FIFTH EDITION

wrox
A Wiley Brand

Professional C++

Marc Gregoire

PROFESSIONAL C++

Continues

▶ PART IV MASTERING ADVANCED FEATURES OF C++

▶ PART V C++ SOFTWARE ENGINEERING

▶ PART VI APPENDICES

PROFESSIONAL

C++

PROFESSIONAL

C++

Fifth Edition

Marc Gregoire

wrox™
A Wiley Brand

Professional C++

Published simultaneously in Canada and the United Kingdom

ISBN: 978-1-119-69540-0
ISBN: 978-1-119-69550-9 (ebk)
ISBN: 978-1-119-69545-5 (ebk)

Manufactured in the United States of America

Limit of Liability/Disclaimer of Warranty: The publisher and the author make no representations or warranties with respect to the accuracy or completeness of the contents of this work and specifically disclaim all warranties, including without limitation warranties of fitness for a particular purpose. No warranty may be created or extended by sales or promotional materials. The advice and strategies contained herein may not be suitable for every situation. This work is sold with the understanding that the publisher is not engaged in rendering legal, accounting, or other professional services. If professional assistance is required, the services of a competent professional person should be sought. Neither the publisher nor the author shall be liable for damages arising herefrom. The fact that an organization or Web site is referred to in this work as a citation and/or a potential source of further information does not mean that the author or the publisher endorses the information the organization or Web site may provide or recommendations it may make. Further, readers should be aware that Internet Web sites listed in this work may have changed or disappeared between when this work was written and when it is read.

For general information on our other products and services please contact our Customer Care Department within the United States at (877) 762-2974, outside the United States at (317) 572-3993 or fax (317) 572-4002.

Wiley publishes in a variety of print and electronic formats and by print-on-demand. Some material included with standard print versions of this book may not be included in e-books or in print-on-demand. If this book refers to media such as a CD or DVD that is not included in the version you purchased, you may download this material at booksupport.wiley.com. For more information about Wiley products, visit www.wiley.com.

Library of Congress Control Number: 2020950208

Dedicated to my wonderful parents and my brother,
who are always there for me. Their support and
patience helped me in finishing this book.

ABOUT THE AUTHOR

MARC GREGOIRE is a software architect from Belgium. He graduated from the University of Leuven, Belgium, with a degree in "Burgerlijk ingenieur in de computer wetenschappen" (equivalent to a master of science in engineering in computer science). The year after, he received an advanced master's degree in artificial intelligence, *cum laude*, at the same university. After his studies, Marc started working for a software consultancy company called Ordina Belgium. As a consultant, he worked for Siemens and Nokia Siemens Networks on critical 2G and 3G software running on Solaris for telecom operators. This required working in international teams stretching from South America and the United States to Europe, the Middle East, Africa, and Asia. Now, Marc is a software architect at Nikon Metrology (`nikonmetrology.com`), a division of Nikon and a leading provider of precision optical instruments, X-ray machines, and metrology solutions for X-ray, CT, and 3-D geometric inspection.

His main expertise is C/C++, specifically Microsoft VC++ and the MFC framework. He has experience in developing C++ programs running 24/7 on Windows and Linux platforms: for example, KNX/EIB home automation software. In addition to C/C++, Marc also likes C#.

Since April 2007, he has received the annual Microsoft MVP (Most Valuable Professional) award for his Visual C++ expertise.

Marc is the founder of the Belgian C++ Users Group (`becpp.org`), co-author of *C++ Standard Library Quick Reference* 1st and 2nd editions (Apress), a technical editor for numerous books for several publishers, and a regular speaker at the CppCon C++ conference. He maintains a blog at `www.nuonsoft.com/blog/` and is passionate about traveling and gastronomic restaurants.

ABOUT THE TECHNICAL EDITORS

PETER VAN WEERT is a Belgian software engineer whose main interests and expertise are application software development, programming languages, algorithms, and data structures.

He received his master of science degree in computer science *summa cum laude* with congratulations from the Board of Examiners from the University of Leuven. In 2010, he completed his PhD thesis on the design and efficient compilation of rule-based programming languages at the research group for declarative programming languages and artificial intelligence. During his doctoral studies he was a teaching assistant for object-oriented programming (Java), software analysis and design, and declarative programming.

Peter then joined Nikon Metrology, where he worked on large-scale, industrial application software in the area of 3-D laser scanning and point cloud inspection for over six years. Today, Peter is senior C++ engineer and Scrum team leader at Medicim, the R&D unit for digital dentistry software of Envista Holdings. At Medicim, he codevelops a suite of applications for dental professionals, capable of capturing patient data from a wide range of hardware, with advanced diagnostic functionality and support for implant planning and prosthetic design.

Common themes in his professional career include advanced desktop application development, mastering and refactoring of code bases of millions of lines of C++ code, high-performant, real-time processing of 3-D data, concurrency, algorithms and data structures, interfacing with cutting-edge hardware, and leading agile development teams.

Peter is a regular speaker at, and board member of, the Belgian C++ Users Group. He also co-authored two books: *C++ Standard Library Quick Reference* and *Beginning C++* (5th edition), both published by Apress.

OCKERT J. DU PREEZ is a self-taught developer who started learning programming in the days of QBasic. He has written hundreds of developer articles over the years detailing his programming quests and adventures. His articles can be found on CodeGuru (`codeguru.com`), Developer.com (`developer.com`), DevX (`devx.com`), and Database Journal (`databasejournal.com`). Software development is his second love, just after his wife and child.

He knows a broad spectrum of development languages including C++, C#, VB.NET, JavaScript, and HTML. He has written the books *Visual Studio 2019 In-Depth* (BpB Publications) and *JavaScript for Gurus* (BpB Publications).

He was a Microsoft Most Valuable Professional for .NET (2008–2017).

ACKNOWLEDGMENTS

I THANK THE JOHN WILEY & SONS AND WROX PRESS editorial and production teams for their support. Especially, thank you to Jim Minatel, executive editor at Wiley, for giving me a chance to write this fifth edition; Kelly Talbot, project editor, for managing this project; and Kim Wimpsett, copy editor, for improving readability and consistency and making sure the text is grammatically correct.

Thanks to technical editor Hannes Du Preez for checking the technical accuracy of the book. His contributions in strengthening this book are greatly appreciated.

A very special thank you to technical editor Peter Van Weert for his outstanding contributions. His considerable advice and insights have truly elevated this book to a higher level.

Of course, the support and patience of my parents and my brother were very important in finishing this book. I would also like to express my sincere gratitude to my employer, Nikon Metrology, for supporting me during this project.

Finally, I thank you, the reader, for trying this approach to professional C++ software development.

—Marc Gregoire

CONTENTS

PART II: PROFESSIONAL C++ SOFTWARE DESIGN

PART III: C++ CODING THE PROFESSIONAL WAY

CHAPTER 15: OVERLOADING C++ OPERATORS 535

CHAPTER 18: STANDARD LIBRARY CONTAINERS 627

CHAPTER 19: FUNCTION POINTERS, FUNCTION OBJECTS, AND LAMBDA EXPRESSIONS 699

PART IV: MASTERING ADVANCED FEATURES OF C++

PART V: C++ SOFTWARE ENGINEERING

CHAPTER 32: INCORPORATING DESIGN TECHNIQUES AND FRAMEWORKS 1083

CHAPTER 33: APPLYING DESIGN PATTERNS 1105

PART VI: APPENDICES

INTRODUCTION

The development of C++ started in 1982 by Bjarne Stroustrup, a Danish computer scientist, as the successor of C with Classes. In 1985, the first edition of *The C++ Programming Language* book was released. The first standardized version of C++ was released in 1998, called C++98. In 2003, C++03 came out and contained a few small updates. After that, it was silent for a while, but traction slowly started building up, resulting in a major update of the language in 2011, called C++11. From then on, the C++ Standard Committee has been on a three-year cycle to release updated versions, giving us C++14, C++17, and now C++20. All in all, with the release of C++20 in 2020, C++ is almost 40 years old and still going strong. In most rankings of programming languages in 2020, C++ is in the top four. It is being used on an extremely wide range of hardware, going from small devices with embedded microprocessors all the way up to multirack supercomputers. Besides wide hardware support, C++ can be used to tackle almost any programming job, be it games on mobile platforms, performance-critical artificial intelligence (AI) and machine learning (ML) software, real-time 3-D graphics engines, low-level hardware drivers, entire operating systems, and so on. The performance of C++ programs is hard to match with any other programming language, and as such, it is the de facto language for writing fast, powerful, and enterprise-class object-oriented programs. As popular as C++ has become, the language is surprisingly difficult to grasp in full. There are simple, but powerful, techniques that professional C++ programmers use that don't show up in traditional texts, and there are useful parts of C++ that remain a mystery even to experienced C++ programmers.

Too often, programming books focus on the syntax of the language instead of its real-world use. The typical C++ text introduces a major part of the language in each chapter, explaining the syntax and providing an example. *Professional* C++ does not follow this pattern. Instead of giving you just the nuts and bolts of the language with little practical context, this book will teach you how to use C++ in the real world. It will show you the little-known features that will make your life easier, as well as the programming techniques that separate novices from professional programmers.

WHO THIS BOOK IS FOR

Even if you have used the language for years, you might still be unfamiliar with the more advanced features of C++, or you might not be using the full capabilities of the language. Perhaps you write competent C++ code, but would like to learn more about design and good programming style in C++. Or maybe you're relatively new to C++ but want to learn the "right" way to program from the start. This book will meet those needs and bring your C++ skills to the professional level.

Because this book focuses on advancing from basic or intermediate knowledge of C++ to becoming a professional C++ programmer, it assumes that you have some knowledge about programming. Chapter 1, "A Crash Course in C++ and the Standard Library," covers the basics of C++ as a refresher, but it is not a substitute for actual training in programming. If you are just starting with C++ but you have significant experience in another programming language such as C, Java, or C#, you should be able to pick up most of what you need from Chapter 1.

In any case, you should have a solid foundation in programming fundamentals. You should know about loops, functions, and variables. You should know how to structure a program, and you should be familiar with fundamental techniques such as recursion. You should have some knowledge of common data structures such as queues, and useful algorithms such as sorting and searching. You don't need to know about object-oriented programming just yet—that is covered in Chapter 5, "Designing with Objects."

You will also need to be familiar with the compiler you will be using to compile your code. Two compilers, Microsoft Visual C++ and GCC, are introduced later in this introduction. For other compilers, refer to the documentation that came with your compiler.

WHAT THIS BOOK COVERS

Professional C++ uses an approach to C++ programming that will both increase the quality of your code and improve your programming efficiency. You will find discussions on new C++20 features throughout this fifth edition. These features are not just isolated to a few chapters or sections; instead, examples have been updated to use new features when appropriate.

Professional C++ teaches you more than just the syntax and language features of C++. It also emphasizes programming methodologies, reusable design patterns, and good programming style. The *Professional C++* methodology incorporates the entire software development process, from designing and writing code to debugging and working in groups. This approach will enable you to master the C++ language and its idiosyncrasies, as well as take advantage of its powerful capabilities for large-scale software development.

Imagine users who have learned all of the syntax of C++ without seeing a single example of its use. They know just enough to be dangerous! Without examples, they might assume that all code should go in the `main()` function of the program or that all variables should be global—practices that are generally not considered hallmarks of good programming.

Professional C++ programmers understand the correct way to use the language, in addition to the syntax. They recognize the importance of good design, the theories of object-oriented programming, and the best ways to use existing libraries. They have also developed an arsenal of useful code and reusable ideas.

By reading and understanding this book, you will become a professional C++ programmer. You will expand your knowledge of C++ to cover lesser known and often misunderstood language features. You will gain an appreciation for object-oriented design and acquire top-notch debugging skills. Perhaps most important, you will finish this book armed with a wealth of reusable ideas that you can actually apply to your daily work.

There are many good reasons to make the effort to be a professional C++ programmer as opposed to a programmer who knows C++. Understanding the true workings of the language will improve the quality of your code. Learning about different programming methodologies and processes will help you to work better with your team. Discovering reusable libraries and common design patterns will improve your daily efficiency and help you stop reinventing the wheel. All of these lessons will make you a better programmer and a more valuable employee. While this book can't guarantee you a promotion, it certainly won't hurt.

HOW THIS BOOK IS STRUCTURED

This book is made up of five parts.

Part I, "Introduction to Professional C++," begins with a crash course in C++ basics to ensure a foundation of C++ knowledge. Following the crash course, Part I goes deeper into working with strings, because strings are used extensively in most examples throughout the book. The last chapter of Part I explores how to write *readable* C++ code.

Part II, "Professional C++ Software Design," discusses C++ design methodologies. You will read about the importance of design, the object-oriented methodology, and the importance of code reuse.

Part III, "C++ Coding the Professional Way," provides a technical tour of C++ from the professional point of view. You will read about the best ways to manage memory in C++, how to create reusable classes, and how to leverage important language features such as inheritance. You will also learn techniques for input and output, error handling, string localization, how to work with regular expressions, and how to structure your code in reusable components called modules. You will read about how to implement operator overloading, how to write templates, how to put restrictions on template parameters using concepts, and how to unlock the power of lambda expressions and function objects. This part also explains the C++ Standard Library, including containers, iterators, ranges, and algorithms. You will also read about some additional libraries that are available in the standard, such as the libraries to work with time, dates, time zones, random numbers, and the filesystem.

Part IV, "Mastering Advanced Features of C++," demonstrates how you can get the most out of C++. This part of the book exposes the mysteries of C++ and describes how to use some of its more advanced features. You will read about how to customize and extend the C++ Standard Library to your needs, advanced details on template programming, including template metaprogramming, and how to use multithreading to take advantage of multiprocessor and multicore systems.

Part V, "C++ Software Engineering," focuses on writing enterprise-quality software. You'll read about the engineering practices being used by programming organizations today; how to write efficient C++ code; software testing concepts, such as unit testing and regression testing; techniques used to debug C++ programs; how to incorporate design techniques, frameworks, and conceptual object-oriented design patterns into your own code; and solutions for cross-language and cross-platform code.

The book concludes with a useful chapter-by-chapter guide to succeeding in a C++ technical interview, an annotated bibliography, a summary of the C++ header files available in the standard, and a brief introduction to the Unified Modeling Language (UML).

This book is not a reference of every single class, method, and function available in C++. The book *C++17 Standard Library Quick Reference* by Peter Van Weert and Marc Gregoire (Apress, 2019.

ISBN: 978-1-4842-4923-9) is a condensed reference to all essential data structures, algorithms, and functions provided by the C++ Standard Library up until the C++17 standard. Appendix B lists a couple more references. Two excellent online references are:

- `cppreference.com`: You can use this reference online or download an offline version for use when you are not connected to the Internet.
- `cplusplus.com/reference/`

When I refer to a "Standard Library Reference" in this book, I am referring to one of these detailed C++ references.

The following are additional excellent online resources:

- `github.com/isocpp/CppCoreGuidelines`: The C++ *Core Guidelines* are a collaborative effort led by Bjarne Stroustrup, inventor of the C++ language itself. They are the result of many person-years of discussion and design across a number of organizations. The aim of the guidelines is to help people to use modern C++ effectively. The guidelines are focused on relatively higher-level issues, such as interfaces, resource management, memory management, and concurrency.
- `github.com/Microsoft/GSL`: This is an implementation by Microsoft of the *Guidelines Support Library* (GSL) containing functions and types that are suggested for use by the C++ Core Guidelines. It's a header-only library.
- `isocpp.org/faq`: This is a large collection of frequently asked C++ questions.
- `stackoverflow.com`: Search for answers to common programming questions, or ask your own questions.

CONVENTIONS

To help you get the most from the text and keep track of what's happening, a number of conventions are used throughout this book.

> **WARNING** *Boxes like this one hold important, not-to-be-forgotten information that is directly relevant to the surrounding text.*

> **NOTE** *Tips, hints, tricks, and asides to the current discussion are placed in boxes like this one.*

As for styles in the text:

Important words are *italic* when they are introduced.

Keyboard strokes are shown like this: Ctrl+A.

Filenames and code within the text are shown like so: `monkey.cpp`.

URLs are shown like this: `wrox.com`.

Code is presented in three different ways:

```
// Comments in code are shown like this.
In code examples, new and important code is highlighted like this.
Code that's less important in the present context or that has been shown before is
formatted like this.
```

C++20 Paragraphs or sections that are specific to the C++20 standard have a little C++20 icon on the left, just as this paragraph does. C++11, C++14, and C++17 features are not marked with any icon.

WHAT YOU NEED TO USE THIS BOOK

All you need to use this book is a computer with a C++ compiler. This book focuses only on parts of C++ that have been standardized, and not on vendor-specific compiler extensions.

Any C++ Compiler

You can use whichever C++ compiler you like. If you don’t have a C++ compiler yet, you can download one for free. There are a lot of choices. For example, for Windows, you can download Microsoft Visual Studio Community Edition, which is free and includes Visual C++. For Linux, you can use GCC or Clang, which are also free.

The following two sections briefly explain how to use Visual C++ and GCC. Refer to the documentation that came with your compiler for more details.

COMPILERS AND C++20 FEATURE SUPPORT

This book discusses new features introduced with the C++20 standard. At the time of this writing, no compilers were fully C++20 compliant yet. Some new features were only supported by some compilers and not others, while other features were not yet supported by any compiler. Compiler vendors are hard at work to catch up with all new features, and I’m sure it won’t take long before there will be fully C++20-compliant compilers available. You can keep track of which compiler supports which features at `en.cppreference.com/w/cpp/compiler_support`.

COMPILERS AND C++20 MODULE SUPPORT

At the time of this writing, there was no compiler available yet that fully supported C++20 modules. There was experimental support in some of the compilers, but it was still incomplete. This book uses modules everywhere. We did our best to make sure all sample code would compile once compilers fully support modules, but since we were not able to compile and test all examples, some errors might have crept in. When you use a compiler with support for modules and you encounter problems with any of the code samples, double-check the list of errata for the book at `www.wiley.com/go/proc++5e` to see if it's a known issue. If your compiler does not yet support modules, you can convert modularized code to non-modularized code, as explained briefly in Chapter 11, "Odds and Ends."

Example: Microsoft Visual C++ 2019

First, you need to create a project. Start Visual C++ 2019, and on the welcome screen, click the Create A New Project button. If the welcome screen is not shown, select File ➪ New ➪ Project. In the Create A New Project dialog, search for the Console App project template with tags C++, Windows, and Console, and click Next. Specify a name for the project and a location where to save it, and click Create.

Once your new project is loaded, you can see a list of project files in the Solution Explorer. If this docking window is not visible, select View ➪ Solution Explorer. A newly created project will contain a file called `<projectname>.cpp`. You can start writing your C++ code in that `.cpp` file, or if you want to compile source code files from the downloadable source archive for this book, select the `<projectname>.cpp` file in the Solution Explorer and delete it. You can add new files or existing files to a project by right-clicking the project name in the Solution Explorer and then selecting Add ➪ New Item or Add ➪ Existing Item.

At the time of this writing, Visual C++ 2019 did not yet automatically enable C++20 features. To enable C++20 features, in the Solution Explorer window, right-click your project and click Properties. In the Properties window, go to Configuration Properties ➪ C/C++ ➪ Language, and set the C++ Language Standard option to ISO C++20 Standard or Preview - Features from the Latest C++ Working Draft, whichever is available in your version of Visual C++. These options are accessible only if your project contains at least one `.cpp` file.

Finally, select Build ➪ Build Solution to compile your code. When it compiles without errors, you can run it with Debug ➪ Start Debugging.

Module Support

At the time of this writing, Visual C++ 2019 did not yet have full support for modules. Authoring and consuming your own modules usually works just fine, but importing Standard Library headers such as the following did not yet work out of the box:

```
import <iostream>;
```

To make such import declarations work, for the time being you need to add a separate header file to your project, for example called `HeaderUnits.h`, which contains an import declaration for every Standard Library header you want to import. Here's an example:

```
// HeaderUnits.h
#pragma once
import <iostream>;
import <vector>;
import <optional>;
import <utility>;
// ...
```

Next, right-click the `HeaderUnits.h` file in the Solution Explorer and click Properties. In Configuration Properties ➪ General, set Item Type to C/C++ Compiler and click Apply. Next, in Configuration Properties ➪ C/C++ ➪ Advanced, set Compile As to Compile as C++ Header Unit (/exportHeader) and click OK.

When you now recompile your project, all import declarations that have a corresponding import declaration in your `HeaderUnits.h` file should compile fine.

If you are using module implementation partitions (see Chapter 11), also known as internal partitions, then right-click all files containing such implementation partitions, click Properties, go to Configuration Properties ➪ C/C++ ➪ Advanced, and set the Compile As option to Compile as C++ Module Internal Partition (/internalPartition) and click OK.

Example: GCC

Create your source code files with any text editor you prefer and save them to a directory. To compile your code, open a terminal and run the following command, specifying all your `.cpp` files that you want to compile:

```
g++ -std=c++2a -o <executable_name> <source1.cpp> [source2.cpp ...]
```

The `-std=c++2a` option is required to tell GCC to enable C++20 support. This option will change to `-std=C++20` once GCC is fully C++20 compliant.

Module Support

At the time of this writing, GCC only had experimental support for modules through a special version of GCC (branch devel/c++-modules). When you are using such a version of GCC, module support is enabled with the `-fmodules-ts` option, which might change to `-fmodules` in the future.

Unfortunately, import declarations of Standard Library headers such as the following were not yet properly supported:

```
import <iostream>;
```

If that's the case, simply replace such import declarations with corresponding `#include` directives:

```
#include <iostream>
```

For example, the `AirlineTicket` example from Chapter 1 uses modules. After having replaced the imports for Standard Library headers with `#include` directives, you can compile the `AirlineTicket` example by changing to the directory containing the code and running the following command:

```
g++ -std=c++2a -fmodules-ts -o AirlineTicket AirlineTicket.cppm AirlineTicket.cpp
AirlineTicketTest.cpp
```

When it compiles without errors, you can run it as follows:

```
./AirlineTicket
```

std::format Support

Many code samples in this book use `std::format()`, introduced in Chapter 1. At the time of this writing, there was no compiler yet that had support for `std::format()`. However, as long as your compiler doesn't support `std::format()` yet, you can use the freely available {fmt} library as a drop-in replacement:

1. Download the latest version of the {fmt} library from `https://fmt.dev/` and extract the code on your machine.
2. Copy the `include/fmt` and `src` directories to `fmt` and `src` subdirectories in your project directory, and then add `fmt/core.h`, `fmt/format.h`, `fmt/format-inl.h`, and `src/format.cc` to your project.
3. Add a file called `format` (no extension) to the root directory of your project and add the following code to it:

   ```
   #pragma once
   #define FMT_HEADER_ONLY
   #include "fmt/format.h"
   namespace std
   {
       using fmt::format;
       using fmt::format_error;
       using fmt::formatter;
   }
   ```

4. Finally, add your project root directory (the directory containing the `format` file) as an additional include directory for your project. For example, in Visual C++, right click your project in the Solution Explorer, click Properties, go to Configuration Properties ➪ C/C++ ➪ General, and add $(ProjectDir); to the front of the Additional Include Directories option.

> **NOTE** *Don't forget to undo these steps once your compiler supports the standard* `std::format()`.

READER SUPPORT FOR THIS BOOK

The following sections describe different options to get support for this book.

Companion Download Files

As you work through the examples in this book, you may choose either to type in all the code manually or to use the source code files that accompany the book. However, I suggest you type in all the code manually because it greatly benefits the learning process and your memory. All of the source code used in this book is available for download at `www.wiley.com/go/proc++5e`.

> **NOTE** *Because many books have similar titles, you may find it easiest to search by ISBN; for this book, the ISBN is 978-1-119-69540-0.*

Once you've downloaded the code, just decompress it with your favorite decompression tool.

How to Contact the Publisher

If you believe you've found a mistake in this book, please bring it to our attention. At John Wiley & Sons, we understand how important it is to provide our customers with accurate content, but even with our best efforts an error may occur.

To submit your possible errata, please e-mail it to our Customer Service Team at `wileysupport@wiley.com` with "Possible Book Errata Submission" as a subject line.

How to Contact the Author

If you have any questions while reading this book, the author can easily be reached at `marc.gregoire@nuonsoft.com` and will try to get back to you in a timely manner.

PART I
Introduction to Professional C++

- **CHAPTER 1:** A Crash Course in C++ and the Standard Library
- **CHAPTER 2:** Working with Strings and String Views
- **CHAPTER 3:** Coding with Style

1

A Crash Course in C++ and the Standard Library

WHAT'S IN THIS CHAPTER?

- A brief overview of the most important parts and syntax of the C++ language and the Standard Library
- How to write a basic class
- How scope resolution works
- What uniform initialization is
- The use of `const`
- What pointers, references, exceptions, and type aliases are
- Basics of type inference

WILEY.COM DOWNLOADS FOR THIS CHAPTER

Please note that all the code examples for this chapter are available as a part of the chapter's code download on this book's website at `www.wiley.com/go/proc++5e` on the Download Code tab.

The goal of this chapter is to cover briefly the most important parts of C++ so that you have a foundation of knowledge before embarking on the rest of this book. This chapter is not a comprehensive lesson in the C++ programming language or the Standard Library. Certain basic points, such as what a program is and what recursion is, are not covered. Esoteric points, such as the definition of a `union`, or the `volatile` keyword, are also omitted. Certain parts of the C language that are less relevant in C++ are also left out, as are parts of C++ that get in-depth coverage in later chapters.

This chapter aims to cover the parts of C++ that programmers encounter every day. For example, if you're fairly new to C++ and don't understand what a reference variable is, you'll learn about that kind of variable here. You'll also learn the basics of how to use the functionality available in the Standard Library, such as `vector` containers, `optional` values, `string` objects, and more. These parts of the Standard Library are briefly introduced in Chapter 1 so that these modern constructs can be used throughout examples in this book from the beginning.

If you already have significant experience with C++, skim this chapter to make sure that there aren't any fundamental parts of the language on which you need to brush up. If you're new to C++, read this chapter carefully and make sure you understand the examples. If you need additional introductory information, consult the titles listed in Appendix B.

C++ CRASH COURSE

The C++ language is often viewed as a "better C" or a "superset of C." It was mainly designed to be an object-oriented C, commonly called as "C with classes." Later on, many of the annoyances and rough edges of the C language were addressed as well. Because C++ is based on C, some of the syntax you'll see in this section will look familiar to you if you are an experienced C programmer. The two languages certainly have their differences, though. As evidence, *The C++ Programming Language* by C++ creator Bjarne Stroustrup (fourth edition; Addison-Wesley Professional, 2013) weighs in at 1,368 pages, while Kernighan and Ritchie's *The C Programming Language* (second edition; Prentice Hall, 1988) is a scant 274 pages. So, if you're a C programmer, be on the lookout for new or unfamiliar syntax!

The Obligatory "Hello, World" Program

In all its glory, the following code is the simplest C++ program you're likely to encounter:

```
// helloworld.cpp
import <iostream>;

int main()
{
    std::cout << "Hello, World!" << std::endl;
    return 0;
}
```

This code, as you might expect, prints the message "Hello, World!" on the screen. It is a simple program and unlikely to win any awards, but it does exhibit the following important concepts about the format of a C++ program:

- Comments
- Importing modules
- The `main()` function
- I/O streams

These concepts are briefly explained in the following sections (along with header files as an alternative for modules, in the event that your compiler does not support C++20 modules yet).

Comments

The first line of the program is a *comment*, a message that exists for the programmer only and is ignored by the compiler. In C++, there are two ways to delineate a comment. In the preceding and following examples, two slashes indicate that whatever follows on that line is a comment:

```
// helloworld.cpp
```

The same behavior (this is to say, none) would be achieved by using a *multiline comment*. Multiline comments start with `/*` and end with `*/`. The following code shows a multiline comment in action (or, more appropriately, inaction):

```
/* This is a multiline comment.
   The compiler will ignore it.
 */
```

Comments are covered in detail in Chapter 3, "Coding with Style."

C++20

Importing Modules

One of the bigger new features of C++20 is support for *modules*, replacing the old mechanism of so-called *header files*. If you want to use functionality from a module, you need to import that module. This is done with an `import` declaration. The first line of the "Hello, World" application imports the module called `<iostream>`, which declares the input and output mechanisms provided by C++:

```
import <iostream>;
```

If the program did not import that module, it would be unable to perform its only task of outputting text.

Since this is a book about C++20, this book uses modules everywhere. All functionality provided by the C++ Standard Library is provided in well-defined modules. Your own custom types and functionality can also be provided through self-written modules, as you will learn throughout this book. If your compiler does not yet support modules, simply replace `import` declarations with the proper `#include` preprocessor directives, discussed in the next section.

Preprocessor Directives

If your compiler does not yet support C++20 modules, then instead of an `import` declaration such as `import <iostream>;`, you need to write the following preprocessor directive:

```
#include <iostream>
```

In short, building a C++ program is a three-step process. First, the code is run through a *preprocessor*, which recognizes meta-information about the code. Next, the code is *compiled*, or translated into machine-readable object files. Finally, the individual object files are *linked* together into a single application.

Directives aimed at the preprocessor start with the `#` character, as in the line `#include <iostream>` in the previous example. In this case, an `#include` directive tells the preprocessor to take everything from the `<iostream>` header file and make it available to the current file. The `<iostream>` header declares the input and output mechanisms provided by C++.

The most common use of header files is to declare functions that will be defined elsewhere. A function *declaration* tells the compiler how a function is called, declaring the number and types of parameters, and the function return type. A *definition* contains the actual code for the function. Before the introduction of modules in C++20, declarations usually went into *header files*, typically with extension `.h`, while definitions usually went into *source files*, typically with extension `.cpp`. With modules, it is no longer necessary to split declarations from definitions, although, as you will see, it is still possible to do so.

NOTE *In C, the names of the Standard Library header files usually end in* `.h`*, such as* `<stdio.h>`*, and namespaces are not used.*

In C++, the `.h` *suffix is omitted for Standard Library headers, such as* `<iostream>`*, and everything is defined in the* `std` *namespace or a subnamespace of* `std`*.*

The Standard Library headers from C still exist in C++ but in two versions.

- *The recommended versions without a* `.h` *suffix but with a* `c` *prefix. These versions put everything in the* `std` *namespace (for example,* `<cstdio>`*).*
- *The old versions with the* `.h` *suffix. These versions do not use namespaces (for example,* `<stdio.h>`*).*

Note that these C Standard Library headers are not guaranteed to be importable with an `import` *declaration. To be safe, use* `#include <cxyz>` *instead of* `import <cxyz>;`*.*

The following table shows some of the most common preprocessor directives:

PREPROCESSOR DIRECTIVE	FUNCTIONALITY	COMMON USES
`#include [file]`	The specified file is inserted into the code at the location of the directive.	Almost always used to include header files so that code can make use of functionality defined elsewhere.

PREPROCESSOR DIRECTIVE	FUNCTIONALITY	COMMON USES
`#define [id] [value]`	Every occurrence of the specified identifier is replaced with the specified value.	Often used in C to define a constant value or a macro. C++ provides better mechanisms for constants and most types of macros. Macros can be dangerous, so use them cautiously. See Chapter 11,"Odds and Ends," for details.

`#ifdef [id]`
`#endif`
`#ifndef [id]`
`#endif`
	Code within the `ifdef` ("if defined") or `ifndef` ("if not defined") blocks are conditionally included or omitted based on whether the specified identifier has been defined with `#define`.	Used most frequently to protect against circular includes. Each header file starts with an `#ifndef` checking the absence of an identifier, followed by a `#define` directive to define that identifier. The header file ends with an `#endif`. This prevents the file from being included multiple times; see the example after this table.

`#pragma [xyz]`	`xyz` is compiler dependent. Most compilers support a `#pragma` to display a warning or error if the directive is reached during preprocessing.	See the example after this table.

One example of using preprocessor directives is to avoid multiple includes, as shown here:

```
#ifndef MYHEADER_H
#define MYHEADER_H
// ... the contents of this header file
#endif
```

If your compiler supports the `#pragma once` directive, and most modern compilers do, then this can be rewritten as follows:

```
#pragma once
// ... the contents of this header file
```

Chapter 11 discusses this in a bit more detail. But, as mentioned, this book uses C++20 modules instead of old-style header files.

The main() Function

`main()` is, of course, where the program starts. The return type of `main()` is an `int`, indicating the result status of the program. You can omit any explicit return statements in `main()`, in which case zero is returned automatically. The `main()` function either takes no parameters or takes two parameters as follows:

```
int main(int argc, char* argv[])
```

`argc` gives the number of arguments passed to the program, and `argv` contains those arguments. Note that `argv[0]` can be the program name, but it might as well be an empty string, so do not rely on it; instead, use platform-specific functionality to retrieve the program name. The important thing to remember is that the actual arguments start at index 1.

I/O Streams

I/O streams are covered in depth in Chapter 13, "Demystifying C++ I/O," but the basics of output and input are simple. Think of an output stream as a laundry chute for data. Anything you toss into it will be output appropriately. `std::cout` is the chute corresponding to the user console, or *standard out*. There are other chutes, including `std::cerr`, which outputs to the error console. The `<<` operator tosses data down the chute. In the preceding example, a quoted string of text is sent to standard out. Output streams allow multiple types of data to be sent down the stream sequentially on a single line of code. The following code outputs text, followed by a number, followed by more text:

```
std::cout << "There are " << 219 << " ways I love you." << std::endl;
```

Starting with C++20, though, it is recommended to use `std::format()`, defined in `<format>`, to perform string formatting. The `format()` function is discussed in detail in Chapter 2, "Working with Strings and String Views," but in its most basic form it can be used to rewrite the previous statement as follows:

```
std::cout << std::format("There are {} ways I love you.", 219) << std::endl;
```

`std::endl` represents an end-of-line sequence. When the output stream encounters `std::endl`, it will output everything that has been sent down the chute so far and move to the next line. An alternate way of representing the end of a line is by using the `\n` character. The `\n` character is an *escape sequence*, which refers to a new-line character. Escape sequences can be used within any quoted string of text. The following table shows the most common ones:

ESCAPE SEQUENCE	MEANING
`\n`	New line: moves the cursor to the beginning of the next line
`\r`	Carriage return: moves the cursor to the beginning of the current line, but does not advance to the next line
`\t`	Tab
`\\`	Backslash character
`\"`	Quotation mark

> **WARNING** *Keep in mind that* `endl` *inserts a new line into the stream and flushes everything currently in its buffers down the chute. Overusing* `endl`*, for example in a loop, is not recommended because it will have a performance impact. On the other hand, inserting* `\n` *into the stream also inserts a new line but does not automatically flush the buffers.*

Streams can also be used to accept input from the user. The simplest way to do this is to use the `>>` operator with an input stream. The `std::cin` input stream accepts keyboard input from the user. Here is an example:

```
int value;
std::cin >> value;
```

User input can be tricky because you can never know what kind of data the user will enter. See Chapter 13 for a full explanation of how to use input streams.

If you're new to C++ and coming from a C background, you're probably wondering what has been done with the trusty old `printf()` and `scanf()` functions. While these functions can still be used in C++, I strongly recommend using `format()` and the streams library instead, mainly because the `printf()` and `scanf()` family of functions do not provide any type safety.

Namespaces

Namespaces address the problem of naming conflicts between different pieces of code. For example, you might be writing some code that has a function called `foo()`. One day, you decide to start using a third-party library, which also has a `foo()` function. The compiler has no way of knowing which version of `foo()` you are referring to within your code. You can't change the library's function name, and it would be a big pain to change your own.

Namespaces come to the rescue in such scenarios because you can define the context in which names are defined. To place code in a namespace, enclose it within a namespace block. Here's an example:

```
namespace mycode {
    void foo()
    {
        std::cout << "foo() called in the mycode namespace" << std::endl;
    }
}
```

By placing your version of `foo()` in the namespace `mycode`, you are isolating it from the `foo()` function provided by the third-party library. To call the namespace-enabled version of `foo()`, prepend the namespace onto the function name by using `::`, also called the *scope resolution operator*, as follows:

```
mycode::foo();    // Calls the "foo" function in the "mycode" namespace
```

Any code that falls within a `mycode` namespace block can call other code within the same namespace without explicitly prepending the namespace. This implicit namespace is useful in making the code more readable. You can also avoid prepending of namespaces with a `using` *directive*. This directive