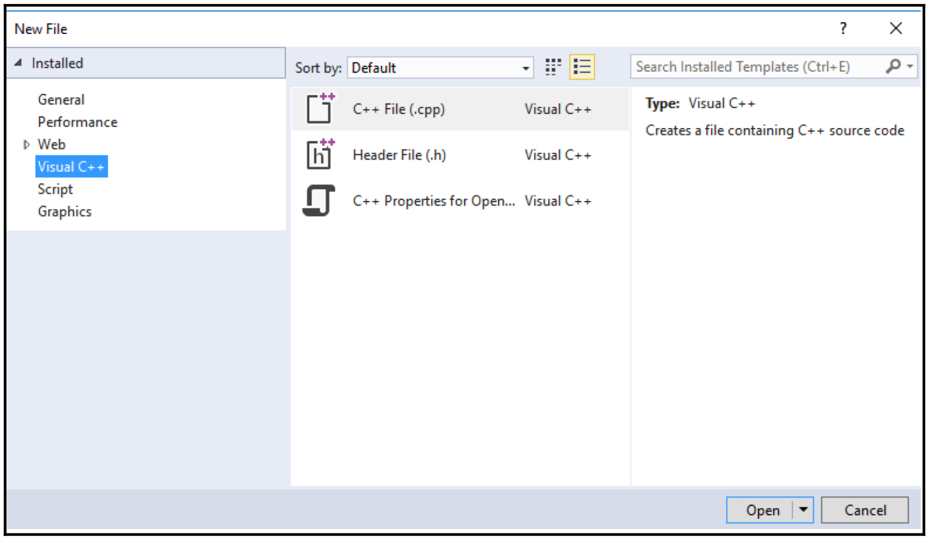
`include`	This folder contains the header files for the C Runtime Library and the C++ Standard Library.
`lib`	This folder contains the static link library files for the C Runtime Library and the C++ Standard Library. Again, there will be separate folders for the CPU type

We will refer back to these folders later in this chapter.

The other thing to point out is the file `vcvarsall.bat` which is under the `VC\Auxillary\Build` folder. When you click on **Developer Command Prompt** for VS2017 on the Start menu, this batch file will be run. If you wish to use an existing command prompt to compile C++ code, you can set that up by running this batch file. The three most important actions of this batch file are to set up the `PATH` environment variable to contain a path to the bin folder, and to set up the `INCLUDE` and `LIB` environment variables to point to the include and lib folders, respectively.

Now navigate to the root directory and create a new folder, `Beginning_C++`, and move to that directory. Next, create a folder for this chapter called `Chapter_01`. Now you can switch to Visual Studio; if this is not already running, start it from the Start menu.

In Visual Studio, click the **File** menu, then **New,** and then the **File...** menu item to get the **New File** dialog, and in the left-hand tree-view, click on the **Visual C++** option. In the middle panel you'll see two options: **C++ File (.cpp)** and **Header File (.h)**, and C++ properties for `Open` folder, as shown in the following screenshot:


New File
Installed
General
Performance
Web
Visual C++
Script
Graphics
Sort by: Default
Search Installed Templates (Ctrl+E)
C++ File (.cpp) Visual C++
Header File (.h) Visual C++
C++ Properties for Open... Visual C++
Type: Visual C++
Creates a file containing C++ source code
Open
Cancel

The first two file types are used for C++ projects, the third type creates a JSON file to aid Visual Studio IntelliSence (help as you type) and will not be used in this book.
Click on the first of these and then click the **Open** button. This will create a new empty file called **Source1.cpp**, so save this to the chapter project folder as **simple.cpp** by clicking on the **File** menu, then **Save Source1.cpp As,** and, navigating to the project folder, change the name in the **File name** box to **simple.cpp** before clicking on the **Save** button.

Now you can type in the code for the simple program, shown as following:

```
#include <iostream>

int main()
{
    std::cout << "Hello, world!n";
}
```

When you have finished typing this code, save the file by clicking on the **File** menu and then the **Save simple.cpp** option in the menu. You are now ready to compile the code.

Compiling the code

Go to the command prompt and type **`cl /?`** command. Since the `PATH` environment is set up to include the path to the `bin` folder, you will see the help pages for the compiler. You can scroll through these pages by pressing the **Return** key until you get back to the command prompt. Most of these options are beyond the scope of this book, but the following table shows some that we will talk about:

Compiler Switch	**Description**
`/c`	Compile only, do not link.
`/D<symbol>`	Defines the constant or macro <symbol>.
`/EHsc`	Enable C++ exception handling but indicate that exceptions from `extern "C"` functions (typically operating system functions) are not handled.
`/Fe:<file>`	Provide the name of the executable file to link to.
`/Fo:<file>`	Provide the name of the object file to compile to.
`/I <folder>`	Provide the name of a folder to use to search for include files.
`/link<linker options>`	Pass <linker options> to the linker. This must come after the source file name and any switches intended for the compiler.

`/Tp <file>`	Compile <file> as a C++ file, even if it does not have `.cpp` or `.cxx` for its file extension.
`/U<symbol>`	Remove the previously defined <symbol> macro or constant.
`/Zi`	Enable debugging information.
`/Zs`	Syntax only, do not compile or link.

Note that some options need spaces between the switch and option, some must not have a space, and for others, the space is optional. In general, if you have the name of a file or folder that contains a space, you should enclose the name in double quotes. Before you use a switch, it is best to consult the help files to find out how it uses spaces.

At the command line, type the **`cl simple.cpp`** command. You will find that the compiler will issue warnings **`C4530`** and **`C4577`**. The reason is that the C++ Standard Library uses exceptions and you have not specified that the compiler should provide the necessary support code for exceptions. It is simple to overcome these warnings by using the `/EHsc` switch. At the command line, type the `cl /EHsc simple.cpp` command. If you typed in the code correctly it should compile:

```
C:\Beginning_C++\Chapter_01>cl /EHsc simple.cpp
Microsoft (R) C/C++ Optimizing Compiler Version 19.00.25017 for x86
Copyright (C) Microsoft Corporation.  All rights reserved

simple.cpp

Microsoft (R) Incremental Linker Version 14.10.25017.0
Copyright (C) Microsoft Corporation.  All rights reserved.
/out:simple.exe

simple.obj
```

By default, the compiler will compile the file to an object file and then pass this file to the linker to link as a command-line executable with the same name as the C++ file, but with the `.exe` extension. The line that says `/out:simple.exe` is generated by the linker, and `/out` is a linker option.

List the contents of the folder. You will find three files: `simple.cpp`, the source file; **simple.obj**, the object file which is the output of the compiler; and `simple.exe`, the output of the linker after it has linked the object file with the appropriate runtime libraries. You may now run the executable by typing `simple` on the command line:

```
C:\Beginning_C++\Chapter_01>simple
Hello, World!
```

Passing parameters between the command-line and an executable

Earlier, you found that the `main` function returned a value and by default this value is zero. When your application finishes, you can return an error code back to the command line; this is so that you can use the executable in batch files and scripts, and use the value to control the flow within the script. Similarly, when you run an executable, you may pass parameters from the command line, which will affect how the executable will behave.

Run the simple application by typing the **`simple`** command on the command line. In Windows, the error code is obtained through the pseudo environment variable `ERRORLEVEL`, so obtain this value through the **`ECHO`** command:

```
C:\Beginning_C++\Chapter_01>simple
Hello, World!

C:\Beginning_C++\Chapter_01>ECHO %ERRORLEVEL%
0
```

To show that this value is returned by the application, change the `main` function to return a value other than 0 (in this case, 99, shown highlighted):

```
    int main()
    {
        std::cout << "Hello, world!n";
        return 99;
    }
```

Compile this code and run it, and then print out the error code as shown previously. You will find that the error code is now given as 99.

This is a very basic mechanism of communication: it only allows you to pass integer values, and the scripts that call your code must know what each value means.

You are much more likely to pass parameters to an application, and these will be passed through your code via parameters to the `main` function. Replace the `main` function with the following:

```
int main(int argc, char *argv[])
{
    std::cout << "there are " << argc << " parameters" <<
    std::endl;
    for (int i = 0; i < argc; ++i)
    {
        std::cout << argv[i] << std::endl;
    }
}
```

When you write the `main` function to take parameters from the command line, the convention is that it has these two parameters.

The first parameter is conventionally called `argc`. It is an integer, and indicates how many parameters were passed to the application. *This parameter is very important.* The reason is that you are about to access memory through an array, and this parameter gives the limit of your access. If you access memory beyond this limit you will have problems: at best you will be accessing uninitialized memory, but at worst you could cause an access violation.

It is important that, whenever you access memory, you understand the amount of memory you are accessing and keep within its limits.

The second parameter is usually called `argv` and is an array of pointers to C strings in memory. You will learn more about arrays and pointers in `Chapter 4`, *Working With Memory, Arrays, and Pointers*, and about strings in `Chapter 9`, *Using Strings*, so we will not give a detailed discussion here. The square brackets (`[]`) indicate that the parameter is an array, and the type of each member of the array is given by the `char *`. The `*` means that each item is a pointer to memory. Normally, this would be interpreted as a pointer to a single item of the type given, but strings are different: the `char *` means that in the memory the pointer points to there will be zero or more characters followed by the `NUL` character (). The length of the string is the count of characters until the `NUL` character.

The third line shown here prints to the console the number of strings passed to the application. In this example, rather than using the newline escape character (n) to add a newline, we use the stream `std::endl`. There are several manipulators you can use, which will be discussed in Chapter 6, *Classes*. The `std::endl` manipulator will put the newline character into the output stream, and then it will flush the stream. This line shows that C++ allows you to chain the use of the << put operator into a stream. The line also shows you that the << put operator is overloaded, that is, there are different versions of the operator for different parameter types (in this case, three: one that takes an integer, used for `argv`, one that takes a string parameter, and another that takes manipulator as a parameter), but the syntax for calling these operators is exactly the same.

Finally, there is a code block to print out every string in the `argv` array, reproduced here:

```
    for (int i = 0; i < argc; ++i)
    {
        std::cout << argv[i] << std::endl;
    }
```

The `for` statement means that the code block will be called until the variable `i` is less than the value of `argc`, and after each successful iteration of the loop, the variable `i` is incremented (using the prefix increment operator `++`). The items in the array are accessed through the square bracket syntax (`[]`). The value passed is an *index* into the array.

Notice that the variable `i` has a starting value of `0`, so the first item accessed is `argv[0]`, and since the `for` loop finishes when the variable `i` has a value of `argc`, it means that the last item in the array accessed is `argv[argc-1]`. This is a typical usage of arrays: the first index is zero and, if there are `n` items in the array, the last item has an index of `n-1`.

Compile and run this code as you have done before, with no parameters:

```
C:\Beginning_C++\Chapter_01>simple
there are 1 parameters
simple
```

Notice that, although you did not give a parameter, the program thinks there is one: the name of the program executable. In fact, this is not just the name, it is the command used to invoke the executable. In this case, you typed the **`simple`** command (without the extension) and got the value of the file `simple` as a parameter printed on the console. Try this again, but this time invoke the program with its full name, `simple.exe`. Now you will find the first parameter is `simple.exe`.

Try calling the code with some actual parameters. Type the **`simple test parameters`** command in the command line:

```
C:\Beginning_C++\Chapter_01>simple test parameters
there are 3 parameters
simple
test parameters
```

This time the program says that there are three parameters, and it has delimited them using the space character. If you want to use a space within a single parameter, you should put the entire string in double quotes:

```
C:\Beginning_C++\Chapter_01>simple "test parameters"
there are 2 parameters
simple
test parameters
```

Bear in mind that `argv` is an array of string pointers, so if you want to pass in a numeric type from the command line and you want to use it as a number in your program, you will have to convert from its string representation accessed through `argv`.

The preprocessor and symbols

The C++ compiler takes several steps to compile a source file. As the name suggests, the compiler preprocessor is at the beginning of this process. The preprocessor locates the header files and inserts them into the source file. It also substitutes macros and defined constants.

Defining constants

There are two main ways to define a constant via the preprocessor: through a compiler switch and in code. To see how this works, let's change the `main` function to print out the value of a constant; the two important lines are highlighted:

```
#include <iostream>
#define NUMBER 4

int main()
{
    std::cout << NUMBER << std::endl;
}
```

The line that starts with `#define` is an instruction to the preprocessor, and it says that, wherever in the text there is the exact symbol `NUMBER`, it should be replaced with 4. It is a text search and replace, but it will replace whole symbols only (so if there is a symbol in the file called `NUMBER99` the `NUMBER` part will not be replaced). After the preprocessor has finished its work the compiler will see the following:

```
    int main()
    {
        std::cout << 4 << std::endl;
    }
```

Compile the original code and run it, and confirm that the program simply prints **4** to the console.

The text search and replace aspect of the preprocessor can cause some odd results, for example, change your `main` function to declare a variable called `NUMBER`, as follows:

```
    int main()
    {
        int NUMBER = 99;
        std::cout << NUMBER << std::endl;
    }
```

Now compile the code. You will get an error from the compiler:

```
C:\Beginning_C++\Chapter_01>cl /EHhc simple.cpp
Microsoft (R) C/C++ Optimizing Compiler Version 19.00.25017 for x86
Copyright (C) Microsoft Corporation.  All rights reserved.

simple.cpp
simple.cpp(7): error C2143: syntax error: missing ';' before 'constant'
simple.cpp(7): error C2106: '=': left operand must be l-value
```

This indicates that there is an error on line 7, which is the new line declaring the variable. However, because of the search and replace conducted by the preprocessor, what the compiler sees is a line that looks as follows:

```
    int 4 = 99;
```

This is not correct C++!

In the code that you have typed, it is obvious what is causing the problem because you have a `#define` directive for the symbol within the same file. In practice, you will include several header files, and these may include files themselves, so the errant `#define` directive could be in one of many files. Equally, your constant symbols may have the same names as variables in header files included after your `#define` directive and may be replaced by the preprocessor.

Using `#define` as a way to define global constants is often not a good idea and there are much better ways in C++, as you'll see in `Chapter 3`, *Exploring C++ Types*.

If you have problems you think are coming from the preprocessor replacing symbols, you can investigate this by looking at the source file passed to the compiler after the preprocessor has done its work. To do this, compile with the `/EP` switch. This will suppress actual compilation and send the output of the preprocessor to `stdout` (the command line). Be aware that this could produce a lot of text, so it is usually better to direct this output to a file and examine that file with the Visual Studio editor.

Another way to provide the values used by the preprocessor is to pass them via a compiler switch. Edit the code and remove the line starting with `#define`. Compile this code as normal (**`cl /EHsc simple.cpp`**), run it, and confirm that the number printed on the console is **99**, the value assigned to the variable. Now compile the code again with the following line:

```
cl /EHsc simple.cpp /DNUMBER=4
```

Note that there is no space between the `/D` switch and the name of the symbol. This tells the preprocessor to replace every `NUMBER` symbol with the text `4` and this results in the same errors as above, indicating that the preprocessor is attempting to replace the symbol with the value provided.

Tools such as Visual C++ and nmake projects will have a mechanism to define symbols through the C++ compiler. The `/D` switch is used to define just one symbol, and if you want to define others they will have their own `/D` switch.

You are now wondering why C++ has such an odd facility that appears only to cause confusing errors. Defining symbols can be very powerful once you understand what the preprocessor is doing.

Using macros

One useful feature of preprocessor symbols is **macros**. A macro has parameters and the preprocessor will ensure that the search and replace will replace a symbol in the macro with the symbol used as a parameter to the macro.

Edit the `main` function to look as follows:

```
#include <iostream>

#define MESSAGE(c, v)
for(int i = 1; i < c; ++i) std::cout << v[i] << std::endl;

int main(int argc, char *argv[])
{
    MESSAGE(argc, argv);
    std::cout << "invoked with " << argv[0] << std::endl;
}
```

The `main` function calls a macro called `MESSAGE` and passes the command line parameters to it. The function then prints the first command line parameters (the invocation command) to the console. `MESSAGE` is not a function, it is a macro, which means that the preprocessor will replace every occurrence of `MESSAGE` with two parameters with the text defined previously, replacing the `c` parameter with whatever is passed as the first parameter of the macro, and replacing `v` with whatever is used as the second parameter. After the preprocessor has finished processing the file, `main` will look as follows:

```
int main(int argc, char *argv[])
{
    for(int i = 1; i < argc; ++i)
        std::cout << argv[i] << std::endl;
    std::cout << "invoked with " << argv[0] << std::endl;
}
```

Note that, in the macro definition, the backslash () is used as a line-continuation character, so you can have multiline macros. Compile and run this code with one or more parameters, and confirm that `MESSAGE` prints out the command-line parameters.

Using symbols

You can define a symbol without a value and the preprocessor can be told to test for whether a symbol is defined or not. The most obvious situation for this is compiling different code for debug builds than for release builds.

Edit the code to add the lines highlighted here:

```
#ifdef DEBUG
#define MESSAGE(c, v)
for(int i = 1; i < c; ++i) std::cout << v[i] << std::endl;
#else
#define MESSAGE
#endif
```

The first line tells the preprocessor to look for the `DEBUG` symbol. If this symbol is defined (regardless of its value), then the first definition of the `MESSAGE` macro will be used. If the symbol is not defined (a release build) then the `MESSAGE` symbol is defined, but it does nothing: essentially, occurrences of `MESSAGE` with two parameters will be removed from the code.

Compile this code and run the program with one or more parameters. For example:

```
C:\Beginning_C++\Chapter_01>simple test parameters
invoked with simple
```

This shows that the code has been compiled without `DEBUG` defined so `MESSAGE` is defined to do nothing. Now compile this code again, but this time with the **/DDEBUG** switch to define the `DEBUG` symbol. Run the program again and you will see that the command-line parameters are printed on the console:

```
C:\Beginning_C++\Chapter_01>simple test parameters
test parameters
invoked with simple
```

This code has used a macro, but you can use conditional compilation with symbols anywhere in your C++ code. Symbols used in this way allow you to write flexible code and choose the code to be compiled through a symbol defined on the compiler command line. Furthermore, the compiler will define some symbols itself, for example, `__DATE__` will have the current date, `__TIME__` will have the current time, and `__FILE__` will have the current file name.

Microsoft, and other compiler producers, defines a long list of symbols that you can access, and you are advised to look these up in the manual. A few that you may find useful are as follows: `__cplusplus` will be defined for C++ source files (but not for C files) so you can identify code that needs a C++ compiler; `_DEBUG` is set for debug builds (note the preceding underscore), and `_MSC_VER` has the current version of the Visual C++ compiler, so you can use the same source for various versions of the compiler.

Using pragmas

Associated with symbols and conditional compilation is the compiler directive, #pragma once. Pragmas are directives specific to the compiler, and different compilers will support different pragmas. Visual C++ defines the #pragma once to solve the problem that occurs when you have multiple header files each including similar header files. The problem is that it may result in the same items being defined more than once and the compiler will flag this as an error. There are two ways to do this, and the <iostream> header file that you include next uses both of these techniques. You can find this file in the Visual C++ include folder. At the top of the file you will find the following:

```
// ostream standard header
#pragma once
#ifndef _IOSTREAM_
#define _IOSTREAM_
```

At the bottom, you will find the following line:

```
#endif /* _IOSTREAM_ */
```

First the conditional compilation: the first time this header is included, the symbol _IOSTREAM_ will not be defined, so the symbol is defined and then the rest of the file will be included until the #endif line.

This illustrates good practice when using conditional compilation. For every #ifndef, there must be a #endif, and often there may be hundreds of lines between them. It is a good idea, when you use #ifdef or #ifundef, to provide a comment with the corresponding #else and #endif, indicating the symbol it refers to.

If the file is included again then the symbol _IOSTREAM_ will be defined, so the code between the #ifndef and #endif will be ignored. However, it is important to point out that, even if the symbol is defined, the header file will still be loaded and processed because the instructions about what to do are contained within the file.

The #pragma once performs the same action as the conditional compilation, but it gets around the problem of using a symbol that could be duplicated. If you add this single line to the top of your header file, you are instructing the preprocessor to load and process this file once. The preprocessor maintains a list of the files that it has processed, and if a subsequent header tries to load a file that has already been processed, that file will not be loaded and will not be processed. This reduces the time it takes for the project to be preprocessed.

The next thing to do is to convert the arguments into a form that the code can use. The command-line parameters are passed to the program in an array of strings; however, we are interpreting some of those parameters as floating-point numbers (in fact, double-precision floating-point numbers). The C runtime provides a function called `atof`, which is available through the C++ Standard Library (in this case, `<iostream>` includes files that include `<cmath>`, where `atof` is declared).

It is a bit counter-intuitive to get access to a math function such as `atof` through including a file associated with stream input and output. If this makes you uneasy, you can add a line after the `include` lines to include the `<cmath>` file. As mentioned in the previous chapter, the C++ Standard Library headers have been written to ensure that a header file is only included once, so including `<cmath>` twice has no ill effect. This was not done in the preceding code, because it was argued that `atof` is a string function and the code includes the `<string>` header and, indeed, `<cmath>` is included via the files the `<string>` header includes.

Add the following lines to the bottom of the `main` function. The first two lines convert the second and fourth parameters (remember, C++ arrays are zero-based indexed) to `double` values. The final line declares a variable to hold the result:

```
    double arg1 = atof(argv[1]);
    double arg2 = atof(argv[3]);
    double result = 0;
```

Now we need to determine which operator was passed and perform the requested action. We will do this with a `switch` statement. We know that the `op` variable will be valid, and so we do not have to provide a `default` clause to catch the values we have not tested for. Add a `switch` statement to the bottom of the function:

```
    double arg1 = atof(argv[1]);
    double arg2 = atof(argv[3]);
    double result = 0;

    switch(op)
    {
    }
```

The division by zero assigned `result` to a value of `NAN`, which is a constant defined in `<math.h>` (included via `<cmath>`), and means "not a number." The `double` overload of the insertion operator for the `cout` object tests to see if the number has a valid value, and if the number has a value of `NAN`, it prints the string **inf**. In our application, we can test for a zero divisor, and we treat the user action of passing a zero as being an error. Thus, change the code so that it reads as follows:

```
case '/':
if (arg2 == 0) {
    cout << endl << "divide by zero!" << endl;
    return 1;
} else {
    result = arg1 / arg2;
}
break;
```

Now when the user passes zero as a divisor, you will get a `divide by zero!` message.

You can now compile the full example and test it out. The application supports floating-point arithmetic using the `+`, `-`, `*`, and `/` operators, and will handle the case of dividing by zero.

Summary

In this chapter, you have learned how to format your code, and how to identify expressions and statements. You have learned how to identify the scope of variables, and how to group collections of functions and variables into namespaces so that you can prevent name clashes. You have also learned the basic plumbing in C++ of looping and branching code, and how the built-in operators work. Finally, you put all of this together in a simple application that allows you to perform simple calculations at the command line.

In the following chapter, you will learn about C++ types and how to convert values from one type to another.

The three functions return a `string` object. In the first two cases, the `string` has the lifetime of the program and so a reference can be returned. In the last function, the function returns a string literal, so a temporary `string` object is constructed. All three can be used to provide a `string` value. For example:

```
    cout << get_global() << endl;
    cout << get_static() << endl;
    cout << get_temp() << endl;
```

All three can provide a string that can be used to assign a `string` object. The important point is that the first two functions return along a lived object, but the third function returns a temporary object, but these objects can be used the same.

If these functions returned access to a large object, you would not want to pass the object to another function, so instead, in most cases, you'll want to pass the objects returned by these functions as references. For example:

```
    void use_string(string& rs);
```

The reference parameter prevents another copy of the string. However, this is just half of the story. The `use_string` function could manipulate the string. For example, the following function creates a new `string` from the parameter, but replaces the letters a, b, and o with an underscore (indicating the gaps in words without those letters, replicating what life would be like without donations of the blood types A, B, and O). A simple implementation would look like this:

```
    void use_string(string& rs)
    {
        string s { rs };
        for (size_t i = 0; i < s.length(); ++i)
        {
            if ('a' == s[i] || 'b' == s[i] || 'o' == s[i])
            s[i] = '_';
        }
        cout << s << endl;
    }
```

The string object has an index operator (`[]`), so you can treat it like an array of characters, both reading the values of characters and assigning values to character positions. The size of the `string` is obtained through the `length` function, which returns an `unsigned int` (`typedef` to `size_t`). Since the parameter is a reference, it means that any change to the `string` will be reflected in the `string` passed to the function. The intention of this code is to leave other variables intact, so it first makes a copy of the parameter. Then on the copy, the code iterates through all of the characters changing the `a`, `b`, and `o` characters to an underscore before printing out the result.