DEITEL® DEVELOPER SERIES

Modern C++
Functional-Style Programming
Concepts
Templates
Copy/Move Semantics
Spaceship Operator
Smart Pointers
Text Formatting
Modules
Standard Library
Security

C++20 for Programmers

An Objects-Natural Approach

Executors
Design Patterns
Coroutines
Ranges/Views
Performance
Open Source Libraries
Contracts
jthread
Parallel Algorithms
Lambdas
Concurrency

PAUL DEITEL • HARVEY DEITEL

FREE SAMPLE CHAPTER

C++20 for Programmers

Learning C++20
(and other Popular Programming Languages)
with Deitel on O'Reilly Online Learning

O'Reilly Online Learning

- This subscription service is popular with millions of developers worldwide.
- Many organizations purchase subscriptions for unlimited employee access.
- The site contains 46,000+ e-books and 5,800+ video products.
- If your organization has a subscription, you can access all this content at no charge.
- Subscribers here get early access to new Deitel e-book "Rough Cuts" and LiveLessons video "Sneak Peeks."

All Deitel C++20 publications on O'Reilly Online Learning Are Based on Their Print Book *C++20 for Programmers*

- Approximately 1,000 pages.
- 200+ complete, working programs, each followed by live execution outputs.
- Approximately 15,000 lines of code.
- Line-by-line code walkthroughs.
- Emphasis on Modern C++ idiom, software engineering, performance and security.
- Real-world applications.
- Interact with the authors at `deitel@deitel.com`.

C++20 LiveLessons Fundamentals Video Product

- 50+ hours of video with Paul Deitel teaching the content of *C++20 for Programmers*.
- Access asynchronously on O'Reilly Online Learning at your convenience.
- Learn at your own pace.
- Interact with the authors at `deitel@deitel.com`.

E-Books

- Same content as *C++20 for Programmers* print book.
- Text searchable.
- Available from popular e-book providers, including O'Reilly, Amazon, Informit, VitalSource, Redshelf and more.
- Interact with the authors at `deitel@deitel.com`.

Full-Throttle Live Training Courses

- Paul Deitel teaches fast-paced, full-day, presentation-only courses.
- Ideal for busy developers and programming managers.
- Ask Paul questions during the course and get answers in real time.
- Still have questions? Email Paul after the course at `deitel@deitel.com`.
- Courses offered monthly or bimonthly.
- C++20 Core Language Full Throttle.
- C++20 Standard Libraries Full Throttle.
- Python Full Throttle.
- Python Data Science Full Throttle.
- Java Full Throttle.

College Textbook Versions of *C++20 for Programmers*

- Available as Pearson interactive eTexts and Revels.
- Both formats offer searchable text, video, Checkpoint self-review questions with answers, flashcards and other student learning aids.
- In addition, Revel offers gradable, interactive, programming and non-programming assessment questions.

Deitel & Associates, Inc. also independently offers customized one- to five-day live courses delivered virtually over the Internet. Contact `deitel@deitel.com` for details.

DEITEL® DEVELOPER SERIES

C++20 for Programmers

An Objects-Natural Approach

Paul Deitel • Harvey Deitel

Pearson

Boston • Columbus • New York • San Francisco • Amsterdam • Cape Town
Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City
São Paulo • Sydney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at `corpsales@pearsoned.com` or (800) 382-3419.

For government sales inquiries, please contact `governmentsales@pearsoned.com`.

For questions about sales outside the U.S., please contact `intlcs@pearson.com`.

Visit us on the Web: `https://informit.com`

Library of Congress Control Number: 2021943762

Cover design by Paul Deitel, Harvey Deitel, and Chuti Prasertsith

ISBN-13: 978-0-13-690569-1
ISBN-10: 0-13-690569-2

ScoutAutomatedPrintCode

Pearson's Commitment to Diversity, Equity, and Inclusion

Pearson is dedicated to creating bias-free content that reflects the diversity of all learners. We embrace the many dimensions of diversity, including but not limited to race, ethnicity, gender, socioeconomic status, ability, age, sexual orientation, and religious or political beliefs.

Education is a powerful force for equity and change in our world. It has the potential to deliver opportunities that improve lives and enable economic mobility. As we work with authors to create content for every product and service, we acknowledge our responsibility to demonstrate inclusivity and incorporate diverse scholarship so that everyone can achieve their potential through learning. As the world's leading learning company, we have a duty to help drive change and live up to our purpose to help more people create a better life for themselves and to create a better world.

Our ambition is to purposefully contribute to a world where:

- Everyone has an equitable and lifelong opportunity to succeed through learning.
- Our educational products and services are inclusive and represent the rich diversity of learners.
- Our educational content accurately reflects the histories and experiences of the learners we serve.
- Our educational content prompts deeper discussions with learners and motivates them to expand their own learning (and worldview).

While we work hard to present unbiased content, we want to hear from you about any concerns or needs with this Pearson product so that we can investigate and address them. Please contact us with concerns about any potential bias at

`https://www.pearson.com/report-bias.html`

To the Members of the ISO C++ Standards Committee:
For your efforts in evolving the world's preeminent language for programming high-performance, mission-critical and business-critical applications.

Paul Deitel
Harvey Deitel

Contents

Online Chapters and Appendices

Preface

Welcome to ***C++20 for Programmers: An Objects-Natural Approach***. This book presents leading-edge computing technologies for software developers. It conforms to the C++20 standard (1,834 pages), which the ISO C++ Standards Committee approved in September 2020.[1,2]

The C++ programming language is popular for building high-performance business-critical and mission-critical computing systems—operating systems, real-time systems, embedded systems, game systems, banking systems, air-traffic-control systems, communications systems and more. This book is an introductory- through intermediate-level tutorial presentation of the C++20 version of C++, which is among the world's most popular programming languages,[3] and its associated standard libraries. We present a friendly, contemporary, code-intensive, case-study-oriented introduction to C++20. In this Preface, we explore the "soul of the book."

P.1 Modern C++

We focus on **Modern C++**, which includes the four most recent C++ standards—**C++20**, **C++17**, **C++14** and **C++11**, with a look toward key features anticipated for **C++23** and later. A common theme of this book is to focus on the new and improved ways to code in C++. We employ best practices, emphasizing current professional software-development Modern C++ idioms, and we focus on performance, security and software engineering issues.

Keep It Topical

"*Who dares to teach must never cease to learn.*"[4] (J. C. Dana)

To "take the pulse" of Modern C++, which changes the way developers write C++ programs, we read, browsed or watched approximately 6,000 current articles, research papers, white papers, documentation pieces, blog posts, forum posts and videos.

---

1. The final draft C++ standard is located at: `https://timsong-cpp.github.io/cppwp/n4861/`. This version is free. The published final version (ISO/IEC 14882:2020) may be purchased at `https://www.iso.org/standard/79358.html`.
2. Herb Sutter, "C++20 Approved, C++23 Meetings and Schedule Update," September 6, 2020. Accessed January 11, 2022. `https://herbsutter.com/2020/09/06/c20-approved-c23-meetings-and-schedule-update/`.
3. Tiobe Index for January 2022. Accessed January 7, 2022. `http://www.tiobe.com/tiobe-index`.
4. John Cotton Dana. From `https://www.bartleby.com/73/1799.html`: "In 1912 Dana, a Newark, New Jersey, librarian, was asked to supply a Latin quotation suitable for inscription on a new building at Newark State College (now Kean University), Union, New Jersey. Unable to find an appropriate quotation, Dana composed what became the college motto."—*The New York Times Book Review*, March 5, 1967, p. 55."

C++ Versions

As a developer, you might work on C++ legacy code or projects requiring specific C++ versions. 20 So, we use margin icons like the **“20” icon** shown here to mark each mention of a Modern C++ language feature with the C++ version in which it first appeared. The icons help you see C++ evolving, often from programming with low-level details to easier-to-use, higher-level forms of expression. These trends help reduce development times, and enhance performance, security and system maintainability.

P.2 Target Audiences

C++20 for Programmers: An Objects-Natural Approach has several target audiences:

- C++ software developers who want to learn the latest C++20 features in the context of a full-language, professional-style tutorial,
- non-C++ software developers who are preparing to do a C++ project and want to learn the latest version of C++,
- software developers who learned C++ in college or used it professionally some time ago and want to refresh their C++ knowledge in the context of C++20, and
- professional C++ trainers developing C++20 courses.

P.3 Live-Code Approach and Getting the Code

At the heart of the book is the Deitel signature **live-code approach**. Rather than code snippets, we show C++ as it’s intended to be used in the context of hundreds of complete, working, real-world C++ programs with live outputs.

Read the **Before You Begin** section that follows this Preface to learn how to set up your **Windows**, **macOS** or **Linux** computer to run the 200+ code examples consisting of approximately 15,000 lines of code. All the source code is available free for download at

- `https://github.com/pdeitel/CPlusPlus20ForProgrammers`
- `https://www.deitel.com/books/c-plus-plus-20-for-programmers`
- `https://informit.com/title/9780136905691` (see Section P.8)

For your convenience, we provide the book’s examples in C++ source-code (`.cpp` and `.h`) files for use with integrated development environments and command-line compilers. See **Chapter 1’s Test-Drives** (Section 1.2) for information on compiling and running the code examples with our three preferred compilers. Execute each program in parallel with reading the text to make your learning experience “come alive.” If you encounter a problem, you can reach us at

`deitel@deitel.com`

P.4 Three Industrial-Strength Compilers

We tested the code examples on the latest versions of

- **Visual C++®** in Microsoft® Visual Studio® Community edition on Windows®,

- **Clang C++** (`clang++`) in Apple® Xcode® on macOS®, and in a Docker® container, and
- **GNU® C++** (`g++`) on Linux® and in the GNU Compiler Collection (GCC) Docker® container.

At the time of this writing, most C++20 features are fully implemented by all three 20 compilers, some are implemented by a subset of the three and some are not yet implemented by any. We point out these differences as appropriate and will update our digital content as the compiler vendors implement the remaining C++20 features. We'll also post code updates to the **book's GitHub repository**:

```
https://github.com/pdeitel/CPlusPlus20ForProgrammers
```

and both code and text updates on the book's websites:

```
https://www.deitel.com/books/c-plus-plus-20-for-programmers
https://informit.com/title/9780136905691
```

P.5 Programming Wisdom and Key C++20 Features

Throughout the book, we use margin icons to call your attention to **software-development wisdom** and **C++20 modules** and **concepts** features:

- **Software engineering observations** highlight architectural and design issues for proper software construction, especially for larger systems. SE
- **Security best practices** help you strengthen your programs against attacks. Sec
- **Performance tips** highlight opportunities to make your programs run faster or minimize the amount of memory they occupy. Perf
- **Common programming errors** help reduce the likelihood that you'll make the same mistakes. Err
- **C++ Core Guidelines** recommendations (introduced in Section P.9). CG
- C++20's new **modules** features. Mod
- C++20's new **concepts** features. Concepts

P.6 "Objects-Natural" Learning Approach

In Chapter 9, we'll cover how to develop **custom C++20 classes**, then continue our treatment of object-oriented programming throughout the rest of the book.

What Is Objects Natural?

In the early chapters, you'll work with **preexisting classes that do significant things**. You'll quickly create objects of those classes and get them to "strut their stuff" with a minimal number of simple C++ statements. We call this the **"Objects-Natural Approach."**

Given the massive numbers of free, open-source class libraries created by the C++ community, **you'll be able to perform powerful tasks long before you study how to create your own custom C++ classes in Chapter 9**. This is one of the most compelling aspects of working with object-oriented languages, in general, and with a mature object-oriented language like C++, in particular.

Free Classes

We emphasize using the huge number of valuable free classes available in the C++ ecosystem. These typically come from:

- the C++ Standard Library,
- platform-specific libraries, such as those provided with Microsoft Windows, Apple macOS or various Linux versions,
- free third-party C++ libraries, often created by the open-source community, and
- fellow developers, such as those in your organization.

We encourage you to view lots of free, open-source C++ code examples (available on sites such as GitHub) for inspiration.

The Boost Project

Boost provides 168 open-source C++ libraries.[5] It also serves as a "breeding ground" for new capabilities that are eventually incorporated into the C++ standard libraries. Some that have been added to Modern C++ include multithreading, random-number generation, smart pointers, tuples, regular expressions, file systems and `string_view`s.[6] The following StackOverflow answer lists Modern C++ libraries and language features that evolved from the Boost libraries:[7]

`https://stackoverflow.com/a/8852421`

Objects-Natural Case Studies

Chapter 1 reviews the basic concepts and terminology of object technology. In the early chapters, you'll then create and use objects of preexisting classes long before creating your own custom classes in Chapter 9 and in the remainder of the book. Our **objects-natural case studies** include:

- Section 2.7—**Creating and Using Objects of Standard-Library Class `string`**
- Section 3.12—**Arbitrary-Sized Integers**
- Section 4.13—**Using the `miniz-cpp` Library to Write and Read ZIP files**
- Section 5.20—**Lnfylun Lhqtomh Wjtz Qarcv: Qjwazkrplm xzz Xndmwwqhlz** (this is the encrypted title of our **private-key encryption case study**)
- Section 6.15—**C++ Standard Library Class Template `vector`**
- Section 7.10—**C++20 `span`s: Views of Contiguous Container Elements**
- Section 8.19—**Reading/Analyzing a CSV File Containing Titanic Disaster Data**
- Section 8.20—**Intro to Regular Expressions**
- Section 9.22—**Serializing Objects with JSON (JavaScript Object Notation)**

---

5. "Boost 1.78.0 Library Documentation." Accessed January 9, 2022. `https://www.boost.org/doc/libs/1_78_0/`.
6. "Boost C++ Libraries." Wikipedia. Wikimedia Foundation. Accessed January 9, 2022. `https://en.wikipedia.org/wiki/Boost_(C%2B%2B_libraries)`.
7. Kennytm, Answer to "Which Boost Features Overlap with C++11?" Accessed January 9, 2022. `https://stackoverflow.com/a/8852421`.

A perfect example of the objects-natural approach is using objects of existing classes, like **`array`** and **`vector`** (Chapter 6), without knowing how to write custom classes in general or how those classes are written in particular. Throughout the rest of the book, we use existing C++ standard library capabilities extensively.

P.7 A Tour of the Book

The full-color table of contents graphic inside the front cover shows the book's modular architecture. As you read this Tour of the Book, also refer to that graphic. Together, the graphic and this section will help you quickly "scope out" the book's coverage.

This Tour of the Book points out many of the book's key features. The early chapters establish a solid foundation in C++20 fundamentals. The mid-range to high-end chapters and the case studies ease you into Modern C++20-based software development. Throughout the book, we discuss C++20's programming models:

- procedural programming,
- functional-style programming,
- object-oriented programming,
- generic programming and
- template metaprogramming.

Part 1: Programming Fundamentals Quickstart

Chapter 1, Intro and Test-Driving Popular, Free C++ Compilers: This book is for professional software developers, so Chapter 1

- presents a brief introduction,
- discusses Moore's law, multi-core processors and why standardized concurrent programming is important in Modern C++, and
- provides a brief refresher on object orientation, introducing terminology used throughout the book.

Then we jump right in with **test-drives** demonstrating how to compile and execute C++ code with our three preferred free compilers:

- **Microsoft's Visual C++** in Visual Studio on Windows,
- **Apple's Xcode** on macOS and
- **GNU's `g++`** on Linux.

We tested the book's code examples using each, pointing out the few cases in which a compiler does not support a particular feature. Choose whichever program-development environment(s) you prefer. The book also will work well with other C++20 compilers.

We also demonstrate GNU `g++` in the GNU Compiler Collection Docker container and Clang C++ in a Docker container. This enables you to run the latest GNU `g++` and clang++ command-line compilers on Windows, macOS or Linux. See Section P.13, Docker, for more information on this important developer tool. See the Before You Begin section for installation instructions.

For Windows users, we point to Microsoft's step-by-step instructions that allow you to install Linux in Windows via the Windows Subsystem for Linux (WSL). This is another way to use the `g++` and `clang++` compilers on Windows.

Chapter 2, Intro to C++ Programming, presents C++ fundamentals and illustrates key language features, including input, output, fundamental data types, arithmetic operators and their precedence, and decision making. **Section 2.7's objects-natural case study** demonstrates **creating and using objects of standard-library class `string`**—without you having to know how to develop custom classes in general or how that large complex class is implemented in particular).

Chapter 3, Control Statements: Part 1, focuses on **control statements**. You'll use the `if` and `if…else` selection statements, the `while` iteration statement for counter-controlled and sentinel-controlled iteration, and the increment, decrement and assignment operators. **Section 3.12's objects-natural case study** demonstrates **using a third-party library to create arbitrary-sized integers**.

Chapter 4, Control Statements: Part 2, presents C++'s other **control statements**—`for`, `do…while`, `switch`, `break` and `continue`—and the logical operators. **Section 4.13's objects-natural case study** demonstrates **using the `miniz-cpp` library to write and read ZIP files** programmatically.

Chapter 5, Functions and an Intro to Function Templates, introduces custom functions. We demonstrate **simulation techniques** with **random-number generation**. The random-number generation function `rand` that C++ inherited from C does not have good statistical
Sec properties and can be predictable.[8] This makes programs using `rand` less secure. We include
11 a treatment of C++11's **more secure library of random-number capabilities** that can produce nondeterministic random numbers—a set of random numbers that can't be predicted. Such random-number generators are used in simulations and security scenarios where predictability is undesirable. We also discuss passing information between functions, and recursion. **Section 5.20's objects-natural case study** demonstrates **private-key encryption**.

Part 2: Arrays, Pointers and Strings

Chapter 6, `array`s, `vector`s, Ranges and Functional-Style Programming, begins our early coverage of the C++ standard library's containers, iterators and algorithms. We present the C++ standard library's **`array` container** for representing lists and tables of values. You'll define and initialize `array`s, and access their elements. We discuss passing `array`s to functions, sorting and searching `array`s and manipulating multidimensional `array`s. We begin our introduction to **functional-style programming with lambda expressions**
20 (anonymous functions) and **C++20's Ranges**—one of C++20's "big four" features. **Section 6.15's objects-natural case study** demonstrates the **C++ standard library class template `vector`. This entire chapter is essentially a large objects-natural case study of both `array`s and `vector`s.** The code in this chapter is a good example of Modern C++ coding idioms.

8. Fred Long, "Do Not Use the `rand()` Function for Generating Pseudorandom Numbers." Last modified by Jill Britton on November 20, 2021. Accessed December 27, 2021. `https://wiki.sei.cmu.edu/confluence/display/c/MSC30-C.+Do+not+use+the+rand%28%29+function+for+generating+pseudorandom+numbers`.

Chapter 7, (Downplaying) Pointers in Modern C++, provides thorough coverage of pointers and the intimate relationship among built-in pointers, pointer-based arrays and pointer-based strings (also called C-strings), each of which C++ inherited from the C programming language. Pointers are powerful but challenging to work with and are error-prone. So, we point out Modern C++ features that **eliminate the need for most pointers** and make your code more robust and secure, including **`array`s** and **`vector`s**, **C++20 `span`s** and **C++17 `string_view`s**. We still cover built-in arrays because they remain useful in C++ and so you'll be able to read legacy code. **In new development, you should favor Modern C++ capabilities. Section 7.10's objects-natural case study** demonstrates one such capability—**C++20 `span`s**. These enable you to view and manipulate elements of contiguous containers, such as pointer-based arrays and standard library `array`s and `vector`s, without using pointers directly. This chapter again emphasizes Modern C++ coding idioms. Sec 20 17 20

Chapter 8, `string`s, `string_view`s, Text Files, CSV Files and Regex, presents many of the standard library `string` class's features; shows how to write text to, and read text from, both plain text files and comma-separated values (CSV) files (popular for representing datasets); and introduces string pattern matching with the standard library's regular-expression (regex) capabilities. C++ offers *two* types of strings—**`string`** objects and **C-style pointer-based strings**. We use **`string`** class objects to make programs more robust and **eliminate many of the security problems of C strings. In new development, you should favor `string` objects**. We also present C++17's **`string_view`s**—a lightweight, flexible mechanism for passing any type of string to a function. This chapter presents **two objects-natural case studies:** Sec 17

- **Section 8.19** introduces **data analysis by reading and analyzing a CSV file containing the Titanic Disaster dataset**—a popular dataset for introducing data analytics to beginners.
- **Section 8.20** introduces **regular-expression pattern matching** and **text replacement**.

Part 3: Object-Oriented Programming

Chapter 9, Custom Classes, begins our treatment of **object-oriented programming** as we **craft valuable custom classes**. C++ is extensible—each class you create becomes a new type you can use to create objects. **Section 9.22's objects-natural case study** uses the third-party library **`cereal`** to convert objects into **JavaScript Object Notation (JSON)** format—a process known as **serialization**—and to **recreate those objects from their JSON representation—**known as **deserialization**.

Chapter 10, OOP: Inheritance and Runtime Polymorphism, focuses on the relationships among classes in an inheritance hierarchy and the powerful runtime polymorphic processing capabilities that these relationships enable. An important aspect of this chapter is understanding how polymorphism works. A key feature of this chapter is its detailed diagram and explanation of how C++ typically implements polymorphism, `virtual` functions and dynamic binding "under the hood." You'll see that it uses an elegant pointer-based data structure. We present other mechanisms to achieve runtime polymorphism, including the **non-virtual interface idiom** (NVI) and **`std::variant`/`std::visit`**. We also discuss **programming to an interface, not an implementation**.

Chapter 11, Operator Overloading, Copy/Move Semantics and Smart Pointers, shows how to enable C++'s existing operators to work with custom class objects, and introduces smart pointers and **dynamic memory management**. Smart pointers help you avoid Err dynamic memory management errors by providing additional functionality beyond that of built-in pointers. We discuss **`unique_ptr`** in this chapter and **`shared_ptr`** and **`weak_ptr`** in online Chapter 20. A key aspect of this chapter is crafting valuable classes. We begin with a **`string` class test-drive**, presenting an elegant use of operator overloading before you implement your own customized class with overloaded operators. Then, in one of the book's most important case studies, you'll build your own custom `MyArray` class using overloaded operators and other capabilities to **solve various problems with C++'s native pointer-based arrays.**[9] We introduce and implement the five **special member functions** you can define in each class—the **copy constructor**, **copy assignment operator**, **move constructor**, **move assignment operator** and **destructor**. We discuss **copy semantics** and Perf **move semantics**, which enable a compiler to move resources from one object to another 20 to avoid costly unnecessary copies. We introduce **C++20's three-way comparison operator** (**<=>**; also called the "**spaceship operator**") and show how to implement custom conversion operators. In Chapter 15, you'll convert `MyArray` to a class template that can store elements of a specified type. You will have truly crafted valuable classes.

Chapter 12, Exceptions and a Look Forward to Contracts, continues our **exception-handling** discussion that began in Chapter 6. We discuss when to use exceptions, exception safety guarantees, exceptions in the context of constructors and destructors, handling Err dynamic memory allocation failures and why some projects do not use exception handling. The chapter concludes with an introduction to **contracts**—a potential future C++ feature that we demonstrate via an experimental contracts implementation available on `godbolt.org`. **A goal of contracts is to make most functions `noexcept`—meaning they** Perf **do not throw exceptions—which might enable the compiler to perform additional optimizations and eliminate the overhead and complexity of exception handling**.

Part 4: Standard Library Containers, Iterators and Algorithms

Chapter 13, Standard Library Containers and Iterators, begins our broader and deeper treatment of three key C++ standard library components:

- **containers** (templatized data structures),
- **iterators** (for accessing container elements) and
- **algorithms** (which use iterators to manipulate containers).

We'll discuss **containers**, **container adaptors** and **near containers**. You'll see that the C++ standard library provides commonly used data structures, so you do not need to create your own—the vast majority of your data structures needs can be fulfilled by reusing these standard library capabilities. We demonstrate most standard library containers and introduce how iterators enable algorithms to be applied to various container types. You'll see that different containers support different kinds of iterators. We continue showing how 20 **C++20 Ranges** can simplify your code.

---

9. In industrial-strength systems, you'll use standard library classes for this, but this example enables us to demonstrate many key Modern C++ concepts.

Chapter 14, Standard Library Algorithms and C++20 Ranges & Views, presents many of 20
the standard library's **115 algorithms**, focusing on common container manipulations, including filling containers with values, generating values, comparing elements or entire containers, removing elements, replacing elements, mathematical operations, searching, sorting, swapping, copying, merging, set operations, determining boundaries, and calculating minimums and maximums. We discuss minimum iterator requirements so you can determine which containers can be used with each algorithm. We begin discussing **C++20 Concepts**— 20
another of C++20's "big four" features. The algorithms in **C++20's `std::ranges` namespace** 20
use **C++20 Concepts** to specify their requirements. We continue our discussion of C++'s functional-style programming features with **C++20 Ranges and Views**.

Part 5: Advanced Topics

Chapter 15, Templates, C++20 Concepts and Metaprogramming, discusses **generic programming with templates**, which have been in C++ since the 1998 C++ standard was released. The importance of **Templates** has increased with each new C++ release. **A major Modern C++ theme is to do more at compile-time for better type checking and better run-** Perf
time performance—anything resolved at compile-time avoids runtime overhead and makes systems faster. As you'll see, templates and especially **template metaprogramming** are the keys to powerful **compile-time operations**. In this chapter, we'll take a deeper look at **templates**, showing how to develop custom class templates and exploring C++20 concepts. 20
You'll create your own concepts, convert Chapter 11's `MyArray` case study to a class template with its own **iterators**, and work with **variadic templates** that can receive any number of template arguments. We'll introduce how to work with **C++ metaprogramming**.

Chapter 16, C++20 Modules, presents another of C++20's "big four" features. **Modules** are Mod
a new way to organize your code, precisely control which declarations you expose to client code and encapsulate implementation details. Modules help developers be more productive, especially as they build, maintain and evolve large software systems. Modules help such systems build faster and make them more scalable. C++ creator Bjarne Stroustrup says, "*Mod-* Perf
ules offer a historic opportunity to improve code hygiene and compile times for C++ (bringing C++ into the 21st century)."[10] You'll see that even in small systems, modules offer immediate benefits in every program by eliminating the need for the C++ preprocessor. We would have SE
liked to integrate modules in our programs but, at the time of this writing, our key compilers are still missing various modules capabilities.

Chapter 17, Parallel Algorithms and Concurrency: A High-Level View, is one of the most important chapters in the book, presenting C++'s features for building applications that create and manage **multiple tasks**. This can significantly improve program perfor- Perf
mance and responsiveness. We show how to use **C++17's prepackaged parallel algorithms** 17
to create **multithreaded programs** that will run faster (often much faster) on today's **multi-core computer architectures**. For example, we sort 100 million values using a sequential sort, then a parallel sort. We use C++'s **`<chrono>` library** features to profile the performance improvement we get on today's popular multi-core systems, as we employ an increasing number of cores. You'll see that the parallel sort runs 6.76 times faster than the

10. Bjarne Stroustrup, "Modules and Macros." February 11, 2018. Accessed January 9, 2022. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0955r0.pdf`.

sequential sort on our Windows 10 64-bit computer using an 8-core Intel processor. We discuss the **producer–consumer relationship** and demonstrate various ways to implement
20 it using low-level and high-level C++ concurrency primitives, including C++20's new latch, barrier and semaphore capabilities. We emphasize that concurrent programming is difficult to get right and that you should aim **to use the higher-level concurrency features whenever possible. Lower-level features like semaphores and atomics can be used to implement higher-level features like latches.**

20 **Chapter 18, C++20 Coroutines**, presents **coroutines**—the last of C++20's "big four" features. **A coroutine is a function that can suspend its execution and be resumed later by another part of the program.** The mechanisms supporting this are handled entirely by code that's written for you by the compiler. You'll see that a function containing any of the keywords **`co_await`**, **`co_yield`** or **`co_return`** is a **coroutine** and that **coroutines enable you to do concurrent programming with a simple sequential-like coding style**. Coroutines require sophisticated infrastructure, which you can write yourself, but doing so is
SE complex, tedious and error-prone. Instead, most experts agree that **you should use high-level coroutine support libraries**, which is the approach we demonstrate. The open-source community has created several experimental libraries for developing coroutines quickly
23 and conveniently—we use two in our presentation. C++23 is expected to have standard library support for coroutines.

Appendices

Appendix A, Operator Precedence Chart, lists C++'s operators in highest-to-lowest precedence order.

Appendix B, Character Set, shows characters and their corresponding numeric codes.

P.8 How to Get the Online Chapters and Appendices

We provide several **online chapters and appendices** on `informit.com`. Perform the following steps to register your copy of ***C++20 for Programmers: An Objects-Natural Approach*** on `informit.com` and access this online content:

1. Go to **`https://informit.com/register`** and sign in with an existing account or create a new one.
2. Under **Register a Product**, enter the ISBN **9780136905691**, then click **Submit**.
3. In your account page's **My Registered Products** section, click the **Access Bonus Content** link under *C++20 for Programmers: An Objects-Natural Approach*.

This will take you to the book's online content page.

Online Chapters

20 **Chapter 19, Stream I/O; C++20 Text Formatting: A Deeper Look**, discusses standard C++ input/output capabilities and legacy formatting features of the `<iomanip>` library. We include these formatting features primarily for programmers who might encounter them in **legacy C++ code**. We also present **C++20's new text-formatting features** in more depth.

Chapter 20, Other Topics, presents miscellaneous C++ topics and looks forward to new
23 features expected in C++23 and beyond.

Online Appendices

Appendix C, Number Systems, overviews the binary, octal, decimal and hexadecimal number systems.

Appendix D, Preprocessor, discusses additional features of the C++ preprocessor. Template metaprogramming (Chapter 15) and C++20 Modules (Chapter 16) obviate many of 20
this appendix's features.

Appendix E, Bit Manipulation, discusses bitwise operators for manipulating the individual bits of integral operands and bit fields for compactly representing integer data.

Web-Based Materials on `deitel.com`

Our `deitel.com` web page for the book

`https://deitel.com/c-plus-plus-20-for-programmers`

contains the following additional resources:

- Links to our **GitHub repository** containing the book's downloadable C++ source code
- Blog posts—`https://deitel.com/blog`
- Book updates

For more information about downloading the examples and setting up your C++ development environment, see the **Before You Begin** section.

P.9 C++ Core Guidelines

The **C++ Core Guidelines** (approximately 500 printed pages) CG

`https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines`

are recommendations "to help people use modern C++ effectively."[11] They're edited by Bjarne Stroustrup (C++'s creator) and Herb Sutter (Convener of the ISO C++ Standards Committee). According to the overview:

> "*The guidelines are focused on relatively high-level issues, such as interfaces, resource management, memory management, and concurrency. Such rules affect application architecture and library design. Following the rules will lead to code that is statically type safe, has no resource leaks, and catches many more programming logic errors than is common in code today. And it will run fast—you can afford to do things right.*"[12]

SE
Err
Perf

Throughout this book, we adhere to these guidelines as appropriate. You'll want to pay close attention to their wisdom. We point out many **C++ Core Guidelines** recommendations with a **CG icon**. There are hundreds of core guidelines divided into scores of categories and subcategories. Though this might seem overwhelming, static code analysis tools (Section P.10) can check your code against the guidelines.

11. C++ Core Guidelines, "Abstract." Accessed January 9, 2020. `https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#S-abstract`.
12. C++ Core Guidelines, "Abstract."

Guidelines Support Library

The C++ Core Guidelines often refer to capabilities of the **Guidelines Support Library (GSL)**, which implements helper classes and functions to support various recommendations.[13] Microsoft provides an open-source GSL implementation on GitHub at

`https://github.com/Microsoft/GSL`

We use GSL features in a few examples in the early chapters. Some GSL features have since been incorporated into the C++ standard library.

P.10 Industrial-Strength Static Code Analysis Tools

Static code analysis tools let you quickly check your code for **common errors** and **security problems** and provide insights for code improvement. Using these tools is like having world-class experts checking your code. To help us adhere to the C++ Core Guidelines and improve our code in general, we used the following static-code analyzers:

- **`clang-tidy`**—`https://clang.llvm.org/extra/clang-tidy/`
- **`cppcheck`**—`https://cppcheck.sourceforge.io/`
- **Microsoft's C++ Core Guidelines static code analysis tools**, which are built into Visual Studio's static code analyzer

We used these three tools on the book's code examples to check for

- adherence to the C++ Core Guidelines,
- adherence to coding standards,
- adherence to modern C++ idioms,
- possible security problems,
- common bugs,
- possible performance issues,
- code readability
- and more.

We also used the compiler flag `-Wall` in the GNU `g++` and Clang C++ compilers to enable all compiler warnings. **With a few exceptions for warnings beyond this book's scope, we ensure that our programs compile without warning messages**. See the **Before You Begin** section for static analysis tool configuration information.

P.11 Teaching Approach

C++20 for Programmers: An Objects-Natural Approach contains a rich collection of live-code examples. We stress program clarity and concentrate on building well-engineered software.

13. C++ Core Guidelines, "GSL: Guidelines Support Library." Accessed January 9, 2022. `https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#S-gsl`.

Using Fonts for Emphasis
We place the key terms and the index's page reference for each defining occurrence in **bold text** for easier reference. C++ code uses a `fixed-width font` (e.g., `x = 5`). We place on-screen components in the **bold Helvetica** font (e.g., the **File** menu).

Syntax Coloring
For readability, we syntax color all the code. In our e-books, our syntax-coloring conventions are as follows:

```
comments appear in green
keywords appear in dark blue
constants and literal values appear in light blue
errors appear in red
all other code appears in black
```

Objectives and Outline
Each chapter begins with objectives that tell you what to expect.

Tables and Illustrations
Abundant tables and line drawings are included.

Programming Tips and Key Features
We call out programming tips and key features with icons in margins (see Section P.5).

Index
For convenient reference, we've included an extensive index, with defining occurrences of key terms highlighted with a **bold** page number.

P.12 Developer Resources

StackOverflow
StackOverflow is one of the most popular developer-oriented, question-and-answer sites. Many problems programmers encounter have already been discussed here, so it's a great place to find solutions to those problems and post questions about new ones. Many of our Google searches for various, often complex, issues throughout our writing effort returned StackOverflow answers as their first results.

GitHub

> *"The best way to prepare [to be a programmer] is to write programs, and to study great programs that other people have written. In my case, I went to the garbage cans at the Computer Science Center and fished out listings of their operating systems."*[14]—William Gates

GitHub is an excellent venue for finding free, open-source code to incorporate into your projects—and for you to contribute your code to the open-source community if you like. Fifty million developers use GitHub.[15] The site hosts over 200 million repositories for

---

14. William Gates, quoted in *Programmers at Work: Interviews with 19 Programmers Who Shaped the Computer Industry* by Susan Lammers. Microsoft Press, 1986, p. 83.
15. "GitHub." Accessed January 7, 2022. `https://github.com/`.

code written in an enormous number of programming languages[16]—developers contributed to 61+ million repositories in the last year.[17] **GitHub** is a crucial element of the professional software developer's arsenal with **version-control tools** that help developer teams manage public open-source projects and private projects.

There is a massive C++ open-source community. On GitHub, there are over 41,000[18] C++ code repositories. You can check out other people's C++ code on GitHub and even build upon it if you like. This is a great way to learn and is a natural extension of our live-code teaching approach.[19]

In 2018, Microsoft purchased **GitHub** for $7.5 billion. As a software developer, you're almost certainly using GitHub regularly. According to Microsoft's CEO, Satya Nadella, the company bought GitHub to "*empower every developer to build, innovate and solve the world's most pressing challenges.*"[20]

We encourage you to study and execute lots of developers' open-source C++ code on GitHub and to contribute your own.

P.13 Docker

We use **Docker**—a tool for packaging software into **containers** that bundle everything required to execute that software conveniently and portably across platforms. Some software packages require complicated setup and configuration. For many of these, you can download free preexisting Docker containers, avoiding complex installation issues. You can simply execute software locally on your desktop or notebook computers, making Docker a great way to help you get started with new technologies quickly, conveniently and economically.

We show how to install and execute Docker containers preconfigured with

- the GNU Compiler Collection (GCC), which includes the `g++` compiler, and
- the latest version of Clang's **`clang++`** compiler.

Each can run in **Docker** on **Windows**, **macOS** and **Linux**.

Docker also helps with **reproducibility**. Custom Docker containers can be configured with the software and libraries you use. This would enable others to recreate the environment you used, then reproduce your work, and will help you reproduce your own results. Reproducibility is especially important in the sciences and medicine—for example, when researchers want to prove and extend the work in published articles.

P.14 Some Key C++ Documentation and Resources

The book includes over 900 citations to videos, blog posts, articles and online documentation we studied while writing the manuscript. You may want to access some of these resources to investigate more advanced features and idioms. The website **`cppreference.com`** has become the defacto C++ documentation site. We reference it frequently so

16. "Where the World Builds Software." Accessed January 7, 2022. `https://github.com/about`.
17. "The 2021 State of the Octoverse." Accessed January 7, 2022. `https://octoverse.github.com`.
18. "C++." Accessed January 7, 2022. `https://github.com/topics/cpp`.
19. Students will need to become familiar with the variety of open-source licenses for software on GitHub.
20. "Microsoft to Acquire GitHub for $7.5 Billion." Accessed January 7, 2022. `https://news.microsoft.com/2018/06/04/microsoft-to-acquire-github-for-7-5-billion/`.

you can get more details about the standard C++ classes and functions we use throughout the book. We also frequently reference the final draft of the **C++20 standard document**, which is available for free on GitHub at

`https://timsong-cpp.github.io/cppwp/n4861/`

You may also find the following C++ resources helpful as you work through the book.

Documentation

- C++20 standard document final draft adopted by the C++ Standard Committee: 20
  `https://timsong-cpp.github.io/cppwp/n4861/`
- C++ Reference at `cppreference.com`:
  `https://cppreference.com/`
- Microsoft's C++ language documentation:
  `https://docs.microsoft.com/en-us/cpp/cpp/`
- The GNU C++ Standard Library Reference Manual:
  `https://gcc.gnu.org/onlinedocs/libstdc++/manual/index.html`

Blogs

- Sutter's Mill Blog—Herb Sutter on software development:
  `https://herbsutter.com/`
- Microsoft's C++ Team Blog:
  `https://devblogs.microsoft.com/cppblog`
- Marius Bancila's Blog:
  `https://mariusbancila.ro/blog/`
- Jonathan Boccara's Blog:
  `https://www.fluentcpp.com/`
- Bartlomiej Filipek's Blog:
  `https://www.cppstories.com/`
- Rainer Grimm's Blog:
  `http://modernescpp.com/`
- Arthur O'Dwyer's Blog:
  `https://quuxplusone.github.io/blog/`

Additional Resources

- Bjarne Stroustrup's website:
  `https://stroustrup.com/`
- Standard C++ Foundation website:
  `https://isocpp.org/`
- C++ Standard Committee website:
  `http://www.open-std.org/jtc1/sc22/wg21/`

P.15 Getting Your Questions Answered

Popular C++ and general programming online forums include

- `https://stackoverflow.com`
- `https://www.reddit.com/r/cpp/`
- `https://groups.google.com/g/comp.lang.c++`
- `https://www.dreamincode.net/forums/forum/15-c-and-c/`

For a list of other valuable sites, see

`https://www.geeksforgeeks.org/stuck-in-programming-get-the-solution-from-these-10-best-websites/`

Sec

Also, vendors often provide forums for their tools and libraries. Many libraries are managed and maintained at `github.com`. Some library maintainers provide support through the **Issues** tab on a given library's GitHub page.

Communicating with the Authors

As you read the book, if you have questions, we're easy to reach at

`deitel@deitel.com`

We'll respond promptly.

P.16 Join the Deitel & Associates, Inc. Social Media Communities

Join the Deitel social media communities on

- LinkedIn®—`https://bit.ly/DeitelLinkedIn`
- YouTube®—`https://youtube.com/DeitelTV`
- Twitter®—`https://twitter.com/deitel`
- Facebook®—`https://facebook.com/DeitelFan`

P.17 Deitel Pearson Products on O'Reilly Online Learning

If you're at a company or college, your organization might have an **O'Reilly Online Learning** subscription, giving you free access to all of Deitel's Pearson e-books and LiveLessons videos hosted on the site, as well as Paul Deitel's live, one-day Full Throttle training courses, offered on a continuing basis. Individuals may sign up for a **10-day free trial** at

`https://learning.oreilly.com/register/`

For a list of all our current products and courses on O'Reilly Online Learning, visit

`https://deitel.com/LearnWithDeitel`

Textbooks and Professional Books

Each Deitel e-book on O'Reilly Online Learning is presented in full color, extensively indexed and text searchable. As we write our professional books, they're posted on

O'Reilly Online Learning for early "rough cut" access, then replaced with the book's final content once published. The final e-book for ***C++20 for Programmers: An Objects-Natu-*** 20
ral Approach is available to O'Reilly subscribers at

```
https://learning.oreilly.com/library/view/c-20-for-programmers/
   9780136905776
```

Asynchronous LiveLessons Video Products

Learn hands-on with Paul Deitel as he presents compelling, leading-edge computing technologies in C++, Java, Python and Python Data Science/AI (and more coming). Access to our ***C++20 Fundamentals LiveLessons*** videos is available to O'Reilly subscribers at

```
https://learning.oreilly.com/videos/c-20-fundamentals-parts/
   9780136875185
```

These videos are ideal for self-paced learning. At the time of this writing, we're still recording this product. Additional videos will be posted as they become available during Q1 and Q2 of 2022. The final video product will contain 50–60 hours of video—approximately the equivalent of two college semester courses.

Live Full-Throttle Training Courses

Paul Deitel's live **Full-Throttle training courses** at O'Reilly Online Learning

```
https://deitel.com/LearnWithDeitel
```

are one-full-day, presentation-only, fast-paced, code-intensive introductions to Python,
Python Data Science/AI, Java, C++20 Fundamentals and the C++20 Standard Library. 20
These courses are for experienced developers and software project managers preparing for projects using other languages. After taking a Full-Throttle course, participants often watch the corresponding *LiveLessons* video course, which has many more hours of classroom-paced learning.

P.18 Live Instructor-Led Training with Paul Deitel

Paul Deitel has been teaching programming languages to developer audiences for three decades. He presents a variety of one- to five-day C++, Python and Java corporate training courses, and teaches Python with an Introduction to Data Science for the UCLA Anderson School of Management's Master of Science in Business Analytics (MSBA) program. His courses can be delivered worldwide on-site or virtually. Please contact **`deitel@deitel.com`** for a proposal customized to meet your company's or academic program's needs.

P.19 College Textbook Version of *C++20 for Programmers*

Our college textbook, ***C++ How to Program, Eleventh Edition***, will be available in three digital formats:

- **Online e-book** offered through popular e-book providers.
- Interactive **Pearson eText** (see below).
- Interactive **Pearson Revel** with assessment (see below).

All of these textbook versions include standard **"How to Program" features** such as:

- A chapter introducing hardware, software and Internet concepts.

- An introduction to programming for novices.
- End-of-section programming and non-programming **Checkpoint self-review exercises with answers**.
- **End-of-chapter exercises**.

Deitel Pearson eTexts and Revels include:

- **Videos** in which Paul Deitel discusses the material in the book's core chapters.
- Interactive programming and non-programming **Checkpoint self-review exercises with answers**.
- **Flashcards** and other learning tools.

In addition, **Pearson Revels** include interactive programming and non-programming automatically graded exercises, as well as instructor course-management tools, such as a grade book.

Supplements available to qualified college instructors teaching from the textbook include:

- **Instructor solutions manual** with solutions to most of the end-of-chapter exercises.
- **Test-item file** with four-part, code-based and non-code-based multiple-choice questions with answers.
- Customizable **PowerPoint lecture slides**.

Please write to **`deitel@deitel.com`** for more information.

P.20 Acknowledgments

We'd like to thank Barbara Deitel for long hours devoted to Internet research on this project. We're fortunate to have worked with the dedicated team of publishing professionals at Pearson. We appreciate the efforts and 27-year mentorship of our friend and colleague Mark L. Taub, Vice President of the Pearson IT Professional Group. Mark and his team publish our professional books and LiveLessons video products, and sponsor our live online training seminars, offered through the O'Reilly Online Learning service:

`https://learning.oreilly.com/`

Charvi Arora recruited the book's reviewers and managed the review process. Julie Nahil managed the book's production. Chuti Prasertsith designed the cover.

Reviewers

We were fortunate on this project to have 10 distinguished professionals review the manuscript. Most of the reviewers are either on the ISO C++ Standards Committee, have served on it or have a working relationship with it. Many have contributed features to the language. They helped us make a better book—any remaining flaws are our own.

- Andreas Fertig, Independent C++ Trainer and Consultant, Creator of `cppinsights.io`, Author of *Programming with C++20* 20
- Marc Gregoire, Software Architect, Nikon Metrology, Microsoft Visual C++ MVP and author of *Professional C++, 5/e* (which is up-to-date with C++20) 20

- Dr. Daisy Hollman, ISO C++ Standards Committee Member
- Danny Kalev, Ph.D. and Certified System Analyst and Software Engineer, Former ISO C++ Standards Committee Member
- Dietmar Kühl, Senior Software Developer, Bloomberg L.P., ISO C++ Standard Committee Member
- Inbal Levi, SolarEdge Technologies, ISO C++ Foundation director, ISO C++ SG9 (Ranges) chair, ISO C++ Standards Committee member
- Arthur O'Dwyer, C++ trainer, Chair of CppCon's Back to Basics track, author of 17 20
several accepted C++17/20/23 proposals and the book *Mastering the C++17 STL* 23
- Saar Raz, Senior Software Engineer, Swimm.io and Implementor of C++20 Con- 20
cepts in Clang
- José Antonio González Seco, Parliament of Andalusia
- Anthony Williams, Member of the British Standards Institution C++ Standards Panel, Director of Just Software Solutions Ltd., Author of C++ *Concurrency in Action, 2/e* (Anthony is the author or co-author of many C++ Standard Committee papers that led to C++'s standardized concurrency features)

Arthur O'Dwyer

We'd like to call out the extraordinary efforts Arthur O'Dwyer put into reviewing our manuscript. While working through his comments, we learned a great deal about C++'s subtleties and especially Modern C++ coding idioms. In addition to carefully marking each chapter PDF we sent him, Arthur provided a separate comprehensive document explaining his comments in detail, often rewriting code and providing external resources that offered additional insights. As we applied all the reviewers' comments, we always looked forward to what Arthur had to say, especially regarding the more challenging issues. He's a busy professional, yet he was generous with his time and always constructive. He insisted that we "get it right" and worked hard to help us do that. Arthur teaches C++ to professionals. He taught us a much about how to do C++ right.

GitHub

Thanks to GitHub for making it easy for us to share our code and keep it up-to-date, and for providing the tools that enable 73+ million developers to contribute to 200 million+ code repositories.[21] These tools support the massive open-source communities that provide libraries for today's popular programming languages, making it easier for developers to create powerful applications and avoid "reinventing the wheel."

Matt Godbolt and Compiler Explorer

Thanks to Matt Godbolt, creator of **Compiler Explorer** at `https://godbolt.org`, which enables you to compile and run programs in many programming languages. Through this site, you can test your code

- on most popular C++ compilers—including our three preferred compilers—and
- across many released, developmental and experimental compiler versions.

21. "Where the World Builds Software." Accessed January 7, 2022. `https://github.com/about`.

For example, we used an experimental `g++` compiler version to demonstrate **contracts** (Chapter 12, Exceptions and a Look Forward to Contracts), which we hope to see standardized in a future C++ language version. Several of our reviewers used `godbolt.org` to demonstrate suggested changes to us, helping us improve the book.

Dietmar Kühl
We would like to thank Dietmar Kühl, Senior Software Developer at Bloomberg L.P. and an ISO C++ Committee member, for sharing with us his views on inheritance and static and dynamic polymorphism. His insights helped us shape our presentations of these topics in Chapters 10 and 15.

Rainer Grimm
Our thanks to Rainer Grimm (`http://modernescpp.com/`), among the Modern C++ community's most prolific bloggers. As we got deeper into C++20, our Google searches frequently pointed us to his writings. Rainer Grimm is a professional C++ trainer who offers courses in German and English. He is the author of several C++ books, including *C++20: Get the Details*, *Concurrency with Modern C++*, *The C++ Standard Library, 3/e* and *C++ Core Guidelines Explained*. He is already blogging about features likely to appear in C++23.

Brian Goetz
We were privileged to have as a reviewer on one of our other books—*Java How to Program, 10/e*—Brian Goetz, Oracle Java Language Architect and co-author of *Java Concurrency in Practice*. He provided us with many insights and constructive comments, especially on

- inheritance hierarchy design, which influenced our design decisions for several examples in **Chapter 10, OOP: Inheritance and Runtime Polymorphism**, and
- Java concurrency, which influenced our approach to C++20 concurrency in **Chapter 17, Parallel Algorithms and Concurrency: A High-Level View**.

Open-Source Contributors and Bloggers
A special note of thanks to the technically oriented people worldwide who contribute to the open-source movement and blog about their work online, and to their organizations that encourage the proliferation of such open software and information.

Google Search
Thanks to Google, whose search engine answers our constant stream of queries, each in a fraction of a second, at any time day or night—and at no charge. It's the single best productivity enhancement tool we've added to our research process in the last 20 years.

Grammarly
We now use the paid version of **Grammarly** on all our manuscripts. They describe their tools as helping you "compose bold, clear, mistake-free writing" with their "AI-powered writing assistant."[22] They also say, "Using a variety of innovative approaches—including advanced machine learning and deep learning—we consistently break new ground in nat-

22. "Grammarly." Accessed January 15, 2022. `https://www.grammarly.com`.

ural language processing (NLP) research to deliver unrivaled assistance."[23] Grammarly provides free tools that you can integrate into several popular web browsers, Microsoft® Office 365™ and Google Docs™. They also offer more powerful premium and business tools. You can view their free and paid plans at

```
https://www.grammarly.com/plans
```

As you read the book and work through the code examples, we'd appreciate your comments, criticisms, corrections and suggestions for improvement. Please send all correspondence, including questions, to

```
deitel@deitel.com
```

We'll respond promptly.

Welcome to the exciting world of C++20 programming. We've enjoyed writing 11 20
editions of our academic and professional C++ content over the last 30 years. We hope you have an informative, challenging and entertaining learning experience with ***C++20 for Programmers: An Objects-Natural Approach*** and enjoy this look at leading-edge, Modern C++ software development.

Paul Deitel
Harvey Deitel

About the Authors

Paul J. Deitel, CEO and Chief Technical Officer of Deitel & Associates, Inc., is an MIT graduate with 42 years in computing. Paul is one of the world's most experienced programming-languages trainers, having taught professional courses to software developers since 1992. He has delivered hundreds of programming courses to academic, industry, government and military clients of Deitel & Associates, Inc. internationally, including UCLA, Cisco, IBM, Siemens, Sun Microsystems (now Oracle), Dell, Fidelity, NASA at the Kennedy Space Center, the National Severe Storm Laboratory, White Sands Missile Range, Rogue Wave Software, Boeing, Puma, iRobot and many more. He and his co-author, Dr. Harvey M. Deitel, are among the world's best-selling programming-language textbook, professional book, video and interactive multimedia e-learning authors, and virtual- and live-training presenters.

Dr. Harvey M. Deitel, Chairman and Chief Strategy Officer of Deitel & Associates, Inc., has 61 years of experience in computing. Dr. Deitel earned B.S. and M.S. degrees in Electrical Engineering from MIT and a Ph.D. in Mathematics from Boston University—he studied computing in each of these programs before they spun off Computer Science departments. He has extensive industry and college teaching experience, including earning tenure and serving as the Chairman of the Computer Science Department at Boston College before founding Deitel & Associates in 1991 with his son, Paul. The Deitels' publications have earned international recognition, with more than 100 translations published in Japanese, German, Russian, Spanish, French, Polish, Italian, Simplified Chinese, Traditional Chinese, Korean, Portuguese, Greek, Urdu and Turkish. Dr. Deitel has delivered hundreds of programming courses to academic, corporate, government and military clients.

23. "Our Mission." Accessed January 15, 2022. `https://www.grammarly.com/about`.

About Deitel® & Associates, Inc.

Deitel & Associates, Inc., founded by Paul Deitel and Harvey Deitel, is an internationally recognized authoring and corporate-training organization, specializing in computer programming languages, object technology, mobile app development and Internet and web software technology. The company's training clients include some of the world's largest companies, government agencies, branches of the military, and academic institutions. The company offers instructor-led training courses delivered virtually and live at client sites worldwide, and virtually for Pearson Education on O'Reilly Online Learning (`https://learning.oreilly.com`), formerly called Safari Books Online.

Through its 47-year publishing partnership with Pearson, Deitel & Associates, Inc., publishes leading-edge programming professional books and college textbooks in print and e-book formats, LiveLessons video courses, O'Reilly Online Learning live training courses and Revel™ interactive multimedia college courses.

To contact Deitel & Associates, Inc. and the authors, or to request a proposal for virtual or on-site, instructor-led training worldwide, write to

```
deitel@deitel.com
```

To learn more about Deitel virtual and on-site corporate training, visit

```
https://deitel.com/training
```

Individuals wishing to purchase Deitel books can do so at

```
https://amazon.com
https://www.barnesandnoble.com/
```

Bulk orders by corporations, the government, the military and academic institutions should be placed directly with Pearson. For corporate and government sales, send an email to

```
corpsales@pearsoned.com
```

Deitel e-books are available in various formats from

```
https://www.amazon.com/              https://www.vitalsource.com/
https://www.barnesandnoble.com/      https://www.redshelf.com/
https://www.informit.com/            https://www.chegg.com/
```

To register for a free 10-day trial to O'Reilly Online Learning, visit

```
https://learning.oreilly.com/register/
```

Before You Begin

Before using this book, please read this section to understand our conventions and set up your computer to compile and run our example programs.

Font and Naming Conventions

We use fonts to distinguish application elements and C++ code elements from regular text:

- We use a **sans-serif bold font** for on-screen application elements, as in "the **File** menu."
- We use a `sans-serif font` for C++ code elements, as in `sqrt(9)`.

Obtaining the Code Examples

We maintain the code examples for *C++20 for Programmers* in a GitHub repository. The **Source Code** section of the book's webpage at

```
https://deitel.com/cpp20fp
```

includes a link to the GitHub repository and a link to a ZIP file containing the code. If you're familiar with Git and GitHub, clone the repository to your system. If you download the ZIP file, be sure to extract its contents. In our instructions, we assume the examples reside in your user account's `Documents` folder in a subfolder named `examples`.

If you're not familiar with Git and GitHub but are interested in learning about these essential developer tools, check out their guides at

```
https://guides.github.com/activities/hello-world/
```

Compilers We Use in *C++20 for Programmers*

Before reading this book, ensure that you have a recent C++ compiler installed. We tested the code examples in *C++20 for Programmers* using the following free compilers:

- For Microsoft Windows, we used Microsoft Visual Studio Community edition, which includes the Visual C++ compiler and other Microsoft development tools.[1]
- For macOS, we used the Apple Xcode[2] C++ compiler, which uses a version of the Clang C++ compiler.
- For Linux, we used the GNU C++ compiler[3]—part of the GNU Compiler Collection (GCC). GNU C++ is already installed on most Linux systems (though

1. Visual Studio 2022 Community at the time of this writing.
2. Xcode 13.2.1 at the time of this writing.
3. GNU `g++` 11.2 at the time of this writing.

you might need to update the compiler to a more recent version) and can be installed on macOS and Windows systems.

- You also can run the latest versions of GNU C++ and Clang C++ conveniently on Windows, macOS and Linux via Docker containers. See the “Docker and Docker Containers” section later in this Before You Begin section.

This Before You Begin describes installing the compilers and Docker. Section 1.2’s test-drives demonstrate how to compile and run C++ programs using these compilers.

Some Examples Do Not Compile and Run on All Three Compilers

At the time of this writing (February 2022), the compiler vendors had not yet fully implemented some of C++20’s new features. As those features become available, we’ll retest the code, update our digital products and post updates for our print products at

```
https://deitel.com/cpp20fp
```

Installing Visual Studio Community Edition on Windows

If you are a Windows user, first ensure that your system meets the requirements for Microsoft Visual Studio Community edition at

```
https://docs.microsoft.com/en-us/visualstudio/releases/2022/system-
  requirements
```

Next, go to

```
https://visualstudio.microsoft.com/downloads/
```

Then perform the following installation steps:

1. Click **Free Download** under **Community**.
2. Depending on your web browser, you may see a pop-up at the bottom of your screen in which you can click **Run** to start the installation process. If not, double-click the installer file in your **Downloads** folder.
3. In the **User Account Control** dialog, click **Yes** to allow the installer to make changes to your system.
4. In the **Visual Studio Installer** dialog, click **Continue** to allow the installer to download the components it needs for you to configure your installation.
5. For this book’s examples, select the option **Desktop Development with C++**, which includes the Visual C++ compiler and the C++ standard libraries.
6. Click **Install**. Depending on your Internet connection speed, the installation process can take a significant amount of time.

Installing Xcode on macOS

On macOS, perform the following steps to install Xcode:

1. Click the Apple menu and select **App Store...**, or click the **App Store** icon in the dock at the bottom of your Mac screen.
2. In the **App Store**’s **Search** field, type **Xcode**.
3. Click the **Get** button to install Xcode.

Installing the Most Recent GNU C++ Version

There are many Linux distributions, and they often use different software upgrade techniques. Check your distribution's online documentation for the proper way to upgrade GNU C++ to the latest version. You also can download GNU C++ for various platforms at

```
https://gcc.gnu.org/install/binaries.html
```

Installing the GNU Compiler Collection in Ubuntu Linux Running on the Windows Subsystem for Linux

You can install the GNU Compiler Collection on Windows via the Windows Subsystem for Linux (WSL), which enables you to run Linux in Windows. Ubuntu Linux provides an easy-to-use installer in the Windows Store, but first you must install WSL:

1. In the search box on your taskbar, type "Turn Windows features on or off," then click **Open** in the search results.
2. In the Windows Features dialog, locate **Windows Subsystem for Linux** and ensure that it is checked. If it is, WSL is already installed. Otherwise, check it and click **OK**. Windows will install WSL and ask you to reboot your system.
3. Once the system reboots and you log in, open the **Microsoft Store** app and search for **Ubuntu**, select the app named **Ubuntu** and click **Install**. This installs the latest version of Ubuntu Linux.
4. Once installed, click the **Launch** button to display the Ubuntu Linux command-line window, which will continue the installation process. You'll be asked to create a username and password for your Ubuntu installation—these do not need to match your Windows username and password.
5. When the Ubuntu installation completes, execute the following two commands to install the GCC and the GNU debugger—you may be asked enter your password for the account you created in Step 4:

   ```
   sudo apt-get update
   sudo apt-get install build-essential gdb
   ```

6. Confirm that `g++` is installed by executing the following command:

   ```
   g++ --version
   ```

To access our code files, use the `cd` command change the folder within Ubuntu to:

`cd /mnt/c/Users/`*YourUserName*`/Documents/examples`

Use your own username and update the path to where you placed our examples on your system.

Docker and Docker Containers

Docker is a tool for packaging software into containers (also called images) that bundle *everything* required to execute that software across platforms, which is particularly useful for software packages with complicated setups and configurations. For many such packages, there are free preexisting Docker containers (often at `https://hub.docker.com`) that you can download and execute locally on your system. Docker is a great way to get started with new technologies quickly and conveniently. It is also a great way to experiment with new compiler versions.

Installing Docker

To use a Docker container, you must first install Docker. Windows and macOS users should download and run the **Docker Desktop** installer from

```
https://www.docker.com/get-started
```

Then follow the on-screen instructions. Also, sign up for a **Docker Hub** account on this webpage so you can take advantage of containers from `https://hub.docker.com`. Linux users should install **Docker Engine** from

```
https://docs.docker.com/engine/install/
```

Downloading the GNU Compiler Collection Docker Container

The GNU team maintains official Docker containers at

```
https://hub.docker.com/_/gcc
```

Once Docker is installed and running, open a Command Prompt[4] (Windows), Terminal (macOS/Linux) or shell (Linux), then execute the command

```
docker pull gcc:latest
```

Docker downloads the GNU Compiler Collection (GCC) container's most current version (at the time of this writing, version 11.2). In one of Section 1.2's test-drives, we'll demonstrate how to execute the container and use it to compile and run C++ programs.

Downloading the GNU Compiler Collection Docker Container

Currently, the Clang team does not provide an official Docker container, but many working containers are available on `https://hub.docker.com`. For this book we used a popular one from

```
https://hub.docker.com/r/teeks99/clang-ubuntu
```

Open a Command Prompt (Windows), Terminal (macOS/Linux) or shell (Linux), then execute the command

```
docker pull teeks99/clang-ubuntu:latest
```

Docker downloads the Clang container's most current version (at the time of this writing, version 13). In one of Section 1.2's test-drives, we'll demonstrate how to execute the container and use it to compile and run C++ programs.

Getting Your C++ Questions Answered

As you read the book, if you have questions, we're easy to reach at

```
deitel@deitel.com
```

and

```
https://deitel.com/contact-us
```

We'll respond promptly.

---

4. Windows users should choose **Run as administrator** when opening the Command Prompt.

The web is loaded with programming information. An invaluable resource for nonprogrammers and programmers alike is the website

```
https://stackoverflow.com
```

on which you can

- search for answers to most common programming questions,
- search for error messages to see what causes them,
- ask programming questions to get answers from programmers worldwide and
- gain valuable insights about programming in general.

For live C++ discussions, check out the Slack channel **cpplang**:

```
https://cpplang-inviter.cppalliance.org
```

and the Discord server **#include<C++>**:

```
https://www.includecpp.org/discord/
```

Online C++ Documentation

For documentation on the C++ standard library, visit

```
https://cppreference.com
```

Also, be sure to check out the C++ FAQ at

```
https://isocpp.org/faq
```

A Note Regarding the {fmt} Text-Formatting Library

Throughout the book many programs include the following line of code:

```
#include <fmt/format.h>
```

which enables our programs to use the open-source `{fmt}` library's text-formatting features.[5] Those programs include calls to the function `fmt::format`.

C++20's new text-formatting capabilities are a subset of the `{fmt}` library's features. In C++20, the preceding line of code should be

```
#include <format>
```

and the corresponding function calls should use the `std::format` function.

At the time of this writing, only Microsoft Visual C++ supported C++20's new text-formatting capabilities. For this reason, our examples use the open-source `{fmt}` library to ensure most of the examples will execute on all of our preferred compilers.

Static Code Analysis Tools

We used the following static code analyzers to check our code examples for adherence to the C++ Core Guidelines, adherence to coding standards, adherence to Modern C++ idioms, possible security problems, common bugs, possible performance issues, code readability and more:

---

5. "`{fmt}`." Accessed February 15, 2022. `https://github.com/fmtlib/fmt`.

- **`clang-tidy`**—`https://clang.llvm.org/extra/clang-tidy/`
- **`cppcheck`**—`https://cppcheck.sourceforge.io/`
- **Microsoft's C++ Core Guidelines static code analysis tools**, which are built into Visual Studio's static code analyzer

You can install `clang-tidy` on Linux with the following commands:

```
sudo apt-get update -y
sudo apt-get install -y clang-tidy
```

You can install `cppcheck` for various operating-system platforms by following the instructions at `https://cppcheck.sourceforge.io/`. For Visual C++, once you learn how to create a project in Section 1.2's test-drives, you can configure Microsoft's C++ Core Guidelines static code analysis tools as follows:

1. Right-click your project name in the **Solution Explorer** and select **Properties**.
2. In the dialog that appears, select **Code Analysis > General** in the left column, then set **Enable Code Analysis on Build** to **Yes** in the right column.
3. Next, select **Code Analysis > Microsoft** in the left column. Then, in the right column you can select a specific subset of the analysis rules in the drop-down list. We used the option **<Choose multiple rule sets...>** to select all the rule sets that begin with **C++ Core Check**. Click **Save As...**, give your custom rule set a name, click **Save**, then click **Apply**. (Note that this will produce large numbers of warnings for the `{fmt}` text-formatting library that we use in the book's examples.)

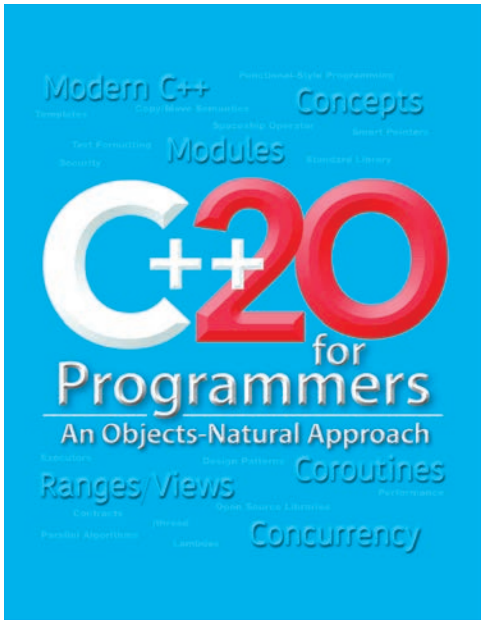
4

Control Statements, Part 2

Objectives

In this chapter, you'll:

- Use the `for` and `do`...`while` iteration statements.
- Perform multiple selection using the `switch` selection statement.
- Use C++17's `[[fallthrough]]` attribute in `switch` statements.
- Use C++17's selection statements with initializers.
- Use the `break` and `continue` statements to alter the flow of control.
- Use the logical operators to form compound conditions in control statements.
- Understand the representational errors associated with using floating-point data to hold monetary values.
- Continue our Objects-Natural approach with a case study that uses an open-source ZIP compression/decompression library to create and read ZIP files.
- Use more C++20 text-formatting capabilities.



Outline

4.1 Introduction

This chapter introduces the `for`, `do…while`, `switch`, `break` and `continue` control statements. We explore the essentials of counter-controlled iteration. We use compound-interest calculations to begin investigating the issues of processing monetary amounts. First, we discuss the representational errors associated with floating-point types. We use a `switch` statement to count the number of A, B, C, D and F grade equivalents in a set of numeric
17 grades. We show C++17's enhancements that allow you to initialize one or more variables of the same type in the headers of `if` and `switch` statements. We discuss the logical operators, which enable you to combine simple conditions to form compound conditions. In our Objects-Natural case study, we continue using objects of preexisting classes with the miniz-cpp open-source library for creating and reading compressed ZIP archive files.
20 Finally, we introduce more of C++20's powerful and expressive text-formatting features.

4.2 Essentials of Counter-Controlled Iteration

This section uses the `while` iteration statement introduced in Chapter 3 to formalize the elements of counter-controlled iteration:

1. a **control variable** (or loop counter)
2. the control variable's **initial value**
3. the control variable's **increment** that's applied during each iteration of the loop
4. the **loop-continuation condition** that determines if looping should continue.

Consider Fig. 4.1, which uses a loop to display the numbers from 1 through 10.

```
// fig04_01.cpp
// Counter-controlled iteration with the while iteration statement.
#include <iostream>
using namespace std;
```

Fig. 4.1 | Counter-controlled iteration with the `while` iteration statement. (Part 1 of 2.)

```
5
6   int main() {
7      int counter{1}; // declare and initialize control variable
8
9      while (counter <= 10) { // loop-continuation condition
10        cout << counter << "  ";
11        ++counter; // increment control variable
12     }
13
14     cout << "\n";
15  }
```

```
1  2  3  4  5  6  7  8  9  10
```

Fig. 4.1 | Counter-controlled iteration with the `while` iteration statement. (Part 2 of 2.)

In Fig. 4.1, lines 7, 9 and 11 define the elements of counter-controlled iteration. Line 7 declares the control variable (`counter`) as an `int`, reserves space for it in memory and sets its initial value to `1`. Declarations that require initialization are executable statements. Variable declarations that also reserve memory are **definitions**. We'll generally use the term "declaration," except when the distinction is important.

Line 10 displays `counter`'s value once per iteration of the loop. Line 11 increments the control variable by 1 for each iteration of the loop. The `while`'s loop-continuation condition (line 9) tests whether the value of the control variable is less than or equal to `10` (the final value for which the condition is true). The loop terminates when the control variable exceeds `10`.

Floating-point values are approximate, so controlling counting loops with floating-point variables can result in imprecise counter values and inaccurate termination tests, which can prevent a loop from terminating. For that reason, always control counting loops with integer variables.

4.3 for Iteration Statement

The **`for` iteration statement** specifies the counter-controlled-iteration details in a single line of code. Figure 4.2 reimplements the application of Fig. 4.1 using a `for` statement.

```
// fig04_02.cpp
// Counter-controlled iteration with the for iteration statement.
#include <iostream>
using namespace std;

int main() {
   // for statement header includes initialization,
   // loop-continuation condition and increment
   for (int counter{1}; counter <= 10; ++counter) {
      cout << counter << "  ";
   }

```

Fig. 4.2 | Counter-controlled iteration with the `for` iteration statement. (Part 1 of 2.)

```
13      cout << "\n";
14   }
```

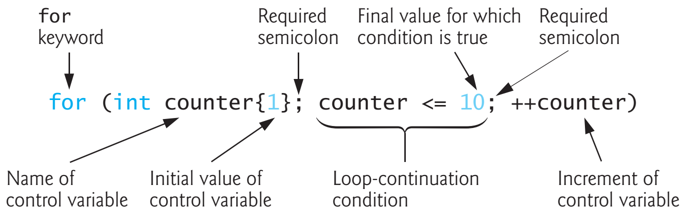
```
1  2  3  4  5  6  7  8  9  10
```

Fig. 4.2 | Counter-controlled iteration with the `for` iteration statement. (Part 2 of 2.)

When the `for` statement (lines 9–11) begins executing, the control variable `counter` is declared and initialized to `1`. Next, the program tests the loop-continuation condition between the two required semicolons (`counter <= 10`). Because `counter`'s initial value is `1`, the condition is true. So, line 10 displays `counter`'s value (`1`). After executing line 10, `++counter` to the right of the second semicolon increments `counter`. Then the program performs the loop-continuation test again to determine whether to proceed with the loop's next iteration. At this point, `counter`'s value is `2` and the condition is still true, so the program executes line 10 again. This process continues until the loop has displayed the numbers 1–10 and `counter`'s value becomes `11`. At this point, the loop-continuation test fails, iteration terminates and the program continues with the first statement after the loop (line 13).

A Closer Look at the `for` Statement's Header

The following diagram takes a closer look at the `for` statement in Fig. 4.2:



The first line—including the keyword `for` and everything in the parentheses after `for` (line 9 in Fig. 4.2)—is sometimes called the **`for` statement header**. The `for` header "does it all"—it specifies each item needed for counter-controlled iteration with a control variable.

General Format of a `for` Statement

The general format of the `for` statement is

```
for (initialization; loopContinuationCondition; increment) {
   statement
}
```

where

- *initialization* names the loop's control variable and provides its initial value,
- *loopContinuationCondition*—between the two required semicolons—determines whether the loop should continue executing, and
- *increment* modifies the control variable's value so that the loop-continuation condition eventually becomes false.

If the loop-continuation condition is initially false, the program does not execute the `for` statement's body. Instead, execution proceeds with the statement following the `for`.

Scope of a for Statement's Control Variable

If the *initialization* expression declares the control variable, it can be used only in that `for` statement—not beyond it. This restricted use is known as the variable's **scope**, which defines its lifetime and where it can be used in a program. For example, a variable's scope is from its declaration point to the right brace that closes the block. As you'll see in Chapter 5, it's good practice to define each variable in the smallest scope needed.

Expressions in a for Statement's Header Are Optional

All three expressions in a `for` header are optional. If you omit the *loopContinuationCondition*, the condition is always true, creating an infinite loop. You might omit the *initialization* expression if the program initializes the control variable before the loop. You might omit the *increment* expression if the program calculates the increment in the loop's body or if no increment is needed.

The increment expression in a `for` acts like a stand-alone statement at the end of the `for`'s body. Therefore, the increment expressions

```
counter = counter + 1
counter += 1
++counter
counter++
```

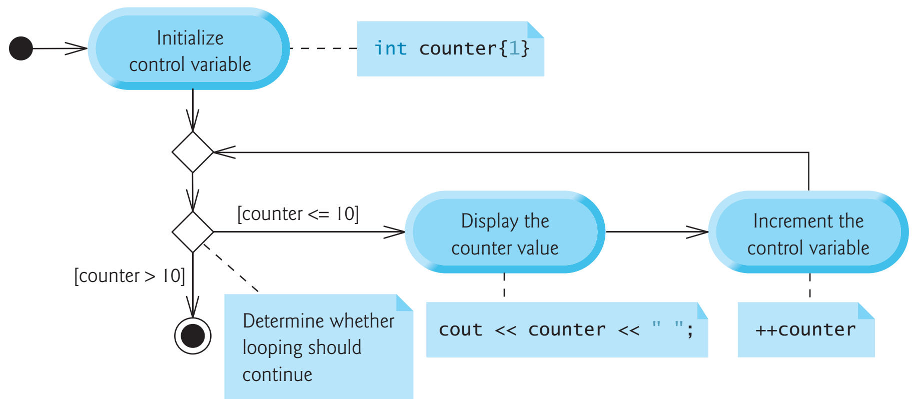
are equivalent in a `for` statement. In this case, the increment expression does not appear in a larger expression, so preincrementing and postincrementing have the same effect. We prefer preincrement. In Chapter 11's operator-overloading discussion, you'll see that preincrement can have a performance advantage.

Using a for Statement's Control Variable in the Statement's Body

Programs frequently display the control-variable value or use it in calculations in the loop body, but this use is not required. The value of the control variable can be changed in a `for` loop's body, but doing so can lead to subtle errors. If a program must modify the control variable's value in the loop's body, prefer `while` to `for`. Err

UML Activity Diagram of the for Statement

Below is the UML activity diagram of the `for` statement in Fig. 4.2—it makes it clear that initialization occurs once, before the condition is tested the first time. Incrementing occurs after the body statement executes:



4.4 Examples Using the for Statement

The following examples show techniques for varying the control variable in a `for` statement. In each case, we write only the appropriate `for` header. Note the change in the relational operator for the loops that decrement the control variable.

a) Vary the control variable from `1` to `100` in increments of `1`.

```
for (int i{1}; i <= 100; ++i)
```

b) Vary the control variable from `100` *down* to `1` in *decrements* of `1`.

```
for (int i{100}; i >= 1; --i)
```

c) Vary the control variable from `7` to `77` in increments of `7`.

```
for (int i{7}; i <= 77; i += 7)
```

d) Vary the control variable from `20` *down* to `2` in *decrements* of `2`.

```
for (int i{20}; i >= 2; i -= 2)
```

e) Vary the control variable over the values `2`, `5`, `8`, `11`, `14`, `17`, `20`.

```
for (int i{2}; i <= 20; i += 3)
```

f) Vary the control variable over the values `99`, `88`, `77`, `66`, `55`, `44`, `33`, `22`, `11`, `0`.

```
for (int i{99}; i >= 0; i -= 11)
```

Do not use equality operators (`!=` or `==`) in a loop-continuation condition if the loop's control variable increments or decrements by more than `1`. For example, in the `for` statement header

```
for (int counter{1}; counter != 10; counter += 2)
```

`counter != 10` never becomes false (resulting in an infinite loop) because `counter` increments by 2 after each iteration, producing only the odd values (3, 5, 7, 9, 11, …).

4.5 Application: Summing Even Integers

The application in Fig. 4.3 uses a `for` statement to sum the even integers from 2 to 20 and store the result in `int` variable `total`. Each iteration of the loop (lines 10–12) adds control variable `number`'s value to variable `total`.

```
1  // fig04_03.cpp
2  // Summing integers with the for statement.
3  #include <iostream>
4  using namespace std;
5
6  int main() {
7     int total{0};
8
9     // total even integers from 2 through 20
10    for (int number{2}; number <= 20; number += 2) {
11       total += number;
12    }
```

Fig. 4.3 | Summing integers with the `for` statement. (Part 1 of 2.)

```
13
14      cout << "Sum is " << total << "\n";
15   }
```

```
Sum is 110
```

Fig. 4.3 | Summing integers with the `for` statement. (Part 2 of 2.)

A `for` statement's initialization and increment expressions can be comma-separated lists containing multiple initialization expressions or multiple increment expressions. Although this is discouraged, you could merge the `for` statement's body (line 11) into the increment portion of the `for` header by using a comma operator as in

```
for (int number{2}; number <= 20; total += number, number += 2) { }
```

The comma between the expressions `total += number` and `number += 2` is the **comma operator**, which guarantees that a list of expressions evaluates from left to right. The comma operator has the lowest precedence of all C++ operators. The value and type of a comma-separated list of expressions is the value and type of the rightmost expression, respectively. The comma operator is often used in `for` statements that require multiple initialization expressions or multiple increment expressions.

4.6 Application: Compound-Interest Calculations

Let's compute compound interest with a `for` statement. Consider the following problem:

> *A person invests $1,000 in a savings account yielding 5% interest. Assuming all interest is left on deposit, calculate and print the amount of money in the account at the end of each year for 10 years. Use the following formula to determine the amounts:*
>
> $$a = p\,(1 + r)^n$$
>
> *where*
>
> p is the original amount invested (i.e., the principal),
> r is the annual interest rate (e.g., use 0.05 for 5%),
> n is the number of years, and
> a is the amount on deposit at the end of the nth year.

The solution (Fig. 4.4) uses a loop to perform the calculation for each of the 10 years the money remains on deposit. We use `double` values here for the monetary calculations. Then we discuss the problems with using floating-point types to represent monetary amounts. For financial applications that require precise monetary calculations and rounding control, consider using an open-source library such as `Boost.Multiprecision`.[1]

Lines 12–13 initialize `double` variable `principal` to `1000.00` and `double` variable `rate` to `0.05`. C++ treats floating-point literals like `1000.00` and `0.05` as type `double`. Sim-

1. John Maddock and Christopher Kormanyos, "Chapter 1. Boost.Multiprecision." Accessed November 19, 2021. `https://www.boost.org/doc/libs/master/libs/multiprecision/doc/html/index.html`.

ilarly, C++ treats whole numbers like `7` and `-22` as type `int`.[2] Lines 15–16 display the initial principal and the interest rate.

```
1  // fig04_04.cpp
2  // Compound-interest calculations with for.
3  #include <iostream>
4  #include <iomanip>
5  #include <cmath> // for pow function
6  using namespace std;
7
8  int main() {
9     // set floating-point number format
10    cout << fixed << setprecision(2);
11
12    double principal{1000.00}; // initial amount before interest
13    double rate{0.05}; // interest rate
14
15    cout << "Initial principal: " << principal << "\n";
16    cout << "    Interest rate:    " << rate << "\n";
17
18    // display headers
19    cout << "\nYear" << setw(20) << "Amount on deposit" << "\n";
20
21    // calculate amount on deposit for each of ten years
22    for (int year{1}; year <= 10; ++year) {
23       // calculate amount on deposit at the end of the specified year
24       double amount{principal * pow(1.0 + rate, year)} ;
25
26       // display the year and the amount
27       cout << setw(4) << year << setw(20) << amount << "\n";
28    }
29 }
```

```
Initial principal: 1000.00
    Interest rate:    0.05

Year   Amount on deposit
   1             1050.00
   2             1102.50
   3             1157.63
   4             1215.51
   5             1276.28
   6             1340.10
   7             1407.10
   8             1477.46
   9             1551.33
  10             1628.89
```

Fig. 4.4 | Compound-interest calculations with `for`.

2. Section 3.12 showed that C++'s integer types cannot represent all integer values. Choose the correct type for the range of values you need to represent. You may designate that an integer literal has type `long` or `long long` by appending `L` or `LL`, respectively, to the literal value.

Formatting with Field Widths and Justification

Line 10 before the loop and line 27 in the loop combine to print the `year` and `amount` values. We specify the formatting with the parameterized stream manipulators `setprecision` and **`setw`** and the nonparameterized stream manipulator `fixed`. The stream manipulator `setw(4)` specifies that the next value output should appear in a **field width** of 4—i.e., `cout <<` prints the value with at least four character positions. If the value to be output requires fewer than four character positions, the value is right-aligned in the field by default. If the value to be output has more than four character positions, C++ extends the field width to the right to accommodate the entire value. To left-align values, output nonparameterized stream manipulator **`left`** (found in header `<iostream>`). You can restore right-alignment by outputting nonparameterized stream manipulator **`right`**.

The other formatting in the output statements displays variable `amount` as a fixed-point value with a decimal point (`fixed` in line 10) right-aligned in a field of 20 character positions (`setw(20)` in line 27) and two digits of precision to the right of the decimal point (`setprecision(2)` in line 10). We applied the sticky stream manipulators `fixed` and `setprecision` to the output stream `cout` before the `for` loop because these format settings remain in effect until they're changed, and they do not need to be applied during each iteration of the loop. However, the field width specified with `setw` applies only to the next value output. Chapter 19 discusses `cin`'s and `cout`'s formatting capabilities in detail. We continue discussing C++20's powerful new text-formatting capabilities in Section 4.14.

Performing the Interest Calculations with Standard Library Function pow

The `for` statement (lines 22–28) iterates 10 times, varying the `int` control variable `year` from 1 to 10 in increments of 1. Variable `year` represents n in the problem statement.

C++ does not include an exponentiation operator, so we use the **standard library function `pow`** (line 24) from the header `<cmath>` (line 5). The call `pow(x, y)` calculates the value of x raised to the yth power. The function receives two `double` arguments and returns a `double` value. Line 24 performs the calculation $a = p(1 + r)^n$, where a is `amount`, p is `principal`, r is `rate` and n is `year`.

Function `pow`'s first argument—the calculation `1.0 + rate`—produces the same result each time through the loop, so repeating it in every iteration of the loop is wasteful. To improve program performance, many of today's optimizing compilers place such calculations before loops in the compiled code. Perf

Floating-Point Number Precision and Memory Requirements

A `float` represents a **single-precision floating-point number**. Most of today's systems store these in four bytes of memory with approximately seven significant digits. A `double` represents a **double-precision floating-point number**. Most of today's systems store these in eight bytes of memory with approximately 15 significant digits—approximately double the precision of `float`s. Most programmers use type `double`. C++ treats floating-point numbers such as 3.14159 in a program's source code as `double` values by default. Such values in the source code are known as **floating-point literals**.

The C++ standard requires only that type `double` provide at least as much precision as `float`. There is also type `long double`, which provides at least as much precision as `double`. For a complete list of C++ fundamental types and their typical ranges, see

`https://en.cppreference.com/w/cpp/language/types`

Floating-Point Numbers Are Approximations

In conventional arithmetic, floating-point numbers often arise as a result of division. Dividing 10 by 3, the result is 3.3333333…, with the sequence of 3s repeating infinitely. The computer allocates a fixed amount of space to hold such a value, so the stored value can be only an approximation. Floating-point types such as `double` suffer from what is referred to as **representational error**. Assuming that floating-point numbers are represented exactly (e.g., using them in comparisons for equality) can lead to incorrect results.

Err

Floating-point numbers have numerous applications, especially for measured values. For example, when we speak of a "normal" body temperature of 98.6 degrees Fahrenheit, we do not need to be precise to a large number of digits. When we read the temperature on a thermometer as 98.6, it actually might be 98.594732103. Calling this number 98.6 is fine for most body temperature calculations. Generally, `double` is preferred over `float`, because `double`s represent floating-point numbers more precisely.[3]

A Warning about Displaying Rounded Values

This example declared `double` variables `amount`, `principal` and `rate` to be of type `double`. Unfortunately, floating-point numbers can cause trouble with fractional dollar amounts. Here's a simple explanation of what can go wrong when floating-point numbers are used to represent dollar amounts that are displayed with two digits to the right of the decimal point. Two calculated dollar amounts stored in the machine could be 14.234 (rounded to 14.23 for display purposes) and 18.673 (rounded to 18.67 for display purposes). When these amounts are added, they produce the internal sum 32.907, which would typically be rounded to 32.91 for display purposes. Thus, your output could appear as

```
  14.23
+ 18.67
  32.91
```

but a person adding the individual numbers as displayed would expect the sum to be 32.90. You've been warned!

Even Common Dollar Amounts Can Have Floating-Point Representational Errors

Even simple dollar amounts can have representational errors when they're stored as `double`s. To see this, we created a simple program that defined the variable `d` as follows:

```
double d{123.02};
```

We displayed `d`'s value with 20 digits of precision to the right of the decimal point. The resulting output showed 123.02 as 123.0199999…, which is another example of a representational error. Though some dollar amounts can be represented precisely as `double`s, many cannot. This is a common problem in many programming languages. Later in the book, we create and use classes that handle monetary amounts precisely.

4.7 do…while Iteration Statement

In a `while` statement, the program tests the loop-continuation condition before executing the loop's body. If it's false, the body never executes. The **`do…while` iteration statement**

3. Nowadays, the standard floating-point representation is IEEE 754 (`https://en.wikipedia.org/wiki/IEEE_754`).

tests the loop-continuation condition after executing the loop's body; so, the body always executes at least once. Figure 4.5 uses a `do…while` to output the numbers 1–10. Line 7 declares and initializes control variable `counter`. Upon entering the `do…while` statement, line 10 outputs `counter`'s value and line 11 increments `counter`. Then the program evaluates the loop-continuation test at the bottom of the loop (line 12). If the condition is true, the loop continues at the first body statement (line 10). If the condition is false, the loop terminates, and the program continues at the next statement after the loop.

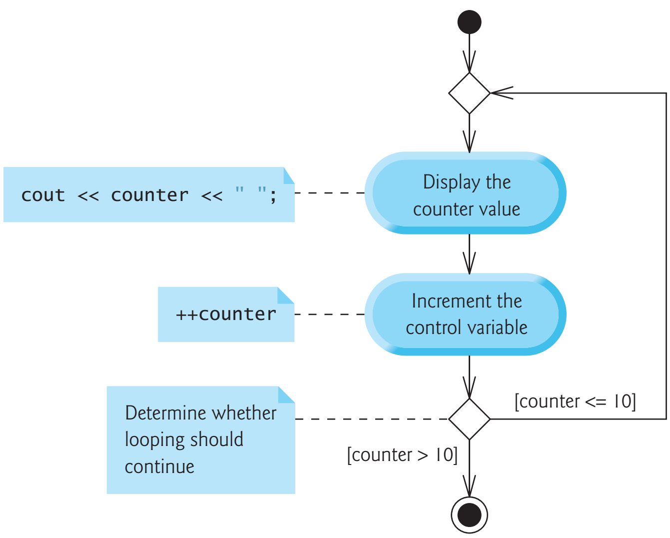
```
1  // fig04_05.cpp
2  // do...while iteration statement.
3  #include <iostream>
4  using namespace std;
5
6  int main() {
7     int counter{1};
8
9     do {
10       cout << counter << "  ";
11       ++counter;
12    } while (counter <= 10); // end do...while
13
14    cout << "\n";
15 }
```

```
1  2  3  4  5  6  7  8  9  10
```

Fig. 4.5 | `do…while` iteration statement.

UML Activity Diagram for the do…while Iteration Statement

The `do…while`'s UML activity diagram makes it clear that the loop-continuation condition is not evaluated until after the loop performs the action state at least once:



4.8 switch Multiple-Selection Statement

C++ provides the **switch multiple-selection** statement to choose among many different actions based on the possible values of a variable or expression. Each action is associated with the value of an **integral constant expression**—any combination of character and integer constants that evaluates to a constant integer value.

Using a switch Statement to Count A, B, C, D and F Grades

Figure 4.6 calculates the class average of a set of numeric grades entered by the user. The `switch` statement determines each grade's letter equivalent (A, B, C, D or F) and increments the appropriate grade counter. The program also displays a summary of the number of students who received each grade.

```
// fig04_06.cpp
// Using a switch statement to count letter grades.
#include <iostream>
#include <iomanip>
using namespace std;

int main() {
   int total{0}; // sum of grades
   int gradeCounter{0}; // number of grades entered
   int aCount{0}; // count of A grades
   int bCount{0}; // count of B grades
   int cCount{0}; // count of C grades
   int dCount{0}; // count of D grades
   int fCount{0}; // count of F grades

   cout << "Enter the integer grades in the range 0-100.\n"
      << "Type the end-of-file indicator to terminate input:\n"
      << "   On UNIX/Linux/macOS type <Ctrl> d then press Enter\n"
      << "   On Windows type <Ctrl> z then press Enter\n";

   int grade;

   // loop until user enters the end-of-file indicator
   while (cin >> grade) {
      total += grade; // add grade to total
      ++gradeCounter; // increment number of grades

      //  increment appropriate letter-grade counter
      switch (grade / 10) {
         case 9:  // grade was between 90
         case 10: // and 100, inclusive
            ++aCount;
            break; // exits switch

         case 8: // grade was between 80 and 89
            ++bCount;
            break; // exits switch

```

Fig. 4.6 | Using a `switch` statement to count letter grades. (Part 1 of 3.)

```
         case 7: // grade was between 70 and 79
            ++cCount;
            break; // exits switch

         case 6: // grade was between 60 and 69
            ++dCount;
            break; // exits switch

         default: // grade was less than 60
            ++fCount;
            break; // optional; exits switch anyway
      } // end switch
   } // end while

   // set floating-point number format
   cout << fixed << setprecision(2);

   // display grade report
   cout << "\nGrade Report:\n";

   // if user entered at least one grade...
   if (gradeCounter != 0) {
      // calculate average of all grades entered
      double average{static_cast<double>(total) / gradeCounter};

      // output summary of results
      cout << "Total of the " << gradeCounter << " grades entered is "
         << total << "\nClass average is " << average
         << "\nNumber of students who received each grade:"
         << "\nA: " << aCount << "\nB: " << bCount << "\nC: " << cCount
         << "\nD: " << dCount << "\nF: " << fCount << "\n";
   }
   else { // no grades were entered, so output appropriate message
      cout << "No grades were entered" << "\n";
   }
}
```

```
Enter the integer grades in the range 0-100.
Type the end-of-file indicator to terminate input:
   On UNIX/Linux/macOS type <Ctrl> d then press Enter
   On Windows type <Ctrl> z then press Enter
99
92
45
57
63
71
76
85
90
100
^Z
```

Fig. 4.6 | Using a switch statement to count letter grades. (Part 2 of 3.)

```
Grade Report:
Total of the 10 grades entered is 778
Class average is 77.80

Number of students who received each grade:
A: 4
B: 1
C: 2
D: 1
F: 2
```

Fig. 4.6 | Using a `switch` statement to count letter grades. (Part 3 of 3.)

Figure 4.6 declares local variables `total` (line 8) and `gradeCounter` (line 9) to keep track of the sum of the grades entered by the user and the number of grades entered. Lines 10–14 declare and initialize to 0 counter variables for each grade category. Lines 24–51 input an arbitrary number of integer grades using sentinel-controlled iteration, update variables `total` and `gradeCounter`, and increment an appropriate letter-grade counter for each grade entered. Lines 54–73 output a report containing the total of all grades entered, the average grade and the number of students who received each letter grade.

Reading Grades from the User

Lines 16–19 prompt the user to enter integer grades or type the end-of-file indicator to terminate the input. The **end-of-file indicator** is a system-dependent keystroke combination used to indicate that there's no more data to input. In Chapter 8, you'll see how the end-of-file indicator is used when a program reads its input from a file.

The keystroke combinations for entering end-of-file are system dependent. On UNIX/Linux/macOS systems, type the sequence

<Ctrl> d

on a line by itself. This notation means to press both the *Ctrl* key and the *d* key simultaneously. On Windows systems, type

<Ctrl> z

On some systems, you must also press *Enter*. Also, Windows typically displays `^Z` on the screen when you type the end-of-file indicator, as shown in the output of Fig. 4.6.

The `while` statement (lines 24–51) obtains the user input. Line 24

```
while (cin >> grade) {
```

performs the input in the `while` statement's condition. In this case, the loop-continuation condition evaluates to `true` if `cin` successfully reads an `int` value. If the user enters the end-of-file indicator, the condition evaluates to `false`.

If the condition is `true`, line 25 adds `grade` to `total`, and line 26 increments `gradeCounter`. These are used to compute the average. Next, lines 29–50 use a `switch` statement to increment the appropriate letter-grade counter based on the numeric grade entered.

Processing the Grades

The `switch` statement (lines 29–50) determines which counter to increment. We assume that the user enters a valid grade in the range 0–100. A grade in the range 90–100 represents A, 80–89 represents B, 70–79 represents C, 60–69 represents D and 0–59 represents F. The `switch` statement's block contains a sequence of **`case` labels** and an optional **`default` case**, which can appear anywhere in the `switch`, but normally appears last. These are used in this example to determine which counter to increment based on the grade.

When the flow of control reaches the `switch`, the program evaluates the **controlling expression** in the parentheses (`grade / 10`) following keyword `switch`. The program compares this expression's value with each `case` label. The expression must have a signed or unsigned integral type—`bool`, `char`, `char8_t`, `char16_t`, `char32_t`, `wchar_t`, `int`, `long` or `long long`.

The controlling expression in line 29 performs integer division, which truncates the fractional part of the result. When we divide a value from 0 to 100 by 10, the result is always a value from 0 to 10. We use several of these values in our `case` labels. If the user enters the integer `85`, the controlling expression evaluates to 8. The `switch` compares `8` with each `case` label. If a match occurs (`case 8:` at line 35), that `case`'s statements execute. For `8`, line 36 increments `bCount`, because a grade in the 80s is a B. The **`break` statement** (line 37) exits the `switch`. In this program, we reach the end of the `while` loop, so control returns to the loop-continuation condition in line 24 to determine whether the loop should continue executing.

The `case`s in our `switch` explicitly test for the values `10`, `9`, `8`, `7` and `6`. Note the cases at lines 30–31 that test for the values `9` and `10` (both of which represent the grade A). Listing cases consecutively in this manner with no statements between them enables the cases to perform the same set of statements—when the controlling expression evaluates to `9` or `10`, the statements in lines 32–33 execute. The `switch` statement does not provide a mechanism for testing ranges of values, so every value you need to test must be listed in a separate `case` label. Each `case` can have multiple statements. The `switch` statement differs from other control statements in that it does not require braces around multiple statements in a `case`, unless you need to declare a variable in a `case`.

case without a break Statement—C++17 [[fallthrough]] Attribute

Without `break` statements, each time a match occurs in the `switch`, the statements for that `case` and subsequent `case`s execute until a `break` statement or the end of the `switch` is reached. This is referred to as "falling through" to the statements in subsequent `case`s.[4]

Forgetting a `break` statement when one is needed is a logic error. To call your attention to this possible problem, many compilers issue a warning when a `case` label is followed by one or more statements and does not contain a `break` statement. For such instances in which "falling through" is the desired behavior, C++17 introduced the **`[[fallthrough]]` attribute**. You can tell the compiler that "falling through" to the next `case` is the correct behavior by placing the statement Err 17

```
[[fallthrough]];
```

where the `break` statement would normally appear.

4. This feature is perfect for writing a concise program that displays the iterative song "The Twelve Days of Christmas." As an exercise, you might write the program, then use one of the many free, open-source text-to-speech programs to speak the song. You might also tie your program to a free, open-source MIDI ("Musical Instrument Digital Interface") program to create a singing version of your program accompanied by music.

The `default` Case

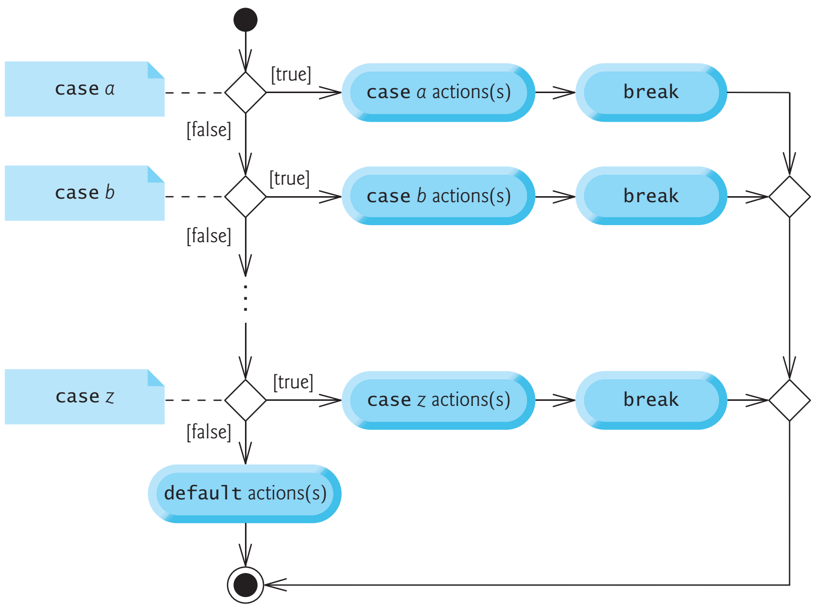
If no match occurs between the controlling expression's value and any of the `case` labels, the `default` case (lines 47–49) executes. We use the `default` case in this example to process all controlling-expression values that are less than `6`—that is, all failing grades. If no match occurs and the `switch` does not contain a `default` case, program control simply continues with the first statement after the `switch`. In a `switch`, it's good practice to test for all possible values of the controlling expression.

Displaying the Grade Report

Lines 54–73 output a report based on the grades entered. Line 60 determines whether the user entered at least one grade—this helps us avoid dividing by zero, which for integer division causes the program to fail and for floating-point division produces the value `nan`—for "not a number." If so, line 62 calculates the average of the grades. Lines 65–69 then output the total of all the grades, the class average and the number of students who received each letter grade. If no grades were entered, line 72 outputs an appropriate message. The output in Fig. 4.6 shows a sample grade report based on 10 grades.

`switch` Statement UML Activity Diagram

The following is the UML activity diagram for the general `switch` statement:



Most `switch` statements use a `break` in each `case` to terminate the `switch` after the `case` is processed. The diagram emphasizes this by including `break` statements and showing that the `break` at the end of a `case` causes control to exit the `switch` statement immediately.

The `break` statement is not required for the `switch`'s last case (or the optional `default` case, when it appears last), because execution continues with the next statement after the `switch`. Provide a `default` case in every `switch` statement to focus you on processing exceptional conditions.

Notes on cases

Each `case` in a `switch` statement must contain a constant integral expression—that is, any expression that evaluates to a constant integer value. You also can use `enum` constants (introduced in Section 5.9) and **character literals**—specific characters in single quotes, such as `'A'`, `'7'` or `'$'`, which represent the integer values of characters. (Appendix B shows the integer values of the characters in the ASCII character set, which is a subset of the Unicode character set.)

In Chapter 10, OOP: Inheritance and Runtime Polymorphism, we present a more elegant way to implement `switch` logic. We use a technique called polymorphism to create programs that are often clearer, easier to maintain and easier to extend than programs using `switch` logic.

4.9 C++17 Selection Statements with Initializers 17

Earlier, we introduced the `for` iteration statement. In the `for` header's initialization section, we declared and initialized a control variable, which limited that variable's scope to the `for` statement. C++17's **selection statements with initializers** enable you to include 17
variable initializers before the condition in an `if` or `if…else` statement and before the controlling expression of a `switch` statement. As with the `for` statement, these variables are known only in the statements where they're declared. Figure 4.7 shows `if…else` statements with initializers. We'll use both `if…else` and `switch` statements with initializers in Fig. 5.5, which implements a popular casino dice game.

```
// fig04_07.cpp
// C++17 if statements with initializers.
#include <iostream>
using namespace std;

int main() {
   if (int value{7}; value == 7) {
      cout << "value is " << value << "\n";
   }
   else {
      cout << "value is not 7; it is " << value << "\n";
   }

   if (int value{13}; value == 9) {
      cout << "value is " << value << "\n";
   }
   else {
      cout << "value is not 9; it is " << value << "\n";
   }
}
```

```
value is 7
value is not 9; it is 13
```

Fig. 4.7 | C++17 `if` statements with initializers.

Syntax of Selection Statements with Initializers

For an `if` or `if…else` statement, you place the initializer first in the condition's parentheses. For a `switch` statement, you place the initializer first in the controlling expression's parentheses. The initializer must end with a semicolon (`;`), as in lines 7 and 14. The initializer can declare multiple variables of the same type in a comma-separated list.

Scope of Variables Declared in the Initializer

Any variable declared in the initializer of an `if`, `if…else` or `switch` statement may be used throughout the remainder of the statement. In lines 7–12, we use the variable `value` to determine which branch of the `if…else` statement to execute, then use `value` in the output statements of both branches. When the `if…else` statement terminates, `value` no longer exists, so we can use that identifier again in the second `if…else` statement to declare a new variable known only in that statement.

To prove that `value` is not accessible outside the `if…else` statements, we provided a second version of this program (`fig04_07_with_error.cpp`) that attempts to access variable `value` after (and thus outside the scope of) the second `if…else` statement. This produces the following compilation errors in our three compilers:

- Visual Studio: `'value': undeclared identifier`
- Xcode: `error: use of undeclared identifier 'value'`
- GNU g++: `error: 'value' was not declared in this scope`

4.10 break and continue Statements

In addition to selection and iteration statements, C++ provides `break` and **`continue`** statements to alter the flow of control. The preceding section showed how `break` could be used to terminate a `switch` statement's execution. This section discusses how to use `break` in iteration statements.

break Statement

Executing a `break` statement in a `while`, `for`, `do…while` or `switch` causes immediate exit from that statement—execution continues with the first statement after the control statement. Common uses of `break` include escaping early from a loop or exiting a `switch` (as in Fig. 4.6). Figure 4.8 demonstrates a `break` statement exiting early from a `for` statement.

```
// fig04_08.cpp
// break statement exiting a for statement.
#include <iostream>
using namespace std;

int main() {
   int count; // control variable also used after loop

   for (count = 1; count <= 10; ++count) { // loop 10 times
      if (count == 5) {
         break; // terminates for loop if count is 5
      }
```

Fig. 4.8 | `break` statement exiting a `for` statement. (Part 1 of 2.)

```
13
14          cout << count << " ";
15       }
16
17       cout << "\nBroke out of loop at count = " << count << "\n";
18    }
```

```
1 2 3 4
Broke out of loop at count = 5
```

Fig. 4.8 | `break` statement exiting a `for` statement. (Part 2 of 2.)

When the `if` statement nested at lines 10–12 in the `for` statement (lines 9–15) detects that `count` is `5`, the `break` statement at line 11 executes. This terminates the `for` statement, and the program proceeds to line 17 (immediately after the `for` statement), which displays a message indicating the value of the control variable when the loop terminated. The loop fully executes its body only four times instead of 10. Note that we could have initialized `count` in line 7 and left the `for` header's initialization section empty, as in:

```
for (; count <= 10; ++count) { // loop 10 times
```

`continue` Statement

Executing the `continue` statement in a `while`, `for` or `do…while` skips the remaining statements in the loop body and proceeds with the next iteration of the loop. In `while` and `do…while` statements, the program evaluates the loop-continuation test immediately after the `continue` statement executes. In a `for` statement, the increment expression executes, then the program evaluates the loop-continuation test.

```
// fig04_09.cpp
// continue statement terminating an iteration of a for statement.
#include <iostream>
using namespace std;

int main() {
   for (int count{1}; count <= 10; ++count) { // loop 10 times
      if (count == 5) {
         continue; // skip remaining code in loop body if count is 5
      }

      cout << count << " ";
   }

   cout << "\nUsed continue to skip printing 5" << "\n";
}
```

```
1 2 3 4 6 7 8 9 10
Used continue to skip printing 5
```

Fig. 4.9 | `continue` statement terminating an iteration of a `for` statement.

Figure 4.9 uses `continue` (line 9) to skip the statement at line 12 when the nested `if` determines that `count`'s value is `5`. When the `continue` statement executes, program control continues with the increment of the control variable in the `for` statement (line 7).

Some programmers feel that `break` and `continue` violate structured programming. Since the same effects are achievable with structured-programming techniques, these programmers prefer to avoid `break` or `continue`.

Perf There's a tension between achieving quality software engineering and achieving the best-performing software. Sometimes one of these goals is achieved at the expense of the other. For all but the most performance-intensive situations, you should first make your code simple and correct, then make it fast and small—but only if necessary.

4.11 Logical Operators

The conditions in `if`, `if...else`, `while`, `do...while` and `for` statements determine how to continue a program's flow of control. So far, we've studied only simple conditions, such as `count <= 10`, `number != sentinelValue` and `total > 1000`. Simple conditions are expressed with the relational operators `>`, `<`, `>=` and `<=` and the equality operators `==` and `!=`. Each tests one condition. Sometimes control statements require more complex conditions to determine a program's flow of control. C++'s **logical operators** enable you to combine simple conditions. The logical operators are `&&` (logical AND), `||` (logical OR) and `!` (logical negation).

4.11.1 Logical AND (&&) Operator

Suppose we wish to ensure at some point in a program that two conditions are both true before we choose a certain path of execution. In this case, we can use the **`&&`** (**logical AND**) operator, as follows:

```
if (gender == FEMALE && age >= 65) {
   ++seniorFemales;
}
```

Assume `FEMALE` is a constant variable. This `if` statement contains two simple conditions. The condition `gender == FEMALE` determines whether a person is female. The condition `age >= 65` might be evaluated to determine whether a person is a senior citizen. The `if` statement considers the combined condition

```
gender == FEMALE && age >= 65
```

which is true if and only if both simple conditions are true. In this case, the `if` statement's body increments `seniorFemales` by `1`. If either or both of the simple conditions are false, the program skips the increment. Some programmers find that the preceding combined condition is more readable when redundant parentheses are added, as in

```
(gender == FEMALE) && (age >= 65)
```

The following **truth table** summarizes the `&&` operator, showing all four possible combinations of the `bool` values `false` and `true` for *expression1* and *expression2*. C++ evaluates to zero (false) or nonzero (true) all expressions that include relational operators, equality operators or logical operators:

expression1	expression2	expression1 && expression2
`false`	`false`	`false`
`false`	`true`	`false`
`true`	`false`	`false`
`true`	`true`	`true`

4.11.2 Logical OR (||) Operator

Now suppose we wish to ensure that either or both of two conditions are true before we choose a certain path of execution. In this case, we use the `||` (**logical OR**) operator, as in the following program segment:

```
if ((semesterAverage >= 90) || (finalExam >= 90)) {
   cout << "Student grade is A\n";
}
```

This statement also contains two simple conditions. The condition `semesterAverage >= 90` determines whether the student deserves an A in the course for a solid performance throughout the semester. The condition `finalExam >= 90` determines whether the student deserves an A in the course for an outstanding performance on the final exam. The `if` statement then considers the combined condition

```
(semesterAverage >= 90) || (finalExam >= 90)
```

and awards the student an A if either or both of the simple conditions are true. The only time the message `"Student grade is A"` is not printed is when both of the simple conditions are false. The following is the truth table for the operator logical OR (`||`):

expression1	expression2	expression1 || expression2
`false`	`false`	`false`
`false`	`true`	`true`
`true`	`false`	`true`
`true`	`true`	`true`

Operator `&&` has higher precedence than operator `||`.[5] Both operators group left-to-right.

4.11.3 Short-Circuit Evaluation

The parts of an expression containing `&&` or `||` operators are evaluated only until it's known whether the condition is true or false. Thus, evaluation of the expression

```
(gender == FEMALE) && (age >= 65)
```

stops immediately if `gender` is not equal to `FEMALE` (i.e., the entire expression is false) and continues if `gender` is equal to `FEMALE` (i.e., the entire expression could still be true if the

5. In general, use parentheses if there is ambiguity about evaluation order.

condition `age >= 65` is true). This feature of logical AND and logical OR expressions is called **short-circuit evaluation**.

In expressions using operator `&&`, a condition—we'll call this the dependent condition—may require another condition to be true for the evaluation of the dependent condition to be meaningful. In this case, the dependent condition should be placed after the `&&` operator to prevent errors. Consider the expression `(i != 0) && (10 / i == 2)`. The dependent condition `(10 / i == 2)` must appear after the `&&` operator to prevent the possibility of division by zero.

4.11.4 Logical Negation (!) Operator

The `!` (**logical negation**, also called **logical NOT** or **logical complement**) operator "reverses" the meaning of a condition. Unlike the logical operators `&&` and `||`, which are binary operators that combine two conditions, the logical negation operator is a unary operator that has only one condition as an operand. To execute code only when a condition is false, place the logical negation operator *before* the original condition, as in the program segment

```
if (!(grade == sentinelValue)) {
   cout << "The next grade is " << grade << "\n";
}
```

which executes the body statement only if `grade` is *not* equal to `sentinelValue`. The parentheses around the condition `grade == sentinelValue` are needed because the logical negation operator has higher precedence than the equality operator.

In most cases, you can avoid using logical negation by expressing the condition differently with an appropriate relational or equality operator. For example, the previous statement may also be written in a more readable manner as

```
if (grade != sentinelValue) {
   cout << "The next grade is " << grade << "\n";
}
```

This flexibility can help you express a condition more conveniently. The following is the truth table for the logical negation operator:

expression	!expression
`false`	`true`
`true`	`false`

4.11.5 Example: Producing Logical-Operator Truth Tables

Figure 4.10 uses logical operators to produce the truth tables discussed in this section. The output shows each expression that's evaluated and its `bool` result. By default, `bool` values `true` and `false` are displayed by `cout` and the stream-insertion operator as `1` and `0`, respectively, but the format function displays the word "true" or the word "false." Lines 10–14, 17–21 and 24–26 produce the truth tables for `&&`, `||` and `!`, respectively.

```cpp
// fig04_10.cpp
// Logical operators.
#include <iostream>
#include <fmt/format.h> // in C++20, this will be #include <format>
using namespace std;
using namespace fmt; // not needed in C++20

int main() {
   // create truth table for && (logical AND) operator
   cout << "Logical AND (&&)\n"
      << format("false && false: {}\n", false && false)
      << format("false && true: {}\n", false && true)
      << format("true && false: {}\n", true && false)
      << format("true && true: {}\n\n", true && true);

   // create truth table for || (logical OR) operator
   cout << "Logical OR (||)\n"
      << format("false || false: {}\n", false || false)
      << format("false || true: {}\n", false || true)
      << format("true || false: {}\n", true || false)
      << format("true || true: {}\n\n", true || true);

   // create truth table for ! (logical negation) operator
   cout << "Logical negation (!)\n"
      << format("!false: {}\n", !false)
      << format("!true: {}\n", !true);
}
```

```
Logical AND (&&)
false && false: false
false && true: false
true && false: false
true && true: true

Logical OR (||)
false || false: false
false || true: true
true || false: true
true || true: true

Logical negation (!)
!false: true
!true: false
```

Fig. 4.10 | Logical operators.

Precedence and Grouping of the Operators Presented So Far

The following table shows the precedence and grouping of the C++ operators introduced so far—from top to bottom in decreasing order of precedence:

Operators	Grouping
`++` `--` `static_cast<type>()`	left to right
`++` `--` `+` `-` `!`	right to left
`*` `/` `%`	left to right
`+` `-`	left to right
`<<` `>>`	left to right
`<` `<=` `>` `>=`	left to right
`==` `!=`	left to right
`&&`	left to right
`||`	left to right
`?:`	right to left
`=` `+=` `-=` `*=` `/=` `%=`	right to left
`,`	left to right

4.12 Confusing the Equality (==) and Assignment (=) Operators

There's one logic error that C++ programmers, no matter how experienced, tend to make so frequently that we feel it requires a separate section. That error is accidentally swapping the operators == (equality) and = (assignment). What makes this so damaging is that it ordinarily does not cause compilation errors. Statements with these errors tend to compile correctly and run to completion, often generating incorrect results through runtime logic errors. Today's compilers generally can issue warnings when = is used in contexts where == is expected (see the end of this section for details on enabling this).

Two aspects of C++ contribute to these problems. One is that any expression that produces a value can be used in the decision portion of any control statement. If the expression's value is zero, it's treated as `false`. If the value is nonzero, it's treated as `true`. The second is that assignments produce a value—namely, the value of the variable on the assignment operator's left side. For example, suppose we intend to write

```
if (payCode == 4) { // good
   cout << "You get a bonus!" << "\n";
}
```

but we accidentally write

```
if (payCode = 4) { // bad
   cout << "You get a bonus!" << "\n";
}
```

The first `if` statement properly awards a bonus to the person whose `payCode` is equal to 4. The second one—which contains the error—evaluates the assignment expression in the `if` condition to the constant 4. Any nonzero value is `true`, so this condition always evaluates as `true` and the person always receives a bonus regardless of the pay code! Even worse, the pay code has been modified when it was only supposed to be examined!

lvalues and *rvalues*

You can prevent this problem with a simple trick. First, it's helpful to know what's allowed to the left of an assignment operator. Variable names are said to be ***lvalues*** (for "left values") because they can be used on an assignment operator's left side. Literals are said to be ***rvalues*** (for "right values")—they can be used on only an assignment operator's right side. You also can use *lvalues* as *rvalues* on an assignment's right side, but not vice versa.

Programmers normally write conditions such as `x == 7` with the variable name (an *lvalue*) on the left and the literal (an *rvalue*) on the right. Placing the literal on the left, as in `7 == x` (which is syntactically correct and is sometimes called a "Yoda condition"[6]), enables the compiler to issue an error if you accidentally replace the `==` operator with `=`. The compiler treats this as a compilation error because you can't change a literal's value.

Using == in Place of =

There's another equally unpleasant situation. Suppose you want to assign a value to a variable with a simple statement like

```
x = 1;
```

but instead write

```
x == 1;
```

Here, too, this is not a syntax error. Rather, the compiler simply evaluates the expression. If `x` is equal to `1`, the condition is true, and the expression evaluates to a nonzero (true) value. If `x` is not equal to `1`, the condition is false and the expression evaluates to 0. Regardless of the expression's value, there's no assignment operator, so the value is lost. The value of `x` remains unaltered, probably causing an execution-time logic error. Using operator `==` for assignment and using operator `=` for equality are logic errors. Use your text editor to search for all occurrences of `=` in your program and check that you have the correct assignment, relational or equality operator in each place.

Enabling Warnings

Xcode automatically issues a warning when you use `=` where `==` is expected. Some compilers require you to enable warnings before they'll issue warning messages. For GNU `g++`, add the **`-Wall`** (enable all warnings) flag to your compilation command—see the `g++` documentation for details on enabling subsets of the potential warnings. For Visual C++:

1. In your solution, right-click the project's name and select **Properties**.
2. Expand **Code Analysis** and select **General**.
3. For **Enable Code Analysis on Build**, select **Yes**, then click **OK**.

---

6. "Yoda conditions." Accessed November 19, 2021. `https://en.wikipedia.org/wiki/Yoda_conditions`.

4.13 Objects-Natural Case Study: Using the `miniz-cpp` Library to Write and Read ZIP files[7]

Perf **Data compression** reduces the size of data—typically to save memory, to save secondary storage space or to transmit data over the Internet faster by reducing the number of bytes. **Lossless data-compression algorithms** compress data in a manner that does not lose information—the data can be uncompressed and restored to its original form. **Lossy data-compression algorithms** permanently discard information. Such algorithms are often used to compress images, audio and video. For example, when you watch streaming video online, the video is often compressed ahead of time using a lossy algorithm to minimize the total bytes transferred over the Internet. Though some of the video data is discarded, a lossy algorithm compresses the data in a manner such that most people do not notice the removed information as they watch the video. The video quality is still "pretty good."

ZIP Files

You've probably used ZIP files—if not, you almost certainly will. The **ZIP file format**[8] is a lossless compression[9] format that has been in use for over 30 years. Lossless compression algorithms use various techniques for compressing data—such as

- replacing duplicate patterns, such as text strings in a document or pixels in an image, with references to a single copy, and
- replacing a group of image pixels that have the same color with one pixel of that color and a count (known as "run-length encoding").

ZIP is used to compress files and directories into a single file, known as an **archive file**. ZIP files are often used to distribute software faster over the Internet. Today's operating systems typically have built-in support for creating ZIP files and extracting their contents.

Open-Source `miniz-cpp` Library

Many open-source libraries support programmatic manipulation of ZIP archive files and other popular archive-file formats, such as TAR, RAR and 7-Zip.[10] Figure 4.11 continues our Objects-Natural presentation by using objects of the open-source `miniz-cpp`[11,12] library's class `zip_file` to create and read ZIP files. The `miniz-cpp` library is a "header-only library"—it's defined in header file `zip_file.hpp`, which you can simply place in the same folder as this example and include the header in your program (line 5). We provide the library in the `examples` folder's `libraries/miniz-cpp` subfolder. Header files are discussed in depth in Chapter 9.

---

7. This example does not compile in GNU C++.
8. "Zip (file format)." Accessed November 19, 2021. `https://en.wikipedia.org/wiki/Zip_(file_format)`.
9. "Data compression." Accessed November 19, 2021. `https://en.wikipedia.org/wiki/Data_compression#Lossless`.
10. "List of archive formats." Wikipedia. Wikimedia Foundation, March 19, 2020. `https://en.wikipedia.org/wiki/List_of_archive_formats`.
11. `https://github.com/tfussell/miniz-cpp`.
12. The `miniz-cpp` library provides capabilities nearly identical to the Python standard library's `zipfile` module (`https://docs.python.org/3/library/zipfile.html`), so the `miniz-cpp` GitHub repository refers you to that documentation page for the list of features.

```
// fig04_11.cpp
// Using the miniz-cpp header-only library to write and read a ZIP file.
#include <iostream>
#include <string>
#include "zip_file.hpp"
using namespace std;

```

Fig. 4.11 | Using the `miniz-cpp` header-only library to write and read a ZIP file.

Inputting a Line of Text from the User with `getline`

The **`getline`** function call reads all the characters you type until you press *Enter*:

```
int main() {
   cout << "Enter a ZIP file name: ";
   string zipFileName;
   getline(cin, zipFileName); // inputs a line of text

```

```
Enter a ZIP file name: c:\users\useraccount\Documents\test.zip
```

Here we use `getline` to read from the user the location and name of a file, and store it in the `string` variable `zipFileName`. Like class `string`, `getline` requires the `<string>` header and belongs to namespace `std`.

Creating Sample Content to Write an Individual File in the ZIP File

The following statement creates a lengthy string named `content` consisting of sentences from this chapter's introduction:

```
   // string literals separated only by whitespace are combined
   // into a single string by the compiler
   string content{
      "This chapter introduces all but one of the remaining control "
      "statements--the for, do...while, switch, break and continue "
      "statements. We explore the essentials of counter-controlled "
      "iteration. We use compound-interest calculations to begin "
      "investigating the issues of processing monetary amounts. First, "
      "we discuss the representational errors associated with "
      "floating-point types. We use a switch statement to count the "
      "number of A, B, C, D and F grade equivalents in a set of "
      "numeric grades. We show C++17's enhancements that allow you to "
      "initialize one or more variables of the same type in the "
      "headers of if and switch statements."};

```

We'll use the `miniz-cpp` library to write this string as a text file that will be compressed into a ZIP file. Each string literal in the preceding statement is separated from the next only by whitespace. The C++ compiler automatically assembles such string literals into a single string literal, which we use to initialize the `string` variable `content`. The following statement outputs the length of `content` (632 bytes).

```
28     cout << "\ncontent.length(): " << content.length();
29
```

```
content.length(): 632
```

Creating a `zip_file` Object

The `miniz-cpp` library's `zip_file` class—located in the library's `miniz_cpp` namespace—is used to create a ZIP file. The statement

```
30     miniz_cpp::zip_file output; // create zip_file object
31
```

creates the `zip_file` object `output`, which will perform the ZIP operations to create the archive file.

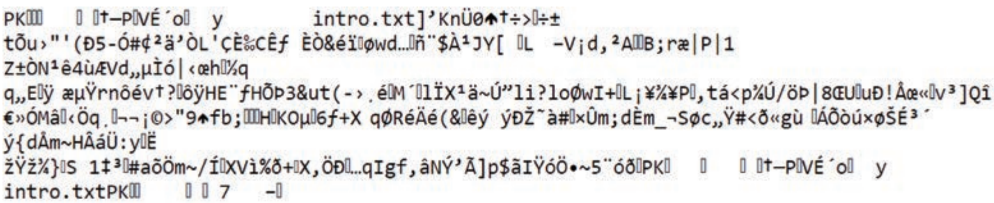
Creating a File in the `zip_file` Object and Saving That Object to Disk

Line 33 calls `output`'s `writestr` member function, which creates one file (`"intro.txt"`) in the ZIP archive containing the text in `content`. Line 34 calls `output`'s `save` member function to store the `output` object's contents in the file specified by `zipFileName`:

```
32     // write content into a text file in output
33     output.writestr("intro.txt", content); // create file in ZIP
34     output.save(zipFileName); // save output to zipFileName
35
```

ZIP Files Appear to Contain Random Symbols

ZIP is a binary format, so if you open the compressed file in a text editor, you'll see mostly gibberish. Below is what the file looks like in the Windows Notepad text editor:



Reading the Contents of the ZIP File

You can locate the ZIP file on your system and extract (decompress) its contents to confirm that the ZIP file was written correctly. The `miniz-cpp` library also supports reading and processing a ZIP file's contents programmatically. The following statement creates a `zip_file` object named `input` and initializes it with the name of a ZIP file:

```
36     miniz_cpp::zip_file input{zipFileName}; // load zipFileName
37
```

This reads the corresponding ZIP archive's contents. We can then use the `zip_file` object's member functions to interact with the archived files.

Displaying the Name and Contents of the ZIP File

The following statements call `input`'s `get_filename` and `printdir` member functions to display the ZIP's file name and a directory listing of the ZIP file's contents, respectively.

```
38     // display input's file name and directory listing
39     cout << "\n\nZIP file's name: " << input.get_filename()
40        << "\n\nZIP file's directory listing:\n";
41     input.printdir();
42
```

```
ZIP file's name: c:\users\useraccount\Documents\test.zip

ZIP file's directory listing:
  Length      Date    Time    Name
---------  ---------- -----   ----
      632  11/28/2021 16:48   intro.txt
---------                     -------
      632                     1 file
```

The output shows that the ZIP archive contains the file `intro.txt` and that the file's length is `632`, which matches that of the string `content` we wrote to the file earlier.

Getting and Displaying Information About a Specific File in the ZIP Archive

Line 44 declares and initializes the `zip_info` object `info`:

```
43     // display info about the compressed intro.txt file
44     miniz_cpp::zip_info info{input.getinfo("intro.txt")};
45
```

Calling `input`'s `getinfo` member function returns a `zip_info` object (from namespace `miniz_cpp`) for the specified file in the archive. Sometimes objects expose data so that you can access it directly using the object's name and a dot (`.`) operator. For example, the object `info` contains information about the archive's `intro.txt` file, including the file's name (`info.filename`), its uncompressed size (`info.file_size`) and its compressed size (`info.compress_size`):

```
46     cout << "\nFile name: " << info.filename
47        << "\nOriginal size: " << info.file_size
48        << "\nCompressed size: " << info.compress_size;
49
```

```
File name: intro.txt
Original size: 632
Compressed size: 360
```

Note that `intro.txt`'s compressed size is 360 bytes—43% smaller than the original file. Compression amounts vary considerably, based on the type of content being compressed.

Extracting "intro.txt" and Displaying Its Original Contents

You can extract the original contents of a compressed file from the ZIP archive. Here we use the `input` object's `read` member function, passing the `zip_info` object (`info`) as an argument. This returns as a `string` the contents of the file represented by the object `info`:

```
      // original file contents
      string extractedContent{input.read(info)};

```

We output `extractedContent` to show that it matches the original string `content` that we "zipped up." This was indeed a lossless compression:

```
      cout << "\n\nOriginal contents of intro.txt:\n"
         << extractedContent << "\n";
   }
```

```
Original contents of intro.txt:
This chapter introduces all but one of the remaining control statements--the
for, do...while, switch, break and continue statements. We explore the
essentials of counter-controlled iteration. We use compound-interest
calculations to begin investigating the issues of processing monetary
amounts. First, we discuss the representational errors associated with
floating-point types. We use a switch statement to count the number of A, B,
C, D and F grade equivalents in a set of numeric grades. We show C++17's
enhancements that allow you to initialize one or more variables of the same
type in the headers of if and switch statements.
```

20 4.14 C++20 Text Formatting with Field Widths and Precisions

Section 3.13 introduced C++20's `format` function (in header `<format>`), which provides powerful new text-formatting capabilities. Figure 4.12 shows how `format` strings can concisely specify what each value's format should be. We reimplement the formatting introduced in Fig. 4.4's compound-interest problem. Figure 4.12 produces the same output as Fig. 4.4, so we'll focus exclusively on the format strings in lines 13, 14, 17 and 22.

```
1  // fig04_12.cpp
2  // Compound-interest example with C++20 text formatting.
3  #include <iostream>
4  #include <cmath> // for pow function
5  #include <fmt/format.h> // in C++20, this will be #include <format>
6  using namespace std;
7  using namespace fmt; // not needed in C++20
8
9  int main() {
10    double principal{1000.00}; // initial amount before interest
11    double rate{0.05}; // interest rate
12
```

Fig. 4.12 | Compound-interest example with C++20 string formatting. (Part 1 of 2.)

```
13      cout << format("Initial principal: {:>7.2f}\n", principal)
14          << format("    Interest rate: {:>7.2f}\n", rate);
15
16      // display headers
17      cout << format("\n{}{:>20}\n", "Year", "Amount on deposit");
18
19      // calculate amount on deposit for each of ten years
20      for (int year{1}; year <= 10; ++year) {
21         double amount = principal * pow(1.0 + rate, year);
22         cout  << format("{:>4d}{:>20.2f}\n", year, amount);
23      }
24   }
```

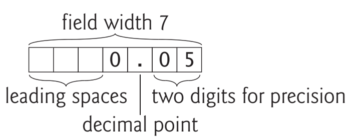
```
Initial principal: 1000.00
    Interest rate:    0.05

Year   Amount on deposit
   1             1050.00
   2             1102.50
   3             1157.63
   4             1215.51
   5             1276.28
   6             1340.10
   7             1407.10
   8             1477.46
   9             1551.33
  10             1628.89
```

Fig. 4.12 | Compound-interest example with C++20 string formatting. (Part 2 of 2.)

Formatting the Principal and Interest Rate

The `format` calls in lines 13 and 14 each use the placeholder `{:>7.2f}` to format the values of `principal` and `rate`. A colon (`:`) in a placeholder introduces a **format specifier** that indicates how a corresponding value should be formatted. The format specifier `>7.2f` is for a floating-point number (`f`) that should be **right-aligned (>)** in a 7-character field width that has two digits of precision (`.2`) to the right of the decimal point. Unlike `setprecision` and `fixed` shown earlier, format settings specified in placeholders are not "sticky"—they apply only to the value that's inserted into that placeholder.

The value of `principal` (`1000.00`) requires exactly seven characters to display, so no spaces are required to fill out the field width. The value of `rate` (`0.05`) requires only four total character positions, so it will be right-aligned in the field of seven characters and filled from the left with leading spaces, as in



Numeric values are right-aligned by default, so the > is not required here. You can **left-align** numeric values in a field width via <.

Formatting the Year and Amount-on-Deposit Column Heads

In line 17's format string

```
"\n{}{:>20}\n"
```

the string `"Year"` is simply placed at the position of the first placeholder, which does not contain a format specifier. The second placeholder indicates that `"Amount on Deposit"` (17 characters) should be right-aligned (>) in a field of `20` characters—`format` inserts three leading spaces to right-align the string. Strings are left-aligned by default, so the > is required here to force right-alignment.

Formatting the Year and Amount-on-Deposit Values in the `for` Loop

The format string in line 22

```
"{:>4d}{:>20.2f}\n"
```

uses two placeholders to format the loop's output. The placeholder `{:>4d}` indicates that `year`'s value should be formatted as an integer (`d` means decimal integer) right-aligned (>) in a field of width `4`. This right-aligns all the year values under the `"Year"` column.

The placeholder `{:>20.2f}` formats `amount`'s value as a floating-point number (`f`) right-aligned (>) in a field width of `20` with a decimal point and two digits to the right of the decimal point (`.2`). Formatting the `amounts` this way *aligns their decimal points vertically*, as is typical with monetary amounts. The field width of 20 right-aligns the `amounts` under `"Amount on Deposit"`.

4.15 Wrap-Up

In this chapter, we completed our introduction to all but one of C++'s control statements, which enable you to control the flow of execution in functions. Chapter 3 discussed `if`, `if…else` and `while`. Chapter 4 demonstrated `for`, `do…while` and `switch`. We showed C++17's enhancements that allow you to initialize a variable in the header of an `if` and `switch` statement. You used the `break` statement to exit a `switch` statement and to terminate a loop immediately. You used a `continue` statement to terminate a loop's current iteration and proceed with the loop's next iteration. We introduced C++'s logical operators, which enable you to use more complex conditional expressions in control statements.

In the Objects-Natural case study, we used the `miniz-cpp` open-source library to create and read compressed ZIP archive files. Finally, we introduced more of C++20's powerful and expressive text-formatting features. In Chapter 5, you'll create your own custom functions.

This page intentionally left blank

Index

Symbols

A

B

D

G

H

M